codemanship

TDD

Jason Gorman

This edition first published in 2016

Codemanship Limited

www.codemanship.com

Printed in Great Britain

ABOUT THE AUTHOR

Jason Gorman is a software developer, trainer and coach based in London. A TDD practitioner since before it had a name, he's helped thousands of developers to learn this essential discipline through his company *Codemanship*. He's the founder of the original international *Software Craftsmanship 20xxx* conference, an activist for software developer apprenticeships, a patron of the Bletchley Park Trust, a one-time-only West End producer, a failed physicist, and a keen amateur musician. His twelve fans know him as *Apes With Hobbies*.

You can follow him on Twitter (@jasongorman), or email jason.gorman@codemanship.com

ABOUT CODEMANSHIP

Founded in 2009, Codemanship provides training, coaching and consulting in the practical software disciplines that enable organisations to sustain the pace of digital innovation. Based in London, Codemanship has trained and guided teams in TDD, refactoring, software design, Continuous Integration and Continuous Delivery, and Agile Software Development for a wide range of clients including the BBC, UBS, Waters plc, Ordnance Survey, salesforce.com, Electronic Arts, John Lewis, Redgate and Sky.

You can find out more about Codemanship training and coaching at www.codemanship.com

REVIEWERS

Will Price

Mark Withall

Phil Proom

Jon Barber

Erik De Bonte

Antony Gorman

Ilya Agoshkov

François Renaud-Philippon

CONTENTS

1. BEFORE WE BEGIN

Summary:

1. To learn TDD, you must do TDD
2. You can tackle the exercises in any OO language
3. You will need:
    a. A unit testing tool, based on the xUnit pattern, that supports parameterized tests
    b. Automated refactoring menu in your code editor
    c. Mock objects framework
    d. Pencil & paper

TDD is a practical discipline, like riding a bicycle or playing the piano. To learn it, you must do it.

The focus of this book will be on doing TDD, and for that reason you will only get the best from it if you try the exercises.

I've tried as much as possible to keep the technology choices open; you can tackle the exercises in any object oriented programming language you like.

But, whether you do them in Java (like I am), or C#, or Ruby, or Python, or C++, you will need a number of things to get started:

- A **unit testing tool** for that language
- Ideally, a menu of **automated refactorings** in your code editor that will do the donkey work of refactoring for you
- A framework for creating **"mock objects"** that can be used in some of your tests
- **Pencil and paper**

All of these things are freely available for most programming languages.

The xUnit unit testing framework design pattern (invented by Kent Beck and others) has been implemented in pretty much every OO language – JUnit for Java, NUnit for .NET, RUnit for Ruby, etc.

Pick the one you're most comfortable with. With one proviso: make sure you pick one that enables you to write *parameterized tests*. As you'll see, they are used extensively by experienced TDD-ers.

Automated refactorings vary from editor to editor and language to language. You will find that dynamically typed languages suffer a disadvantage, as there is usually type information missing about methods and method calls that a tool would need for some refactorings. Java, Smalltalk and C# have excellent automated refactoring support. JavaScript arguably has the worst. If you're working in a scripting language like JS, expect to have to do some refactoring by hand.

Mock object frameworks, again, vary in quality. But, in this book, we will use them in quite specific ways that pretty much all of them can handle.

Finally, have a pencil and paper handy. Always. Throughout the book, we'll see situations where we might want to note things down, or make a list of tests, or sketch a simple design. Not all our thinking gets done in code.

2. WHY DO TDD?

Summary:

- TDD helps us to build the right software
- TDD helps to avoid building features we don't need, and making the design too complicated
- Refactoring is a key part of TDD. It helps us to keep code easy to change
- The short cycles of TDD, together with fast-running automated tests, help us to keep our software always working
- TDD helps us to deliver working software sooner, and for longer

Popularised in the late 1990s by Kent Beck, Test-Driven Development ("TDD") combines practices that the best programmers have used since the 1950s.

Done well, it helps us to address some key problems in the way we write software:

- Building the right thing
- Keeping the design simple
- Producing code that's easy to change
- Making sure the software always works
- Sustaining the pace of development

BUILDING THE RIGHT THING

Imagine we're designing a new kitchen. We could make a list of all the things we think the kitchen needs: a cooker, a sink, a refrigerator, a toaster, a kettle, cupboards, and so on.

What happens when we approach design as a “shopping list” of features is that, after it’s built, we discover we left stuff out that we needed. For example, we might want to make fresh pasta, but didn’t put a pasta machine on our list.

To avoid finding out too late that our shopping list of features is wrong, we start instead by considering examples of *how the kitchen will be used*, and figure out what features it needs to do that.

TDD works this way. We use *tests as specifications* for what we want to do using the software.

KEEPING THE DESIGN SIMPLE

Along with the risk of leaving important features out of our design, there’s also the risk of including features that aren’t needed at all.

In software (as well as kitchens), unneeded features and unnecessary complexity add costs, both initial and ongoing.

TDD encourages us to write the simplest code possible to pass our tests. If the test doesn’t specifically ask for it, *you ain’t gonna’ need it*.

PRODUCING CODE THAT’S EASY TO CHANGE

Seven decades of computer programming history has taught that us that our code will almost certainly need to change.

If code is difficult to understand, complicated, full of duplication, and too interconnected, then it will be expensive to change.

TDD explicitly includes a discipline called *refactoring* that helps us to keep our code as easy to change as possible.

After we write the code to pass each test, we stop to *refactor* the code to make it simpler and easier to understand, to eliminate duplication, and to manage the dependencies in the code to localise the impact of changes.

MAKING SURE THE SOFTWARE ALWAYS WORKS

Software that doesn't work has no value. While we're editing the code, the code isn't working.

TDD breaks development down into small cycles. These *micro-iterations* typically last just a few minutes, at the end of which we have tested, working code that *could* be shipped if necessary.

We automate the tests so they can be run quickly. This way, after each small change, we can re-test the software to make sure it still works.

SUSTAINING THE PACE OF DEVELOPMENT

Keeping the code always working means we can deliver working software sooner.

And TDD also helps us sustain the pace of development for longer.

Adding new features to new software is easy, and our initial productivity is high.

But as the code grows, it becomes harder and harder to change it without breaking it.

The rising cost of change hinders teams trying to respond to the changing needs of end users. The software becomes a liability instead of a benefit.

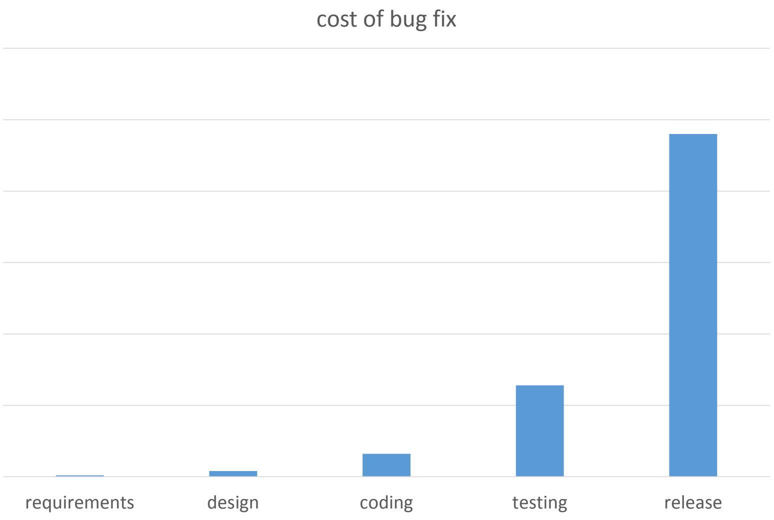
TDD tackles the factors that make code harder to change head-on.

RELIABILITY VS. PRODUCTIVITY

Too many developers have an unrealistic view of the relationship between the quality of their software and the time and cost of creating it. The received wisdom is that more reliable software takes longer to write.

A mountain of good industry data, however, paints a different picture. Far from costing more, in the vast majority of cases improving the reliability of our code would actually end up costing us *less*.

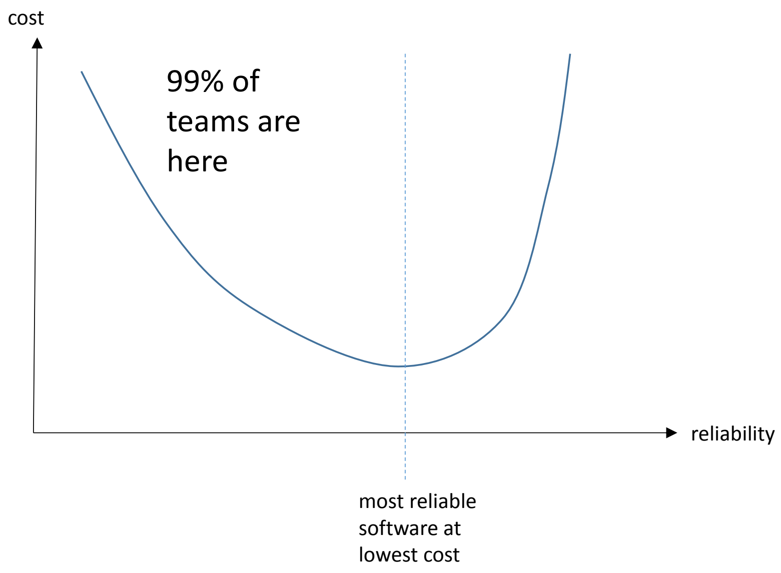
The counterintuitive causal mechanism for this strange effect has been known for several decades. The later we discover them, the more bugs cost to fix. A bug discovered by users in production can cost 100x more to fix than if it had been caught as soon as the programmer made the error.



The difference in cost of fixing bugs at later stages in development can be so large that, by taking more care to catch them sooner, we can actually end up going faster.

This is a strategy called *defect prevention*, and it has been hugely successful at not only helping teams to improve the reliability of their code, but also to save time and money delivering working software. It's a win-win.

The net result is that better software usually costs less to create.



TDD can help get us into a "sweet spot" of the most reliable code at the lowest cost in four ways:

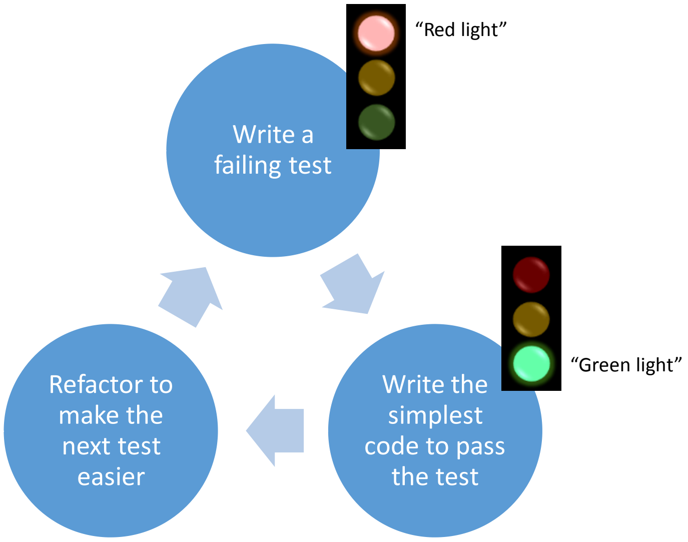
- Agreeing executable tests catches many requirements misunderstandings before we've written any code. These requirements "bugs" can cost hundreds of times more to fix in user testing or production
- TDD breaks coding down into "baby steps", bringing more focus to every line of code we write and highlighting more errors that we might have missed taking bigger bites
- TDD encourages us to keep our code simple, and simpler code is less likely to be wrong
- The automated tests TDD creates enable us to check for new bugs we might have introduced immediately after making a change

Studies done of teams adopting TDD have convincingly shown that, on average, test-driven code is much more reliable, but doesn't cost any more – and in many cases, costs less – to deliver working software.

TDD is arguably the first defect prevention technique to have gained widespread adoption.

3. WHAT IS TDD?

In essence, TDD is an iterative process that involves three steps:


"Red light"
Write a failing test
"Green light"
Write the simplest code to pass the test
Refactor to make the next test easier

The tests can be at any level of abstraction. They can be system tests, or component/service tests, or tests for individual classes.

Some developers use a traffic light analogy to remember the steps.

Each new failing test specifies something we want the software to do that it currently doesn't. (That's why it's failing.)

We flesh out our design one failing test at a time, adding *just enough* implementation to pass each new test, and keeping the code as easy to change as possible by refactoring.

4. HOW TO TDD

Summary:

- Start with the simplest failing test you can think of
- Write the simplest code you can think of to pass the test quickly
- If no need to refactor, move on to the next failing test
- Refactor your test code, too!
- Parameterized tests are a useful way to consolidate similar test methods
- Leave in duplication when it makes tests easier to understand
- Aim for one test method for each distinct rule. Use the test name to clearly convey the rule
- Tests should read like a specification
- Localise dependencies on the objects under test
- In TDD, we're done when we can't think of any more tests that *should* fail
- TDD is a process of design discovery
- Tests make changes safer and easier

The best way to explain how to test-drive a software design is with a simple example.

We're going to create some code that will calculate numbers in the Fibonacci sequence.

The Fibonacci sequence starts with zero and one, and then all subsequent numbers are the sum of the previous two.

i.e. 0, 1, 0+1=1, 1+1=2, 2+1=3, 5, 8, 13, 21 etc

FAILING TEST #1

We'll start by writing a failing test. (I'm doing it in Java, with the Junit testing framework.)

Try to think of the simplest test you could start with – the one that would be easiest to pass.

```
public class FibonacciTests {

   @Test
   public void firstNumberInSequenceIsZero() {
      assertEquals(0, new Fibonacci().getNumber(0));
   }

}
```

Let's write the *simplest* code that will pass the test:

```
public class Fibonacci {

   public int getNumber(int index) {
      return 0;
   }

}
```

Next, let's look at the code and see if we need to refactor it to make the next test easier.

At this point, it's hard to see how we could make this code easier to change.

So let's move on to the next failing test.

FAILING TEST #2

```
public class FibonacciTests {

   @Test
   public void firstNumberInSequenceIsZero() {
      assertEquals(0, new Fibonacci().getNumber(0));
   }

   @Test
   public void secondNumberInSequenceIsOne() {
      assertEquals(1, new Fibonacci().getNumber(1));
   }

}
```

Again, we write the simplest code that will pass *both* of these tests.

```
public class Fibonacci {

   public int getNumber(int index) {
      return index;
   }

}
```

Now that we're back on a green light, it's time to think about refactoring again.

The implementation code looks okay, but there's some very obvious duplication in the test code. *(Remember: test code needs to be easy to change, too!)*

The most direct way we could eliminate this duplication would be to turn these two very similar test methods into a single *parameterized test* covering both cases.

The built-in mechanism in JUnit for writing parameterized tests is a bit clunky, so I'm going to use JUnitParams (github.com/Pragmatists/JUnitParams) to make life easier.

```
@RunWith(JUnitParamsRunner.class)
public class FibonacciTests {

   @Test
   @Parameters({"0,0","1,1"})
   public void firstTwoNumbersAreSameAsIndex(int index,
                                             int expected) {
      assertEquals(expected,
                              new Fibonacci().getNumber(index));
   }

}
```

Now, for another failing test.

FAILING TEST #3

```
@RunWith(JUnitParamsRunner.class)
public class FibonacciTests {

   @Test
   @Parameters({"0,0","1,1"})
   public void firstTwoNumbersAreSameAsIndex(int index,
                                             int expected) {
      assertEquals(expected,
                          new Fibonacci().getNumber(index));
   }

   @Test
   public void thirdNumberInSequenceIsOne(){
      assertEquals(1, new Fibonacci().getNumber(2));
   }

}
```

And then the simplest code to pass all three tests:

```
public class Fibonacci {

  public int getNumber(int index) {
    if(index < 2)
      return index;
    return 1;
  }

}
```

Notice the branch in our implementation code. There are two distinct rules (or patterns) in our solution: one for the first two numbers, and another for the rest.

If our tests are to serve as specification, it helps enormously if the rules are obvious from reading the test code.

So, even though there's some obvious duplication of test code, in this instance *readability is more important*.

For this reason, I choose *not* to add this third test case to the parameterized test for the first two Fibonacci numbers.

This way, we end up with a *test method for each rule*, and we can use the names of those test methods to clearly communicate the rules.

But there's another bit of duplication in the test code we should get rid of.

Both tests know how to instantiate a *Fibonacci* object and invoke the *getNumber()* method. If the interface of Fibonacci changes, we'll need to change multiple tests. Let's refactor the test code to put that knowledge in one place.

```
@RunWith(JUnitParamsRunner.class)
public class FibonacciTests {

   @Test
   @Parameters({"0,0","1,1"})
   public void firstTwoNumbersAreSameAsIndex(int index,
                                             int expected) {
      assertEquals(expected, getFibonacciNumber(index));
   }

   @Test
   public void thirdNumberInSequenceIsOne(){
      assertEquals(1, getFibonacciNumber(2));
   }

   private int getFibonacciNumber(int index) {
      return new Fibonacci().getNumber(index);
   }

}
```

We find it's generally a good idea to limit the knowledge our test code has of the interfaces of the objects being tested.

Let's move on to another failing test.

FAILING TEST #4

```
@Test
public void fourthNumberInSequenceIsTwo(){
   assertEquals(2, getFibonacciNumber(3));
}
```

To pass this test, the simplest solution I could think of is:

```
public class Fibonacci {

   public int getNumber(int index) {
      if(index < 2)
         return index;
      return index - 1;
   }

}
```

We discovered one rule for the first two numbers, and a second rule for the next two.

Let’s refactor the test code to reflect that, with another parameterized test.

```
@Test
@Parameters({"2,1", "3,2"})
public void thirdNumberOnIsIndexMinusOne(int index,
                                          int expected){
   assertEquals(expected, getFibonacciNumber(index));
}
```

But we’re not done yet. How do we know that? We know because we can think of *more failing test cases*.

FAILING TEST #5

The sixth Fibonacci number has an index of 5 and a value of 5.

```
@Test
public void sixthNumberIsFive() {
   assertEquals(5, getFibonacciNumber(5));
}
```

To pass this test, the simplest change we can make to the implementation is:

```
public class Fibonacci {

   public int getNumber(int index) {
      if(index < 2)
         return index;
      return getNumber(index - 1) + getNumber(index - 2);
   }

}
```

The fifth number obeys the same rule as the third and fourth, so that extra test is duplication that doesn’t make the specification any easier to understand. Let’s merge it into the parameterized test for third and fourth, and rename the test method to more accurately describe the rule.

```
@RunWith(JUnitParamsRunner.class)
public class FibonacciTests {

   @Test
   @Parameters({"0,0","1,1"})
   public void firstTwoNumbersAreSameAsIndex(
                                    int index,
                                    int expected) {
      assertEquals(expected, getFibonacciNumber(index));
   }

   @Test
   @Parameters({"2,1", "3,2", "5,5"})
   public void thirdNumberOnIsSumOfPreviousTwo(int index,
                                    int expected){
      assertEquals(expected, getFibonacciNumber(index));
   }

   private int getFibonacciNumber(int index) {
      return new Fibonacci().getNumber(index);
   }

}
```

To finish up, let's see if we can think of any more failing test cases.

FAILING TEST #6

What would happen if we asked for the -1th Fibonacci number? We'd expect that to fail, because there is no -1th number.

```
@Test(expected=IllegalArgumentException.class)
public void indexMustBePositiveInteger() {
   getFibonacciNumber(-1);
}
```

To pass this test, we just need to check the parameter value satisfies the rule, and throw the specified exception if it doesn't.

```
public class Fibonacci {

   public int getNumber(int index) {
      if(index < 0)
         throw new IllegalArgumentException();
      if(index < 2)
         return index;
      return getNumber(index - 1) + getNumber(index - 2);
   }

}
```

Our tests now read like a specification for our Fibonacci calculator. Just by looking at the names of the test methods, we can see there are three distinct rules, and the names clearly convey what those rules are.

We *discovered* this design by working through a sequence of examples – failing tests – and doing the simplest things we could think of to pass them.

The end result is a working Fibonacci calculator, with a suite of fast-running automated tests that will help us if we need to change the calculator later.

WHY GO TO ALL THE TROUBLE?

Now, imagine we deliver this code to our end users, who complain that it's too slow on higher indexes.

This is because our algorithm is recursive, recalculating the same numbers many times.

We decide to replace it with an iterative solution that remembers and reuses numbers once they've been calculated.

```
public class Fibonacci {

   public int getNumber(int index) {
      if(index < 0)
         throw new IllegalArgumentException();

      int[] sequence = new int[index+1];

      for (int i = 0; i < sequence.length; i++) {
         if(i < 2){
            sequence[i] = i;
         }else{
            sequence[i] = sequence[i - 1] + sequence[i-2];
         }
      }

      return sequence[index];
   }
}
```

It's much safer to make this change because we have a good set of automated tests that will alert us straight away if we break the software.

This is a very important thing to remember about TDD: it may seem like overkill to take such baby steps and write so many tests for such a simple problem. But we've learned that by far the greater cost in software development is the cost of changing code later, and for the extra up-front investment of TDD, we get a potentially much larger pay-off.

EXERCISE #1

a. Test-drive some code that will generate a list of prime numbers that are less than 1,000
b. Test-drive some code that will convert integers from 1 to 4,000 into Roman Numerals

EXERCISE #2

Test-drive some code that will calculate the total net value of items in a shopping cart represented as a list of unit price and quantity – e.g., {{10.0, 5}, {25.5, 2}}, with the following discounts applied:

1. If total gross value > £100, apply a 5% discount
2. If total gross value > £200, apply a 10% discount

5. THE GOLDEN RULE

Summary:

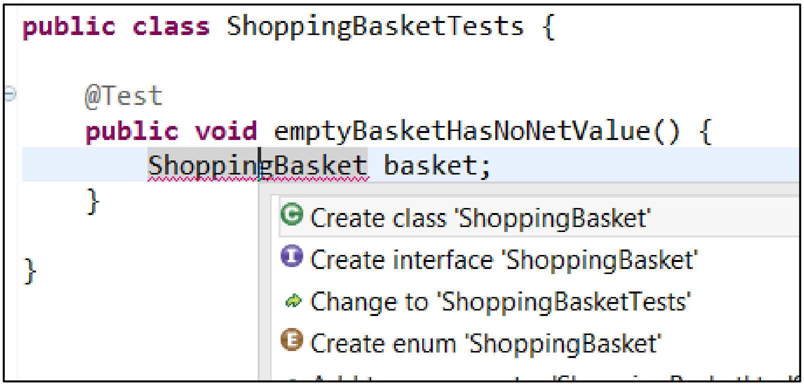
- Don’t write source code until a test requires it
- Reference new classes, methods, variables etc in your test first, so the code won’t compile, and then fix it by declaring them
- Aim to have just one thing broken at a time if possible

As the name implies, Test-Driven Development drives software design directly from tests.

In practice, what this means is:

> We do not write any source code until we have a failing test that requires it

So, when we’re test-driving a class, we don’t declare the class and then start writing tests for it. We start by writing a test, and only declare the class when the test needs us to.



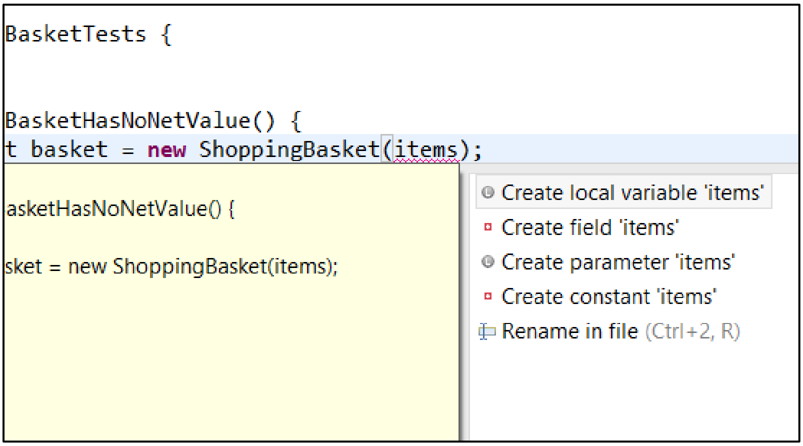
```
public class ShoppingBasketTests {

    @Test
    public void emptyBasketHasNoNetValue() {
        ShoppingBasket basket;
    }

}
```

Create class 'ShoppingBasket'
Create interface 'ShoppingBasket'
Change to 'ShoppingBasketTests'
Create enum 'ShoppingBasket'

As I tackle the shopping basket exercise, I start by writing a failing test that uses the *ShoppingBasket* class I intend to create.

My editor flags up that there's no such class, and prompts me to create one.

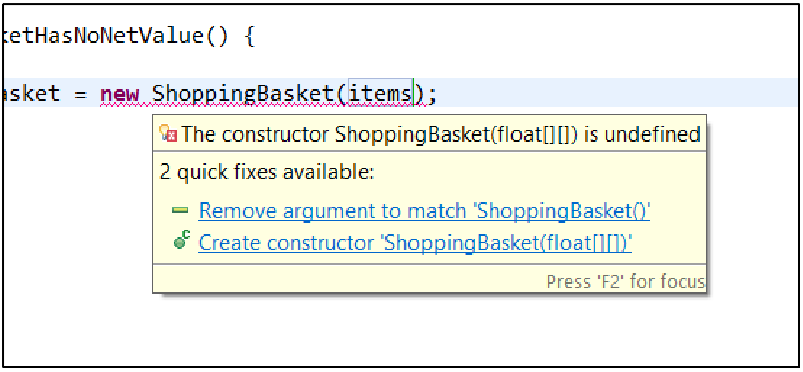
Until I do that, the code won't compile. It's a *broken* test. The Golden Rule gives me permission to fix it so I can move on. In TDD, a broken test is a failing test.



```
BasketTests {

BasketHasNoNetValue() {
t basket = new ShoppingBasket(items);
```

```
asketHasNoNetValue() {

sket = new ShoppingBasket(items);
```

Next, I write code that passes a variable called *items* into the – as yet non-existent - constructor of *ShoppingBasket*. Again, Eclipse tells me there's no such variable, and prompts me to fix it by declaring one.



```
etHasNoNetValue() {

sket = new ShoppingBasket(items);
```

The constructor ShoppingBasket(float[][]) is undefined

2 quick fixes available:

Remove argument to match 'ShoppingBasket()'
Create constructor 'ShoppingBasket(float[][])'
Press 'F2' for focus

Moving on, we create the constructor. And keep going in this fashion, only declaring source code when the test requires it.

Of course, we *could* write the entire test, and then declare everything it needs. But in TDD, we favour the shortest feedback cycles, and so prefer to have one thing broken at a time if possible.

TEST-DRIVEN DESIGN VS. DESIGN-DRIVEN TESTING

A classic mistake programmers new to TDD make is to write failing tests that assert a design they have in mind, rather than a behaviour or a rule they want the software to handle.

For example, some people will write a test for a class they want to declare:

```
@Test
public void forecastIsNotNull() {
   WeatherForecast forecast = new WeatherForecast();
   assertNotNull(forecast);
}
```

In a literal interpretation of the Golden Rule, this gives them permission to declare the class *WeatherForecast*. But, as we'll see in their next test, it's redundant.

```
@Test
public void forecastForTodayIsAverageOfPreviousTwo(){
   double[] previousDays = new double[]{17, 18};
   assertEquals(17.5,
         new WeatherForecast(previousDays).forecast());
}
```

If *WeatherForecast* doesn't exist, this second test won't even compile. Most importantly, we only need to declare the class so that we can test the result of *forecast()*.

Be wary of writing tests like this, or that test getters, or other aspects of the implementation's structure. Chances are, you're doing what some of us call "design-driven testing", and not "test-driven design".

Focus your failing tests on the results of desired behaviour, and details like this will fall out naturally as we work our way to a solution.

EXERCISE #3

Repeat exercises #1 and #2, applying the Golden Rule

6. START WITH THE QUESTION

Summary:

- Write the test assertion first and work backwards to the set-up
- Tests have 3 components – set-up, action & assertions
- Starting with the assertion helps us to discover what set-up we need

Functional tests have three components:

- The set-up: arranges objects and test data for the test
- The action: invokes the method or function being tested
- The assertion(s): asks the questions that will tell us if the action worked

Intuitively, we tend to write test code in that order. But that can lead us into difficulties.

How do we know what set-up we need for the test? It's not uncommon, when we write tests in the Arrange->Act->Assert order, to get to the assertion and realise we've written the wrong set-up for the question we want to ask.

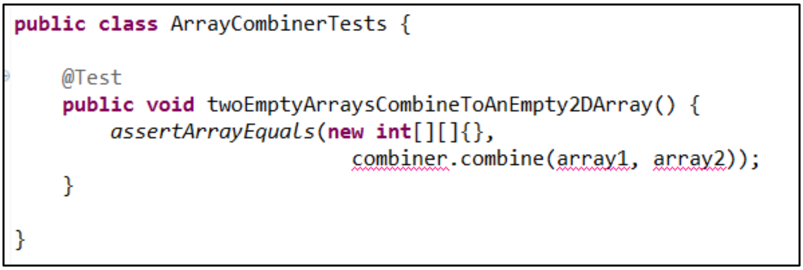
The test is all about the question, so in TDD we recommend you start there and work *backwards* to the set-up you need to ask it.

This may take some getting used to, but – with practice – you'll start to feel comfortable doing it this way.

Let's look at an example to illustrate how to work backwards from assertions.

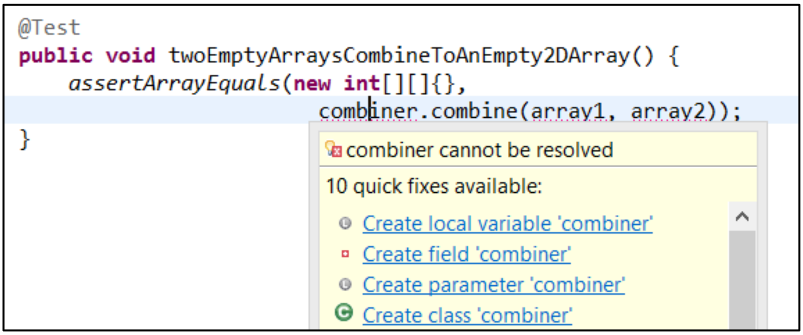
In this example, we're test-driving some code to combine 2 1-dimensional arrays into a single 2D array.

We start by writing the assertion:


```
public class ArrayCombinerTests {

    @Test
    public void twoEmptyArraysCombineToAnEmpty2DArray() {
        assertArrayEquals(new int[][]{},
                              combiner.combine(array1, array2));
    }

}
```

Notice that our assertion references three local variables that haven't been declared yet. By writing the assertion first, we've discovered what set-up our test will need.

Now, let's work backwards to create the set-up.

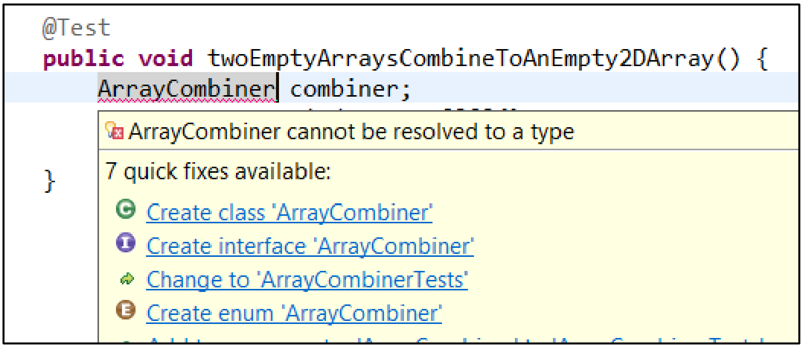

```
@Test
public void twoEmptyArraysCombineToAnEmpty2DArray() {
    assertArrayEquals(new int[][]{},
                          combiner.combine(array1, array2));
}
```

combiner cannot be resolved

10 quick fixes available:

- Create local variable 'combiner'
- Create field 'combiner'
- Create parameter 'combiner'
- Create class 'combiner'

My editor prompts me to create a local variable called *combiner*, which I declare as type *ArrayCombiner*.


```
@Test
public void twoEmptyArraysCombineToAnEmpty2DArray() {
    ArrayCombiner combiner;

}
```

ArrayCombiner cannot be resolved to a type

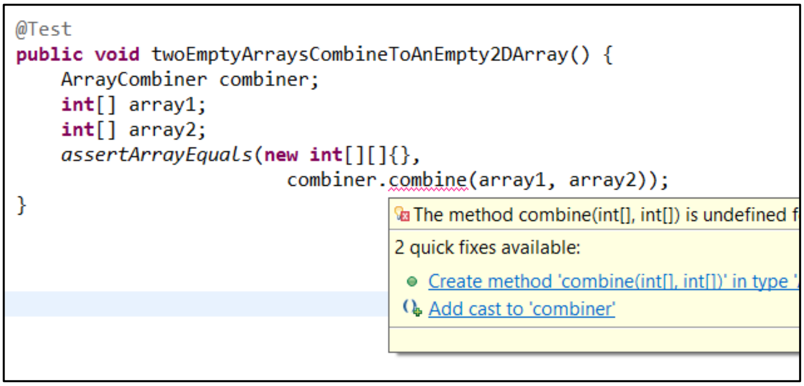
7 quick fixes available:

- Create class 'ArrayCombiner'
- Create interface 'ArrayCombiner'
- Change to 'ArrayCombinerTests'
- Create enum 'ArrayCombiner'

I'm then prompted to create that class.

In a similar fashion, I work my way backwards to declaring local variables called *array1* and *array2.*

Then I'm prompted to declare the *combine()* method, which is the action we're testing.



```
@Test
public void twoEmptyArraysCombineToAnEmpty2DArray() {
    ArrayCombiner combiner;
    int[] array1;
    int[] array2;
    assertArrayEquals(new int[][]{},
                        combiner.combine(array1, array2));
}
```

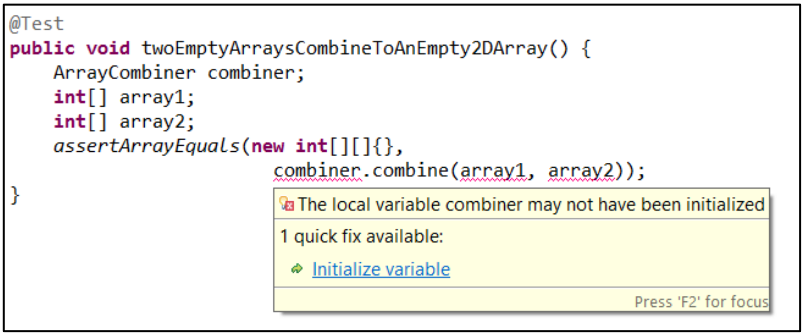
The method combine(int[], int[]) is undefined f

2 quick fixes available:

Create method 'combine(int[], int[])' in type '

Add cast to 'combiner'

Next, I'm asked to instantiate the variables in our test set-up.



```
@Test
public void twoEmptyArraysCombineToAnEmpty2DArray() {
    ArrayCombiner combiner;
    int[] array1;
    int[] array2;
    assertArrayEquals(new int[][]{},
                        combiner.combine(array1, array2));
}
```

The local variable combiner may not have been initialized

1 quick fix available:

Initialize variable

Press 'F2' for focus

Once *combiner*, *array1* and *array2* have been initialised in the correct state for our test, we have the complete set-up.

```java
public class ArrayCombinerTests {

   @Test
   public void twoEmptyArraysCombineToAnEmpty2DArray(){
      ArrayCombiner combiner = new ArrayCombiner();
      int[] array1 = new int[]{};
      int[] array2 = new int[]{};
      assertArrayEquals(new int[][]{},
                        combiner.combine(array1, array2));
   }
}
```

EXERCISE #4

Writing the assertions first and working backwards to the set-up, test-drive some code to calculate how much water will be needed to fill the following:

1. A cube
2. A cylinder
3. A pyramid

7. TEST YOUR TESTS

Summary:

- See the test assertion fail, so you know that if the result is wrong, the test will catch that
- Implement just enough to see the assertion fail
- Test names should clearly convey what's supposed to happen, to help developers fix it when a test fails
- How we write assertions can make a difference to how helpful test failure messages are in identifying the cause
- Expected exceptions and mock object expectations are kinds of assertions

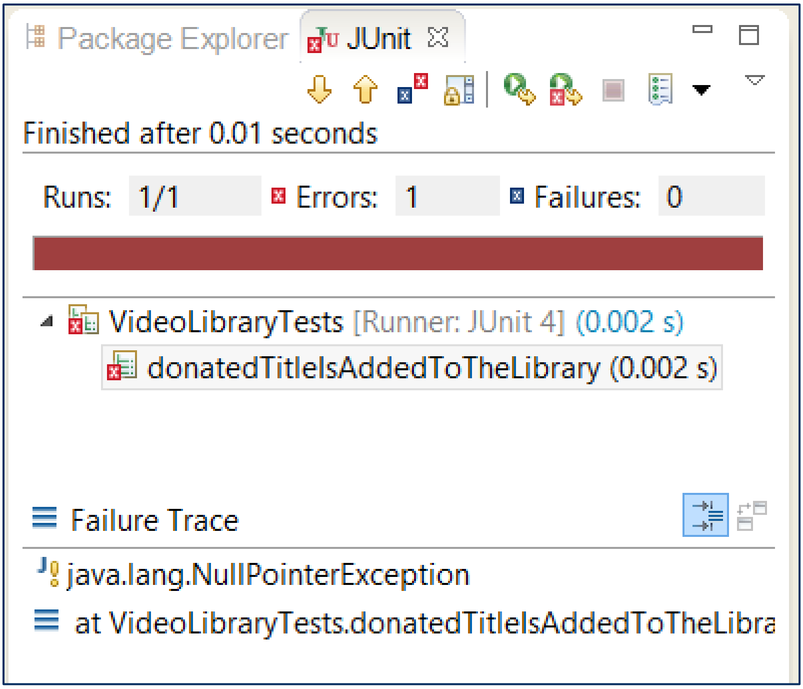
In order for our automated tests to give us good assurance that the code's working, they need to be *good* tests.

It's important to check that, if the result we get is wrong, the test will fail.

For this reason, it's highly recommended that – before you write the code to pass the test – you see the test fail for the right reason.

```
public class VideoLibraryTests {

   @Test
   public void donatedTitleIsAddedToTheLibrary() {
      VideoTitle title = new VideoTitle();
      VideoLibrary library = new VideoLibrary();
      library.donate(title);
      assertTrue(library.getTitles().contains(title));
   }
}
```

When I run this test for donating a video title to a community library, I get the result:



Finished after 0.01 seconds

Runs: 1/1 Errors: 1 Failures: 0

VideoLibraryTests [Runner: JUnit 4] (0.002 s)
donatedTitleIsAddedToTheLibrary (0.002 s)

Failure Trace

```
java.lang.NullPointerException
at VideoLibraryTests.donatedTitleIsAddedToTheLibra
```

This is because, at the moment, *VideoLibrary.getTitles()* returns null (because I haven't written that code yet).

The test assertion hasn't been evaluated. It didn't get that far because of the unhandled *NullPointerException.*

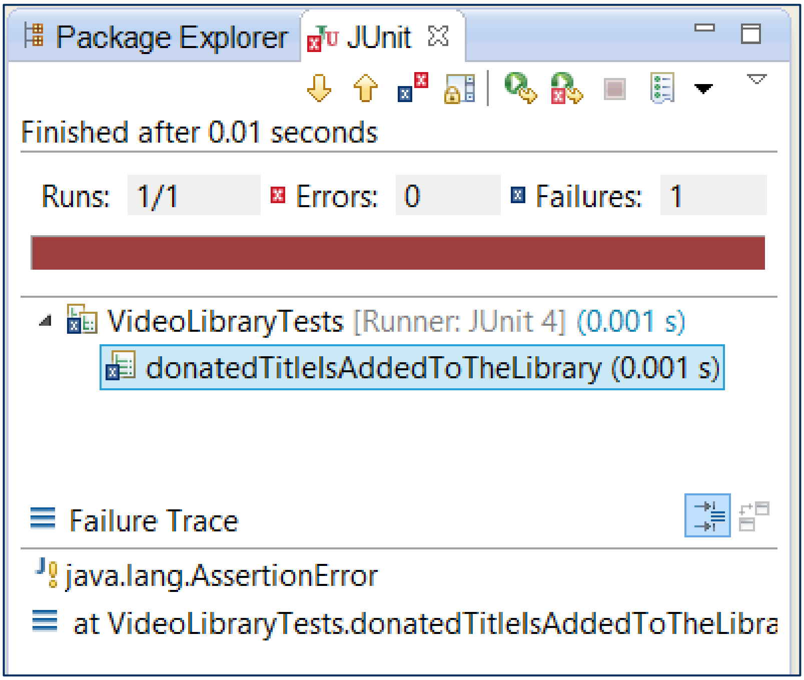
To have confidence in this test, what I need to know is if the assertion will fail when the donated title isn't added to that collection. So I must add just enough implementation to see that happen.

```
public class VideoLibrary {

   public List<VideoTitle> getTitles() {
      return new ArrayList<VideoTitle>();
   }

   public void donate(VideoTitle title) {

   }
}
```

Now we can see that the test does indeed fail if the donated title isn't in added to the library.

SIDENOTE

Assertions don't just come in the assert…() variety. Expected exceptions, and mock object expectations (which we'll cover later), are also kinds of assertions. Make sure you see them fail, too.



Finished after 0.01 seconds

Runs: 1/1 Errors: 0 Failures: 1

VideoLibraryTests [Runner: JUnit 4] (0.001 s)
donatedTitleIsAddedToTheLibrary (0.001 s)

Failure Trace

```
java.lang.AssertionError
at VideoLibraryTests.donatedTitleIsAddedToTheLibra
```

When tests fail, this is our opportunity to send a message to some developer in the future who might be asked to change our code (and that could be us!)

The most important piece of information is *"What should have happened?"* And the best place to convey this is in the name of the test.

FLUENT ASSERTIONS

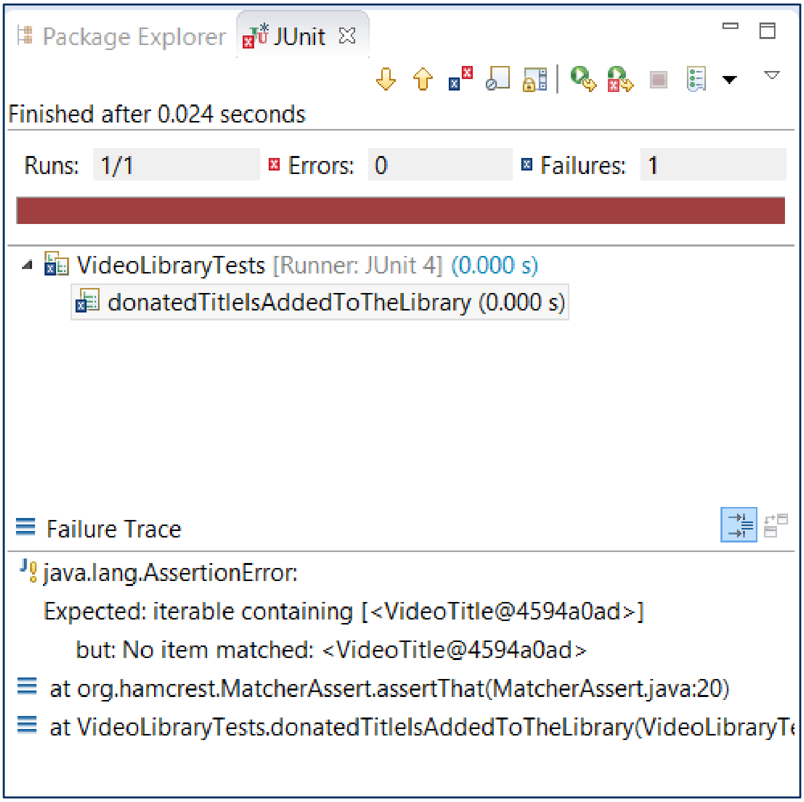
Although we now have confidence that if the donated title wasn't added to the library, this test would catch that, we have to read the test method name to know what wasn't true. In this example, it may be obvious, but often we need more information than a test name can give us.

It's becoming more popular for developers to write what are called "fluent assertions" – assertions that can provide extra information about exactly which part of the assertion failed.

For example, using Hamcrest (www.hamcrest.org), I could rewrite my assertion:

```
@Test
public void donatedTitleIsAddedToTheLibrary() {
   VideoTitle title = new VideoTitle();
   VideoLibrary library = new VideoLibrary();
   library.donate(title);
   assertThat(library.getTitles(), contains(title));
}
```

When this test fails, we get more information in the failure trace.



Finished after 0.024 seconds

Runs: 1/1 Errors: 0 Failures: 1

VideoLibraryTests [Runner: JUnit 4] (0.000 s)
donatedTitleIsAddedToTheLibrary (0.000 s)

Failure Trace

```
java.lang.AssertionError:
Expected: iterable containing [<VideoTitle@4594a0ad>]
     but: No item matched: <VideoTitle@4594a0ad>
at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
at VideoLibraryTests.donatedTitleIsAddedToTheLibrary(VideoLibraryTe
```

EXERCISE #5

Test-drive code to leave reviews for movies, with:

- A rating from 1-5
- The name of the reviewer (defaulted to “Anonymous” if not supplied)
- The text of the review

It should calculate an average rating for a movie, and also report the number of reviews for each rating. E.g.,

The Abyss

Rating	No. of Reviews
5	13
4	11
3	4
2	5
1	2

Make sure you apply all the ideas we’ve seen up to this point, including seeing the test assertions fail for the right reasons.

8. ONE REASON TO FAIL

Summary:

- Tests should ask a single question, so that:
    - We can bring more focus to each design decision
    - Get more feedback with each decision
    - More easily debug when tests fail
    - Test code is easier to understand

When we test-drive the design of our code, we strive to take baby steps, making one decision at a time and getting feedback with each step.

For this and other reasons, it works best when each test asks only one question.

```
public class LibraryTests {

  @Test
  public void donatedTitlesAddedToLibrary() {
     Library library = new Library();
     VideoTitle title = new VideoTitle();
     Member donor = new Member();
     library.donate(title, donor);
     assertTrue(library.contains(title));
     assertEquals(1, title.getRentalCopyCount());
     assertEquals(10, donor.getPriorityPoints());
  }
}
```

In this example, our test asks three questions. We've made three design decisions in a single step, and will have to do more to get it the test to pass.

Think, too, about what will happen if this test fails. Which part of the implementation is broken? Tests that ask too many questions are harder to debug when things break.

Tests that ask too many questions bring less focus on each design decision and less feedback as we go - with the inevitable impact on code quality that we observe as feedback cycles get longer.

It's better to tackle this example in three tests, each one asking a specific question.

```
public class LibraryTests {

   private Library library;
   private VideoTitle title;
   private Member donor;

   @Before
   public void donateTitle() {
      library = new Library();
      title = new VideoTitle();
      donor = new Member();
      library.donate(title, donor);
   }

   @Test
   public void donatedTitlesAddedToLibrary(){
      assertTrue(library.contains(title));
   }

   @Test
   public void donatedTitlesHaveOneDefaultRentalCopy(){
      assertEquals(1, title.getRentalCopyCount());
   }

   @Test
   public void donorsGetTenPriorityPoints(){
      assertEquals(10, donor.getPriorityPoints());
   }
}
```

Notice how giving each question its own test enables us to document each rule with the method names, making the tests easier to understand.

Some people naively interpret the need for tests to ask only question as meaning literally "every test should only have one assertion". It's not that simple, though.

```
@Test
public void fibonacciSequenceIsGenerated() {
   Fibonacci fibonacci = new Fibonacci();
   assertEquals("0,1,1,2,3,5,8,13",
                      fibonacci.generateSequence(8));
}
```

How many reasons does this test have to fail? I can see nine: each individual number in the sequence has to be calculated correctly, and they have to be separated by commas.

This approach means taking big leaps instead of baby steps, making multiple design decisions before getting any feedback.

Better to break it down, like:

```
@Test
public void firstNumberInSequenceIsZero() {
   Fibonacci fibonacci = new Fibonacci();
   assertEquals("0",
        fibonacci.generateSequence(8).split(",")[0]);
}
```

In TDD, the ability to break problems down into the smallest questions is key.

Finally, be careful about alternative kinds of test assertions. How many reasons does this test have to fail?

```
@Test
public void donatedTitlesAddedToLibrary() {
   Library library = new Library();
   VideoTitle title = new VideoTitle();
   Member donor = mock(Member.class);
   library.donate(title, donor);
   assertTrue(library.contains(title));
   verify(donor).awardPoints(10);
}
```

9. TESTS SHOULD BE SELF-EXPLANATORY

Summary:

- Choose names of test methods to clearly convey what the test is
- Use names for helper methods, objects, fields, constants and variables that clearly convey their role in the tests
- Use test fixture names that make it easy to find tests
- Pick test data that highlights boundaries in the logic
- Name literal values – using constants or variables – if it makes their significance clearer
- Some duplication in test code is fine when it makes the test easier to understand

```
public class Tests1 {

   private BankAccount a1;
   private BankAccount a2;

   @Before
   public void init() {
      a1 = new BankAccount();
      a2 = new BankAccount();
      a1.credit(100);
   }

   @Test
   public void transferTest1() {
      doAction();
      assertEquals(50, a1.getBalance(), 0);
   }

   @Test
   public void transferTest2() {
      doAction();
      assertEquals(50, a2.getBalance(), 0);
   }

   private void doAction() {
      a1.transfer(a2, 50);
   }
}
```

At first glance, it's not immediately obvious what these tests are about. Poor choices of names for the test fixture, test methods, fields and helper methods make it harder to see that we're testing a funds transfer between a payer bank account and a payee.

If we refactor this code, we can make the intent clearer. Let's start with the test method names.

```
@Test
public void transferDebitsAmountFromPayer() {
   doAction();
   assertEquals(50, a1.getBalance(), 0);
}

@Test
public void transferCreditsAmountToPayee() {
   doAction();
   assertEquals(50, a2.getBalance(), 0);
}
```

Test method names should clearly convey what the test is. Not how the test works, or what method or class is being test: *what is the test?*

Don't worry if you have to write a long, verbose test method name. We're not designing an API, and we'll probably never need to write code that calls our test methods. Think like a newspaper headline writer.

Now, how about those fields, *a1* and *a2*?

```
private BankAccount payer;
private BankAccount payee;
```

Try to name test objects and test data (fields, variables, constants) so they convey the *role* that object plays in the test. Ask "What does the customer/user call this?"

Now, how about that unhelpfully general helper method, *doAction()*?

```
@Test
public void transferCreditsAmountToPayee() {
   transferFunds(payer, payee, 50);
   assertEquals(50, payee.getBalance(), 0);
}

private void transferFunds(BankAccount payer,
                           BankAccount payee,
                           int amount) {
   payer.transfer(payee, amount);
}
```

Renaming it to *transferFunds()* makes it much clearer what it does.

I've also introduced parameters for *payer*, *payee* and *amount*, so we can better interpret what happens just by looking at the call to that method in the test.

The *init()* method sets up our accounts before each test method is run. We could make it a bit more obvious by renaming it.

```
@Before
public void setupAccounts() {
    payer = new BankAccount();
    payee = new BankAccount();
    payer.credit(100);
}
```

And finally, *Tests1* isn't a very illuminating name for a test fixture. When someone asks "Where are the tests for bank accounts?", it won't be of much help in finding them. Let's rename it to make it obvious what these are the tests for.

```
public class BankAccountTests {
```

As well as naming, our choice of test data can also help to make tests clearer.

```
@Test(expected=InsufficientFundsException.class)
public void cannotWithdrawMoreThanBalance() {
    BankAccount account = new BankAccount();
    account.credit(100);
    account.debit(100.01);
}
```

In this example, we could have chosen any amount to debit great than 100, but by choosing 100.01, we more clearly communicate where the boundary is. Debiting 100 will work just fine. Debiting a penny more will cause an exception to be thrown.

If we wanted to make it even more obvious, we could name the opening balance.

```
private static final int BALANCE = 100;

@Test(expected=InsufficientFundsException.class)
public void cannotWithdrawMoreThanBalance() {
   BankAccount account = new BankAccount();
   account.credit(BALANCE);
   account.debit(BALANCE + 0.01);
}
```

Naming literal values like this can sometimes help to clarify its significance in the test.

Lastly, don’t forget that – although we should seek to remove duplication from our test code - if it makes it easier to understand, leave it in. *Readability trumps reuse.*

EXERCISE #6

Revisit the code you write for exercises 1-5, and see if you can make the tests easier to understand by refactoring them.

If you can find someone to help, ask them to read your tests and comment on anything that isn’t totally clear.

A great way to practice choosing test method names when you’re pair programming is for one person to declare the test, and then let the other person write the test code *based only on the name*.

10. SPEAKING THE CUSTOMER'S LANGUAGE

Summary:

- The key to communicating on a software project is to establish a shared language
- Use the customer's language when choosing names in your code
- Requirements documents and acceptance tests are a good source of inspiration
- A tag cloud generator is a cheap way of building a visual glossary of customer terms

The names we choose for classes, methods, variables and other items can have a profound effect on the way we understand code.

```
public class PlaceRepositoryTests {

   @Test
   public void allocateFlagsPlaceForUser() {
      PlaceRepository placeRepository =
                                 new PlaceRepository();
      User user = new User();
      Place place =
             placeRepository.allocate("A", 1, user);
      assertEquals(user, place.flaggedFor());
   }

}
```

If I asked you what business domain this code comes from, could you tell by looking at the code?

How about if we change some of the names?

```
public class FlightSeatingTests {

   @Test
   public void seatIsReservedForPassenger() {
      FlightSeating seating = new FlightSeating();
      Passenger passenger = new Passenger();
      SeatReservation reservation
               = seating.reserve("A", 1, passenger);
      assertEquals(passenger,
                         reservation.getPassenger());
   }

}
```

The key to communication is ensuring every stakeholder's internal mental model is roughly the same. That means we all need to be *speaking the same language*.

If software design is all about solving the customer's problem, it stands to reason that the language we should all be speaking is the *customer's language*.

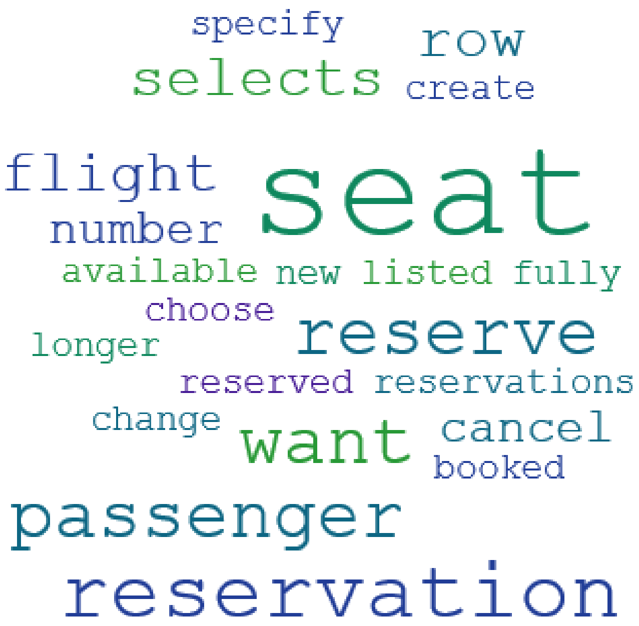
Here's their description of how reserving seats should work:

The passenger selects the flight they want to reserve a seat on. They choose the seat by row and seat number (e.g., row A, seat 1) and reserve it. We create a reservation for that passenger in that seat.

When you're searching for a name for a new class, a new method, or a new variable, look to the customer's description for inspiration. What do they call it?

Some teams take establishing a common language so seriously that they create and maintain glossaries of terms. A cheaper way of achieving something similar might be to run requirements documents – including acceptance tests - through a tag cloud generator.

Here's one I made from some user stories for an airline's seat reservation system.


specify
row
selects
create
flight
seat
number
available
new
listed
fully
choose
reserve
longer
reserved
reservations
change
want
cancel
booked
passenger
reservation

EXERCISE #7

Test-drive some code to automatically play a game based on the following problem. Run the description below through a tag cloud generator, and use it for inspiration when choosing names in your code.

```
Rock-Paper-Scissors is a game for two players.

Each player simultaneously reveals whether
they have randomly selected Rock, Paper, or
Scissors. The winner of each round is
determined as follows:

Rock blunts Scissors – Rock wins

Scissors cuts Paper – Scissors wins

Paper wraps Stone – Paper wins

If both players select the same, then that
round is a draw.

The game consists of three rounds, but if
there’s no clear winner after three, they
continue playing until one of them wins.
```

11. TRIANGULATING

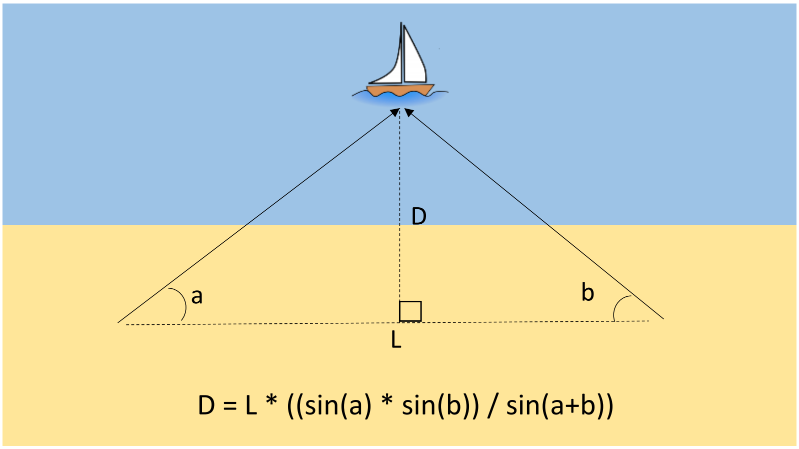
Summary:

- Triangulation allows us to discover the simplest design one test case at a time
- Like triangulating a position on a map, it works by choosing 2 or more data points and finding the simplest solution that satisfies them
- Taking baby steps brings more focus on each design decision and leads to better test assurance
- Starting with the simplest failing test we can think of, we gradually generalise the design *just enough* with each new test
- It requires at least 2 tests to generalise to a pattern or rule
- Use test names to document the patterns/rules as they emerge
- As we triangulate our design, we may notice patterns to the way code generalises that can help guide us
- Sometimes, the implementation to pass a test is obvious and trivial, and we don't need to triangulate

Creating designs that are as simple as possible, and that work reliably, requires us to apply more focus to every design decision.

In TDD, instead of leaping for a general solution, we *triangulate*.

Triangulation is the term we use for the process of pinpointing a solution using *multiple examples*. It comes from trigonometry, where we use triangles to determine the distance and location of a point (e.g., on a map).



We take multiple bearings to an object we wish to know the location of, and that object is where the lines meet – the location that exists on all those bearings.

Triangulating in TDD is similar. We pick a failing test case, and come up with the simplest solution *just to pass that test*. And then we pick another failing test, and generalise to the simplest solution that passes *both* tests. And we keep going until we can't think of any more failing tests, looking for the simplest solution that passes all of our tests.

We've already seen an example of triangulation, when we test-drove code to calculate Fibonacci numbers in the chapter *How to TDD?*

We could have started by writing a single test.

```
public class FibonacciTests {

   @Test
   public void fibonacciIsSumOfPreviousTwoNumbers() {
      assertEquals(21, new Fibonacci().getNumber(8));
   }

}
```

And then implemented a general algorithm to pass it.

```
public class Fibonacci {

   public int getNumber(int index) {
      if(index < 0) throw new IllegalArgumentException();

      if(index < 2)
         return index;
      return getNumber(index-1) + getNumber(index-2);
   }
}
```

But this is something of a leap. Already, we have things in our solution that no test requires (namely, the guard condition about negative indexes).

How did we know this is the right solution? How did we know this is the simplest solution? And how confident are we that if someone breaks this code later, our single test will catch it? How easy would it be to debug it?

Instead, what we did was take baby steps, starting with the simplest failing test we could think of (the one that would be easiest to pass).

```
public class FibonacciTests {

   @Test
   public void firstNumberIsZero() {
      assertEquals(0, new Fibonacci().getNumber(0));
   }
}
```

And then did the simplest thing possible to pass just that test.

```
public class Fibonacci {

   public int getNumber(int index) {
      return 0;
   }
}
```

Then we picked the next simplest failing test we could think of.

```
  @Test
  public void secondNumberIsOne() {
     assertEquals(1, new Fibonacci().getNumber(1));
  }
```

And then we generalised our solution *just enough* to pass both of these tests.

```
  public int getNumber(int index) {
     return index;
  }
```

What we're looking for is *patterns* (or rules). It's not possible to spot a pattern or generalise to a rule from just one example. With two or more examples, we can begin to generalise.

The simplest pattern that fits the first two tests is that the Fibonacci number is the same as its index.

Notice how we documented the pattern using a parameterized test that consolidated those two examples.

```
@Test
@Parameters({"0,0", "1,1"})
public void firstTwoNumbersAreSameAsIndex(int expected,
                                          int index) {
   assertEquals(expected,
                          new Fibonacci().getNumber(index));
}
```

The third Fibonacci number follows a different pattern to the first two, implying a branch in the logic.

```
  @Test
  public void thirdNumberIsOne() {
     assertEquals(1, new Fibonacci().getNumber(2));
  }
```

Many developers would, at this point, leap straight for:

```
public int getNumber(int index) {
   if(index < 2)
      return index;
   return getNumber(index-1) + getNumber(index-2);
}
```

But this would be premature. It’s impossible to infer a much simpler general solution from just from “third number is 1”.

Instead, let’s triangulate this new pattern, starting with the simplest possible solution to pass the third test.

```
public int getNumber(int index) {
   if(index < 2)
      return index;
   return 1;
}
```

Notice that, for indexes of 2 or higher, we’re returning a literal value. That’s all we need to do to pass this third test. It’s a new pattern, and we can’t generalise with just one example of it.

After a spot of refactoring to localise the knowledge of how to get Fibonacci numbers in the test code, it’s time to think about our next failing test. How about the fourth Fibonacci number?

```
@Test
public void fourthNumberIsTwo() {
   assertEquals(2, getFibonacciNumber(3));
}
```

Surely, at this point, it’s time to implement the general algorithm?

Actually, no. There’s a simpler generalisation.

```
public int getNumber(int index) {
   if(index < 2)
      return index;
   return index - 1;
}
```

And now it’s time to refactor our test code again to consolidate these two examples of this new rule.

```
@Test
@Parameters({"1,2", "2,3"})
public void thirdNumberOnIsIndexMinusOne(int expected,
                                          int index) {
   assertEquals(expected, getFibonacciNumber(index));
}
```

What's our next failing test? Well, the fifth Fibonacci number has an index of 4 and a value of 3, so our current code would actually pass that test. But the sixth has an index and value both of 5, so that would fail.

```
@Test
@Parameters({"1,2", "2,3", "5,5"})
public void thirdNumberOnIsIndexMinusOne(int expected,
                                          int index) {
   assertEquals(expected, getFibonacciNumber(index));
}
```

The simplest solution that will pass all these tests is, in fact:

```
public int getNumber(int index) {
   if(index < 2)
      return index;
   return getNumber(index-1) + getNumber(index-2);
}
```

We *discovered* this algorithm by taking baby steps and doing the simplest thing with each step, generalising with each new test.

It took us two tests to discover the rule about the first two Fibonacci number being the same as their index. It took us three tests to discover the rule about the third and above numbers being the sum of the previous two.

As a final step, we add an "edge case" test to require a guard condition for negative indexes.

```
   @Test(expected=IllegalArgumentException.class)
   public void indexMustBePositiveInteger() {
      getFibonacciNumber(-1);
   }
```

The resulting tests read like a specification for these three rules, and provide good test assurance that the rules have been correctly implemented. If we broke the code so that it breaks one of the rules, there's a very good chance at least one test will fail, giving us a vital early warning and making it easier to pinpoint exactly what's gone wrong.

Of course, we "know" the general solution, because we thought about it in advance. Thinking about designs in advance is a good thing. I highly recommend it!

But, even though it's a good idea to think ahead, it's not such a good idea to *code ahead*. A trivial example like the Fibonacci calculator tests our discipline in not leaping ahead for general solutions and speculating about what the best design will be.

With programming, the devil is in the detail. Triangulating brings more focus to getting those details right. Start simple, take baby steps, and generalise only when you see a pattern.

TRIANGULATION PATTERNS

Observant readers may have noticed that there are loose patterns to the way we generalise our solutions as we triangulate.

- To pass a single test, we might need to do nothing more than return the result the tests expects as a literal value.
- To pass two tests that expect two different results, we might generalise that literal value to a variable (or a parameter).
- When that value is accessed by more than one method (so our implementation has to remember it), a variable might become a field.
- When a variable can have multiple values at the same time, it can become a collection.
- When that collection is a sequence that follows a rule, it can become a loop – or a lambda expression - that generates the collection, applying the rule to every element.

As you get more experience with TDD, you'll develop an instinctive feel for these patterns of generalisation, learning to let the tests guide your designs.

OBVIOUS IMPLEMENTATIONS & TDD "GEARS"

Sometimes, though, triangulating is overkill. Imagine test-driving a simple function to add two numbers together, for example.

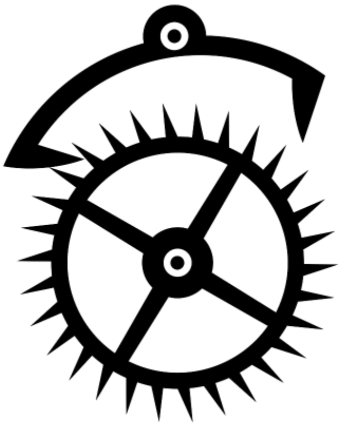
```
@Test
public void sumOfTwoPlusTwoIsFour() {
   assertEquals(4, Maths.sum(2,2), 0);
}
```

Would we go to the trouble of triangulating this, starting by just returning the literal result 4? Arguably, there'd be little value gained for something this straightforward, so instead we might just implement the simplest general solution.

```
public static double sum(double i, double j) {
   return i + j;
}
```

Beware, though; it takes considerable experience to be able to effectively judge when a design really is too trivial to take baby steps. We recommend erring on the side of caution, especially when you're relatively new to TDD. With time, you'll develop better judgement about how small your baby steps need to be.

Kent Beck, author of *Test-Driven Development By Example*, likens it to pulling a bucket of water up from a well using a ratchet and pulley.



The teeth on the ratchet gear lock it in position every time we raise the bucket by a certain amount. This means all our effort up to that point won't be wasted if we let go of the rope.

The heavier the bucket of water, the smaller we'll want the teeth to be, so we can pull it up in shorter bursts of energy.

But if we're raising only a teaspoonful of water, we could raise the bucket much faster with a ratchet gear that has larger teeth.

TDD is a bit like this. The tests lock our solution code in place, so we don't risk wasting all our effort by breaking the code we already wrote.

The more complex and "heavy" the problem we're trying to solve, the smaller the steps we might want to take. The simpler and more trivial it is, the bigger the steps we can comfortably take.

Your ability to "switch gears" when doing TDD will grow as you get more and more practice.

EXERCISE #8

Triangulate some code that sorts a set of playing cards into ascending order (Aces count as 1). Start with the simplest example you can think of (e.g., what happens if we sort a single card?), and discover a design, taking the smallest steps forward possible.

12. REFACTORING

Summary:

- Refactoring is improving the internal design of software without changing what it does
- Refactorings are small, atomic code re-writes that preserve behaviour
- Many refactorings can be automated
- Run the tests after every refactoring to check nothing's broken
- Refactorings are well-defined and have names like *Rename*, *Extract Method*, *Extract Class* and *Inline*
- Pay special attention to code duplication, as it can reveal useful abstractions
- In TDD, designs emerge through triangulation and refactoring
- Keep refactoring until you're happy leaving the code as it is

So far, we've seen several examples of something programmers call "refactoring".

Refactoring is *improving the internal design of our software without changing what it does*.

We refactor our code to:

- make it easier to understand
- make it simpler
- remove duplication
- localise the impact of making changes

More generally, we refactor the code to make it *easier to change*.

The danger in changing code is that we might break the software. Refactoring minimises this risk in 3 ways:

1. Refactorings are **small and atomic**

   The smaller the change, the less can go wrong. And if it does go wrong, we want to be able to easily undo it. Refactorings succeed or fail as a whole.

2. Refactorings **preserve behaviour**

   After each refactoring, we want the code to do exactly what it did before. We can check that it does using automated tests

3. Often, refactorings can be **automated**

   Automated refactorings, which are supported to some extent in most editors, help us by automatically updating the code so that it should still work, and also by offering a single-step *Undo* in case anything goes wrong

Think of your source code as a data structure made of "stuff" like classes, methods, parameters, variables, identifiers, statements, expressions and so on.

A refactoring rewrites this "stuff" to make it easier to change in one or more ways.

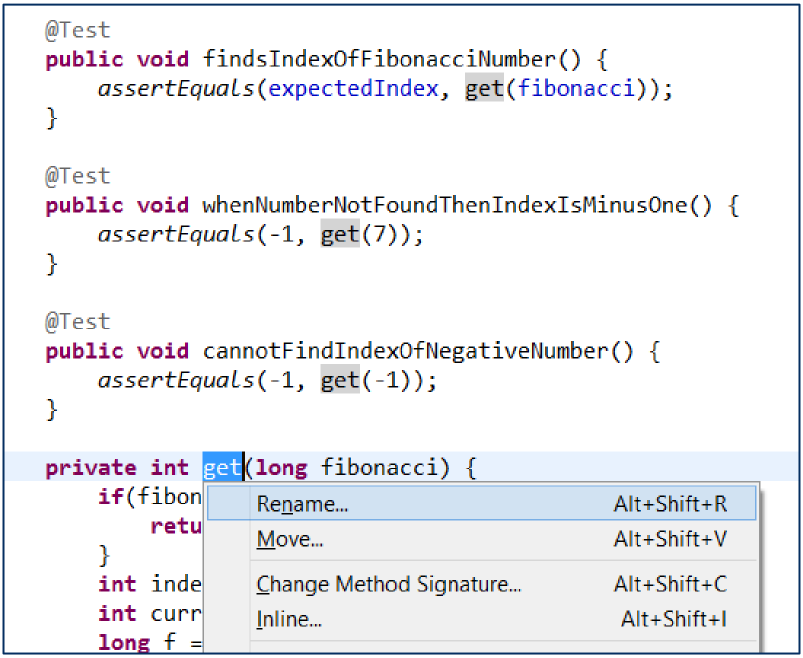
Very importantly, at the end of each refactoring, the *code still works*. We check this by running our tests.

It's important to become familiar with the most commonly used refactorings, and get practice at applying them to your code.

Let's look at some examples in Java using the popular Eclipse IDE (www.eclipse.org).

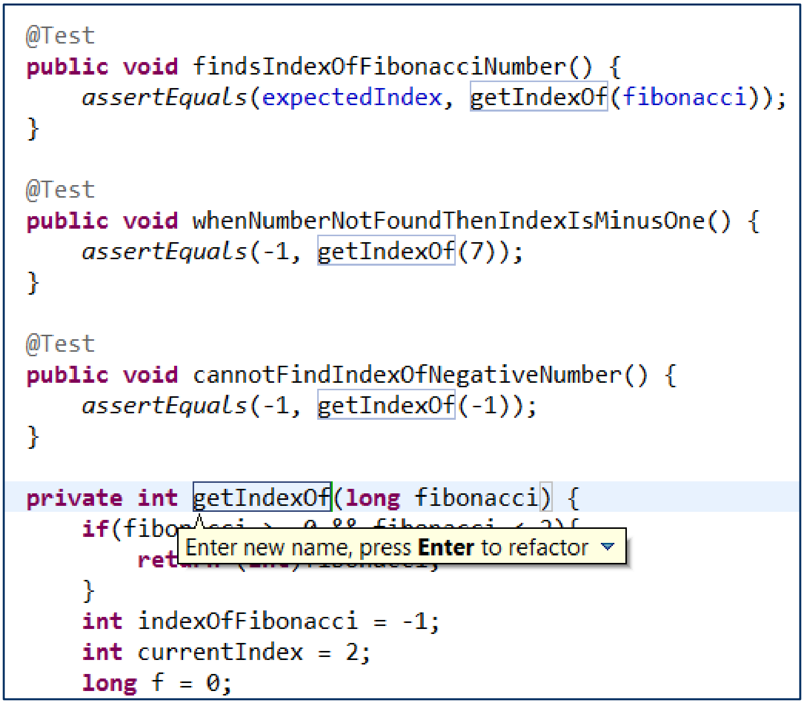
RENAME

To make its meaning clearer, we may wish to rename a class, a method, a variable and other things that have names. When we change the name of, say, a method, that change breaks all of the code that calls that method. So the Rename refactoring has to update all of the references so that the code still works.



```
@Test
public void findsIndexOfFibonacciNumber() {
    assertEquals(expectedIndex, get(fibonacci));
}

@Test
public void whenNumberNotFoundThenIndexIsMinusOne() {
    assertEquals(-1, get(7));
}

@Test
public void cannotFindIndexOfNegativeNumber() {
    assertEquals(-1, get(-1));
}

private int get(long fibonacci) {
    if(fibon
        retu
    }
    int inde
    int curr
    long f =
```

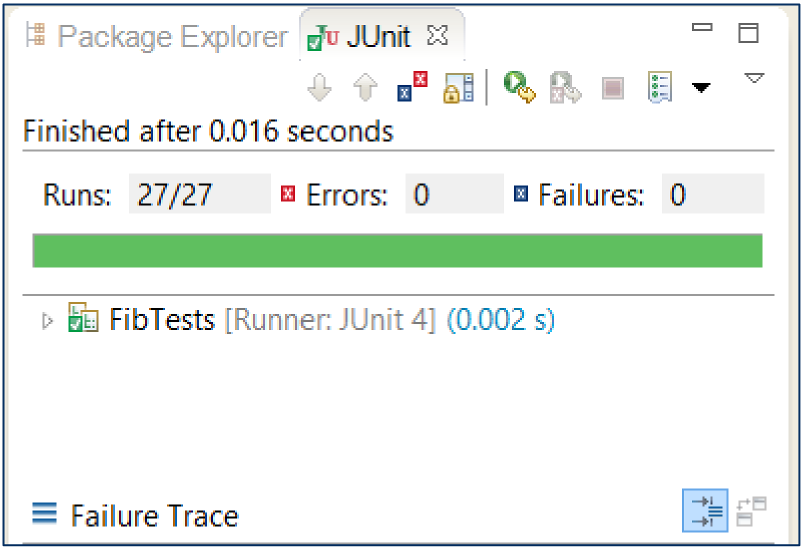
In my editor, I select the thing I want to rename (in this case, a method ambiguously called *get*). I launch the context-sensitive refactoring menu, and select the *Rename* refactoring.



```
@Test
public void findsIndexOfFibonacciNumber() {
    assertEquals(expectedIndex, getIndexOf(fibonacci));
}

@Test
public void whenNumberNotFoundThenIndexIsMinusOne() {
    assertEquals(-1, getIndexOf(7));
}

@Test
public void cannotFindIndexOfNegativeNumber() {
    assertEquals(-1, getIndexOf(-1));
}

private int getIndexOf(long fibonacci) {
    if(fibonacci > 0 && fibonacci < 2){
        return (int)fibonacci;
    }
    int indexOfFibonacci = -1;
    int currentIndex = 2;
    long f = 0;
```

I can edit the method name in place in my editor. Notice how, as I type the new name, calls to *get()* are automatically updated. After I hit *Enter,* the automated refactoring will save my source files.

As soon as the refactoring is done, I run my tests to make sure it hasn't broken the code.

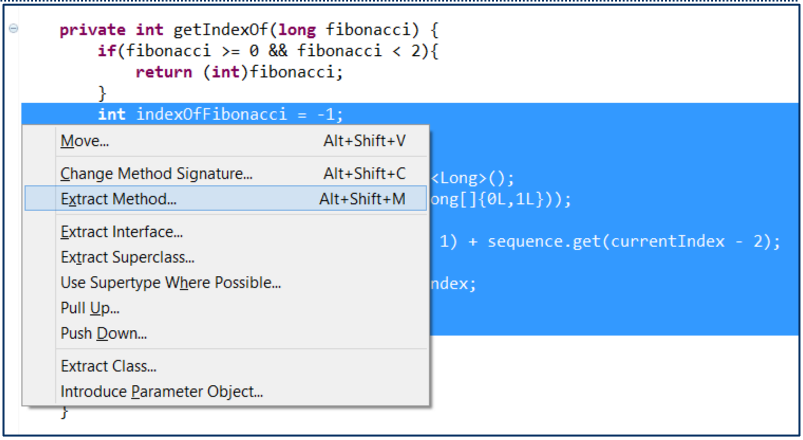


```
Finished after 0.016 seconds

Runs: 27/27    Errors: 0    Failures: 0

FibTests [Runner: JUnit 4] (0.002 s)

Failure Trace
```

The method name makes more sense now, but I can still see problems that will make this code harder to change.

Let’s do another refactoring.

EXTRACT METHOD

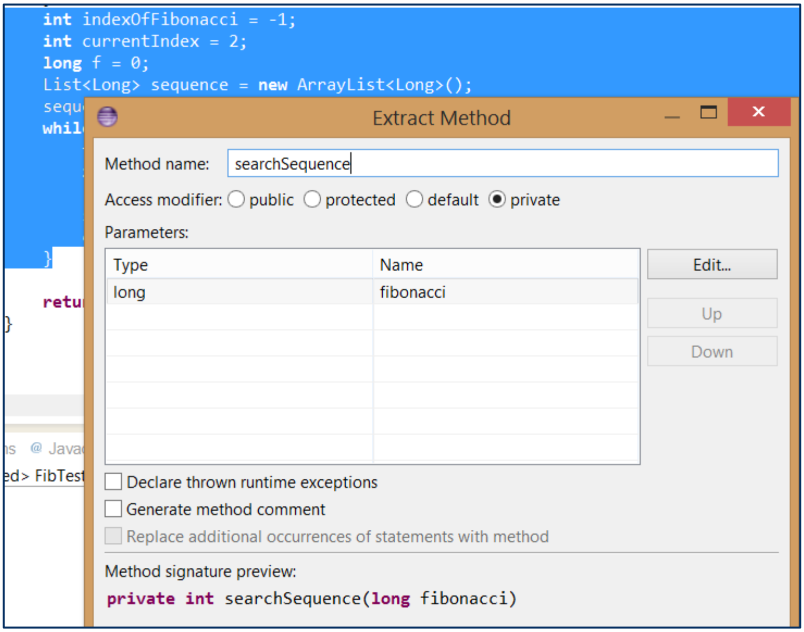


```
    private int getIndexOf(long fibonacci) {
        if(fibonacci >= 0 && fibonacci < 2){
            return (int)fibonacci;
        }
        int indexOfFibonacci = -1;
<Long>();
ong[]{0L,1L}));
 1) + sequence.get(currentIndex - 2);
ndex;
    }
```

Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Interface...
Extract Superclass...
Use Supertype Where Possible...
Pull Up...
Push Down...
Extract Class...
Introduce Parameter Object...

The *getIndexOf()* method does rather a lot, and is difficult to read. We can simplify things and make the code clearer by breaking the method down.

I select a block of code that does a specific chunk of the work and bring up the refactoring menu again.



```
    int indexOfFibonacci = -1;
    int currentIndex = 2;
    long f = 0;
    List<Long> sequence = new ArrayList<Long>();
    sequ
    whil

    }

    retu
}
```

Method name: searchSequence

Access modifier: public protected default [✓ private]

Parameters:

Type	Name
long	fibonacci

Declare thrown runtime exceptions ☐

Generate method comment ☐

Replace additional occurrences of statements with method ☐

Method signature preview:

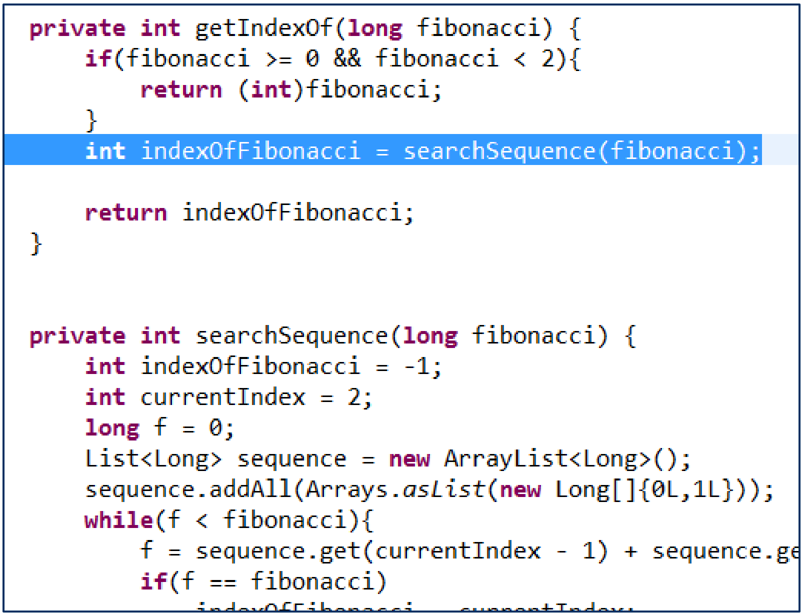
```
private int searchSequence(long fibonacci)
```

A dialogue pops up for the *Extract Method* refactoring, prompting me to give this new method a name. This is an opportunity to convey what this block of code does, using the method name.

Notice how it automatically adds a parameter for a variable *fibonacci* that's declared before this block of code. It has to pass this value in, or the code won't work.

It knows to return any data value that is referenced after this block of code, too.

Let's complete this refactoring.



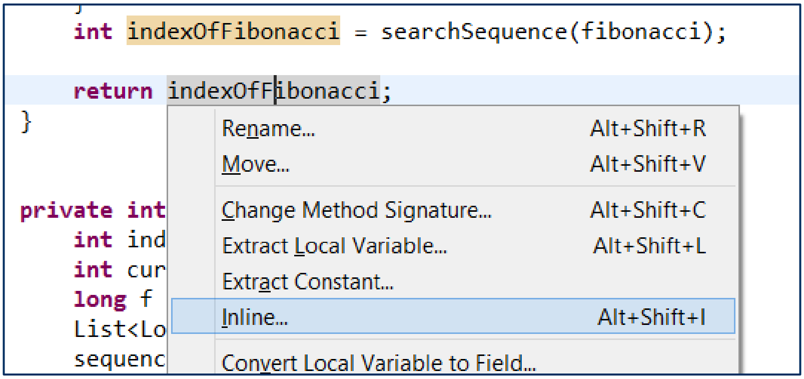
```
private int getIndexOf(long fibonacci) {
    if(fibonacci >= 0 && fibonacci < 2){
        return (int)fibonacci;
    }
    int indexOfFibonacci = searchSequence(fibonacci);

    return indexOfFibonacci;
}


private int searchSequence(long fibonacci) {
    int indexOfFibonacci = -1;
    int currentIndex = 2;
    long f = 0;
    List<Long> sequence = new ArrayList<Long>();
    sequence.addAll(Arrays.asList(new Long[]{0L,1L}));
    while(f < fibonacci){
        f = sequence.get(currentIndex - 1) + sequence.ge
        if(f == fibonacci)
            indexOfFibonacci = currentIndex;
```

Immediately, we run the tests to make sure nothing's broken.

There are still issues that might need addressing in our code. First of all, some low-hanging fruit.

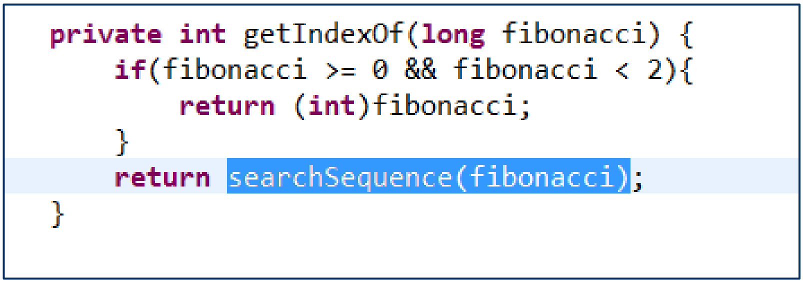
INLINE

Inlining replaces a reference to a thing with the thing itself. For example, we could inline the local variable *indexOfFibonacci*, because we don't really need it anymore.



```
        int indexOfFibonacci = searchSequence(fibonacci);

        return indexOfFibonacci;
    }


    private int
        int ind
        int cur
        long f
        List<Lo
        sequenc
```

Rename... Alt+Shift+R
Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Local Variable... Alt+Shift+L
Extract Constant...
Inline... Alt+Shift+I
Convert Local Variable to Field...



```
    private int getIndexOf(long fibonacci) {
        if(fibonacci >= 0 && fibonacci < 2){
            return (int)fibonacci;
        }
        return searchSequence(fibonacci);
    }
```

Again, we run the tests immediately to check everything still works. This is a habit you must get into to refactor safely.

There are still more issues to address. Does this code really belong in a test fixture at all? Probably not. Let's put it in its own place, so it can be more easily found and reused.

EXTRACT CLASS

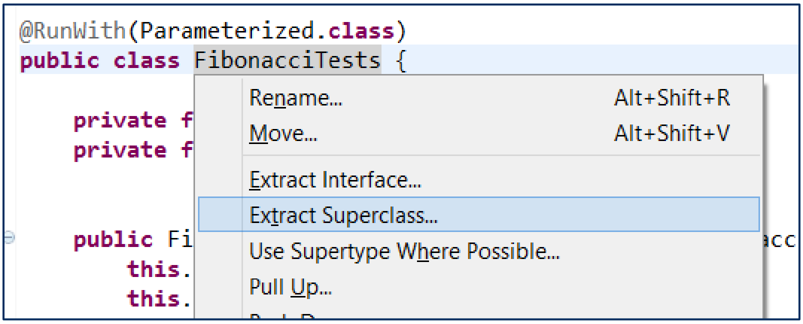
Extract Class moves selected features of an existing class into their own new class, and replaces them in the old code with an instance of the new class.

My editor's refactoring menu doesn't have a proper automated Extract Class, so we're going to go a bit around the houses here to make it happen. Many refactorings require us to perform a sequence of smaller refactorings.

Our goal is to – as much as possible – keeping the code working. So we're going to do this in a number of small steps, and run the tests after each step.

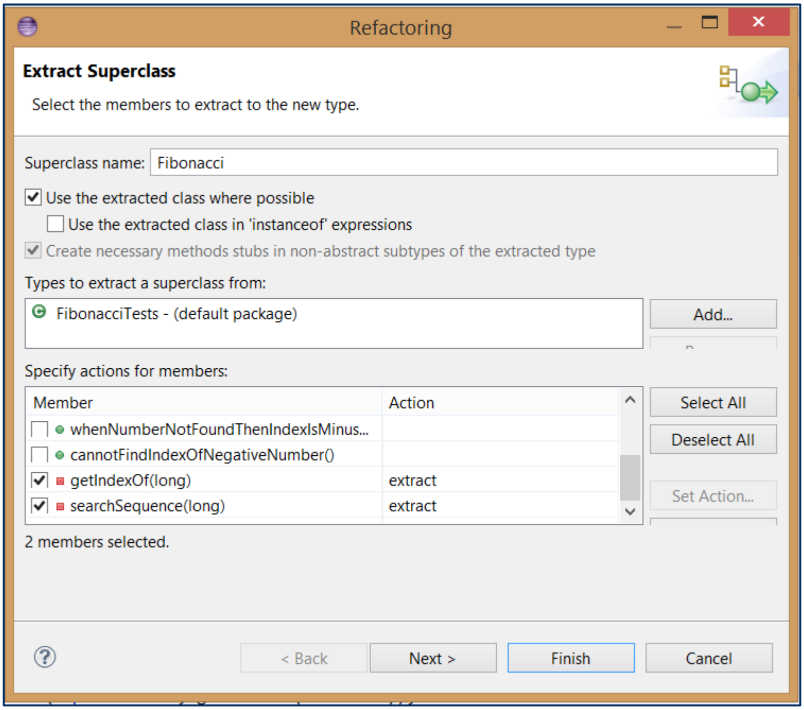
First, let's use the Extract Superclass refactoring to move *getIndexOf()* and *searchSequence()* in a new class, from which the test fixture will inherit so that it all still works.

This new superclass will just be a stepping stone. Ultimately we won't want it to be a superclass of the test fixture.



```
@RunWith(Parameterized.class)
public class FibonacciTests {

    private f
    private f


    public Fi                                                    acc
        this.
        this.
```

A dialog pops up prompting us to give this new superclass a name, and to select the features we want to move into it.



Extract Superclass

Select the members to extract to the new type.

Superclass name: Fibonacci

[✓] Use the extracted class where possible

[ ] Use the extracted class in 'instanceof' expressions

[✓] Create necessary methods stubs in non-abstract subtypes of the extracted type

Types to extract a superclass from:

FibonacciTests - (default package)

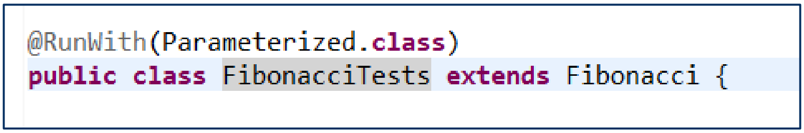
Specify actions for members:

	Member	Action
[ ]	whenNumberNotFoundThenIndexIsMinus...	
[ ]	cannotFindIndexOfNegativeNumber()	
[✓]	getIndexOf(long)	extract
[✓]	searchSequence(long)	extract

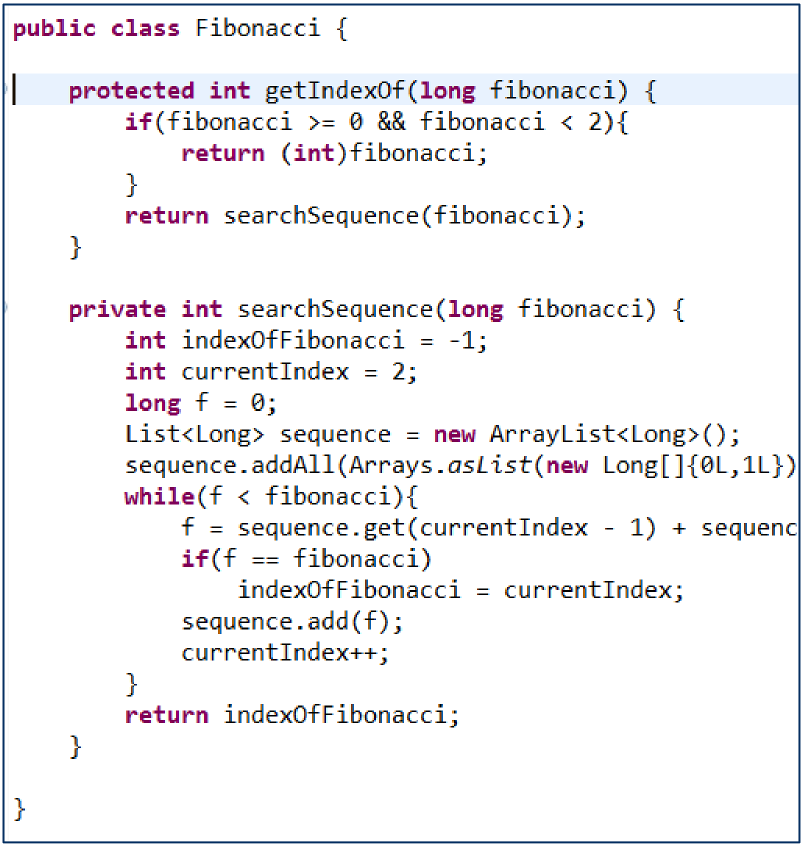
2 members selected.

In this instance, that's all we need to specify – though the *Extract Superclass* dialog has a lot more options – so we just click *Finish*.

It warns us that the visibility of *getIndexOf()* need to be changed for the subclass to continue using it. This is fine. It's just to make sure the code still works.



```
@RunWith(Parameterized.class)
public class FibonacciTests extends Fibonacci {
```

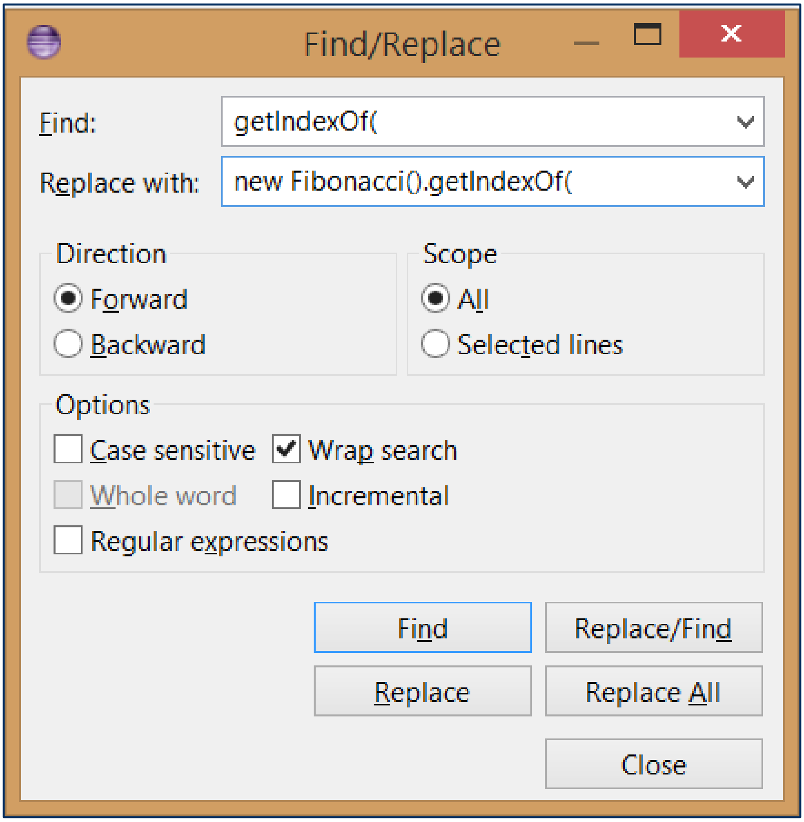


```
public class Fibonacci {

    protected int getIndexOf(long fibonacci) {
        if(fibonacci >= 0 && fibonacci < 2){
            return (int)fibonacci;
        }
        return searchSequence(fibonacci);
    }

    private int searchSequence(long fibonacci) {
        int indexOfFibonacci = -1;
        int currentIndex = 2;
        long f = 0;
        List<Long> sequence = new ArrayList<Long>();
        sequence.addAll(Arrays.asList(new Long[]{0L,1L})
        while(f < fibonacci){
            f = sequence.get(currentIndex - 1) + sequenc
            if(f == fibonacci)
                indexOfFibonacci = currentIndex;
            sequence.add(f);
            currentIndex++;
        }
        return indexOfFibonacci;
    }

}
```

Again, we run the tests at this point.

Now that we have a *Fibonacci* class, we want to change the tests so they invoke methods on an instance of that class, and not on the superclass.

We can achieve this using *Find/Replace*.


Find/Replace

Find: getIndexOf(

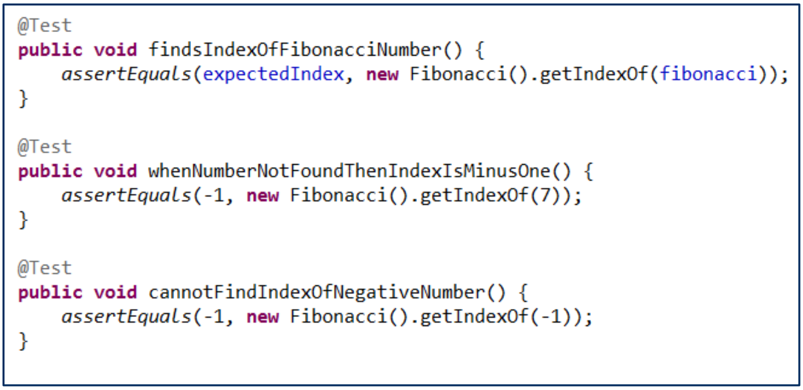
Replace with: new Fibonacci().getIndexOf(

Direction: [● Forward] ○ Backward

Scope: [● All] ○ Selected lines

Options:
- ☐ Case sensitive
- [✓ Wrap search]
- ☐ Whole word
- ☐ Incremental
- ☐ Regular expressions

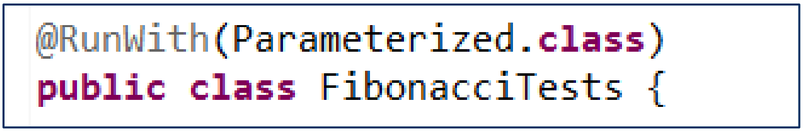
We replace all the calls to *getIndexOf()* on the superclass with calls to the same method on a new *Fibonacci* object.



```
@Test
public void findsIndexOfFibonacciNumber() {
    assertEquals(expectedIndex, new Fibonacci().getIndexOf(fibonacci));
}

@Test
public void whenNumberNotFoundThenIndexIsMinusOne() {
    assertEquals(-1, new Fibonacci().getIndexOf(7));
}

@Test
public void cannotFindIndexOfNegativeNumber() {
    assertEquals(-1, new Fibonacci().getIndexOf(-1));
}
```

And run the tests again.

Finally, there's no need any longer for *FibonacciTests* to extend *Fibonacci*, so we can remove that stepping stone.

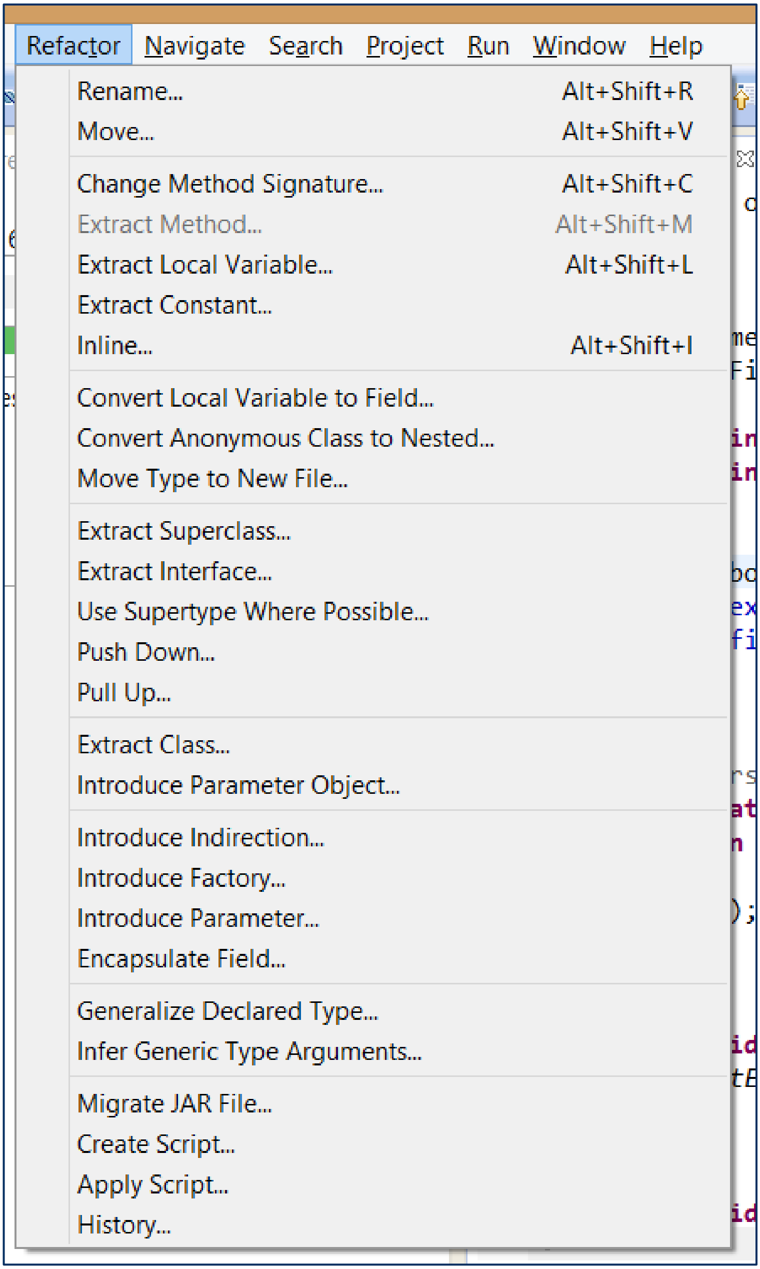


```
@RunWith(Parameterized.class)
public class FibonacciTests {
```

And then… yep, you guessed it… RUN THE TESTS.

THE REFACTORING MENU

The Eclipse editor offers a useful range of automated refactorings.


Refactor
Navigate
Search
Project
Run
Window
Help
Rename... Alt+Shift+R
Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Inline... Alt+Shift+I
Convert Local Variable to Field...
Convert Anonymous Class to Nested...
Move Type to New File...
Extract Superclass...
Extract Interface...
Use Supertype Where Possible...
Push Down...
Pull Up...
Extract Class...
Introduce Parameter Object...
Introduce Indirection...
Introduce Factory...
Introduce Parameter...
Encapsulate Field...
Generalize Declared Type...
Infer Generic Type Arguments...
Migrate JAR File...
Create Script...
Apply Script...
History...

Support for automated refactorings varies from editor to editor and language to language. It's typically better in languages that have compile-time type checking than in dynamic languages, because - in some refactorings - the tool needs to know what types of objects are involved.

In scripting languages like JavaScript and Ruby, programmers may have to learn how to do some refactorings by hand. It's important to be especially disciplined in these cases.

DUPLICATION & EMERGENT DESIGN

Although it's not as important as readability and simplicity, the duplication in our code – including our test code – offers useful clues about what might be a good design for our solution. This is because *the opposite of duplication is reuse*.

When we see two blocks of code that are almost the same, we can extract a parameterised method that performs the common logic. When we see two classes that are very similar, we can extract a common base class. Or if they do similar things, but in different ways, we can extract a common interface.

Duplication is often a good thread to pull on, as it can reveal abstractions that will make our designs better.

For this reason, many people recommend we *refactor to remove duplication* as the third step in the TDD cycle.

More generally, a design is revealed to us as we refactor. A method may be too long or doing too many things, so we break it up into multiple methods. A class may be getting too big or have too many responsibilities, so we split it up into new classes.

Starting from the single simplest solution, a complex design can emerge through the process of triangulation and refactoring. The aim is to discover the design that will pass the tests *and* be easy to change.

WHEN ARE WE DONE?

In our Fibonacci example, we still have issues we might want to address left in our code. The *getIndexOf()* method is pretty long, and does a lot. We could break it down by extracting the different pieces of work into their own private helper methods. Also, our test fixture mixes a single parameterised test with several ordinary tests for edge cases. The edge case tests are run unnecessarily for every parameterised test case, leaving potential confusion about how many tests there really are.

When it comes to the quality of our code, we often have the best of intentions to go back and code issues that might get in our way later.

Inspection of hundreds of code bases, however, teaches us that – nine times out of ten – we never actually get around to it fixing problems we leave behind.

For that reason, I strongly recommend that you refactor until you're happy leaving the code as it is – because you very probably will leave it like that *forever*.

That makes the third step in the TDD cycle extremely important. It reminds us to clean up our code to make it as readable, as simple, as free of duplication and as modular as we can before moving on to the next failing test.

EXERCISE #9

Look through the code you wrote for earlier exercises in this book for anything that you're not 100% happy with – names you think could be made clearer, methods that do more than one thing, nested IF statements, and so on.

Refactor the code until your confident that it will be easy to understand and easy to change.

Explore the refactoring menu in your editor and try each automated refactoring works on a copy of your code.

And DON'T FORGET TO KEEP RUNNING THE TESTS!

13. DESIGN PRINCIPLES

Summary:

- A Simple Design (in order of priority):
  - Works (i.e., passes all the tests)
  - Is easy to understand
  - Has minimal duplication
  - Is as simple as possible
- Design classes that Tell, Don't Ask, sharing as little internal detail as possible
- Give methods and classes a single responsibility, so they offer more possibilities for combinations and reuse
- Compose objects from the outside, using dependency injection, to offer greater flexibility for design and testing
- Expose client-specific interfaces to hide methods that client code doesn't need to use
- Use contract tests to ensure different implementations of the same abstraction fulfil the contract of their super-type

In previous chapters, we've touched on some goals for the design of our code that will make it easier to change, so we can keep adding new tests and new features, and sustain the pace of development for longer.

We're going to dwell on the principles of good design, as they're important enough to warrant a chapter all of their own.

SIMPLE DESIGN

Simple Design, also popularised by Kent Beck, is a set of design principles that developers can apply to most any kind of software.

Rather than having to learn a whole encyclopaedia of design rules and design patterns, Simple Design sets just four goals, in order of importance.

1. The code works
2. The code is easy to understand
3. The code has minimal duplication
4. The code is as simple as possible

THE CODE WORKS

Most important of all is that the code works. We check that it does by running our tests. If it doesn't pass the tests, fixing that is priority number one.

THE CODE IS EASY TO UNDERSTAND

When we're happy the code works, we next concern ourselves with how easy it is to understand. It's estimated developers spend between 50-80% of our time just reading code. Time invested in making the code clearer is almost always profitable later.

THE CODE HAS MINIMAL DUPLICATION

If we're satisfied that the code works, we turn our attention to duplication. The mantra to remember here is *Don't Repeat Yourself* (D.R.Y.). When we have to change duplicated code, we have to make that change multiple times.

One exception to D.R.Y. is when a bit of duplication makes the code easier to understand. In our test fixtures, for example, I left in some duplication – separate test methods for cases that could have been incorporated into an existing parameterised test – to make it easier to see this was a different rule being tested, and not just a different example of the same rule.

THE CODE IS AS SIMPLE AS POSSIBLE

Simpler designs are quicker to get working, easier to understand, and less likely to go wrong. For all these reasons, TDD recommends we do the simplest thing possible that will pass our tests.

Again, the exception is when simplicity conflicts with our higher-priority design goals. Sometimes the simplest solution isn't necessarily the easiest to understand, for example. On occasion, it may be better to solve a problem a longer way, if that longer way can be understood faster.

TELL, DON'T ASK

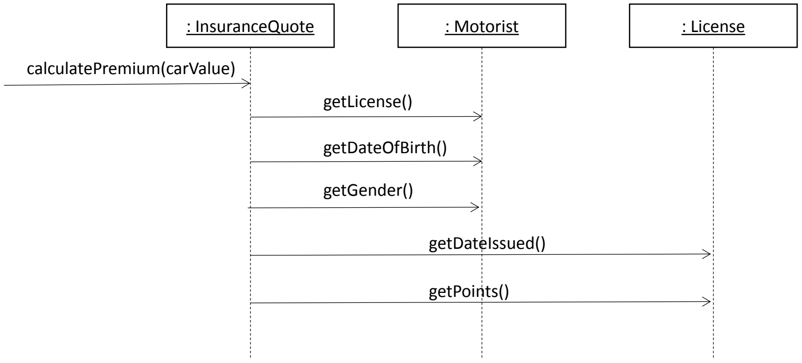
The four principles of Simple Design take us a long way towards a good design, when they're applied rigorously to the code as it grows.

But Simple Design doesn't directly address one potential obstacle to changing our code that's actually pretty important: *dependencies*.

Consider a class that calculates insurance premiums for motorists. To decide what premiums to apply it needs to know the age of the motorist, their gender (men tend to have more accidents), how long they've been driving legally, and how many points they have on their driver's license.

```
public class InsuranceCalculator {

  private Motorist motorist;

  public InsuranceCalculator(Motorist motorist) {
    this.motorist = motorist;
  }

  public double calculatePremium(double carValue) {
    License license = motorist.getLicense();

    double premiumPercent = 0;

    premiumPercent +=
      calculateAgePremium(
          calculateAge(motorist.getDateOfBirth()));
    premiumPercent +=
          calculateGenderPremium(motorist.getGender());
    int yearsOfExperience =
          calculateExperience(license.getDateIssued());
    premiumPercent +=
          calculateExperiencePremium(yearsOfExperience);
    premiumPercent +=
          calculatePointsPremium(license.getPoints());

    return carValue * premiumPercent;
  }
```

To get these pieces of information, it has to ask *Motorist* and *License* for them. Let's visualise the interactions between our objects using a UML sequence diagram:



Because *InsuranceQuote* is doing all the work, but *Motorist* and *License* have all the data, this design creates a lot of low-level *coupling* between our objects.

The more objects know about each other, the more likely it is that a change to one object will affect others. Changing *License* might break *InsuranceQuote*, which might in turn break any code that depends on *InsuranceQuote*.

Another goal of good design is to *localise the impact of change*. We can achieve this by, as much as possible, internalising dependencies within classes, which reduces the coupling between them.

Code that needs to know a motorist's date of birth should be packaged where that data is. Code that needs to know how many points there are on a motorist's license should be packaged where that points data is.

More generally, put the work where the data is.

Let's refactor our code to reduce the coupling between the classes, by putting our calculations in the same classes as the data they use.

```
public class InsuranceCalculator {

   private Motorist motorist;

   public InsuranceCalculator(Motorist motorist) {
      this.motorist = motorist;
   }

   public double calculatePremium(double carValue)  {
      return motorist.calculatePremium(carValue);
   }

}
```

Instead of asking for the data, *InsuranceQuote* now delegates the work to *Motorist*.

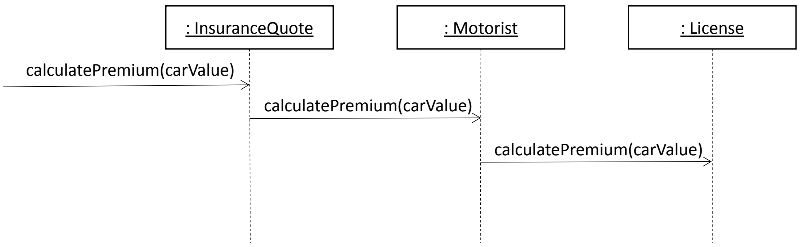
```
public class Motorist {

   private final String dateOfBirth;
   private final Gender gender;
   private final License license;

   public Motorist(String dateOfBirth,
                   Gender gender,
                   License license) {
      this.dateOfBirth = dateOfBirth;
      this.gender = gender;
      this.license = license;
   }

   public double calculatePremium(double carValue)  {
      return calculateMotoristPremium(carValue) +
                   license.calculatePremium(carValue);
   }

   private double calculateMotoristPremium(double carValue)  {
      double premiumPercent = calculateAgePremium()
                        + calculateGenderPremium ();
      return premiumPercent * carValue;
   }
```

Motorist does the work relating to what it knows: *dateOfBirth* and *gender*. It delegates the rest of the work to *License*, because that class has the rest of the data.

```
public class License {

   private int points;
   private final String dateIssued;

   public License(String dateIssued){
      this.dateIssued = dateIssued;
   }

   double calculatePremium(double carValue) {
      return calculateExperiencePremium(carValue) +
                        calculatePointsPremium(carValue);
   }
```

Instead of asking *Motorist* and *License* for their data, *InsuranceQuote* tells them to do the work themselves. That's why this style of design is sometimes referred to as "Tell, Don't Ask".

When we visualise the interactions between the different objects after this refactoring, it looks like this:


: InsuranceQuote
: Motorist
: License
calculatePremium(carValue)
calculatePremium(carValue)
calculatePremium(carValue)

Just at a glance, we can see there are far fewer object couplings. Note that, because we're sharing less data, there's no need for all those *getter* methods any more.

This design principle goes by several names, including *data hiding* and *encapsulation*. All you need to remember is that the less objects know about each other, the better.

SINGLE RESPONSIBILITY

Consider a method that credits a bank account:

```
public void credit(double amount){
   this.balance += amount;
   SimpleDateFormat sdfDate =
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
   Date now = new Date();
   String dateTime = sdfDate.format(now);
   String creditXml = "<credit>"+
          " <account>"+ accountNumber + "</account>" +
          " <amount>" + amount + "</amount>" +
          " <datetime>" + dateTime + "</dateTime>" +
          "</credit>";
   AccountLogger.log(creditXml);
   rewardPoints += Math.floor(amount/100);
}
```

This method does a whole bunch of stuff.

- Adds the amount to the balance
- Formats the current date & time
- Creates an XML string that represents this transaction for logging
- Calculates and adds reward points at 1% of the credit amount

Not only does it make this method harder to understand, but what happens if we want to format the current date and time for some other purpose? What happens if we want to calculate reward points for other kinds of transactions?

As it stands, *credit()* is an all-or-nothing affair. We don't get the option to reuse any of its logic by itself, and this can present a barrier to change.

To better explain, here's a thought experiment: how many different ways can we combine the string "ABCD"? *Only one*. It's always "ABCD".

How many different ways can we combine "AB" and "CD"? *Four*. We can make "AB", "CD", "ABCD" and "CDAB".

How many different ways can we combine "A", "B", "C" and "D"? *Sixty four*.

By breaking “ABCD” into “A”, “B”, “C” and “D” we give ourselves sixty four times as many possible combinations of letters.

Likewise, by breaking *credit()* down into four separate methods, each with one distinct job, we create many more opportunities to create new logic by combining one or all of those methods.

```
public void credit(double amount){
   updateBalance(amount);
   String dateTime = formatCurrentDateTime();
   AccountLogger.log(serialize(amount, dateTime));
   rewardPoints += calculateRewardPoints(amount);
}
```

credit() is now what we call a *composed method*; that is, a method composed of calls to other methods. The method names tell the story of what work is being done, but the actual work is delegated to these new methods.

This makes *credit()* easier to understand, and it also means that we can write new code reusing methods like *formatCurrentDateTime()*, *serialize()* and *calculateRewardPoints().*

We could also extend our account class, and override those individual methods without having to change the code in *credit()*. This refactored design opens up many new possibilities.

The same principle applies at the class level; should we have to use an account every time we want to format the current date and time? That smacks a little of buying a Mercedes just to use the cigarette lighter.

And if we wanted to change the format of the current date and time, should we have to edit – and risk breaking – the account class? There will be other classes depending on it. It might break them, too. Better, surely, for that formatting code to go in its own class, where we can change it by itself.

```
public void credit(double amount){
   updateBalance(amount);
   String dateTime =
      new DateTimeFormatter().formatCurrentDateTime();
   AccountLogger.log(serialize(amount, dateTime));
   rewardPoints += calculateRewardPoints(amount);
}
```

While we're about it, should the account class be responsible for creating the XML string? Again, it's foreseeable wanting to change the XML format independently of how the account works. So, that, too, belongs in its own class.

```
public void credit(double amount){
   updateBalance(amount);
   String dateTime =
      new DateTimeFormatter().formatCurrentDateTime();
   AccountLogger.log(
      new XmlSerializer().serialize(this,
                                     amount,
                                     dateTime));
   rewardPoints += calculateRewardPoints(amount);
}
```

I can also see us needing to change how reward points are calculated independently of how a bank account works. Let's extract a class for that, too.

```
public void credit(double amount){
   updateBalance(amount);
   String dateTime =
      new DateTimeFormatter().formatCurrentDateTime();
   AccountLogger.log(
      new XmlSerializer().serialize(this,
                                     amount,
                                     dateTime));
   rewardPoints +=
      new RewardPointsCalculator().calculate(amount);
}
```

Extracting these separate responsibilities into their own classes gives us more options for reusing and extending our code. For example, if we wanted to, we could package *DateTimeFormatter* in its own library and reuse it on other projects.

Splitting *credit()* into separate methods, and then moving some of those methods into new classes – each with a distinct job – has bought us considerably more flexibility to keep evolving our design.

But we need to go further to buy us the kind of flexibility we're going to need later, as we'll discover in upcoming chapters.

What happens when we want to use different date-time formats for different kinds of output? What happens when we want to represent our credit transaction in different report formats, like CSV or HTML? What happens when we want to calculate reward points differently in different countries?

There's no easy way to get *BankAccount* to use a different implementation of *DateTimeFormatter*, *XmlSerializer* or *RewardPointsCalculator*.

Imagine we have two different implementations of a *DateTimeFormatter* interface, one for US date formats and one for the UK.

```
public interface DateTimeFormatter {

  public abstract String formatCurrentDateTime();

}
```

Similarly, imagine we have a US reward points calculator and a UK calculator that both implement a *RewardPointsCalculator* interface.

```
public interface RewardPointsCalculator {

  public abstract double calculate(double amount);

}
```

Finally, imagine we have two ways of representing a credit transaction: as XML and as HTML, both of which implement a *Serializer* interface.

```
public interface Serializer {

  public abstract String serialize(
                              BankAccount acccount,
                              double amount,
                              String dateTime);

}
```

How about, instead of instantiating these objects inside *BankAccount*, we pass them into the constructor?

```
public class BankAccount {

  private double balance;
  private final String accountNumber;
  private int rewardPoints;
  private final DateTimeFormatter dateTimeFormatter;
  private final Serializer serializer;
  private final RewardPointsCalculator rewardPointsCalculator;

  public BankAccount(String accountNumber,
             DateTimeFormatter dateTimeFormatter,
             Serializer serializer,
             RewardPointsCalculator rewardPointsCalculator){
    this.accountNumber = accountNumber;
    this.dateTimeFormatter = dateTimeFormatter;
    this.serializer = serializer;
    this.rewardPointsCalculator = rewardPointsCalculator;
  }

  public void credit(double amount){
    updateBalance(amount);
    String dateTime =
        dateTimeFormatter.formatCurrentDateTime();
    AccountLogger.log(
        serializer.serialize(this, amount, dateTime));
    rewardPoints += rewardPointsCalculator.calculate(amount);
  }
```

BankAccount is now composed from the outside by whichever code calls the constructor. If we *abstract* the classes it collaborates with, binding *BankAccount* to our pure interfaces, it becomes possible to vary *BankAccount*'s composition dynamically by plugging in different implementations.

```
BankAccount accountUS = new BankAccount("12345678",
                        new USDateTimeFormatter(),
                        new HtmlSerializer(),
                        new USRewardPointsCalculator());

BankAccount accountUK = new BankAccount("23456789",
                        new UKDateTimeFormatter(),
                        new XmlSerializer(),
                        new UKRewardPointsCalculator());
```

When we compose objects from the outside, by passing their collaborators in to the constructor or as method parameters, we call that *dependency injection*.

We now have the ability to swap collaborators easily, and this gives us even greater flexibility for future changes.

As we'll see in the chapter on *Test Doubles*, it also comes in very useful for writing fast-running automated tests by allowing us to test our code against pretend versions of things like database connections and web service calls.

FAKE IT 'TIL YOU MAKE IT

It also allows us to defer thinking about the design of other parts of our software while we focus on the logic of the part we're working on. E.g., Perhaps we don't want to think about how reward points are calculated. We can inject a placeholder for a calculator and carry on testing credit()

CLIENT-SPECIFIC INTERFACES

The less objects in our software know about each other, the better. As well as hiding internal features by applying Tell, Don't Ask, we also need to hide external features that our classes don't need to use.

To illustrate, look at this code from a community video library.

```
public class Library {

   private final List<VideoTitle> titles;

   public Library(){
      titles = new ArrayList<>();
   }

   public boolean hasTitle(String name){
      for (VideoTitle title : titles) {
         if(title.getName().equals(name)){
            return true;
         }
      }
      return false;
   }

   public void add(VideoTitle title){
      titles.add(title);
   }
}

public class VideoStats {

   private final VideoTitle title;

   public VideoStats(VideoTitle title){
      this.title = title;
   }

   public double averageRating(){
      List<Rating> ratings = title.getRatings();
      double totalRating = 0;
      for (Rating rating : ratings) {
         totalRating += rating.getValue();
      }
      return totalRating/ratings.size();
   }
}
```

Both *Library* and *VideoStats* use *VideoTitle*, but they use different methods of it. *Library* just needs to know the name of the title, while *VideoStats* just needs to access its ratings.

If we decide to change the details of either of these methods of *VideoTitle*, then both clients will be affected.

We can hide methods that clients don't need to see by splitting up the interface, creating client-specific interfaces for *Library* and *VideoStats* that only expose the methods they need.

```
public class VideoTitle implements Named, Rated {

   private final String name;
   private final List<Rating> ratings;

   public VideoTitle(String name){
      this.name = name;
      this.ratings = new ArrayList<>();
   }

   @Override
   public String getName() {
      return name;
   }

   @Override
   public List<Rating> getRatings() {
      return ratings;
   }

   public void rate(int value){
      ratings.add(new Rating(value));
   }
}
```

Note that the names of these new interfaces reflect the *role* the objects play with respect to each client. These are not the names of "things", like *Library* and *VideoTitle*.

Now we can refactor *Library* and *VideoStats* so they depend only on the interfaces they require.

```
public class Library {

  private final List<Named> titles;

  public Library(){
    titles = new ArrayList<>();
  }

  public boolean hasTitle(String name){
    for (Named title : titles) {
      if(title.getName().equals(name)){
        return true;
      }
    }
    return false;
  }

  public void add(Named title){
    titles.add(title);
  }
}

public class VideoStats {

  private final Rated title;

  public VideoStats(Rated title){
    this.rated = rated;
  }

  public double averageRating(){
    List<Rating> ratings = title.getRatings();
    double totalRating = 0;
    for (Rating rating : ratings) {
      totalRating += rating.getValue();
    }
    return totalRating/ratings.size();
  }
}
```

Notice that we didn't include the method *rate()* on the *Rated* interface. Although you might think it makes sense to include it, based on the name, in fact there's no reason for *VideoStats* to be exposed to it. Some other client uses that method, and if *rate()* is the only method it uses, we could again create a client-specific interface called, say, *Rateable*.

When our objects implement abstractions, like pure interfaces, or extend existing classes and override their methods, there's one thing we need to be mindful of – that they fulfil the original contracts of their super-types.

For example, there are many different ways we could sort an array of numbers, ranging from the brute force method of looping through the array until we find what we're looking for, to faster sorting algorithms like Bubble Sort and Insertion Sort.

But, however we do it, the end result must be the same.

```
public abstract class Sort {

   public abstract int[] sortAsc(int[] input);

   void swap(int[] input, int index1, int index2) {
      int first = input[index1];
      int second = input[index2];
      input[index1] = second;
      input[index2] = first;
   }
}
```

In this design, we have an abstract base class for sorting arrays of integers. Imagine we started by test-driving an implementation of Bubble Sort, and then moved on to an implementation of Insertion Sort, and extracted a common superclass with the shared *swap()* method and an abstract *sortAsc()* method they each override.

```java
public class BubbleSort extends Sort {

   @Override
   public int[] sortAsc(int[] input) {
      boolean sorted = false;
      while(!sorted){
         sorted = true;
         for (int i = 0; i < input.length - 1; i++) {
            if(input[i] > input[i+1]){
               swap(input, i, i+1);
               sorted = false;
            }
         }
      }
      return input;
   }
}
```

```java
public class InsertionSort extends Sort {

   @Override
   public int[] sortAsc(int[] input) {
      for (int i = 0; i < input.length - 1; i++) {
         for(int j = i+1;j > 0;j--){
            if(input[j] < input[j-1]){
               swap(input, j, j-1);
            }
         }
      }
      return input;
   }
}
```

After refactoring the duplication between these two classes, we should also refactor duplication between their test fixtures. So we end up extracting a common test base class.

```
@RunWith(JUnitParamsRunner.class)
public abstract class SortTests {

   private Object data(){
      return new Object[][]{
            {new int[]{1}},
            {new int[]{2,1}},
            {new int[]{3,2,1}},
            {new int[]{2,3,1}},
            {new int[]{5,2,3,4,1}},
            {new int[]{2,1,2,3}},
            {new int[]{12,2,6,1,7,6,13,0}}
      };
   }

   @Test
   @Parameters(method="data")
   public void arrayIsSortedInAscendingOrder(int[] input) {
      int[] output = createSort().sortAsc(input);
      assertThat(Arrays.asList(output),
                        containsInAnyOrder(input));
      for (int i = 0; i < output.length - 1; i++) {
         assertThat(output[i],
                        is(lessThanOrEqualTo(output[i + 1])));
      }
   }

   abstract Sort createSort();

}
```

Note the abstract method *createSort()*; this is a factory method for instantiating sorting implementations that we override in the test fixtures that extend *SortTests*.

```
public class BubbleSortTests extends SortTests {

   @Override
   protected Sort createSort() {
      return new BubbleSort();
   }
}

public class InsertionSortTests extends SortTests {

   @Override
   protected Sort createSort() {
      return new InsertionSort();
   }
}
```

The tests in *SortTests* effectively define an abstract contract that all sorting implementations must satisfy, no matter how they work internally. This test design technique is therefore sometimes referred to as *contract testing*.

EXERCISE #10

Test-drive some code that manages the stock and orders of a CD warehouse. Customers can buy CDs, searching on the title and the artist. Record labels send batches of CDs to the warehouse. Keep a stock count of how many copies of each title are in the warehouse. Customers can only order titles that are in stock. Use dependency injection to fake credit card payment processing, so we can get on with our CD warehouse design without worrying about how that will be done.

Customers can leave reviews for CDs they've bought through the warehouse, which gives each title an integer rating from 1- 10 and the text of their review if they want to say more.

As well as applying all of the ideas we've covered about TDD so far, make sure your code is:

- Working
- Easy to understand
- Has minimal duplication
- Is as simple as possible

…and is made from classes that:

- Tell, don't ask
- Have one distinct responsibility
- Can be composed from the outside
- Expose client-specific interfaces
- Use contract tests for shared abstractions

14. TEST DOUBLES

Summary:

- Test doubles are objects used in tests that aren't the real thing
- They can help us write fast-running tests by decoupling from external dependencies like databases and web services
- They can help us defer implementation details by "faking it 'til we make it"
- They can help make tests that depend on changing or random data repeatable
- Stubs are test doubles that provide test data
- Mocks are test doubles that allow us to test object interactions, and help us to design objects that Tell, Don't Ask
- Over-reliance on mock object frameworks can "bake in" a tightly-coupled design
- Dummies are test doubles that allow the test to compile and run, but aren't used
- Test doubles should implement interfaces that we control, to protect our application code from external dependencies
- Whether a test double is a stub, a mock or a dummy depends on how it's used, not how it's implemented

There are often times, when we're writing automated tests, that we need to use an object that – for a number of possible reasons – is not the real thing.

It could be:

- For performance reasons (e.g., connecting to an external service would not be desirable in a suite of fast-running unit tests.)
- For cost reasons (e.g., requiring Oracle licenses to use a database in a test.)
- Because the type of object we want to use doesn't even exist yet ("Fake it 'til you make it").
- Because we know it won't be used in our test.
- Because the object in question can only exist running inside a container process, like the HTTP context of a web server.
- To make tests repeatable when object behaviour might vary (e.g., getting today's date)

Test doubles come in several flavours:

- **Stubs** – objects that supply test data
- **Mocks** – objects that require interactions to happen
- **Fakes** – objects that exhibit all the behaviour of the real thing (e.g., an in-memory relational database)
- **Dummies** – objects that aren't used, but need to be there to compile and run the test
- **Spies** – objects that remember when their methods are called, so we can query that in our tests

In TDD, stubs, mocks and dummies come up most often. We'll explore their use in this chapter.

STUBS

A stub is a test double that presents an expected interface to our class under test, and has a test-specific implementation that returns data that will be used in our test. More simply, a stub's job is to provide test data. In that sense, a stub's implementation is part of the set-up for a test.

```
public class TradeQuoteTests {

   @Test
   public void tradePriceIsStockPriceTimesQuantity() {
      StockPricer pricer = new StockPricerStub(10);
      TradeQuote trade = new TradeQuote(pricer);
      assertEquals(1000, trade.quote("X" , 100), 0);
   }
}
```

In this test, we want to check that a quote for a stock market trade is calculated correctly. Our *TradeQuote* object will get a price from a *StockPricer*. When the software is in production, an implementation of the *StockPricer* interface would connect to an external web service. For the purposes of our test, though, we write our own test-specific implementation that returns a price of 10.

Note the use of dependency injection here to plug the *StockPricer* stub into the *TradeQuote* object (this is a great illustration of the kind of flexibility we get by composing objects from the outside).

Internally, *TradeQuote* depends only on the interface, and knows nothing about the stub.

```
public class TradeQuote {

   private final StockPricer pricer;

   public TradeQuote(StockPricer pricer) {
      this.pricer = pricer;
   }

   public double quote(String stock, int quantity) {
      return pricer.getPrice(stock) * quantity;
   }
}
```

Notice also how I passed the test data value into the constructor of my stub, rather than hardcoding it into the stub's implementation. I've done this for two reasons; firstly, it means I can specify the value in the actual test code, making it easier to understand. Secondly, I can reuse this stub implementation with different values, meaning less code duplication.

The stub's implementation is simply:

```
public class StockPricerStub implements StockPricer {

   private final double price;

   public StockPricerStub(double price) {
      this.price = price;
   }

   @Override
   public double getPrice(String stock) {
      return price;
   }
}
```

Sometimes, instead of returning test data, we might want a stub to throw an exception to test how our object handles it.

```
   @Test(expected=InvalidTradeException.class)
   public void tradeNotValidIfStockNotFound()
                              throws InvalidTradeException {
      StockPricer pricer = new StockNotFoundStockPricerStub();
      TradeQuote trade = new TradeQuote(pricer);
      assertEquals(1000, trade.quote("X" , 100), 0);
   }
```

When the stub throws a *StockNotFoundException*, *TradeQuote* should catch that and throw an *InvalidTradeException*.

```
public class StockNotFoundStockPricerStub implements
                                                StockPricer {

   @Override
   public double getPrice(String stock)
                              throws StockNotFoundException {
      throw new StockNotFoundException(stock);
   }
}
```

In both tests, I used a stock symbol "X". It doesn't matter what stock symbol we use, as our stubs will return the data we want them to regardless.

Two important things to remember when using stubs:

1. Do not test the stub! Our goals here is to test the object that uses the data the stub provides

2. Stubs are test code

Stubs can also be used to fix test data that would usually change when using the real object - like a person's age – making the test repeatable.

```
@Test
public void driverUnder25PaysFivePercentPremium() {
  Motorist motorist = new Motorist("01/01/1900",
                                   Gender.MALE,
                                   null,
                                   new AgeCalculatorStub(24));
  assertEquals(0.05, motorist.calculateAgePremium(), 0);
}
```

```
public class Motorist {

   private final String dateOfBirth;
   private final Gender gender;
   private final DriversLicense license;
   private final AgeCalculator ageCalculator;

   public Motorist(String dateOfBirth,
            Gender gender,
            DriversLicense license,
            AgeCalculator ageCalculator) {
      this.dateOfBirth = dateOfBirth;
      this.gender = gender;
      this.license = license;
      this.ageCalculator = ageCalculator;
   }

   private double calculateAgePremium() {
      int age = ageCalculator.calculateAge(dateOfBirth);
      double agePremium;
      if(age < 25){
         agePremium = 0.05;
      } else
         if (age > 70){
         agePremium = 0.04;
      } else {
         agePremium = 0.03;
      }
      return agePremium;
   }
```

In our test, it makes no difference what we specify the motorist's date of birth to be. His age will always be "calculated" as 24.

MOCK OBJECTS

Mocks often get mixed up with stubs (and it doesn't help that many developers use mock object frameworks to create stubs). The terms are routinely used interchangeably, even by renowned experts in TDD.

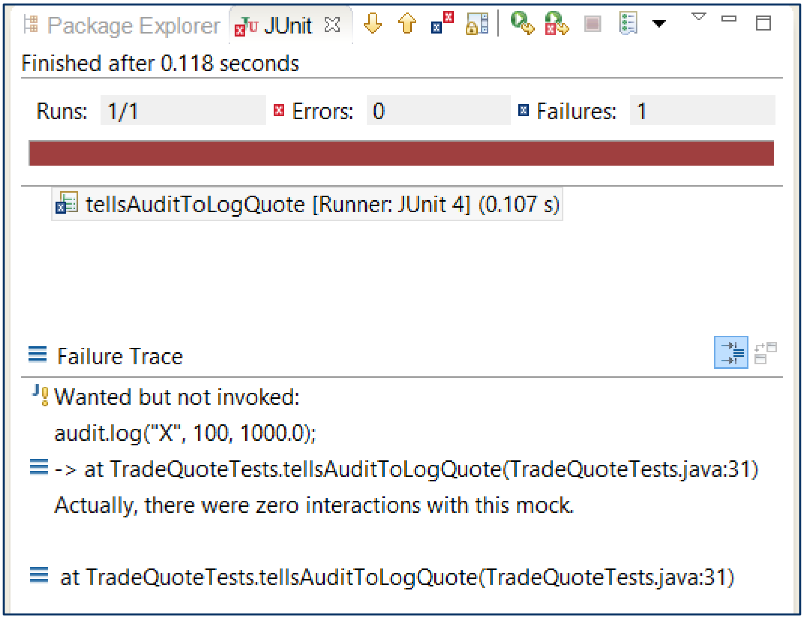
But, strictly speaking, a mock isn't a stub. The purpose of a stub is to provide test data. The purpose of a mock is to allow us to write tests that will fail when an interaction between our object under test and one of its collaborators doesn't happen in the way we say it should.

```
@Test
public void tellsAuditToLogQuote() throws Exception {
   int quantity = 100;
   String stock = "X";
   StockPricer pricer = new StockPricerStub(10);
   Audit audit = mock(Audit.class);
   double quotedPrice =
        new TradeQuote(pricer, audit)
                                 .quote(stock, quantity);
   verify(audit).log(stock , quantity , quotedPrice);
}
```

Suppose we get a new requirement for our *TradeQuote* to log each quote generated for audit purposes.

We don't want to have to inspect the audit log to find out if *TradeQuote* called the *log()* method. And if logs are written to a file or a database, we definitely don't want *TradeQuote* to talk to the real thing in our fast-running unit test.

We can mock *Audit* – in this example using Mockito (www.mockito.org) – and then verify that the interaction took place. Before we write the code to pass this interaction test, we run the test to see that our mock assertion (i.e., *verify*) fails.



```
Finished after 0.118 seconds
Runs: 1/1    Errors: 0    Failures: 1
tellsAuditToLogQuote [Runner: JUnit 4] (0.107 s)
Failure Trace
Wanted but not invoked:
audit.log("X", 100, 1000.0);
-> at TradeQuoteTests.tellsAuditToLogQuote(TradeQuoteTests.java:31)
Actually, there were zero interactions with this mock.

at TradeQuoteTests.tellsAuditToLogQuote(TradeQuoteTests.java:31)
```

To pass the test, *TradeQuote* needs to call *log()* with the right parameter values.

```
public double quote(String stock, int quantity)
                              throws InvalidTradeException {
  try {
    double quotedPrice =
                  pricer.getPrice(stock) * quantity;
    audit.log(stock, quantity, quotedPrice);
    return quotedPrice;
  } catch (StockNotFoundException e) {
    throw new InvalidTradeException(e);
  }
}
```

Note that, although we used the *StockPricer* stub, this test isn't about the calculation of the quote. It's about whether or not *TradeQuote* tells *Audit* to log the quote.

Think back to the chapter on design principles, and Tell, Don't Ask. Using traditional test assertions, we would have needed to provide a way for our test to query the internal state of *Audit* to check if the log had been written. This breaks encapsulation unnecessarily.

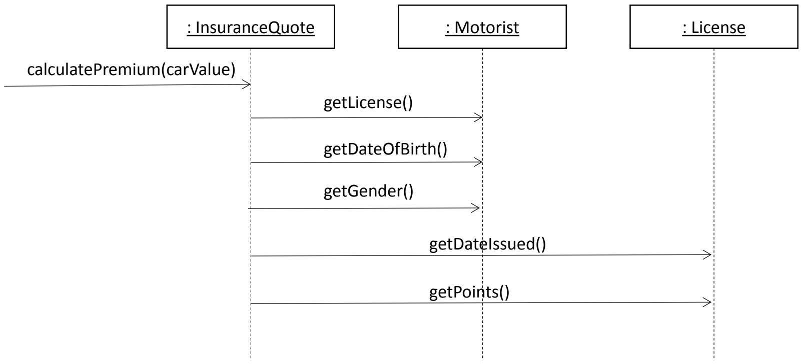
Logging quotes isn't *TradeQuote*'s job. Telling *Audit* to log the quote is.

This is why mock objects were invented: to allow us to more easily test-drive designs made up of objects that Tell, Don't Ask. In this sense, mocks are not really a testing tool. They're a design tool, helping us to test-driven designs that are more loosely coupled.

ABUSING MOCK OBJECT FRAMEWORKS

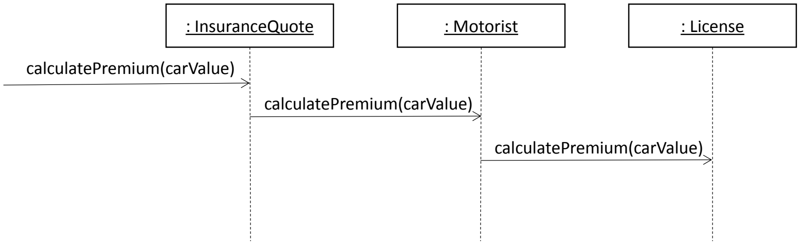
Originally intended as a design tool for TDD, mock object frameworks can help us to test-drive objects that are loosely coupled and that Tell, Don't Ask. But they can be easily abused, ending up with code that is more difficult to change.

Many developers rely on mocks as a crutch for writing tests for poorly designed code. When your designs look like this:



Then things can get a bit sticky in our test code. The problem is that mocking frameworks expose internal details about which methods should get called. Just as surely as lots of getters break object encapsulation, so too does lots of mocking code.

If we wanted to refactor this design to make it more loosely coupled:


: InsuranceQuote
: Motorist
: License
calculatePremium(carValue)
calculatePremium(carValue)
calculatePremium(carValue)

It would break a whole bunch of tests that explicitly rely on there being getters instead.

The whole purpose of mocks is to help us come to a Tell, Don't Ask design in the first place. Abuse and over-reliance on mock objects can effectively *bake in a bad design*.

DUMMIES

Blink and you might have missed the fact that we already used dummy objects in some of the tests in this chapter.

A dummy is an object that won't be used in our test – of, if it is used, we don't care about it – but that has to be included so that we can compile and run the test.

```
@Test
public void driverUnder25PaysFivePercentPremium() {
   Motorist motorist = new Motorist("01/01/1900",
                                    Gender.MALE,
                                    null,
                                    new AgeCalculatorStub(24));
   assertEquals(0.05,
                    motorist.calculateAgePremium(), 0);
}
```

In this test, notice how we pass in a null value for *license* to the *Motorist* constructor. We have to pass in something, or the test code won't compile. But this test doesn't involve a *DriversLicense*, so null is the simplest thing we can use.

It might be that the code we're testing calls methods on a dummy – but those methods don't return any data (so we don't need to use a stub) – in which case we can use the Null Object design pattern.

A Null Object is an empty implementation of an interface that we can call methods on, but those methods don't do anything.

A Null Object implementation for *DriversLicense* would require a pure interface, with a dummy implementation that looks like this:

```
public interface License {

   public abstract void addPoints(int points);

}


public class LicenseDummy implements License {

   @Override
   public void addPoints(int points) {

   }

}
```

When our code under test invokes *addPoints()* on our dummy license, nothing happens. But if the *license* parameter value was actually null, we'd get an unhandled exception.

Another way of creating Null Objects is using a mock objects framework.

```
@Test
public void tradePriceIsStockPriceTimesQuantity(){
   String stock = "INTEL";
   StockPricer pricer = new StockPricerStub(10);
   TradeQuote trade =
               new TradeQuote(pricer, mock(Audit.class));
   assertEquals(1000, trade.quote(stock , 100), 0);
}
```

In this example, we use a mock *Audit* object as a dummy. The test isn't about the interaction with the mock. It's about the calculation of the quote. But we know that *Audit.log()* will be invoked, so passing in a mock object takes care of that. Mockito will generate an implementation of the *Audit* interface that's effectively a Null Object.

WHOSE INTERFACE IS IT ANYWAY?

Imagine, in our example, that our external stock price provider has created a convenient Java API for using their service.

```
public interface AcmeStocks {

  public double price(String stockSymbol);

}
```

Why not use implementations of this to create our test doubles?

If we did, this could cause problems later on. First of all, the design of this interface is beyond our control. We'll need to keep our Acme Stocks API up-to-date, because it connects to a live web service. So every time Acme Stocks change our API, we'll have to change our code that depends on it.

Also, what happens if Acme Stocks go bust? Or if we find a provider who offer better terms and want to switch? If our *TradeQuote* logic depends directly on their interface, we may have to rewrite all that code.

It's best to protect our code from direct external dependencies like this, by declaring our own interfaces, that we control, that will allow us to swap implementations without rewriting big chunks of our application logic.

True that, somewhere in our code, we'll have to live with the direct dependency. But aim to isolate that dependency, keeping it as small as possible, and in one easily-swapped placed. We'll discuss test-driving integration code in the next chapter.

MOCKS VS. STUBS VS. DUMMIES

What distinguishes a mock from a stub from a dummy is not how these test doubles are implemented, but *how they are used* in our tests.

We can create stubs and dummies using mock object frameworks. E.g.

```
@Test
public void tradePriceIsStockPriceTimesQuantity(){
   String stock = "INTEL";
   StockPricer pricer = mock(StockPricer.class);
   when(pricer.getPrice(stock)).thenReturn(10.0);
   TradeQuote trade =
                new TradeQuote(pricer, mock(Audit.class));
   assertEquals(1000, trade.quote(stock , 100), 0);
}
```

We created *pricer* using the *mock()* method, but set it up to return test data. This test isn't about the interaction with the *StockPricer*, it's about the calculation of the quote. Therefore *pricer* is a stub, not a mock.

And, in the same test, we use *mock()* to create a dummy implementation of *Audit*. Again, it's not a mock if our intention isn't to test that methods on the *Audit* object are invoked.

Finally, we can create mock objects without using mocking frameworks. At their essence, mock objects are just implementations of interfaces that remember when their methods are invoked (and with what parameter values), allowing us to test the interactions between objects in our designs.

There are many ways this could be achieved in code. A simple way in Java might be to use anonymous classes to implement interfaces, with method implementations that record interactions.

(Indeed, according to a pioneer of mock objects, Steve Freeman, this is how they started.)

```
public class LibraryTests {

   private boolean awardPriorityPointsInvoked;
   private boolean registerCopyInvoked;

   @Test
   public void tellsTitleToRegisterCopy() {
      registerCopyInvoked = false;
      Member member = new Member(){
         public void awardPriorityPoints(int points){}
      };
      Title title = new Title(){
         public void registerCopy(){
            registerCopyInvoked = true;
         }
      };
      new Library().donate(title, member);
      assertTrue("title.registerCopy() was not invoked",
                                          registerCopyInvoked);
   }

   @Test
   public void tellsMemberToAwardTenPriorityPoints() {
      awardPriorityPointsInvoked = false;
      Member member = new Member(){
         public void awardPriorityPoints(int points){
            awardPriorityPointsInvoked = (points == 10);
         }
      };
      Title title = new Title(){ public void registerCopy(){}};
      new Library().donate(title, member);
      assertTrue(
            "member.awardPriorityPoints(10) was not invoked",
            awardPriorityPointsInvoked);
   }

}
```

So, a dummy isn't a mock just because it was created using a mocking framework. And you don't need to use a mocking framework to create mock objects.

Remember:

1. If it's there to provide test data, it's a **stub**.
2. If it's not important, but has to be there for the test to compile and run, it's a **dummy**.
3. If we're using it to test object interactions, it's a **mock**.

EXERCISE #11

Test-drive some code that compares prices on TVs from three different sources:

1. *Screen Bargains* – an online TV retailer with a web API
2. *Acme TV* – a retail chain with an old-fashioned TCP/IP Electronic Data Interchange interface
3. *Televizion* – a mail order company who provide a monthly price list in an Excel spreadsheet

By specifying a make and model of television, your code will find the best price and recommend that retailer. If more than one retailer is offering the same best price, your code will list them all.

Searches also trigger a message to be sent to your ad targeting engine, detailing the make and model of TV the user is interested in.

Apply all of the TDD principles and practices we've looked at so far, and use test doubles appropriately to provide the test data that would normally come from these 3 external sources, and to test-drive sending a message to the ad targeting engine. For any objects in your test that need to be there so it will run, but won't be used, use a dummy.

15. TEST-DRIVING INTEGRATION CODE

Summary:

- Minimise code that needs to be integration tested, so you have to live with as few slow-running tests as possible
- Aim for < 5% integration code (and <5% integration tests)
- Isolate and minimise duplication of code that has external dependencies
- Use dependency injection to make integration code easily swappable
- Group fast-running and slow-running tests separately, so we can easily choose which kind to run
- For ultimate flexibility, package integration code separately

Imagine we needed to test-drive some code that calculates average ratings of video titles supplied by an external website called Rotten Potatoes.

We could stub the service that fetches the reviews for a title, so we can test the calculation of the average.

```
@Test
public void averageVideoRatingIsTotalDividedByCount() {
      String name = "Jaws 3D";
      Title title = new Title(name);
      Review[] reviews = new Review[2];
      reviews[0] = new Review(name, 3, "");
      reviews[1] = new Review(name, 2, "");
      ReviewsService reviewsService =
                     new ReviewsServiceStub(reviews );
      VideoStats videoRating =
                     new VideoStats(title, reviewsService);
      assertEquals(2.5, videoRating.average(), 0);
}
```

This gives us a fast-running test for the calculation. But at some point, surely, we're going to have to write some code that actually connects to Rotten Potatoes' API, right?

Let's write a test for a production implementation of *ReviewsService*.

```
public class JSONReviewsServiceTests {

   @Test
   public void reviewsTestServiceHasTwoReviewsOfJaws3D() {
      ReviewsService service =
           new JSONReviewsService(
        "http://localhost:8080/rottenpotatoes/json/reviews/");
      Review[] reviews = service.fetchReviews("Jaws 3D");
      assertEquals(2, reviews .length);
   }

}
```

When we run this test, it will connect to a test reviews server at the URL specified and use an HTTP GET to retrieve all reviews for Jaws 3D (of which we know there are two, because we control that test data.)

In our implementation, a bunch of stuff happens:

```
public class JSONReviewsService implements ReviewsService {

   private final String url;

   public JSONReviewsService(String REST_url) {
      this.url = REST_url;
   }

   @Override
   public Review[] fetchReviews(String titleName) {
      String json = "";
      try {
         url += URLEncode.encode(titleName, “UTF-8”) + “/get”;
         CloseableHttpClient httpClient =
                      HttpClients.createDefault();

         HttpGet getRequest = new HttpGet(url);
         getRequest.addHeader("accept", "application/json");
         HttpResponse response;
         response = httpClient.execute(getRequest);
```

```
            if (response.getStatusLine().getStatusCode() != 200) {
               throw new RuntimeException(
                    "Failed : HTTP error code : "
                    + response.getStatusLine().getStatusCode());
            }

            BufferedReader br =
                    new BufferedReader(new InputStreamReader(
                         (response.getEntity().getContent())));

            String output;

            while ((output = br.readLine()) != null) {
               json += output;
            }

            httpClient.close();

         } catch (ClientProtocolException e1) {
            e1.printStackTrace();
         } catch (IOException e1) {
            e1.printStackTrace();
         }

         JSONArray jsonReviews = new JSONArray(json);
         Review[] reviews = new Review[jsonReviews.length()];

         for (int i = 0; i < jsonReviews.length(); i++) {
            JSONObject obj = jsonReviews.getJSONObject(i);
            reviews[i] =
                  new Review(  obj.optString("title"),
                               obj.optInt("rating"),
                               obj.optString("comment"));
         }
         return reviews;
      }
   }
```

If we write a data service like this for every kind of externally-provided data in our application, we could wind up with a lot of code that has to be integration tested, and a large suite of slow-running tests.

Remember our design principles: is this *JSONReviewsService* doing one specific thing?

In fact, it does two things:

1. Fetch the JSON data from the reviews server

2. Parse the data and build an array of reviews

Let’s refactor this design into two classes.

```
@Override
public Review[] fetchReviews(String titleName) {
   RESTClient client = new RESTClient(url);
   String json = client.get(titleName);

   JSONArray jsonReviews = new JSONArray(json);
   Review[] reviews = new Review[jsonReviews.length()];

   for (int i = 0; i < jsonReviews.length(); i++) {
      JSONObject obj = jsonReviews.getJSONObject(i);
      reviews[i] =
            new Review(  obj.optString("title"),
                         obj.optInt("rating"),
                         obj.optString("comment"));
   }
   return reviews;
}
```

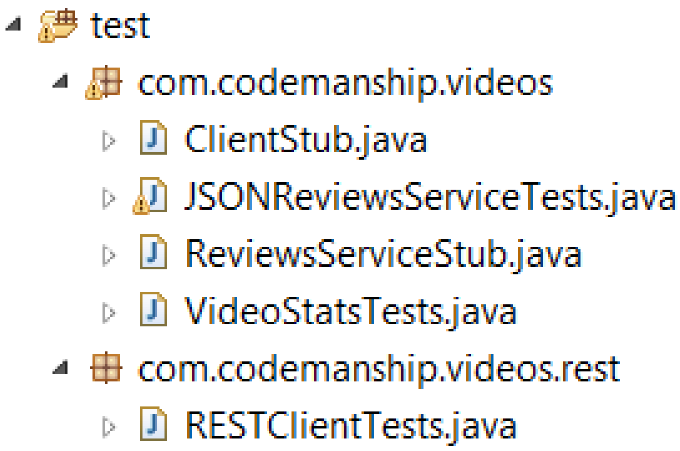
Next, let’s compose it from the outside, using dependency injection to make *RESTClient* swappable.

```
public class JSONReviewsService implements ReviewsService {

   private final Client client;

   public JSONReviewsService(Client client) {
      this.client = client;
   }

   @Override
   public Review[] fetchReviews(String titleName) {
      String json = client.get();
```

RESTClient – from which we extracted the *Client* interface - gets its own integration test, which has nothing to do with reviews or ratings.

```
public class RESTClientTests {

   @Test
   public void returnsDataFromSpecifiedRESTurl() {
      String url = "http://localhost:8080/resttest/json/test";
      RESTClient client = new RESTClient(url);
      assertEquals("[{ foo : 0 }]", client.get("foo"));
   }
}
```

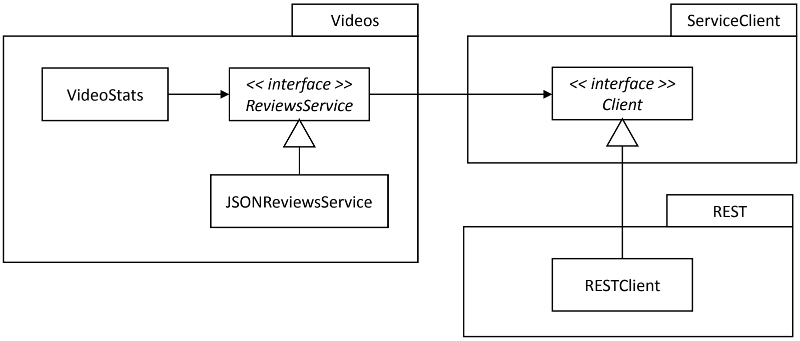
We can easily separate this slow-running integration test from the fast-running tests, enabling us to choose whether to run only unit tests, or only integration tests. (Or all tests).



```
test
   com.codemanship.videos
      ClientStub.java
      JSONReviewsServiceTests.java
      ReviewsServiceStub.java
      VideoStatsTests.java
   com.codemanship.videos.rest
      RESTClientTests.java
```

We can reuse *RESTClient* for other services. Say, for example, we're asked to pull a release schedule of new video titles from an online retailer's REST API.

We can even go a step further, and package our integration code (and associated tests) separately, so it can be reused in other development projects. (NB: in this context, "package" means a unit of release, like a Java JAR file, or a DLL in .NET.)



The *Videos* package only depends directly on the *ServiceClient* package, which the *REST* package extends. This would give us ultimate flexibility. We could even swap in new *Client* implementations without stopping the application.

Our refactored design offers us three opportunities we didn't have before:

- We can stub *Client* when testing *JSONReviewsService,* and test that the JSON data is parsed correctly by itself

```
@Test
public void fetchesReviewsForTitle() {
   String reviewsJson = "[" +
      "{title : \"Jaws 3D\", rating : 3, comment: \"\"}," +
      "{title : \"Jaws 3D\", rating : 3, comment: \"\"}," +
      "]";
   ReviewsService service =
         new JSONReviewsService(
                                     new ClientStub(reviewsJson ));
   Review[] reviews = service.fetchReviews("Jaws 3D");
   assertEquals(2, reviews.length);
}
```

- We can reuse *RESTClient* for other kinds of data that needs to be retrieved from a REST service. All it needs is the URL and parameter values.
- We can substitute a different client implementation dynamically, which can help us if there are multiple data sources, or if we're load-balancing across multiple REST servers.

In practice, code that has direct external dependencies can be greatly minimised by following the design principles of minimising duplication, giving methods and classes a single responsibility, and composing objects from the outside. I typically find integration code need only make up less than 5% of the code in an application, and therefore less than 5% of the tests.

We can do the maths; integration code is – by its very nature - at the edges of our system, meaning that changes to inner code (UI logic, controllers, business logic, etc) usually can't break it. And, as it's less than 5% of the total code, we might expect to be changing it less than 5% of the time. Which means we need to run our integration tests 20x less often than our unit tests.

If we're well-organised about it, slow-running integration tests don't have to be a burden.

There's more refactoring that needs be done to improve this code. We've made it easier by minimising and isolating the integration code.

EXERCISE #12

Continuing with the same code you write for Exercise #11 ("Test-drive some code that compares prices on TVs from three different sources"), rig up test versions of those 3 data sources (a web service, a simple TCP/IP daemon, and an Excel spreadsheet). Set-up a local file to store audit logs.

Test-drive implementations that will get data from or write data to these external sources. Try as much as possible to isolate the external dependencies and minimise the code that really needs to be integration tested.

16. TDD WITH THE CUSTOMER

Summary:

- Examples help us to pin down the precise meaning of requirements
- We can extract data from customer examples to use in tests
- A user story is a placeholder to have a conversation with the customer where we agree tests that will act as our requirements specification
- Writing tests is a skilled job, and the customer will probably require our assistance to produce effective tests
- The customer's tests must define every input scenario the software will need to handle
- Negotiate feature scope and complexity by negotiating tests
- If you realise test cases have been missed, go back to the customer to agree new tests. You are not the customer
- A feature isn't "done" until it passes the customer's tests
- Work in vertical slices, delivering working software that passes the customer's tests
- Making customer tests machine-executable guarantees absolute precision
- Tools like FitNesse allow customers to provide test data we can use in *executable specifications*
- Once we have a failing customer test, we can implement a design that will pass the test
- Close customer involvement is vital. There's no workaround or substitute that works anywhere near as well.

A common misconception about TDD is that it focuses on unit tests and the internal design of our software. In fact, the tests that drive our designs can be written at any level of design. They could be system tests that drive the software through an external interface, integration tests that drive the interactions between systems, services or components, or unit tests that drive the design of our classes.

An increasingly popular application of using tests as specifications helps us to communicate with our customers, building a precise shared understanding of what is required from the software.

SPECIFICATION BY EXAMPLE

Decades of experience working with customers to understand their requirements has taught us that the best way to pin down exactly what the customer wants is to use examples.

In real life, someone might specify that they like their coffee "hot" and "sweet". But how hot is "hot", and how sweet is "sweet"?

We could ask the customer to specify the precise temperature they like their coffee served at (e.g., 90°C), and the exact sugar content they desire (40g/L). But, chances are, they don't know what the precise temperature is, or exactly how many grams of sugar per litre. As expert baristas, we may think in those terms: our customer probably doesn't.

To understand how our customer really likes their coffee, we could ask them to give us an example cup that they believe is just right, and extract data from that example about the precise temperature and sugar content.

To flesh out our understanding of how customers want their coffee, we could ask for more examples. Maybe Jack likes his coffee "hot and sweet", but Jane likes it "white with no sugar" and Rajesh likes it "milky with one lump". Exactly how much milk do we put in to make the coffee "white"? How much more to make it "milky"? How much sugar is there in "one lump"? And so on.

We can apply the same technique to pinning down software requirements. A customer may ask that:

"When a movie title is added to the library, members who expressed an interest in borrowing it are alerted"

Which movie title? Who expressed an interest in borrowing it? How do we know they're interested?

By asking the customer to give a specific example, we can remove the ambiguity from their specification:

*"When movie title **The Abyss** is added to library, members **joepublic**, **janedoe** and **fredbloggs** are alerted because they expressed an interest in borrowing titles containing **'abyss'** "*

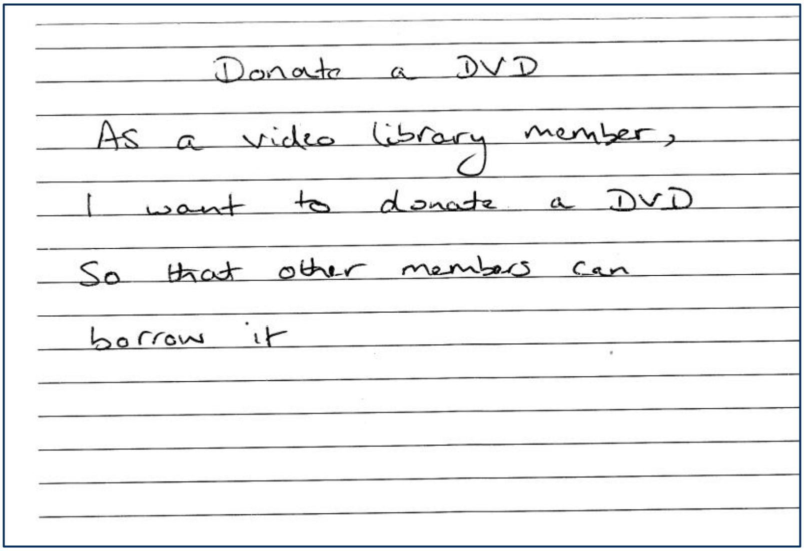
In Extreme Programming, we agree the precise details of user stories using customer test examples as our specifications.

This requires us to work very closely with our customers. Don't let them leave the room until you've got a good set of customer tests to work from. And don't write a line of implementation code unless you have a failing customer test that requires it.

If you are disciplined and rigorous about it, your customer will soon learn that if there isn't a test for it, they ain't getting it.

USER STORIES – PLACEHOLDERS FOR CONVERSATIONS

In Extreme Programming, customers request new features and changes to existing features by writing user stories. A user story is not, in itself, a requirements specification. It contains just enough information to uniquely identify the requirement, and serves purely as a placeholder to remind the developers to have a conversation with the person who wrote the user story to agree the details.



Donate a DVD

As a video library member,
I want to donate a DVD
So that other members can
borrow it

In a test-driven approach, when developers pick up a user story to work on, the output of this conversation with the customer should include a set of tests that precisely specify what's required.

Customers are usually not software testers, so we must offer them guidance on this process and help them to identify the test scenarios we'll need to consider (e.g., if they ask for new library members to choose a password when they join, we might ask the customer to consider what should happen if the password they choose is too weak, or what should happen if the password field is left blank, and so on.)

Teams that expect customers to go away and write the tests themselves could be waiting a long time. This is a technical skill that takes a long time to master. If you have dedicated testers on your team, this is one area where they can prove very useful, helping the customer to articulate their needs as tests.

In our example, working with the customer, we identify several tests that the system will need to pass:

- Donating a movie title that isn't in the library (the "happy path")
- Donating multiple copies of the same movie title
- Donating a copy of a movie title that the library already has copies of
- Donating a copy of a "blockbuster" movie title (one that's highly sought after by members, earning double the reward points)

TEST COMPLETENESS & TEST SCOPE

Writing good tests for a user story can require a considerable time investment from everyone involved, and this can encourage teams to rush the process. When we miss test cases that our code will need to handle, we end up with an incomplete specification, and – ultimately – incomplete software.

The software must meaningfully handle every input that its interface allows, so we'll need at least one test to cover every unique possibility.

If a user story generates too many tests, then that is a sign that it's too complicated. We can break complex stories down into sub-requirements, as well as limiting test cases by simplifying or constraining the allowable inputs.

For example, we could split "Donate a DVD" into "Donate a single copy of a DVD" and "Donate multiple copies of a DVD". Or we could decide that users can only donate one copy at a time (since it will probably be a rare occurrence for them to own multiple copies of the same movie title.)

What we must *never* do is allow an input that the software doesn't handle. For example, if the library's user interface allows members to donate more than one copy, but the code only registers one copy.

Writing tests with the customer is often a negotiation over the software's scope, so be prepared to help them get working software sooner by limiting that scope.

THE TESTS WE DIDN'T THINK OF

Try as we might to identify every test case for a user story before we start writing code, the maxim "the map is not the terrain" will inevitably apply.

While test-driving an implementation of our movie title class, we might discover that it's possible for there to be two different movies with the same name. (For example, there are two movies called "The Thing".) How do we disambiguate them in the library?

We could identify movies by both the name and the year of release (e.g., "The Thing (1982)" and "The Thing (2011)").

But this is not a change we can make without rethinking our user interface. As developers, we must be aware that every line of code we write in some way defines the user's experience.

If a change to the code will mean a change to the externally visible or measurable functioning of the software, then we shouldn't make that decision by ourselves. It's really a decision for the customer.

When you hit new test cases during implementation, take them to the customer and specify the changes with them as part of their tests for that feature.

DEFINITION OF "DONE"

In a test-driven approach to development, the customer's tests provide us with a clear understanding of what they need from the software.

Going back to our coffee example, we can deliver as many cups of coffee to the customer as we like, but we're not done until we've delivered a cup that passes their test (90°C with 10g/L of sugar).

The customer should not accept a delivery until it passes their tests, and this is why we often refer to them as *acceptance tests*.

This not only helps us to pin down requirements, clearing up possibly very costly misunderstandings, it can also help us to measure our progress much more objectively.

Software developers are notorious for saying they are "90% done" when completion of really still a long way off. But when we assess completeness based on passing customer tests (e.g., it passes 90% of the customer's tests), we find we get a much more realistic picture of where we are.

GETTING TO "DONE" IN VERTICAL SLICES

Some teams make the mistake of working on application layers, instead of cutting vertical slices through those layers. So by the release date they may end up writing, say, two thirds of the code, but not get as far as implementing the user interface, or wiring in the database, so none of the features can be used.

Feature	**Progress %**	**UI**	**Services**	**Domain**	**DB**
Donate a DVD	70%	0%	80%	100%	100%
Borrow a DVD	75%	0%	100%	100%	100%
Join the library	65%	0%	60%	100%	100%
Refer a friend	75%	0%	100%	100%	100%
Review a movie	75%	0%	100%	100%	100%
Search for titles	50%	0%	0%	100%	100%
Report DVD lost or damaged	50%	0%	0%	100%	100%
Reverse a DVD	50%	0%	0%	100%	100%
Spend reward points	75%	0%	100%	100%	100%
Transfer reward points	75%	0%	100%	100%	100%
Total progress	66%				

Other teams make the mistake of going through a specific development activity for all of the features (i.e., "we'll design it all, then code it all, then we'll test it all"). Again, the risk if they only manage to get two thirds of the work done before the release date, they'll have a whole bunch of untested features at the finish line.

Feature	Progress %	Analysis	Design	Coding	Testing
Donate a DVD	75%	100%	100%	100%	0%
Borrow a DVD	75%	100%	100%	100%	0%
Join the library	68%	100%	100%	70%	0%
Refer a friend	70%	100%	100%	80%	0%
Review a movie	50%	100%	100%	0%	0%
Search for titles	50%	100%	100%	0%	0%
Report DVD lost or damaged	63%	100%	100%	50%	0%
Reverse a DVD	63%	100%	100%	50%	0%
Spend reward points	75%	100%	100%	100%	0%
Transfer reward points	75%	100%	100%	100%	0%
Total progress	66%				

Driving development with customer tests encourages to organise ourselves around delivery of working features. If we only manage to do two-thirds of the work, we should finish up with two-thirds of the features tested and *working*.

Feature	Progress %	Total Tests	Passed
Donate a DVD	60%	5	3
Borrow a DVD	100%	4	4
Join the library	100%	2	2
Refer a friend	100%	2	2
Review a movie	100%	4	4
Search for titles	0%	4	0
Report DVD lost or damaged	0%	2	0
Reserve a DVD	0%	2	0
Spend reward points	100%	2	2
Transfer reward points	100%	1	1
Total progress	66%		

Cut vertical slices through both your architecture – UI, services, domain, database - and your development process – analysis, design, coding, testing, release – to ensure that when you say you're 66% "done", you really are 66% done, and the customer can benefit from their investment.

Organise your team around the question “who do we need to deliver this working feature?”

EXECUTABLE SPECIFICATIONS

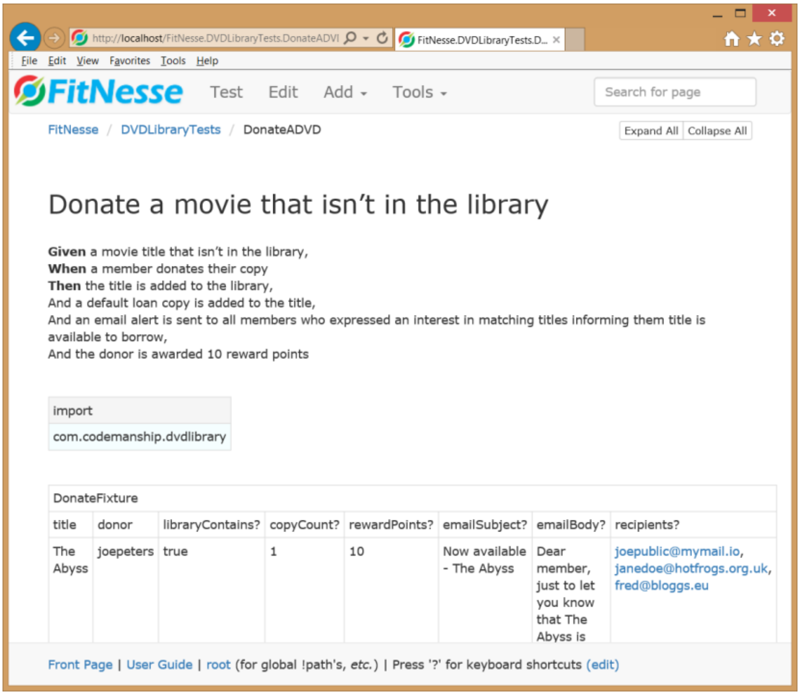
When it comes to specifications, there’s “precise”, and then there’s “precise enough to be executed by a computer”.

To completely eliminate ambiguity from customer specifications, many development teams write automated tests that check the software works as desired for each example.

There are many tools available for providing customer example data to automated tests, but the basic design pattern is always the same: *paramaterised test with customer data*.

We write a parameterised test – much as we’ve done throughout this book – and then data provided by the customer, captured in a file format they themselves can edit (e.g., a table in a Wiki page, or a worksheet in a spreadsheet), is sucked in to provide the parameter values.

A popular tool is FitNesse (www.fitnesse.org), written by Robert C. Martin. It enables customers to write their examples on Wiki pages, providing the example data in tables which can then be extracted and used by automated tests.


http://localhost/FitNesse.DVDLibraryTests.DonateADVI
FitNesse.DVDLibraryTests.D...
File Edit View Favorites Tools Help
FitNesse Test Edit Add Tools
Search for page
FitNesse / DVDLibraryTests / DonateADVD
Expand All Collapse All

Donate a movie that isn't in the library

Given a movie title that isn't in the library,
When a member donates their copy
Then the title is added to the library,
And a default loan copy is added to the title,
And an email alert is sent to all members who expressed an interest in matching titles informing them title is available to borrow,
And the donor is awarded 10 reward points

import
com.codemanship.dvdlibrary

DonateFixture							
title	donor	libraryContains?	copyCount?	rewardPoints?	emailSubject?	emailBody?	recipients?
The Abyss	joepeters	true	1	10	Now available - The Abyss	Dear member, just to let you know that The Abyss is	joepublic@mymail.io, janedoe@hotfrogs.org.uk, fred@bloggs.eu

Front Page | User Guide | root (for global !path's, etc.) | Press '?' for keyboard shortcuts (edit)

In this example, the customer has written a general description of their test in the *Given…When…Then* format prescribed by a variant of TDD called *Behaviour-Driven Development*.

The *Given* clause describes the setup for the test. The *When* clause describes the action being tested. And the *Then* clause describes the desired outcomes (essentially, the test assertions.)

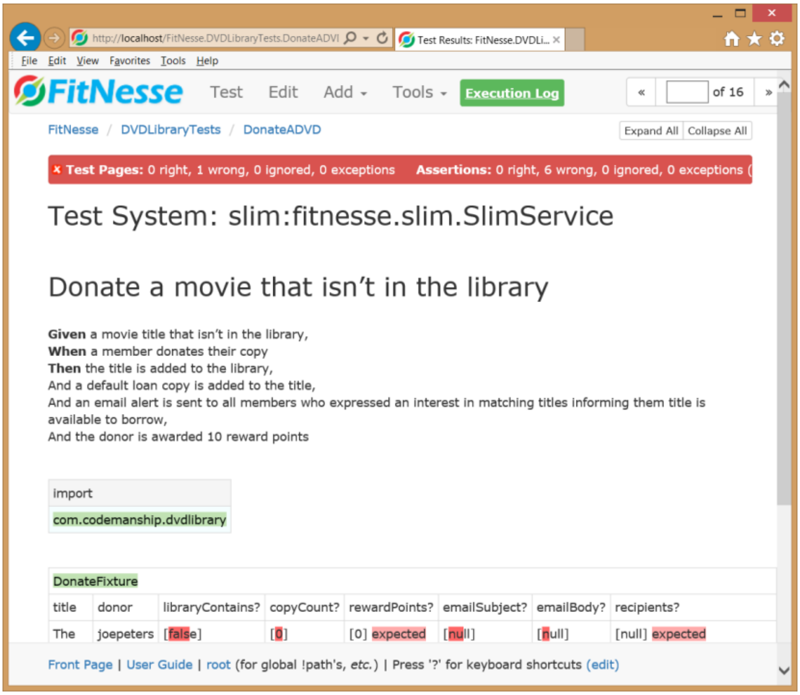
Underneath that, our customer has provided test data in a table for a specific example, which we will use in our automated FitNesse test.

To automate a FitNesse test like this one, we just need to write a fixture – a plain old Java object that has the name we assigned to the table, *DonateFixture*.

The inputs will be provided through setters on our object with names that match the columns *title* and *donor*. The outputs will be accessed through getters that match the columns *libraryContains*, *copyCount*, *rewardPoints*, *emailSubject*, *emailBody* and *recipients*. If the values of these outputs don't precisely match the data in the table, the test will fail.

```
public class DonateFixture {

  public void setDonor(String memberId){}

  public void setTitle(String name){}

  public boolean libraryContains(){  return false; }

  public int copyCount(){ return 0; }

  public int rewardPoints(){  return 0; }

  public String emailSubject(){  return null; }

  public String emailBody(){  return null; }

  public String recipients(){  return null; }
}
```

If we were to run our FitNesse test – by clicking on the *Test* link at the top of the page – we would see that, for now, we have a failing customer test.



http://localhost/FitNesse.DVDLibraryTests.DonateADVD
Test Results: FitNesse.DVDLi...
File Edit View Favorites Tools Help
FitNesse
Test
Edit
Add
Tools
Execution Log
of 16
FitNesse / DVDLibraryTests / DonateADVD
Expand All
Collapse All

Test Pages: 0 right, 1 wrong, 0 ignored, 0 exceptions   Assertions: 0 right, 6 wrong, 0 ignored, 0 exceptions (

Test System: slim:fitnesse.slim.SlimService

Donate a movie that isn't in the library

Given a movie title that isn't in the library,
When a member donates their copy
Then the title is added to the library,
And a default loan copy is added to the title,
And an email alert is sent to all members who expressed an interest in matching titles informing them title is available to borrow,
And the donor is awarded 10 reward points

import
com.codemanship.dvdlibrary

DonateFixture							
title	donor	libraryContains?	copyCount?	rewardPoints?	emailSubject?	emailBody?	recipients?
The	joepeters	[false]	[0]	[0] expected	[null]	[null]	[null] expected

Front Page | User Guide | root (for global !path's, *etc.*) | Press '?' for keyboard shortcuts (edit)

In the next chapter, we'll test-drive an implementation that will pass this test using the techniques we've learned so far.

THE CUSTOMER CANNOT BE REPLACED

Usually, for organisational reasons, development teams struggle to get the level of customer involvement a test-driven approach to requirements specification requires.

It can take up hours of their time each week thrashing out tests for user stories, and they also need to be available to confirm that the software we delivered does indeed pass their acceptance tests.

They might prefer to do all of this in the early stages of development, and then not get involved again until it's ready to be released. This traditional model of customer engagement has dominated software development for decades, and it’s the expectation of many customers and managers.

And, although tools like FitNesse are far more customer-friendly than unit tests, they still present a non-trivial learning curve for non-technical stakeholders. This is why 80% of teams end up writing the “customer tests” themselves, or employ a dedicated analyst or tester to do it on behalf of the customer.

Be absolutely clear that just because the software passes the “acceptance tests”, that doesn’t necessarily mean it will be accepted. Only the customer can decide that. They must be closely involved in writing the tests, and must see the software pass them with their own eyes.

You must move mountains if necessary to get the customer involved. And don’t compromise on the Golden Rule – if the customer hasn’t agreed a failing test for it, we don’t write that code.

EXERCISE #13

Working with a friend or colleague who will act as your customer, agree a set of user stories (no more than 6) for a software application that solves a problem for them.

Ask your customer to pick what they believe is the most important user story, and work with them to define a set of executable customer tests – using real example data – and capture those tests using a tool like FitNesse.

Starting with the most useful test example (usually the “happy path” where the end user achieves their goal), implement just enough of an automated test to see it fail.

17. DRIVING DESIGN FROM CUSTOMER TESTS

Summary:

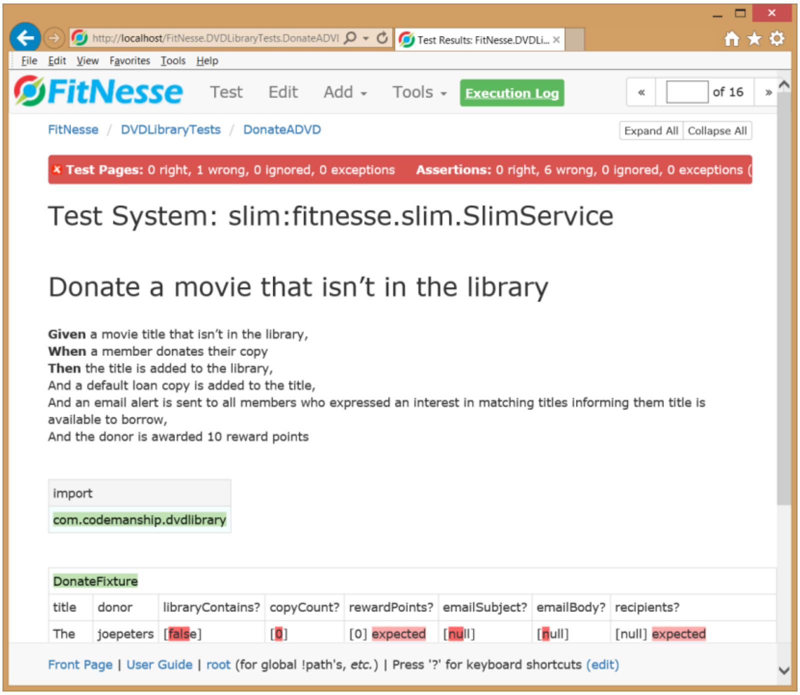
- The design process starts with a failing customer test
- Identify the work the software has to do to pass the test
- Identify what data is needed to do each piece of work
- Assign responsibility for doing the work to objects that will own the needed data (remember Tell, Don't Ask)
- Choose meaningful names for those objects, drawing inspiration directly from the customer's test
- For each unique assertion in the customer's test, test-drive an implementation that will make that part "go green" using the techniques we've explored in this book so far
- Wire each worker object into the automated customer test fixture, and see it pass before moving on to the next assertion
- Keep running the customer test. Feedback, feedback, feedback!
- Use test doubles (stubs, mocks, dummies) to exclude external dependencies from the automated customer test, and to allow us to "fake it 'til we make it" for any components we don't want to implement yet
- When the whole customer test is passing, consider speeding up execution by adapting the test fixture to also run as an xUnit test, if possible
- Just because we've passed the automated customer test, that doesn't mean we're "done" yet

In the previous chapter, we explored how we can pin down precise requirements specifications by agreeing executable tests with our customers.

In this chapter, we'll look at how we can drive the internal design of our software directly from these tests.

START WITH A FAILING CUSTOMER TEST

For the user story "Donate a DVD", we agreed a test with our customer for the happy path, which we captured on a FitNesse Wiki page, and made it into an executable specification with an empty Java test fixture.



http://localhost/FitNesse.DVDLibraryTests.DonateADVI
Test Results: FitNesse.DVDLi...
File Edit View Favorites Tools Help
FitNesse
Test
Edit
Add
Tools
Execution Log
of 16
FitNesse / DVDLibraryTests / DonateADVD
Expand All
Collapse All

Test Pages: 0 right, 1 wrong, 0 ignored, 0 exceptions Assertions: 0 right, 6 wrong, 0 ignored, 0 exceptions

Test System: slim:fitnesse.slim.SlimService

Donate a movie that isn't in the library

Given a movie title that isn't in the library,
When a member donates their copy
Then the title is added to the library,
And a default loan copy is added to the title,
And an email alert is sent to all members who expressed an interest in matching titles informing them title is available to borrow,
And the donor is awarded 10 reward points

import
com.codemanship.dvdlibrary

DonateFixture							
title	donor	libraryContains?	copyCount?	rewardPoints?	emailSubject?	emailBody?	recipients?
The	joepeters	[false]	[0]	[0] expected	[null]	[null]	[null] expected

Front Page | User Guide | root (for global !path's, *etc.*) | Press '?' for keyboard shortcuts (edit)

Remember the first of the four principles of Simple Design: the software must work.

Our ultimate goal, which we will return to continuously throughout the implementation process, is to pass this test.

We want short feedback loops, and therefore to tackle the design in small discrete “chunks”. We’ll tick off each outcome our test requires one a time, verifying that it works by running the FitNesse test.

IDENTIFY THE WORK

The first outcome we need to satisfy is that, after it’s been donated, the library should contain that title.

So, the first piece of work our implementation needs to do is *add the donated title to the list of available titles in the library*.

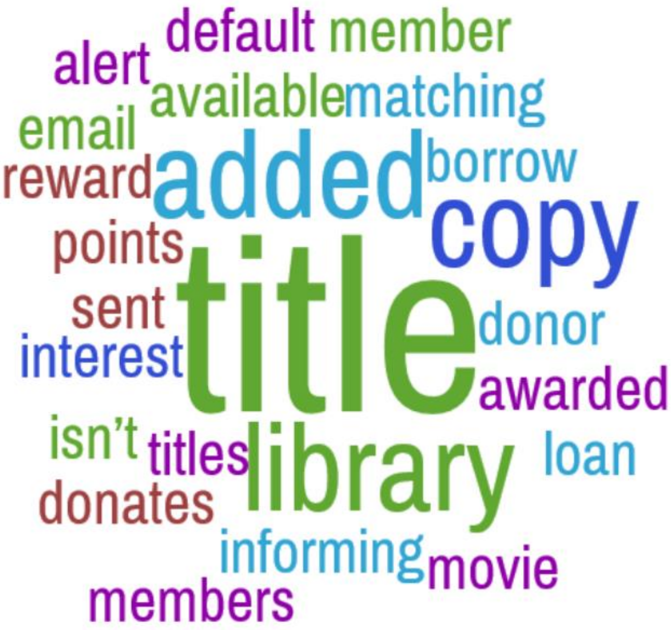
IDENTIFY THE KNOWLEDGE NEEDED TO DO THE WORK

If our design is going to be effectively modular, then we should aim for classes that are cohesive and loosely coupled. The design principle of Tell, Don’t Ask applies here.

The work of adding the donated title to the list of available titles should be placed in the class that knows about available titles. What we need next is to think of a good name for this class.

NAME THE WORKER

Using an idea we explored in *Speaking The Customer’s Language*, let’s run our requirements specification through a tag cloud generator and see if there’s a name that jumps out at us.



Since the list of available titles is part of the library, it makes sense to call this class *Library*.

Let's test-drive an implementation of *Library* to make it do this work.

TEST-DRIVING ADDING A TITLE TO THE LIBRARY

First, observing the Golden Rule, we write a failing unit test for *Library*.

```
public class LibraryTests {

  @Test
  public void donatedTitlesAreAddedToAvailableTitles(){
    Library library = new Library();
    Title title = mock(Title.class);
    library.donate(title);
    assertTrue(library.getAvailableTitles()
                                    .contains(title));
  }
}
```

Notice how I've used Mockito to create a dummy *Title* for this test. Title, at this point, needs no implementation, only an object identity. No need to think about the implementation of *Title* while we're focusing on *Library*.

```
public class Title {

}
```

Next, we write the simplest code to make this test pass.

```
public class Library {

  private List<Title> availableTitles;

  public Library() {
    this.availableTitles = new ArrayList<>();
  }


  public void donate(Title title) {
    availableTitles.add(title);
  }

  public List<Title> getAvailableTitles() {
    return availableTitles;
  }
}
```

Next, we refactor to clean up any design problems before we move on. In this instance, we might not be happy about exposing the internal collection *availableTitles* to enable the test assertion.

Let’s extract a new method that encapsulates checking if the *Library* contains a title, and move it to where it belongs.

```
@Test
public void donatedTitlesAreAddedToAvailableTitles(){
   Library library = new Library();
   Title title = mock(Title.class);
   library.donate(title);
   assertTrue(library.contains(title));
}
```

Finally, *getAvailableTitles()* is no longer being used, so let’s inline it.

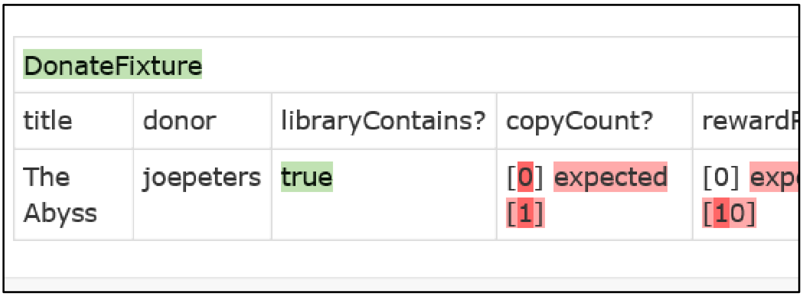
```
public class Library {

   private List<Title> availableTitles;

   public Library() {
      this.availableTitles = new ArrayList<>();
   }

   public void donate(Title title) {
      availableTitles.add(title);
   }

   public boolean contains(Title title) {
      return availableTitles.contains(title);
   }
}
```

Now that we have *Library* doing the first piece of work, we should wire it into the customer’s test so we can tick off that outcome.

```
public class DonateFixture {

   private Library library;
   private Title title;

   public void setDonor(String memberId){
      library = new Library();
      library.donate(title);
   }

   public void setTitle(String name){
      title = new Title();
   }

   public boolean libraryContains(){
      return library.contains(title);
   }
```

Notice this time, we're using a real instance of *Title*. As much as possible, customer tests should use the real objects doing the work, so we can better assure ourselves that our implementation works end-to-end. The exception to this is objects that have direct external dependencies, which will make our customer test fixtures more complex (e.g., requiring a database set-up) and slower-running.

Let's run our customer test to see if we can tick off the first outcome.



DonateFixture				
title	donor	libraryContains?	copyCount?	rewardP
The Abyss	joepeters	true	[0] expected [1]	[0] exp [10]

So far, so good. Let's move on to the next outcome: there should be one default loan copy added for the donated title. Who knows about loan copies? A quick glance at our tag cloud suggests that

Title might be the best name for the class that has this responsibility.

TEST-DRIVING ADDING A DEFAULT LOAN COPY TO THE TITLE

First, let's write a failing unit test for *Title*.

```
public class TitleTests {

   @Test
   public void defaultLoanCopyIsAdded(){
      Title title = new Title();
      title.addLoanCopy();
      assertEquals(1, title.getCopyCount());
   }
}
```

The simplest code to pass this test would just be to have *getCopyCount()* return 1. We could triangulate the code to add loan copies, but, in this instance, a general solution would be quite trivial and obvious, and triangulating would be a lot of effort for very little reward. So we're just going to write the *obvious implementation* to pass the test.

```
public class Title {

   private int loanCopies;

   public int getCopyCount() {
      return loanCopies;
   }

   public void addLoanCopy() {
      loanCopies++;
   }
}
```

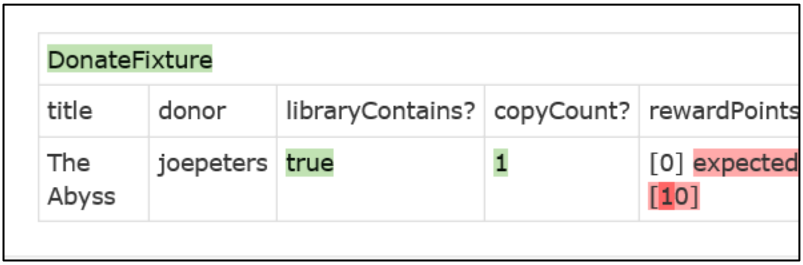
At this point, there's nothing in this code we might need to refactor, so we can move on and wire this piece of work into our customer test.

```
    public void setDonor(String memberId){
       library = new Library();
       library .donate(title);
    }

    public int copyCount(){
       return title.getCopyCount();
    }
```

To get this part of the customer's test passing, we just need *Library* to tell *Title* to add a loan copy.

```
    public void donate(Title title) {
        availableTitles.add(title);
        title.addLoanCopy();
    }
```

Now we can tick the second piece of work off in our customer test.



DonateFixture				
title	donor	libraryContains?	copyCount?	rewardPoints
The Abyss	joepeters	true	1	[0] expected [10]

Before we move on to the next outcome in our customer test, we should take review our code to see if it needs refactoring.

We notice that *Library* only uses the *addLoanCopy()* method of *Title*. Recall the design principle that classes should present client-specific interfaces, to limit coupling to only what they need to see.

Let's extract a new interface on *Title* for *Library* to use. Using our tag cloud for inspiration, we choose the name *Copyable*.

```
public class Title implements Copyable {

   private int loanCopies;

   public int getCopyCount() {
      return loanCopies;
   }

   @Override
   public void addLoanCopy() {
      loanCopies++;
   }
}
```

Now we can bind *Library* to this interface instead of directly to *Title*.

```
public class Library {

   private List<Copyable> availableTitles;

   public Library() {
      this.availableTitles = new ArrayList<>();
   }

   public void donate(Copyable title) {
      availableTitles.add(title);
      title.addLoanCopy();
   }

   public boolean contains(Copyable title) {
      return availableTitles.contains(title);
   }
}
```

TEST-DRIVING REWARD POINTS

Next, we turn our attention to the third outcome of the customer's test.

```
public class MemberTests {

   @Test
   public void rewardingMemberAddsPointsToTotal() {
      Member member = new Member();
      member.reward(10);
      assertEquals(10, member.getRewardPoints());
   }
}
```

Again, the implementation is obvious, so we'll just go straight for it rather than triangulating.

```
public class Member {

   private int rewardPoints;

   public void reward(int points) {
      this.rewardPoints += points;
   }

   public int getRewardPoints() {
      return rewardPoints;
   }
}
```

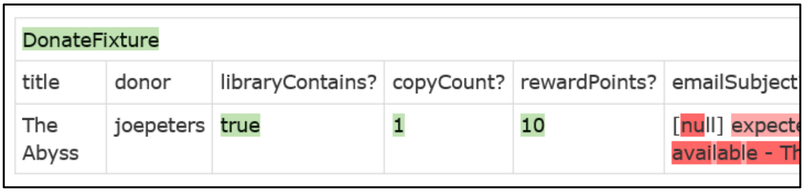
And now we can wire this into the customer's test.

```
   public void setDonor(String memberId){
      donor = new Member();
      library = new Library();
      library .donate(title, donor);
   }


   …

   public int rewardPoints(){
      return donor.getRewardPoints();
   }
```

Library will need to tell the donor (Member) to reward itself with 10 points to pass the customer's test.

```
    public void donate(Copyable title, Member donor) {
        availableTitles.add(title);
        title.addLoanCopy();
        donor.reward(10);
    }
```

Now we can run the FitNesse test to check the result.



DonateFixture					
title	donor	libraryContains?	copyCount?	rewardPoints?	emailSubject
The Abyss	joepeters	true	1	10	[null] expecte available - Th

We're nearly there, but before we move on to the email alert, there's a little bit of refactoring we need to do. *Library* only uses the *reward()* method of Member, so let's extract a client-specific interface for that.

```
public class Member implements Rewardable {

    private int rewardPoints;

    @Override
    public void reward(int points) {
        this.rewardPoints += points;
    }

    public int getRewardPoints() {
        return rewardPoints;
    }
}
```

Library now only needs to bind directly to the *Rewardable* interface.

```
    public void donate(Copyable title, Rewardable donor){
        availableTitles.add(title);
        title.addLoanCopy();
        donor.reward(10);
    }
```

As a final note before we move on to the email alert, we should update *LibraryTests* so that it, too, only binds to these interfaces.

This will decouple the tests as well as the implementation from details they don't need to know.

```
@Test
public void donatedTitlesAreAddedToAvailableTitles (){
    Library library = new Library();
    Copyable title = mock(Copyable.class);
    Rewardable donor = mock(Rewardable.class);
    library.donate(title, donor);
    assertTrue(library.contains(title));
}
```

TEST-DRIVING EMAIL ALERTS

The last piece of the jigsaw in our design for passing the customer's test is sending email alerts to members who expressed an interest in matching titles.

There are four elements to this, so we'll be doing it in four steps, getting feedback with each.

1. Formatting the subject line of the email
2. Sending the email
3. Formatting the body of the email
4. Selecting the recipients

The last part will involve an external dependency. The plan will be to do the first three pieces of work, and use a mock object to test-driven the client-side code for the fourth part, all using fast-running unit tests. Then we will test-drive an integration test for pushing the email alert onto a queue, to be picked up and processed by an external email server asynchronously.

FORMATTING THE SUBJECT LINE

In our design, we decide it makes most sense for an *EmailAlert* to be created, passing the new *Title* into the constructor.

```
public class EmailAlertTests {

   @Test
   public void subjectLineIncludesNewTitleName() {
      EmailAlert alert =
                  new EmailAlert("The Abyss”);
      assertEquals("Now available - The Abyss",
                                 alert.getSubject());
   }
}
```

Again, passing this test is trivial.

```
public class EmailAlert {

   private final String titleName;

   public EmailAlert(String titleName) {
      this.titleName = titleName;
   }

   public String getSubject() {
      return "Now available - " + titleName();
   }
}
```

Notice that this is the point where we implemented the *name* field of *Title*, because this is the first outcome where it’s actually used. We knew all along that this would be needed, but we only implemented it when a test required it.

In TDD, it’s highly recommended that you think ahead about the design, just as long as you don’t code ahead.

TEST-DRIVING SENDING THE EMAIL ALERT

Before we can wire our new functionality into the customer’s test, we’ll need to test-drive code to send the alert to an external message queue.

As we don’t want our FitNesse test to actually send an email, we’ll use a mock object for this external dependency, and test that a method is called to push the alert onto the queue.

```
   @Test
   public void sendingAlertPushesItOntoEmailQueue(){
      EmailQueue queue = mock(EmailQueue.class);
      EmailAlert alert =
                        new EmailAlert("The Abyss", queue);
      alert.send();
      verify(queue).send();
   }
```

Passing this test is easy.

```
   public void send() {
      queue.send(this);
   }
```

Before we move on, let's get refactor away the duplicate set-up code.

```
public class EmailAlertTests {

   private EmailAlert alert;
   private EmailQueue queue;

   @Before
   public void setupAlert() {
      queue = mock(EmailQueue.class);
      alert = new EmailAlert("The Abyss", queue);
   }

   @Test
   public void subjectLineIncludesNewTitleName() {
      assertEquals("Now available - The Abyss",
                                       alert.getSubject());
   }

   @Test
   public void sendingAlertPushesItOntoEmailQueue(){
      alert.send();
      verify(queue).send(alert);
   }
```

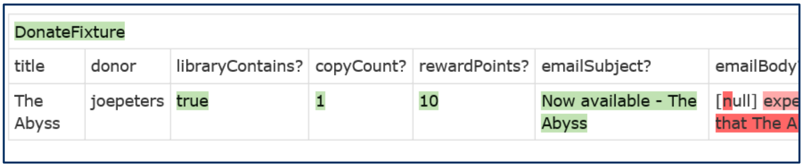
Now we can wire this into the customer test, and use a mock *EmailQueue* - at this point it's just an interface – to capture the *subject* value of the *EmailAlert*.

```
private EmailQueue queue;
private ArgumentCaptor<EmailAlert> alert;

…

public void setDonor(String memberId){
   donor = new Member();
   queue = mock(EmailQueue.class);
   alert = ArgumentCaptor.forClass(EmailAlert.class);
   library = new Library(queue);
   library.donate(title, donor);
   verify(queue).send(alert.capture());
}

…

public String emailSubject(){
   return alert.getValue().getSubject();
}
```

To pass the customer test, the *donate()* method of Library must create an *EmailAlert* and invoke *send()*.

```
public Library(EmailQueue queue) {
   this.queue = queue;
   this.availableTitles = new ArrayList<>();
}

public void donate(Copyable title, Rewardable donor){
   availableTitles.add(title);
   title.addLoanCopy();
   donor.reward(10);
   new EmailAlert(title.getName(), queue).send();
}
```

Now we can run the customer test.



DonateFixture						
title	donor	libraryContains?	copyCount?	rewardPoints?	emailSubject?	emailBody
The Abyss	joepeters	true	1	10	Now available - The Abyss	[null] expe that The A

FORMATTING THE EMAIL ALERT'S BODY

```
@Test
public void bodyIncludesNewTitleName() {
   String expectedText = "Dear member, " +
                  "just to let you know that " +
                  "The Abyss is now available to borrow";
   assertEquals(expectedText, alert.getBody());
}
```

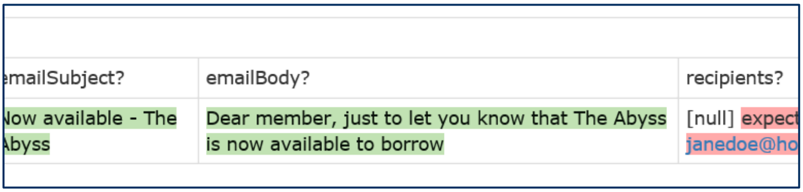
The implementation of this is trivial, too.

```
public String getBody() {
   return "Dear member, " +
         "just to let you know that " +
         titleName +
         " is now available to borrow";
}
```

Now we can write the code in our FitNesse fixture to make this part of the customer test pass.

```
public String emailBody(){
   return alert.getValue().getBody();
}
```

Let's run it and get some feedback.



emailSubject?	emailBody?	recipients?
Now available - The Abyss	Dear member, just to let you know that The Abyss is now available to borrow	[null] expect janedoe@ho

POPULATING THE RECIPIENTS LIST – FAKE IT 'TIL YOU MAKE IT

In order to build a list of recipients' email addresses, we will first need to identify which members have expressed an interest in this new title.

This opens a whole can of worms that we might not want to do deal with right now. So we're going to fake this functionality in order to keep progressing with our “Donate a movie that isn’t in the library” customer test.

All we need to know is which members expressed an interest, according to our customer’s test data. A stub that provides that list will enable us to move forward. We can revisit how matches are found with the customer later.

Also, this piece of behaviour is not as trivial as that needed to format the subject and the body of the alert, so we’re going to triangulate it, starting with the simplest test case we can think of.

```
@Test
public void whenNoMembersInterestedRecipientsIsEmpty(){
   InterestedMemberSearch search
        = new InterestedMemberSearchStub(new String[]{});
   EmailAlert emailAlert
        = new EmailAlert("X", null, search);
   assertEquals("", emailAlert .getRecipients());
}
```

InterestedMemberSearchStub implements an interface that *EmailAlert* will use to get a list of email addresses of members who expressed an interest in “X”.

We don’t know how it will select that list in the final implementation. We just need to know that it will.

Note how we’re passing the stub data in to the constructor, so it’s visible inside the test.

The simplest way to pass this test is:

```
   public String getRecipients() {
      return "";
   }
```

Let’s move on to another failing test.

```
@Test
public void whenOneMemberInterestedRecipientsIsSingleEmail(){
   InterestedMemberSearch search
               = new InterestedMemberSearchStub(
                                    new String[]{"x@y.com"});
   EmailAlert emailAlert =
                  new EmailAlert("X", null, search);
   assertEquals("x@y.com", emailAlert .getRecipients());
}
```

Which we can pass by doing:

```
   public String getRecipients() {
      String recipients = "";
      String[] memberEmails = search.byTitle(titleName);
      for (int i = 0; i < memberEmails.length; i++) {
         recipients += memberEmails[i];
      }
      return recipients;
   }
```

Next, let’s refactor the test code to consolidate these two very similar tests into a single JUnitParams parameterised test.

```
   @Test
   @Parameters(method="recipientsParams")
   public void recipientsListIsInterestedMemberEmails(
                              String[] memberEmails,
                              String recipients) {
      InterestedMemberSearch search
               = new InterestedMemberSearchStub(memberEmails);
      EmailAlert emailAlert = new EmailAlert("X", null, search);
      assertEquals(recipients, emailAlert .getRecipients());
   }

   private Object[] recipientsParams(){
      return new Object[][]{
            {new String[]{}, ""},
            {new String[]{"x@y.com"}, "x@y.com"}
         };
   }
```

Lastly, we need to handle lists of more than one interested member, adding comma separation.

```
private Object[] recipientsParams(){
   return new Object[][]{
      {new String[]{}, ""},
      {new String[]{"x@y.com"}, "x@y.com"},
      {new String[]{"x@y.com", "a@b.com"},
                                    "x@y.com, a@b.com"}
   };
}
```

Which we can pass by doing:

```
public String getRecipients() {
   String recipients = "";
   String[] memberEmails = search.byTitle(titleName);
   for (int i = 0; i < memberEmails.length; i++) {
      recipients += memberEmails[i];
      if(i < memberEmails.length - 1){
         recipients += ", ";
      }
   }
   return recipients;
}
```

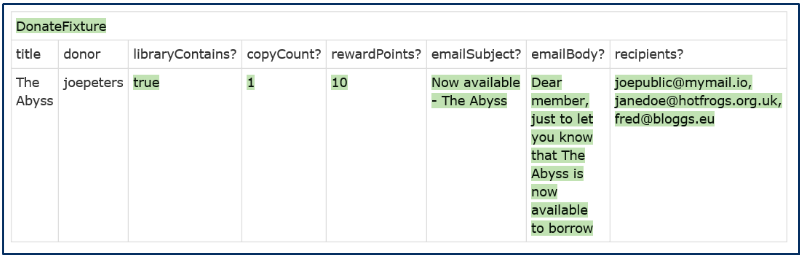
This is pretty old-fashioned imperative code for Java 8. Let's refactor it to make it declarative and less prone to breaking if it has to change.

```
public String getRecipients() {
   String[] memberEmails = search.byTitle(titleName);
   if(memberEmails.length > 0)
      return Arrays.asList(memberEmails)
            .stream()
            .reduce((current, next) -> current + ", " + next)
            .get();
   return "";
}
```

This should handle any valid list of member emails. Now let's wire it into the FitNesse test.

```
private InterestedMemberSearch search;

public DonateFixture(){
   search = new InterestedMemberSearchStub(
         new String[]{
               "joepublic@mymail.io",
               "janedoe@hotfrogs.org.uk",
               "fred@bloggs.eu"
               }
         );
}

…

public void setDonor(Strting memberId){
   donor = new Member();
   queue = mock(EmailQueue.class);
   alert = ArgumentCaptor.forClass(EmailAlert.class);
   library = new Library(queue, search);
   library .donate(title, donor);
   verify(queue).send(alert.capture());
}

…

public String recipients(){
   return alert.getValue().getRecipients();
}
```

Let's run the test to get our final piece of feedback.



DonateFixture							
title	donor	libraryContains?	copyCount?	rewardPoints?	emailSubject?	emailBody?	recipients?
The Abyss	joepeters	true	1	10	Now available - The Abyss	Dear member, just to let you know that The Abyss is now available to borrow	joepublic@mymail.io, janedoe@hotfrogs.org.uk, fred@bloggs.eu

THE "LONDON SCHOOL" OF TDD

In our worked example, we mostly wrote unit tests in the familiar style using assertions to check that the work got done. We only wired the objects doing the work in to pass our failing customer test.

An increasingly popular alternative approach is to focus our unit tests on the wiring itself, allowing us to test-drive the interactions, and therefore the design of interfaces to support those interactions, directly.

We use mock objects to write tests that fail until an interaction between two objects occurs – indeed, that is the purpose for which mock objects were invented.

Working again from the outside in, we start with failing tests for an implementation of *Library*, and mock its collaborators.

```java
public class LibraryTests {

   @Test
   public void tellsDonatedTitleToAddLoanCopy(){
      Copyable title = mock(Copyable.class);
      Rewardable donor = mock(Rewardable.class); // dummy
      Library library = new Library();
      library.donate(title, donor);
      verify(title).addLoanCopy();
   }
}
```

In this failing test, we define that *Library* will communicate with the donated title through a *Copyable* interface, which has an *addLoanCopy()* method.

We don’t test that the donated title is added to the library here. Instead, we test that the library invokes the *addLoanCopy* method. The FitNesse test checks that the title was added to the library.

Once we have Library working to our satisfaction, we move inwards to test-driving implementations of its collaborators, mocking their collaborators, until our design’s working end to end, and we’re able to pass the customer test completely.

```java
   public boolean libraryContains(){
      return library.contains(title);
   }
```

This is a reversal of the way we did things before, where we test-driven the work the objects do with unit tests and wire them together to pass the customer test. In this style of TDD – commonly referred to as the "London school", because that's where it originated from practitioners like Steve Freeman and Nat Pryce, who wrote the book *Growing Object Oriented Software Guided By Tests* – we drive the wiring using unit tests and let the customer test check the work got done.

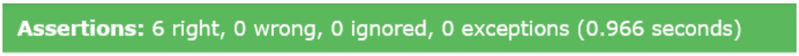
There are advantages and disadvantages to both approaches. The first approach we saw duplicates testing of the work objects do, but has the advantage of putting the tests closer to the modules being tested, which can make debugging easier when tests fail, and makes the tests themselves more portable. Also, an over-reliance on mock objects can lead to issues with maintainability of the design emerging is not modular enough.

The London School places greater emphasis on the object oriented design and encourages a 'Tell, Don't Ask' style of design, where we focus on roles, responsibilities and especially collaborations between objects.

On balance, we find that either approach can be successful. It's therefore a question of trying both and seeing which feels right for you. Be sure, though: both approaches require considerable practice to master.

MAKING CUSTOMER TESTS RUN FASTER

By excluding external dependencies like the email message queue, we can write a FitNesse test that runs faster. But it still has to read test data values from a Wiki file, and reading things from files slows execution down very significantly.

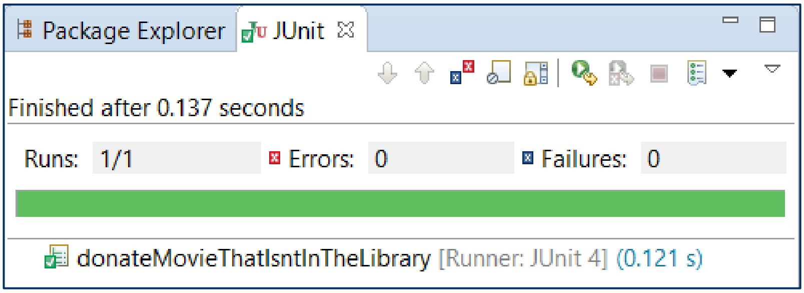

Assertions: 6 right, 0 wrong, 0 ignored, 0 exceptions (0.966 seconds)

Our customer test for donating a DVD takes almost a whole second to run. If we have a couple of hundred such customer tests, we'd have to wait 3 minutes to run the suite. In TDD, we seek the shortest feedback loops possible, so can we speed this up?

One way to achieve that would be to adapt the test's Java fixture so it can also be run as a JUnit test.

```
@Test
public void donateMovieThatIsntInTheLibrary(){
   setTitle("The Abyss");
   setDonor("joepeters");
   assertTrue(libraryContains());
   assertEquals(1, copyCount());
   assertEquals(10, rewardPoints());
   assertEquals("Now available - The Abyss", emailSubject());
   assertEquals("Dear member, just to let you know that " +
                     "The Abyss is now available to borrow",
                emailBody());
   assertEquals("joepublic@mymail.io, " +
                "janedoe@hotfrogs.org.uk, " +
                "fred@bloggs.eu",
                recipients());
   }
```

This test takes a fraction of the time to run.



Finished after 0.137 seconds

Runs: 1/1 Errors: 0 Failures: 0

donateMovieThatIsntInTheLibrary [Runner: JUnit 4] (0.121 s)

Be careful, though. We copied and pasted the customer's data directly from the FitNesse Wiki page, and if that data changes, our JUnit version could get out of step. We'll need to re-run the FitNesse test every time the Wiki page is edited.

ARE WE "DONE" YET?

Our implementation passes the customer test we wrote in FitNesse, but is this actually a working feature?

Email alerts aren't actually sent to recipients, and the list of email addresses of members who expressed an interest in the title is faked.

On top of that, how will people be able to use this functionality? There's no user interface (unless they know Java, of course!) And the minute the program stops running, all of our data will be lost.

This is quite some way from being a working feature that end users could benefit from.

But our customer test has helped us flesh out an internal design on which these extra elements can now be built. We have placeholders, in the shape of interfaces, for sending emails to the queue, and for retrieving interested members' email addresses.

The next step would be to test-drive implementations for these interfaces using integration tests. (See the chapter "Test-Driving Integration Code".)

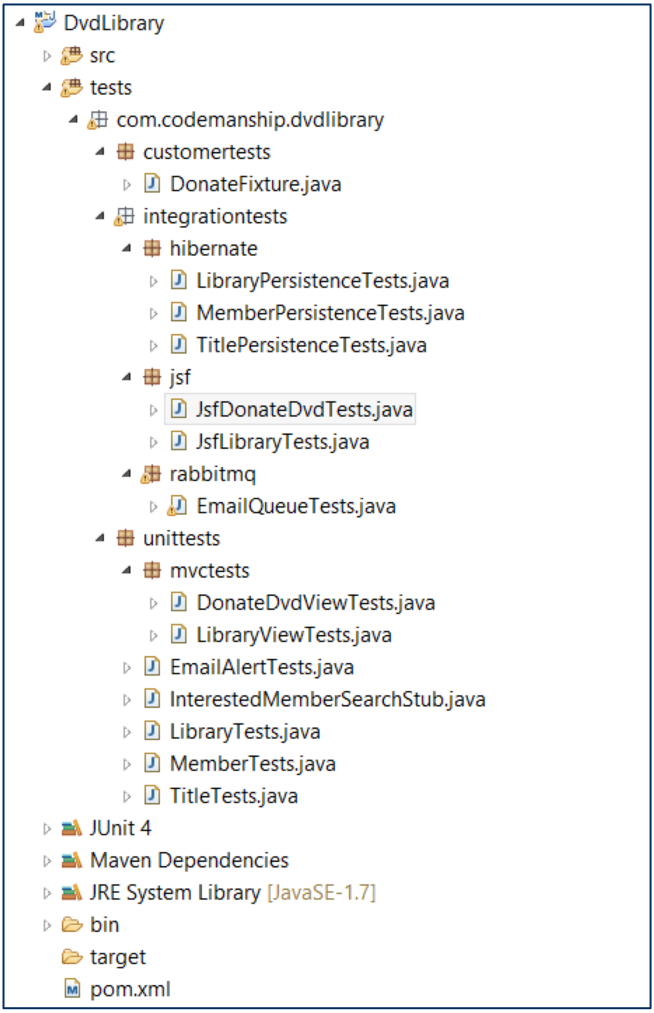
EXERCISE #15

Following a similar process to that illustrated in this chapter, pick one of your customer's happy path tests, and implement an internal design that passes that test. Use test doubles for any external dependencies or components you want to defer thinking about for now ("fake it 'til you make it").

When you've passed that customer test, adapt it to run as a faster unit test. Then pick another and implement that. After you've implemented the code to pass 2-3 customer tests, you will get a feel for how the process works.

18. THE TESTING PYRAMID

In our previous two chapters, we walked through a process for driving the design of some software directly from an executable specification written with the customer.



```
DvdLibrary
  src
  tests
    com.codemanship.dvdlibrary
      customertests
        DonateFixture.java
      integrationtests
        hibernate
          LibraryPersistenceTests.java
          MemberPersistenceTests.java
          TitlePersistenceTests.java
        jsf
          JsfDonateDvdTests.java
          JsfLibraryTests.java
        rabbitmq
          EmailQueueTests.java
      unittests
        mvctests
          DonateDvdViewTests.java
          LibraryViewTests.java
        EmailAlertTests.java
        InterestedMemberSearchStub.java
        LibraryTests.java
        MemberTests.java
        TitleTests.java
  JUnit 4
  Maven Dependencies
  JRE System Library [JavaSE-1.7]
  bin
  target
  pom.xml
```

To deliver a working piece of software, we may end up with several different kinds of tests. To allow us to run the different kinds of tests separately (or all together as a single suite), we package them separately, too.

Our tests can be roughly split into three categories here:

1. Customer tests

2. Integration tests
3. Unit tests

We may also wish to consider – probably in a separate project and using a different toolset – writing a *system* test that checks that the whole thing hangs together correctly end-to-end, including the Java Server Faces UI, the core logic we created to pass the FitNesse test, the email message queue, the email server that reads messages from that queue and sends the emails, and a back-end database mapped on to our core objects using Hibernate.

The question arises: how much of each kind of testing do we need?

Some development teams automate all the checking of their application's core logic using tests that drive the user interface (e.g., driving a website using Selenium). We find that this produces test suites that run very slowly. It's not unheard of for teams to have to wait an hour or more to find out if a change they made has broken any of the code.

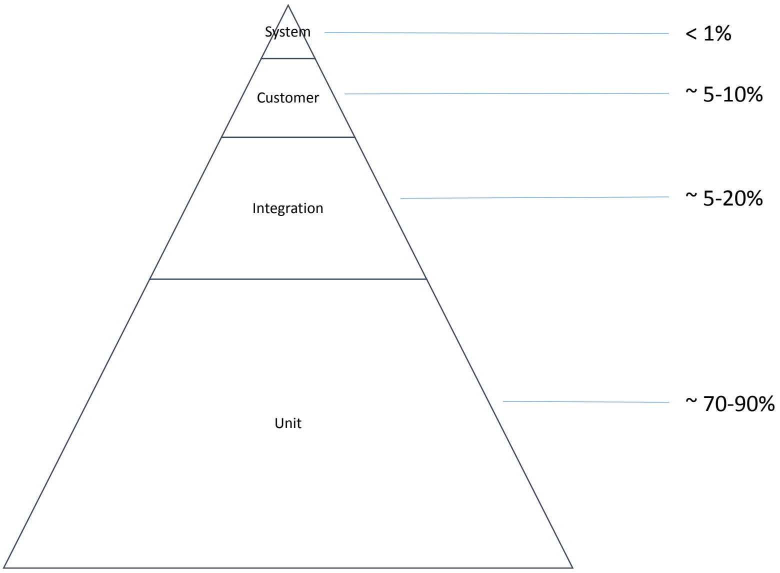
Some development teams automate all the checking of their core logic using a tool like FitNesse, bypassing the UI and driving the application through controllers or services, and excluding external dependencies using test doubles. While this can produce test suites that run much faster than UI tests, they still can take far too long to enable the rapid, frequent feedback we're going to need.

A majority of development teams have learned to check as much of the logic of their application as possible using fast-running unit tests.

There's still a need to test the integration with external dependencies, and still a need to test that the software does exactly what the customer asked for. There's also a need to test that, when all the pieces are wired together, the system as a whole works.

The trick is to recognise the purpose of each kind of testing, understand the risks that each level of testing can address, and then apportion the right amount of test automation to each.

Typically, what teams end up with is a “testing pyramid”.



Although it can vary depending on the kind of application and on the technology used, we would expect the great majority of automated tests to be unit tests. A 100,000 line application may have 10,000 unit tests (yes, really that many!), 1,000 integration tests, a few hundred customer tests, and maybe a few dozen system-level “smoke tests” to make sure it all works when it’s plugged together.

We would rely most of the time on the unit tests to catch bugs we may have introduced, running them many times in an hour.

We would run our integration tests whenever we’ve changed integration code, which might apply to 5-20% of the changes we make in total.

We’d run our customer tests frequently, for which we might have adapted them to run faster as unit tests like we did in the previous chapter. But, as we saw, for every time we ran the customer test, we ran our unit tests multiple times, getting feedback at a lower level until we were ready to wire the new code into the customer test.

We might run our system-level tests before committing our changes to a shared repository like Git or Subversion, just to make sure that there aren't any configuration problems we've missed.

If we've divided up the testing work effectively, then our entire suite of tests – including system tests – might run in just a few minutes, so we could potentially run them all as part of a build.

EXERCISE #16

Complete the features you worked on in the previous exercises, writing integration tests for any external dependencies it requires (e.g., storing object data), and adding a simple user interface on your preferred technology stack (e.g., a web interface, or a desktop GUI).

Organise your tests into the categories: Customer, Integration and Unit. Count the number of each kind of test, and draw your own testing pyramid showing how they're split as a percentage. Time how long it takes to run each set of tests, and also how long to run the entire suite of tests.

19. TDD & CONTINUOUS INTEGRATION

Summary:

- The automated tests we create doing TDD can give us confidence that our software is always working
- Before committing code changes to a shared repository, merge other people's changes into your local copy and make sure it passes all the tests
- Run the tests as part of the build to catch problems caused by local configurations
- Use a "build token" to prevent issues caused by developers committing conflicting changes on top of each other
- Optimise test suite execution times to speed up builds
- Create a build tree for large systems, where every component has its own build process, and dependencies are drawn from the outputs of successful sub-builds
- Continuous Delivery is enabled by TDD
- Avoid feature branching as a strategy for hiding unfinished features from end users, because you will lose the benefits of Continuous Integration
- Use feature toggles to hide unfinished features until they're ready

Continuous Integration is a popular practice on software development teams where multiple programmers working on different parts of the software frequently commit their changes to a shared repository.

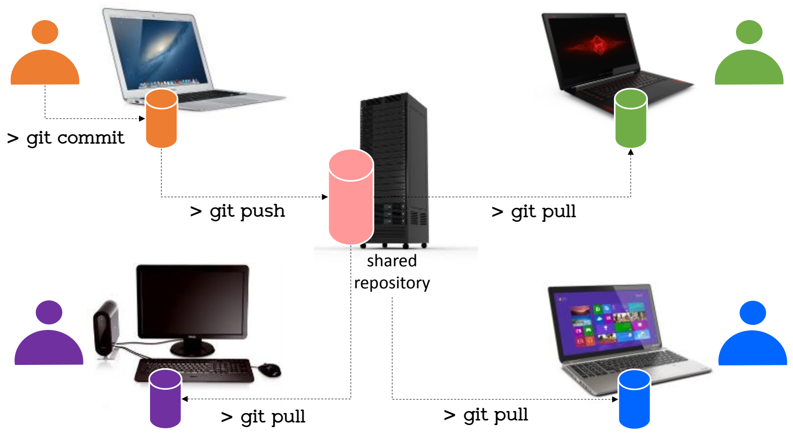
To eliminate the possibility that the code only works on the developer's own computer, a Continuous Integration (or "build") server will build the code from the shared repository on another machine every time someone commits changes.

They do this for several reasons:

1. To check that their changes work with everyone else's changes
2. To check that their changes work on a different machine
3. To communicate their changes to the other developers
4. To make their changes available to the customer as soon as possible

TDD and Continuous Integration work well together. The suite of automated tests it produces can give us higher confidence that the code does indeed work and the changes we've made haven't broken it.

BEFORE WE COMMIT: UPDATE/MERGE & TEST LOCALLY



A pre-condition to committing changes to code is to check that our changes work with other people's changes. It's vital, therefore, to make sure that the local copy you're working on is up to date with other people's changes.

Whenever we plan to commit our changes, we should first get an update from the shared repository that includes any changes that have been made since our last update.

We merge those changes into our local copy of the code, resolving any conflicts that may have arisen from edits to the same files that we've changed.

Our local copy, merged with other people's changes, now represents what the code in the repository will look like after we commit.

Before we commit, we need to check that it works on our computer. We do this by building it locally and running our automated tests.

Only if all the tests pass should we consider committing our changes. If any tests fail, we must fix the problems, and repeat this process when we think we're ready.

AFTER WE COMMIT: WAIT FOR THE TESTS TO PASS ON A BUILD SERVER

It's possible that the tests all passed on your local machine because of some quirk of that machine's configuration. (For example, there may be a version of a library installed on your machine that other machines don't have.)

To help us eliminate this possibility, we wait to see that the code builds and passes the tests on a different machine, often referred to as a "build server".

Most teams have a CI server like CruiseControl (cruisecontrol.sourceforge.net) or Jenkins (jenkins.io) that "listens" for new commits and triggers a build – complete with tests – automatically.

Until the build succeeds on the build server, we don't know for sure that we haven't broken the software. It's highly advisable that nobody else commits changes until the build has succeeded. Most CI servers will notify the team of the outcome.

USE A “BUILD TOKEN” TO PREVENT OVERLAPPING COMMITS

On development teams, there’s a real risk of two programmers committing conflicting changes on top of each other. Imagine Dave plans to commit his changes, so he gets an update from the repository and runs his tests locally. While he’s doing that, Sofia commits her changes, one of which conflicts with Dave’s. Dave’s tests all pass, so he commits his changes, and “breaks the build”.

The most disciplined practitioners of Continuous Integration mitigate this risk by having a system that “locks” the repository while one person is committing their changes.

A low-tech approach is to have an easily identifiable “build token”; an object that developers need to grab in order to commit. They take the token when they’re about to update, and don’t return it until their build has succeeded on the CI server.

One team I know uses a felt beef burger as their build token. If a team member plans to commit, they take the burger to their desk.

They only return it when their build has succeeded on the CI server. Taking too long over commits has become known as "hogging the burger".

MAKING BUILDS FAST

Some teams have automated tests suites that take a long time to run. This slows Continuous Integration down, sometimes to a point where there's really nothing "continuous" about it.

A build – complete with automated testing – needs to take a few minutes at the most. Firstly, this means we need to put significant effort into optimising our test suites. If most of our tests drive the software through the user interface, and/or include external dependencies like reading and writing files, or using web services, then it's not uncommon for teams to have to wait more than an hour to get feedback from the build server. This is an hour when it's not safe for anyone else to commit their changes.

Slow builds block teams.

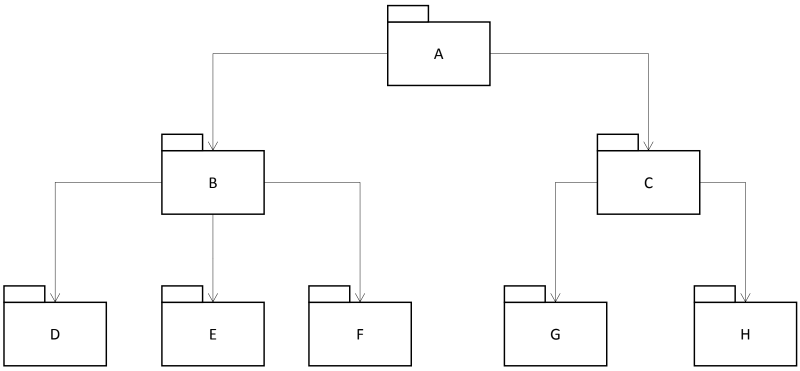
Aim for a pyramid of tests: with mostly unit tests, fewer integration tests, and just a handful of system tests. Some teams exclude the bulk of their slow-running tests from the build process, falling back on a handful of "smoke tests" that might catch any obvious configuration problems. But this increases the risk of the build testing missing more subtle conflicts in the logic. The more of your tests you can run in the build, the lower that risk.

Instead of excluding slow-running tests, explore how they can be speeded up – e.g., by caching data read from files, using in-memory databases, or reusing datasets once they're loaded so there's less need to set up multiple test databases, and so on.

BUILD TREES

On large software systems (millions of lines of code), building and testing the whole thing – even when we've optimised those tests – can still take a long time.

Look to break down the system architecture into smaller, loosely coupled sub-systems or components, each of which can be built and tested separately.

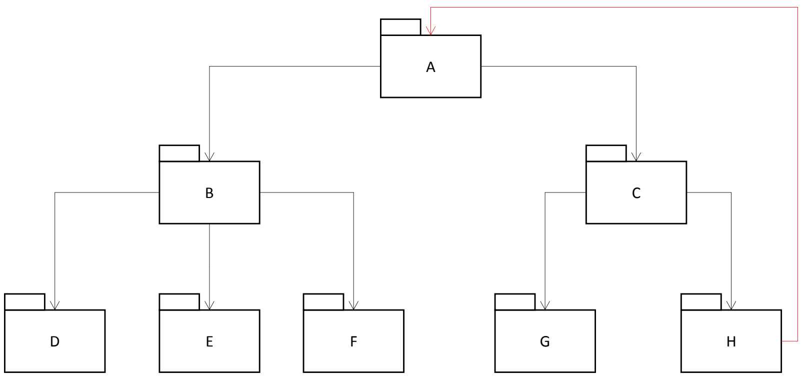


In this theoretical example, every component has its own build process, which includes running its own suite of tests.

We can build and test D, E, and F by themselves. To build and test B, we just use the outputs of the most recent successful builds of its dependencies D, E and F. Likewise, we can built and test C using the outputs of the last successful builds of G and H.

And to build and test component A, we use the outputs of the most recent successful builds of B and C.

Take extra care to avoid introducing cycles into the component architecture, where one component becomes indirectly dependent on itself.



In order to build and test A, we'll have to build and test all of the other components in the cycle (C & H).

TDD & CONTINUOUS DELIVERY

If we have good tests, and fast builds, we dramatically increase the confidence our customer can have that – at any given time - there is a working version of the software that could potentially be put to use.

At any point in time, the software is potentially *shippable*. We call this "Continuous Delivery". It puts control into the hands of the customer. If they need the new software right now, it's ready to go.

This can have a profound effect on the economics of making software. The typical experience of a customer, when they request a new feature or a change to the software, is having to wait months for the next big release until they can use it and get value from that investment.

With Continuous Delivery, they can have it as soon as it's ready. And if we're doing TDD well, then they can have high confidence that when we say it's ready, it *really* is ready.

It should come as no surprise that there are no development teams doing Continuous Delivery who aren't doing TDD. TDD can enable continuously shippable software.

FEATURE BRANCHING & FEATURE TOGGLES

If we're committing changes to our code many times a day, there's a danger we may end up releasing a half-finished feature. End users should not have access to unfinished features, for the same reason that motorists should not be given access to unfinished bridges.

Teams employ two approaches to hiding unfinished work from the users.

Feature branching has become quite popular, and involves developers creating their own branch of the master repository into which they commit their code for the feature they're working on. They continuously build and test their branch (each branch has its own build process), but don't merge their changes into the master until the feature is ready to be released.

Strictly speaking, this isn't Continuous Integration. My changes would not be made visible to my team mates until I did that final merge into master. This can store up some nasty surprises when that time comes, and rather defeats the object of doing CI. For this reason, many experienced practitioners advise against feature branching.

A better approach used by some teams is to employ what they call "feature toggles" in the user interface. Everyone commits their changes into master – so it's genuine Continuous Integration – but the button, or link, or menu item users would use to invoke the new feature is hidden until it's ready.

The easiest way to achieve this is to comment out the line of code that displays the user control until the feature's working. A more sophisticated approach would be to use a configuration file that can dynamically toggle features on and off. (This can have other useful applications, like allowing users or administrators to customise an application for different groups of users.)

Feature toggles that are configurable without having to recompile the code have the advantage of allowing us to deploy a version of the software that exposes the new feature for user testing, while keeping it hidden in real deployments.

EXERCISE #16

If you haven't already, commit the code you created in the previous two chapters into a shared online repository like GitHub.

Set up a Continuous Integration server – for example, using Jenkins or CruiseControl – that will build the code and run all the automated tests whenever changes are committed to that shared repository.

Working with your customer, dream up one or two new user stories that you believe will add value for any end users of the software.

Ask a friend or colleague – or your "customer", if she is a developer too – to share the work of implementing these new stories. Apply the ideas covered in this chapter while you work together on different parts of the code.

Use a feature toggle to hide the new functionality until it's ready to go.

20. TDD & LEGACY CODE

Summary:

- Legacy code is code for which we have no automated tests
- The "Catch 22" with legacy code is that we need to refactor to make automating tests easy, but it's not safe to refactor without automated tests
- Start by identifying the "change point": the part of the code that will need to change to accommodate a new requirement
- Then identify "inflection points": parts of the software that directly depend on the change point, where – if we broke the code – it would show
- Introduce tests around the inflection points. These could be unit tests, integration tests, system tests, or even manual tests
- Refactor the code to break the external dependencies preventing us from making our tests fast-running
- Keep running your inflection point tests after every refactoring – no matter how long this takes. After refactoring, it will get easier
- Be a good "boy scout", and leave code you work on in better order than you found it to make the going easier in future

Most books about software development focus on new code. (And this one, so far, has been no exception.)

But the reality for most developers is that, most of the time, we're trying to add features or make changes to existing software.

In a straw poll, 43% of developers who found they were prevented from doing TDD cited legacy code as the reason.

WHAT MAKES CODE "LEGACY"?

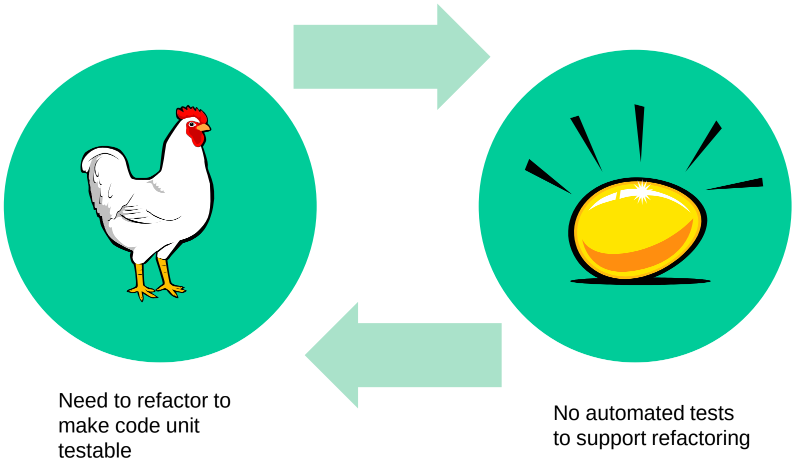
The impact of automated tests on the cost of changing software can be profound. In his book *Working Effectively With Legacy Code*, Michael Feathers defines "legacy code" as code for which we have no automated tests.

The risk when changing code is that we'll break the software, and having good automated tests can help us to catch those *regressions* sooner, when they're much easier and cheaper to fix.

When we inherit legacy code, and are asked to make changes to it by our customer, we'll probably want to start by writing a failing test.

Typically, what prevents us from doing this is dependencies in the existing code; talking to databases, reading files, and so on.

To write a good fast-running automated test, we'll need to refactor the existing code to make those external dependencies swappable, so we can mock or stub them.


Need to refactor to make code unit testable
No automated tests to support refactoring

But we can't *safely* refactor the code because there are no tests. It's a chicken-and-egg situation.

START BY IDENTIFYING THE CHANGE POINT(S)

Imagine we inherit the code for a web-based video-on-demand system. Rental prices are currently calculated based on information about the video title. Recent releases command a $1 premium, and an extra dollar is charged for especially good movies with an IMDb rating greater than 8.0.

```
public class Pricer {

  public float calculatePrice(String imdbID){
    Video video = ImdbService.fetchInfo(imdbID);
    float price = 2.95f; // default rental price
    // recent releases command premium
    if(video.getYear() == currentYear()){
      price += 1.0f;
    }
    // best films command premium
    if(video.getRating() > 8.0f){
      price += 1.0f;
    }
    return price;
  }

  int currentYear() {
    return Calendar.getInstance().get(Calendar.YEAR);
  }
}
```

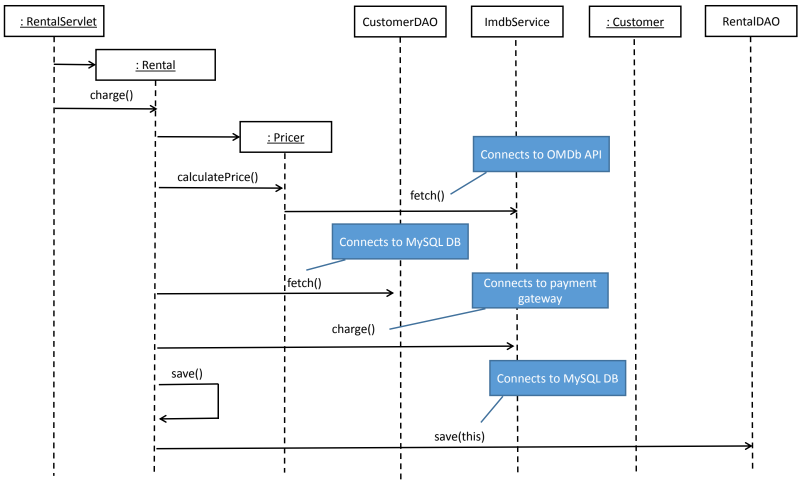
Our customer also wishes to change the pricing logic so that movies with IMDb ratings less than 4.0 get a $1 discount (they call it their "bargain bin" price).

The *calculatePrice()* method of *Pricer* is our "change point" – the part of the software we'll need to change to accommodate the customer's new requirement.

NEXT, IDENTIFY INFLECTION POINTS

Then we ask ourselves "if we changed *calculatePrice()*, what modules that depend on that could be broken?"

To make it safe to change *calculatePrice()*, we'll want to write automated tests for the modules that could be broken by that change.



In his book, Michael Feathers calls these “inflection points” (some people call them “test points”). An inflection point is the position in the call stack where, if the change has broken the software, it will be immediately evident. They’re where we’ll need to write some automated tests before we make the change.

In our case, changing *calculatePrice()* could break the *charge()* method of the Rental class that uses the Pricer, so we’ll be looking to write automated tests for Rental.

INTRODUCE TESTS. ANY KIND OF TESTS.

Remember, at this point our code has no automated tests. Unfortunately, our architecture makes fast-running unit tests impossible without refactoring because of the external dependencies.

The temptation here is to skip the testing part and just start hacking away at the code. But refactoring without tests is dangerous. So *resist that temptation*. It may be hard work at first, but potentially a walk in the park compared to how hard it might later be if we break the code without realising it.

In this instance, we could probably write some integration tests against a test MySQL database, and use the real IMDb API, as it's read-only and free.

Worst case, if even automated integration or system tests are not currently possible, we would need to write some manual test scripts, run the software and verify it ourselves after each refactoring. That takes considerable discipline and patience, but is usually worth the effort in the long run.

BREAK THE EXTERNAL DEPENDENCIES

Once we have some tests around the inflection points, the next priority is to turn those into fast-running unit tests to make the going easier.

Our goal is to test Rental without hitting the database or the IMDb API, which means we need to make the classes that contain those direct dependencies swappable.

Let's start with the IMDb dependency, which exists in a static method *fetch()* on the *ImdbService* class.

```
public class Pricer {

  public float calculatePrice(String imdbID){
    Video video = ImdbService.fetchInfo(imdbID);
```

First, let's turn that non-swappable static method into an instance method.

```
public class Pricer {

  public float calculatePrice(String imdbID){
    Video video = new ImdbService().fetchInfo(imdbID);
```

Now, we should run our inflection point tests. (Yes, even for a change this small. Even if they're manual tests.)

Next, let's use dependency injection to make *ImdbService* swappable via a *MovieInfo* interface that it implements.

```
public class Pricer {

   private final MovieInfo imdb;

   public Pricer(MovieInfo imdb) {
      this.imdb = imdb;
   }

   public float calculatePrice(String imdbID){
      Video video = imdb.fetchInfo(imdbID);
```

And now we should run our tests again.

Next, we need to inject an instance of *Pricer* into *Rental*, so we can wire in the *ImdbService* object from the outside in our tests. Again, we want *Pricer* to be easily swappable, so we need to extract an interface for *Rental* to use. But, in this case, *Pricer* would actually be the best name for that interface. So, let's rename the class to make it more specific.

```
public class ImdbPricer implements Pricer {
```

Which we inject into Rental.

```
public class Rental {

   private final String imdbID;
   private final long customerID;
   private float amountCharged;
   private final Pricer pricer;

   public Rental(String imdbID,
                 long customerID,
                 Pricer pricer){
      this.imdbID = imdbID;
      this.customerID = customerID;
      this.pricer = pricer;
   }

   public void charge(){
      Customer customer = CustomerDAO.fetch(customerID);
      amountCharged = pricer.calculatePrice(imdbID);
      customer.charge(amountCharged);
      save();
   }
```

And... run the tests!

Then we can turn our attention to the classes that access the MySQL database, starting with *CustomerDAO*.

```
   public void charge(){
      Customer customer = CustomerDAO.fetch(customerID);
      amountCharged = pricer.calculatePrice(imdbID);
      customer.charge(amountCharged);
      save();
   }
```

The process is quite similar; make static methods instance methods.

```
   public void charge(){
      Customer customer =
                   new CustomerDAO().fetch(customerID);
      amountCharged = pricer.calculatePrice(imdbID);
      customer.charge(amountCharged);
      save();
   }
```

And then inject the dependency in through the constructor, so that it can be wired in – and substituted when necessary – from the outside in our test code.

```
private final DAO customerDAO;

public Rental(String imdbID, long customerID,
              Pricer pricer, DAO customerDAO){
    this.imdbID = imdbID;
    this.customerID = customerID;
    this.pricer = pricer;
    this.customerDAO = customerDAO;
}

public void charge(){
    Customer customer = customerDAO.fetch(customerID);
    amountCharged = pricer.calculatePrice(imdbID);
    customer.charge(amountCharged);
    save();
}
```

Rinse and repeat for *RentalDAO,* which can implement the same *DAO* interface as *CustomerDAO*.

```
   private final DAO rentalDAO;

   public Rental(String imdbID,
                 long customerID,
                 Pricer pricer,
                 DAO customerDAO,
                 DAO rentalDAO){
      this.imdbID = imdbID;
      this.customerID = customerID;
      this.pricer = pricer;
      this.customerDAO = customerDAO;
      this.rentalDAO = rentalDAO;
   }

…

   private void save() {
      rentalDAO.save(this);
   }
}

…

public interface DAO {
   Entity fetch(long entityId);
   void save(Entity entity);
}
```

We can now re-write our slow-running integration test to use test doubles for these external dependencies, making it easier and safer to test-drive the changes to the pricing logic the customer wants.

Or can we? There's a dependency we missed that might make automated tests problematic. Some of the pricing logic is date-driven. We might write a test for a movie that, at the time we wrote it, was a brand new release. What would happen if we ran that test years later?

To make the tests totally repeatable and predictable, we need to make the current year swappable, too.

```
public class Pricer {

   private final MovieInfo imdb;
   private final CurrentYear currentYear;

   public Pricer(MovieInfo imdb,
                 CurrentYear currentYear) {
      this.imdb = imdb;
      this.currentYear = currentYear;
   }

   public float calculatePrice(String imdbID){
      Video video = imdb.fetchInfo(imdbID);
      float price = 2.95f; // default rental price
      if(video.getYear() == currentYear.get()){
         price += 1.0f;
      }
```

Don’t forget to run the inflection points tests after *every* refactoring. It may seem like a total drag, but an even bigger drag is debugging broken code without fast-running automated tests to support us.

Now we can write fast-running, repeatable tests.

```
public class RentalTests {

   @Test
   public void videosRatedLessThanFourOnImdbGetDollarOff() {
      String imdbID = "tt2975590";
      int customerID = 999;
      Video video =
            new Video(imdbID,
                      "Batman vs Superman",
                      2016,
                      3.9f);
      Pricer pricer = new Pricer(createImdbStub(video),
                                 createYearStub(2016));
      Rental rental =
            new Rental(imdbID,
                       customerID,
                       pricer ,
                       createCustomerDAOStub(customerID),
                       createRentalDAODummy());
      rental.charge();
      assertEquals(2.95,rental.getAmountCharged(),0.01);
   }
```

Good scouts leave their campsites tidier than they found them. Good developers leave code they change in better order than they found it, too, to make future changes easier.

Notice there’s a fair amount of set-up code hidden away behind helper methods in our test code. This is indicative of dependency problems. Rental knows too much about other objects in the system – it has too many collaborators.

A cleaner version might remove all the data access dependencies from *Rental* and *Pricer*, so that they just handle the logic, and we can handle fetching and saving data outside of that as a separate concern (with its own set of tests.)

With fast-running tests to support us, this is a refactoring that will be much easier and safer to do.

```
public class Rental {

   private float amountCharged;
   private final Video video;
   private final Customer customer;
   private final Pricer pricer;

   public Rental(Video video, Customer customer, Pricer pricer){
      this.video = video;
      this.customer = customer;
      this.pricer = pricer;
   }

   public void charge(){
      amountCharged = pricer.calculatePrice(video);
      customer.charge(amountCharged);
   }

   public float getAmountCharged() {
      return amountCharged;
   }
}
```

The refactored test tells a story of a much simplified design.

```
@Test
public void videosRatedLessThanFourOnImdbGetDollarOff() {
   String imdbID = "tt2975590";
   Video video = new Video(imdbID,
                           "Batman vs Superman",
                           2016,
                           3.9f);
   Customer customer = mock(Customer.class);
   Rental rental = new Rental(video,
                              customer,
                              new Pricer(
                                   createYearStub(2016))
                              );
   rental.charge();
   assertEquals(2.95,rental.getAmountCharged(),0.01);
}
```

EXERCISE #17

Find some legacy code that you could add value to. It could be code you've worked on, or code from another project – maybe an Open Source project – that uses technology you're familiar with.

Build and run the software, and familiarise yourself with its features.

Think about 1-3 small ways in which the software could be improved by adding or changing features.

Apply the ideas covered in this chapter to test-drive the addition of those changes, paying special attention to the discipline of re-running the inflection point tests after ever refactoring.

21. BEYOND TEST-DRIVEN DEVELOPMENT

One of the things that makes TDD such a useful approach to software design is that, aside from the benefits we've already looked at, it can be a jumping off point for more advanced kinds of software testing.

DATA-DRIVEN & PROPERTY-BASED TESTS

Throughout this book, I've encouraged you to refactor duplication in your test code by consolidating multiple similar test cases into a single parameterised test.

Not only is this a great way to remove test code duplication, helping us to write tests that read more like a specification in the process, we can leverage those parameterised tests to buy us much greater assurance with relatively little extra code.

Imagine we've test-driven some code to calculate square roots.

```
@Test
@Parameters({"0,0", "1,1", "4,2", "9,3", "0.25,0.5"})
public void rootOfPositiveInputIsFound(double input,
                                       double expected){
  assertEquals(expected, Maths.sqrt(input), 0.00001);
}
```

Our final solution passes all of these tests, and – while we can't think of any more test cases that we'd expect to fail – maybe we don't have 100 confidence that our design will always work.

With a bit of extra work, we have the potential to test our code against a much larger set of cases.

If we wanted to perform tests on 1,000 different inputs, we're not going to code every test case by hand. We're going to want to generate the input data, and therefore we'll have to calculate the expected output.

So, the first thing we'll need to do is generalise our test assertion so that expected result is calculated rather than supplied as a parameter value.

```
@Test
@Parameters({"0", "1", "4", "9", "0.25"})
public void squareOfSquareRootSameAsInput(double input){
   double sqrt = Maths.sqrt(input);
   assertEquals(input, sqrt * sqrt, 0.00001);
}
```

Notice how I renamed the test to more accurately describe the rule, which is now explicitly described in the assertion. Generalising our tests like this can make them more self-describing.

Now the expected result's being calculated, we can generate as much input data to drive this test as we like.

For example, we could generate a range of inputs from 0 to 100, incrementing by 0.1 each time.

```
@Test
@Parameters(method="inputs")
public void squareOfSquareRootSameAsInput(double input){
   double sqrt = Maths.sqrt(input);
   assertEquals(input, sqrt * sqrt, 0.00001);
}

private Object[] inputs(){
   List<Double> numbers = new ArrayList<>();
   double  i = 0d;
   while(i < 100){
      numbers.add(i);
      i += 0.1;
   }
   return numbers.toArray();
}
```

These tests take a while to run, so we might not want to include them every time we run our unit tests. We could have our cake and eat here by having two versions of the same test fixture – one that tests a handful of cases, and one that does a more exhaustive set of tests.

```
@RunWith(JUnitParamsRunner.class)
public class ExhaustiveMathsTests extends MathsTests {

   @Test
   @Parameters(method="inputs")
   public void squareOfSquareRootIsSameAsInput(double input) {
      super.squareOfSquareRootIsSameAsInput(input);
   }

   private Object[] inputs(){
      List<Double> numbers = new ArrayList<>();
      double  i = 0d;
      while(i < 100){
         numbers.add(i);
         i += 0.1;
      }
      return numbers.toArray();
   }
}
```

I've extended the original test fixture, and added our test data generator to this new subclass. When I want to run these exhaustive tests, I run this test fixture. When I just want to run the original unit tests, I run the original fixture. Easy as peas!

We can use any algorithm we like to generate our test data. It could be a range, like we did here for this simple one-input problem.

If there are multiple input parameters that interact with each other in our solution's logic (e.g., "when a customer is over 18 AND has more than 12 loyalty points THEN they get a free loan of any video title"), then we could generate different combinations of inputs.

And we could generate random input values, just to see what happens - a sort of automated exploratory testing.

```
@RunWith(JCheckRunner.class)
public class RandomMathsTests extends MathsTests {

   @Test
   @Configuration(tests=1000)
   public void squareOfSquareRootIsSameAsInput(double input){
      imply(input >= 0);
      super.squareOfSquareRootIsSameAsInput(input);
   }
}
```

In this example, I've used a tool called JCheck (www.jcheck.org - a Java implementation of Haskell's QuickCheck) to generate 1,000 random test cases, all of which must have positive input values. For just a few extra lines of code, we can get 1,000 extra tests!

Tests that express the rules in general terms are called *property-based tests*, and they can be a gateway to much more powerful kinds of testing that are able to produce very high levels of reliability.

CRITICAL CODE

Would we use techniques like this all the time, on all of our code? Probably not. Though it varies from application to application, we tend to find that parts of our code are more critical than others.

As a default, I recommend that you try to test-driven as much of your code as possible. Not only does it lead to simpler, cleaner designs, but the level of reliability basic TDD can achieve is good enough for the majority of situations.

But some of your code is likely to be especially critical, and with that code we may choose to go beyond TDD. Here are three factors you may want to look out for when deciding how much extra testing code might need:

1. **Business risk** – starting with customer tests, identify scenarios where the stakes are high if the software breaks. Talk to the customer and ask "how big a deal will it be if this doesn't work 1 time in 1,000? 1 time in 10,000? 1 time in 1

million?" Here, software can be a victim of its own success. Seemingly unimportant code running a website like, say, Facebook, can become a big deal when it's being executed billions of times every day. There may only be a 1:1,000,000 chance of failure, but on some websites that means it will happen hundreds of times every day.

2. **Complexity** – how likely is it that a piece of code could be broken? Typically, the more complex code is, the more things that can go wrong. We won't need to test a method that adds a list of numbers together more than once. But a Fast Fourier Transform might need thousands of tests. Target your testing at code that presents the greatest risk due to its complexity.
3. **Dependencies** – a piece of code may be simple, but if it's used (directly or indirectly) by millions of lines of client code then, when it breaks, *everything* breaks. Analyse the dependencies in your software to determine which modules and methods are carrying the biggest load in terms of reuse (a useful metric here is called the *rank* of that module or method), and target your testing where the load is heaviest.

The important thing is to be aware of the risk your code presents and, with practice, you'll learn how to apply an appropriate amount of effort to making sure the most critical parts of your software are good enough for use in the real world.

MUTATION TESTING

Although TDD can produce tests that give us confidence that our code is working, there will always be times when we have doubts. Just because the tests are all green, that doesn't necessarily mean our code has no bugs we need to worry about.

Experienced TDD practitioners recommend testing your tests by running them to make sure they actually do fail when the code gives the wrong result.

We can take this further with a technique called *mutation testing*.

Let's ask ourselves how confident are we that if our square root solution was broken in any way, one of our tests would catch it.

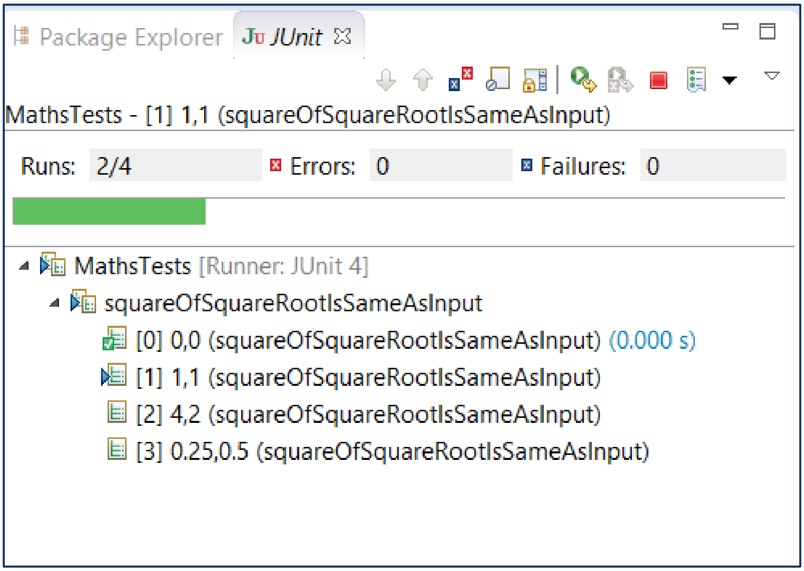
To do this, we can deliberately introduce an error, like changing a + into a - .

```
do {
   t = squareRoot;
   squareRoot = (t + (number / t)) / 2;
} while ((t - squareRoot) != 0);
```

We could "mutate" this code into:

```
do {
   t = squareRoot;
   squareRoot = (t - (number / t)) / 2;
} while ((t - squareRoot) != 0);
```

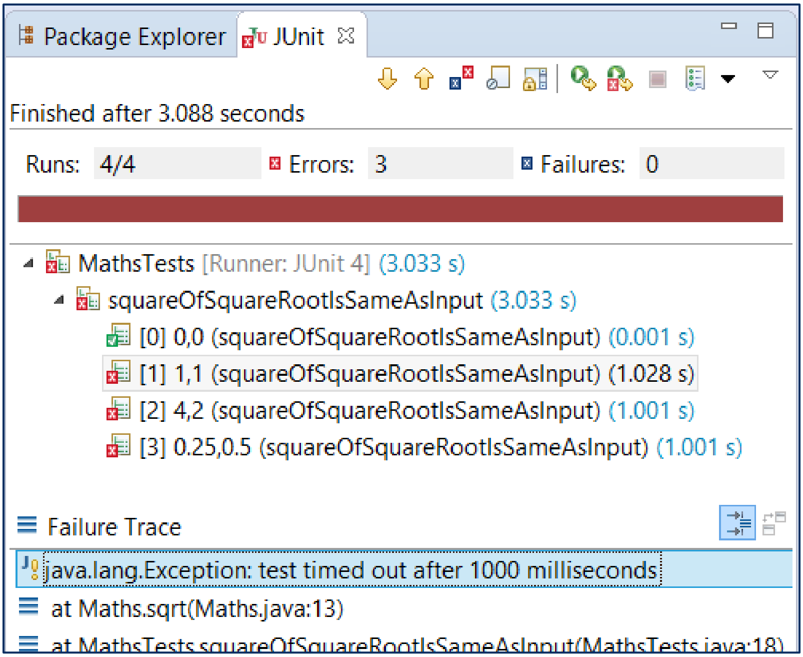
Then we run our unit tests to see if they "kill the mutant".



```
MathsTests - [1] 1,1 (squareOfSquareRootIsSameAsInput)
Runs: 2/4    Errors: 0    Failures: 0

MathsTests [Runner: JUnit 4]
  squareOfSquareRootIsSameAsInput
    [0] 0,0 (squareOfSquareRootIsSameAsInput) (0.000 s)
    [1] 1,1 (squareOfSquareRootIsSameAsInput)
    [2] 4,2 (squareOfSquareRootIsSameAsInput)
    [3] 0.25,0.5 (squareOfSquareRootIsSameAsInput)
```

The first test passes, but the second test gets stuck in an infinite loop. None of these calculations should take more than a few milliseconds. Let's add a timeout to the tests, so we can see them fail.

```
@Test(timeout=1000)
@Parameters({"0,0","1,1","4,2","0.25,0.5"})
public void squareOfSquareRootIsSameAsInput(
                                double input,
                                double expected) {
    assertEquals(expected, Maths.sqrt(input), 0.00001);
}
```



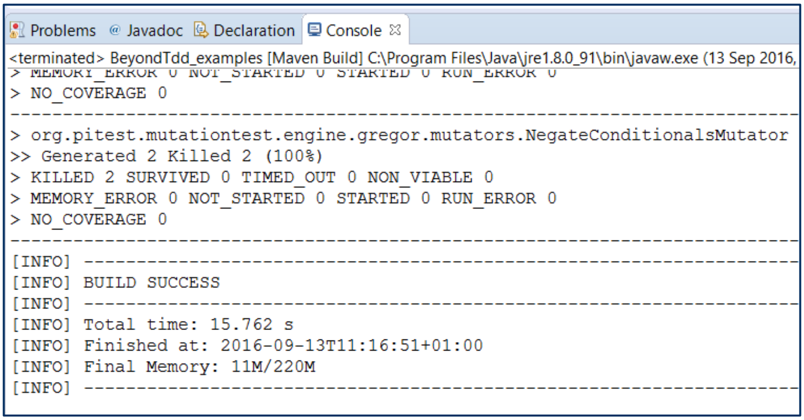
```
Finished after 3.088 seconds

Runs: 4/4    Errors: 3    Failures: 0

MathsTests [Runner: JUnit 4] (3.033 s)
    squareOfSquareRootIsSameAsInput (3.033 s)
        [0] 0,0 (squareOfSquareRootIsSameAsInput) (0.001 s)
        [1] 1,1 (squareOfSquareRootIsSameAsInput) (1.028 s)
        [2] 4,2 (squareOfSquareRootIsSameAsInput) (1.001 s)
        [3] 0.25,0.5 (squareOfSquareRootIsSameAsInput) (1.001 s)

Failure Trace
java.lang.Exception: test timed out after 1000 milliseconds
at Maths.sqrt(Maths.java:13)
at MathsTests.squareOfSquareRootIsSameAsInput(MathsTests.java:18)
```

Our tests have caught the error we deliberately introduced, so we have greater confidence that part of our code is being effectively tested.

Let's change it back, and "mutate" another part of our code, repeating this process until we're confident that our tests would catch errors in all of it.

Boy, that sounds like laborious work, doesn't it? Make a single "mutation" like turning + into -, or > into <, run all the tests, change it back. For every line of code? Phew!

Thankfully, there are tools available that can do this for us. I'm using a mutation testing tool called PIT (www.pitest.org) for my example. Conveniently, I can run it from within Eclipse using Maven.



```
<terminated> BeyondTdd_examples [Maven Build] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (13 Sep 2016,
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
> org.pitest.mutationtest.engine.gregor.mutators.NegateConditionalsMutator
>> Generated 2 Killed 2 (100%)
> KILLED 2 SURVIVED 0 TIMED_OUT 0 NON_VIABLE 0
> MEMORY_ERROR 0 NOT_STARTED 0 STARTED 0 RUN_ERROR 0
> NO_COVERAGE 0
--------------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 15.762 s
[INFO] Finished at: 2016-09-13T11:16:51+01:00
[INFO] Final Memory: 11M/220M
[INFO] ------------------------------------------------------------------------
```

My square root tests have survived comprehensive automated mutation testing, and my confidence in those tests is now much higher.

TEST-DRIVING THE "UNTESTABLE"

Some programming languages and platforms lend themselves more readily to TDD than others. Programmers looking for an excuse not to write automated tests will sometimes cite the technology as the main obstacle: e.g., "We'd TDD this, but it's SQL", or "We can't do TDD here because these classes can only run inside a web server."

IF YOU CAN CALL IT, YOU CAN AUTOMATE IT

The fact is, though, that we can test-driven pretty much anything that we can automate in code, and we can automate pretty much anything in any programming language. It just takes some ingenuity and imagination.

Programming language choice is rarely the barrier it appears to be. There's a SQLUnit, a COBOLUnit, and even frameworks for unit testing microprocessor designs. If it offers the ability to invoke programs or functions written in that language, then we can test-drive it.

SEPARATE CONCERNS

The question we should ask ourselves when we're faced with a tricky technology for TDD is "how much code do we really need to write in this technology?"

Take user interfaces, for example. Most UI frameworks, like Java Swing or JSF, present challenges for writing fast-running automated tests.

But how much of our code really needs to be inside a Swing control, or a JSF Facelet? Typically, we find our UI classes do more than they need to, including processing that really has nothing to do with how data is displayed or users interact with the application.

The trick here is to treat UI frameworks as external dependencies, like we did in a previous chapter, and to factor our code to minimise the amount that directly depends on the framework.

Specifically, the code that knows how to populate, say, a text box or a listbox from application data, and that knows which method on which application object to call when a user performs and action.

It's just two questions, really, we need to answer that directly involves the UI:

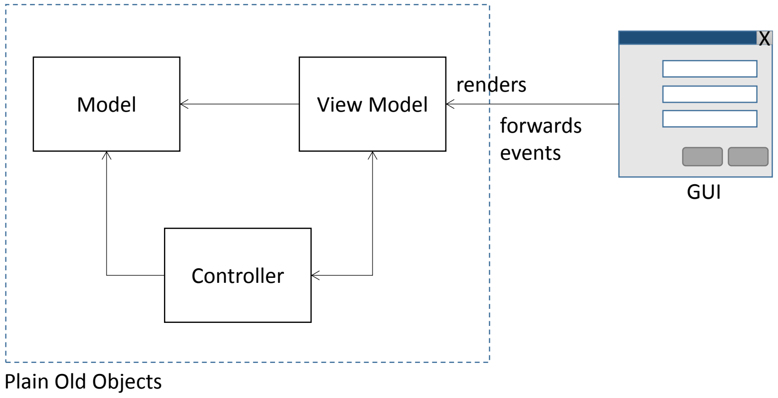
1. Is the data rendered correctly?
2. Was the action/event handled?

If clicking a button calculates insurance premiums and then displays them on the screen, our UI tests only need concern themselves with whether the right method on the right object was invoked (for which we could use a mock of that object), and whether the result is displayed in the way the customer wants (for which we could use a stub to get a test value). Whether or not the actual premium is calculated correctly is *somebody else's problem*.

A good way to minimise UI-specific code is to apply the Model-View-Controller pattern, but with an extra layer of classes that represent the logic of the views (a "view model").

The view model represents an abstract concept of a UI component. If the Insurance Premium web page has a premium amount displayed on it, then the equivalent view model would have a *premiumAmount* field that represents the underlying data.

If clicking the OK button calculates the premium amount, then that event should be forwarded to a *calculatePremium()* method on the view model.



The flow and logic of user interactions are handled by a core of plain old objects with no direct external dependencies on a UI framework, and therefore completely amenable to unit testing.

What's left is a thin sliver of code that binds directly to the UI framework's API. In our testing pyramid, these bindings can be checked using a much smaller number of system-level tests to make sure everything's wired together correctly.

FAKE IT

If you're *really* determined to write automated tests for, say, JSF Facelets or ASP.NET web forms, there's usually a way to fake the container that those classes need to run in.

A good example of this is MockServer (www.mock-server.com), which can act like a web server in which your code is running. It recreates web request scenarios so that we can repeatedly test how our code handles them.

NON-FUNCTIONAL TDD

Another psychological barrier developers new to TDD have trouble overcoming is the idea that not all our tests need to be about the logic of our code.

It's entirely possible to ask other kinds of questions, like "Does this function execute fast enough?" and "How much memory does this algorithm use?"

We've already seen an example of specifying a *timeout* for a JUnit test using the built-in parameter of the *@test* annotation.

```
@Test(timeout=1000)
@Parameters({"0,0","1,1","4,2","0.25,0.5"})
public void squareOfSquareRootIsSameAsInput(
                                     double input,
                                     double expected) {
   assertEquals(expected, Maths.sqrt(input), 0.00001);
}
```

If we wanted to see how our code performs under a bigger load than a single client thread, we could use a tool like TestNG (www.testng.org)

```
@Test(threadPoolSize = 3,
      invocationCount = 10,
      timeOut = 50)
public void millionItemsSearchedInUnder50ms() {
   String[] items = new String[1000000];
   for (int i = 0; i < items.length; i++) {
      items[i] = "BLAH" + i;
   }
   int ITEM_INDEX = 555555;
   items[ITEM_INDEX] = "Jason";
   assertEquals(ITEM_INDEX,
               new BinarySearch(items).find("Jason"));
}
```

WHERE DO NON-FUNCTIONAL TESTS COME FROM?

How do we know if our code needs to be tested non-functionally? Do we just test all of it to make sure it's as fast and efficient and scalable and secure and maintainable and accessible and all the other –ables as a matter of course?

No, that would be a silly waste of effort. Not all code is performance-critical, for example. A function may run slow, but if it's only run once a month that might not be a problem.

We should look to our customers to learn what parts of our software might need other kinds of testing. When discussing a user story, for example, ask questions like "How often will this be used?" and "What sort of hardware will this be running on?"

A banking feature might be used thousands of times every second, and therefore we might want to test that it can scale up to handle that sort of load.

A feature in a computer game might be timing-critical: not much point shooting your weapon if the computer's going to take 3 seconds to process that.

Code might be running on devices with very limited memory, in which case using a gigabyte of RAM in an algorithm probably isn't an option.

What we should generally not do is invent non-functional requirements without consulting the customer, and speculatively optimise our code "just in case". Let the customer's needs and *real* technical constraints (budget, hardware, legal compliance, etc) drive your non-functional tests.

And relate the non-functional tests directly to functional customer tests wherever possible, so you can be sure that you and the customer are talking specifics. When you talk about a feature being used very frequently, be sure which scenario you're referring to. Withdrawing cash from an ATM may occur frequently, but an edge case like the mechanism that dispenses the notes jamming occurs very rarely.

UNPREDICTABLE BEHAVIOUR/DATA & "FLICKERING BUILDS"

One of the risks when we write tests that, for example, specify a maximum response time for a method, is that there may be times when the computer is busy doing whatever it is that computers do when we're not looking.

The threading models of modern operating systems can be unpredictable, and we might see a performance test very occasionally fail.

This leads to what some programmers call "Heisenbugs" (a pun on the discoverer of the Uncertainty Principle in quantum physics), and *flickering builds*; builds that sometimes succeed and sometimes fail.

Three ways to reduce this risk are:

a. **Make sure you leave a good margin for error**. If you believe a method might take 10ms on average to complete, setting the timeout for your test to 11ms is very likely to produce flickering builds. Think "orders of magnitude": setting the timeout to 10x the average (100ms) should make failures vanishingly rare. But not extinct.

b. **Approach performance in a more deterministic way**. A binary search execution time may vary, but on the same input, it will always do the exact same number of comparisons. So instead of writing a test that requires the search to take less than 10ms, write a test that requires the search to have no worse than O(log N) performance.
c. **Fake the unpredictable part**. Stubs and mocks can be used to return unpredictable data, like the current date and time, or a "random number" used in a game, to ensure repeatable tests

CLEAN CODE & CONTINUOUS INSPECTION

Some developers would argue that there's one exception to the rule about non-functional tests coming from the customer, and that's tests that are about the maintainability of our code.

For sure, the customer's not likely to be directly interested in how readable the code is, or how complex the modules and functions in it are, or how much code duplication there is.

But when it comes to maintainability requirements, the clue is in the name. What almost certainly *will* interest the customer is how much it will cost to change the code at a later date.

Cost of change has a direct impact on our ability to sustain the pace of innovation, delivering more value for longer through a software product or system.

Without paying continuous attention to factors that increase the cost of adding or changing code, change can become so expensive as to effectively rule it out. When businesses can't change their software easily, then they can't change the way their business works easily.

It's important to understand when we embark on a new piece of software what its lifespan is intended to be, so we can judge the importance of maintainability.

But beware: there's a lot of software out there that was originally intended as a short-term, tactical, throwaway solution to a problem, but has ended up in use for decades.

Better to ask not how long the solution needs to last, but how long the problem will be around in some form. A mobile app for a one-off promotion is likely to only be needed for the duration of the promotion. A ticketing system for a venue is likely to be around for as long as that venue sells tickets.

This is one of many situations where we need to guide our customers firmly. Given a choice, most non-technical stakeholders will reject putting significant technical considerations like code quality in favour of "more features".

The conversation you need to have with them needs to be a non-technical conversation about what happens after the first release, and how that might impact their business.

Establishing measures for things the customer cares about, like cost of change, as well as lead time on delivering features and change requests, can help to "join the dots" between the code-level benefits of TDD and the business benefits. The relatively few organisations that track these things have learned that cutting corners on code quality leads to a rapidly slowing pace of development. Some can even tell you what that has cost them as a business in terms of extra development effort and missed business opportunities.

Developers also need code-level indicators to give them an early warning of maintainability issues. Maintainability "bugs" are like functional bugs, in so much as the longer they're left in the code, the more they cost to fix. So we want to catch them as early as we can.

Every change we make potentially introduces a code quality bug; adding a new branch to an IF statement can make it too complicated, for example. *Code goes bad one change at a time.*

Teams that consistently deliver maintainable code apply some kind of continuous code quality testing that checks their code very frequently for new problems.

Pair programming, when pairing partners keep one eye on code quality issues as they work, can help keep quality up. And some teams have a policy of performing code review before anyone can commit their code to the shared repository.

A pre-commit code inspection is what we call a *quality gate*. Changes only get integrated into the code base if other members of the team consider that they're good enough.

In practice, experience teaches us that pair programming and code reviews are useful, not sufficient to achieve the level of maintainability most software really needs.

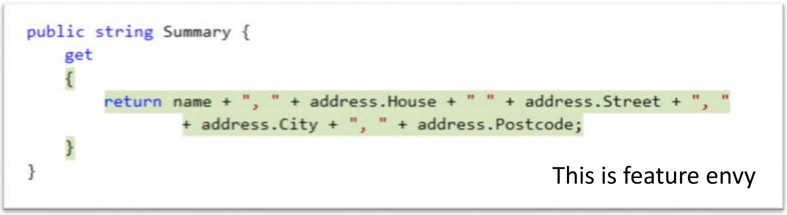
More advanced development teams *automate* their code reviews so that they can be performed very frequently across all of the code at minimal cost. This practice is called Continuous Inspection.

The next Codemanship book, "Software Design Principles", will cover Continuous Inspection in some depth, but we present a brief explanation here to hopefully get you started.

Identify goals - Continuous Inspection starts by identifying design goals; for example, to have loosely coupled modules in your code.

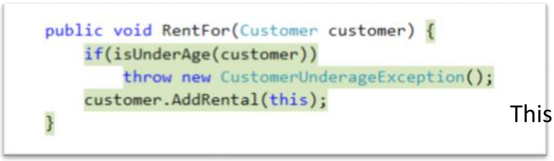
Agree indicators - Then the team agree indicators which would alert us if any code fails that goal. E.g., if there are any examples of "feature envy" in the code. Feature envy is when a method in one class calls multiple methods in another class, giving a strong indication that the method is in the wrong class, prompting us to move it.

Select examples - Next, we gather some representative code – test cases –that we agree are and are not examples of feature envy. Here are some C# test samples.



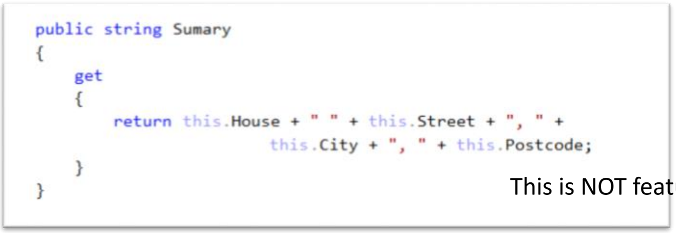
```
public string Summary {
    get
    {
        return name + ", " + address.House + " " + address.Street + ", "
            + address.City + ", " + address.Postcode;
    }
}
```

This is feature envy



```
public void RentFor(Customer customer) {
    if(isUnderAge(customer))
        throw new CustomerUnderageException();
    customer.AddRental(this);
}
```

This is NOT feature envy



```
public string Sumary
{
    get
    {
        return this.House + " " + this.Street + ", " +
            this.City + ", " + this.Postcode;
    }
}
```

This is NOT feature envy

Automate quality gates – In a test-driven manner, one test case at a time, we automate a quality gate that will catch the bad examples and allow the good examples through.

There are many tools available for writing automated code quality gates. At the very least, you need a parser to read code and extract in-memory models of it that we can query programmatically. Here, I've used a tool called FxCop (it comes with Visual Studio) to analyse C# looking for examples of feature envy.

```
private void InspectMethodBody(Method method)
{
     if (method != null)
     {
         var visitor = VisitStatements(method);
         CheckForFeatureEnvy(method, visitor);
     }
}

private MethodInvocationVisitor VisitStatements(Method method)
{
    MethodInvocationVisitor visitor =
          new MethodInvocationVisitor(method.DeclaringType);
    visitor.VisitStatements(method.Body.Statements);
    return visitor;
}

private void CheckForFeatureEnvy(Method method,
                             MethodInvocationVisitor visitor)
{
    var enviedTypes = visitor.EnviedTypes;
    if (enviedTypes.Count > 0)
    {
        AddFeatureEnvyProblem(method, enviedTypes);
    }
}
```

And, of course, I used TDD to implement this custom FxCop rule.

```
[TestFixture]
public class FeatureEnvyRuleTests
{
    [Test]
    [TestCase("MethodThatMakesOneCallOnOneSupplier", 0)]
    [TestCase("MethodThatMakesTwoCallsOnOneSupplier", 1)]
    [TestCase("MethodThatMakesTwoCallsOnDifferentSuppliers", 0)]
    public void MethodsThatMakeMultipleCallsHaveFeatureEnvy(
                                        string methodName,
                                        int expectedProblemCount)
    {
        FeatureEnvyRule rule = new FeatureEnvyRule();
        rule.Check(GetMemberToCheck(methodName));
        Assert.AreEqual(expectedProblemCount,
                                         rule.Problems.Count);
    }
```

Deploy Quality Gate – once we're satisfied that our quality gate works well enough to be tried out on real projects, we integrate it into our build cycle. Before we do this, the team needs to agree on the policy that they'll stick to when this particular gate fails. A *hard gate* will fail the build if any code is caught in the gate and the developer who committed that code will have to address the issue.

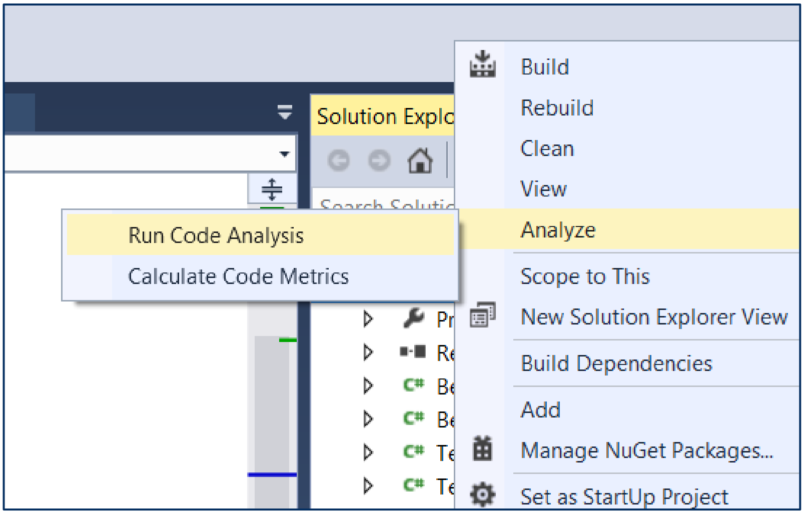
A *soft gate* will report issues, but the team may have a process for reviewing the suspect code and – if they feel it's not a problem – let it through.

Of course, like Jones the cat in the movie *Alien*, letting code quality issues through the gate means they will show up again in later inspections. If we're happy to let code through, we need a mechanism for ignoring previously addressed issues. There's nothing less useful than a continuous inspection report that's stuffed full of potential issues we already know about and chose to ignore.

This is why many teams exclude previously raised issues from inspection reports, so only new issues that get picked up are highlighted. FxCop has a feature that enables this, but it can be a bit hit and miss. In my implementation, I use an XML diff tool to compare the new report with the previous one.

Clean Check-In – just as it's a good idea to make sure the code passes the functional tests before we commit it, we should also check that it passes the code quality gates, so as not to risk breaking the build and wasting the team's time. I'm in the habit of running code quality checks continuously on my desktop, typically only going 10-15 minutes between inspections.

It helps enormously if you can run code analysis from within your IDE. Visual Studio is quite advanced in this respect.



Feedback – As with all code, we're unlikely to be so lucky as to get it exactly right first time. Using code quality gates on real projects will be a learning experience, and we need to feedback those lessons and evolve our gates as well as our Continuous Inspection processes. It can take several months for them to mature and bed in.

Reuse – The good news is that, for a specific technology, once we've learned lessons about Continuous Inspection on one project, much of that can be reused on new projects. Feature envy is feature envy, and it doesn't really matter if we're building a mobile weather app or a cloud-based trading system.

EXERCISE #18

Convert one of the parameterised tests you write for a previous exercise into a property-based test, and drive it using programmatically generated data.

EXERCISE #19

Performing code mutations manually by making a single edit to the code (e.g., turning a + into a -), mutation test the code you wrote for a previous exercise. Remember to do one simple mutation at a time, and revert the code after each one.

If your tests fail to "kill the mutant", look at how the tests could be improved, and then check that improvement by repeating the mutation.

If a mutation testing tool exists for the programming language you're working in, try using the tool to do it, too.

EXERCISE #20

Using mock objects to count the number of times a "swap" method is invoked, write a non-functional property-based test that will fail if an implementation of a sorting algorithm has worse than O(N log N) performance for random (very unsorted) lists of strings of up to 1,000,000 length. (HINT: you'll be needing a sorting algorithm that's *quick*).

EXERCISE #21

Design and implement an automated code quality gate to flag up methods that are too long and/or too complicated.

22. MASTERING TDD

In this book, we've explored TDD in depth. Hopefully, by now, you've learned there's much more to it than "red-green-refactor".

I've been writing tests first since before it had a name, and have trained and coached thousands of other developers in these disciplines.

If there's one thing I've learned from all that TDD experience, it's this: *if you don't want to learn TDD, you won't.*

TDD takes longer than a weekend to become proficient enough in to use on real projects. Making the move to TDD requires a longer-term commitment, typically 4-6 months.

There's a whole bunch of stuff you need to know, and there's also a bunch of habits you need to build to make TDD work for you; habits like remembering to start by writing a failing test, remembering to refactor your code after passing a test, and remembering to see the test fail for the right reasons before writing the code to make it pass.

When you first start doing TDD, it will probably feel uncomfortable, like asking a violinist to hold the bow differently to the way they've been doing it for years.

TDD will "click" for you after many hours of mindful practice, which is why it's important to try the exercises in this book. You can no more learn TDD just by reading this book than you could learn to ride a bicycle that way.

BUILDING HABITS

Kent Beck, author of Test-Driven Development By Example, once said of himself: “'I'm not a great programmer; I'm just a good programmer with great habits”.

Mastering TDD requires mastery of two things: the “art” of test-driving designs, which takes years and years of experience, and the basic habits of TDD, which you can build in a few months with regular practice.

The habits are important because, once they become second nature, your mind is then free to focus on solving the problem.

An experienced guitar player is usually not consciously aware of her picking technique. She is focused on the music.

And experienced TDD practitioner is usually not consciously aware of running the tests after every change, or writing the assertion first and working backwards, or avoiding doing any refactoring when tests are failing. She is focused on solving the problem.

Here’s a list of my TDD habits:

- I start by writing a failing test. Always.
- I write the simplest code I can think of to get it passing quickly
- I refactor the code to make the next test easier
- I don’t write source code unless there’s a failing test that requires it
- I write tests that only have on reason to fail
- I make sure I’ve seen the test fail for the right reason
- I triangulate when the solution’s not obvious & trivial
- I write the assertion first and work backwards
- I don’t refactor when a test is failing
- I make sure the test code is self-explanatory and reads like a specification

- I refactor duplicate tests into parameterised tests (for all the reasons we've seen)
- I write my code using the customer's language whenever possible
- I drive internal designs directly from failing customer tests
- I use dependency injection and test doubles to isolate external dependencies in my tests
- I favour fast-running tests whenever possible
- I use mock objects to write tests that are about collaborations between objects, and assertions to write tests that are about the work objects do
- I organise my test code so it's obvious what's being tested, and what kinds of tests they are (unit, integration, customer, performance etc)
- I keep my test code and source code separate so it's obvious which is which and they can easily be packaged separately for DevOps reasons
- I don't write tests that can have a domino effect on other tests (e.g., if they're sharing data that one test changes and the next test relies on). I can run my tests individually and in any order.
- I avoid tests that rely on unpredictable behaviour or data that changes (e.g., a test that uses the current date and time)
- I maintain my test code so that it remains valuable, up-to-date documentation

MAKE TDD YOUR DEFAULT BEHAVIOUR

By far the most commonly cited cause of failing with TDD is teams who attempt to learn it while under delivery pressure. When people are under pressure, we revert to our default behaviour.

At first, TDD feels clunky and difficult and slow. Developers who are climbing the TDD learning curve see their productivity drop. Teams underestimate the steepness of the curve, and the length of climb. It's not days, or weeks; it's months.

No wonder, when the heat is on, so many developers get out of the TDD kitchen.

The breakthrough happens when TDD becomes your default behaviour. The more pressure you're under, the smaller the steps you take, the more you triangulate, the more your refactor, the more you "TDD".

This is healthy. Your boss probably won't believe you, but the way the most advanced software teams get back on schedule is not by cutting corners, but by taking *more* care.

UNDER THE RADAR

For these reasons, I strongly recommend against attempting to test-drive everything on real projects from day one. Your productivity will circle the drain, and your customer and your boss will not be happy. There are far too many development teams out there who are *forbidden* to do TDD

Instead, adopt TDD by stealth. Make a bit of time available every day to work on your TDD skills and build those habits.

As your confidence grows, you'll feel more comfortable doing TDD for extended periods on increasingly more challenging problems, until – one day – you'll be doing it every day as your default way of working. You'll find it hard *not* to do TDD.

UNDER-PROMISE, OVER-DELIVER

The purported benefits of TDD, like more reliable software, shorter cycle times for delivering features and a lower cost of changing code, really only apply to developers who can do TDD reasonably well. There's going to be quite a lead time before you notice these improvements – typically more than a year.

The experience of many teams is that if you over-sell the benefits and how soon the business will feel them, you can set expectations unrealistically high. Disappointment is likely to follow.

The smarter way to adopt TDD is to adopt by stealth, gradually building up, and then – rather than promise future benefits – draw attention to benefits already being felt.

Would you rather be in a meeting persuading your boss to let you try TDD because "it will reduce bugs in production by 90%", or in a meeting proudly revealing that bug counts are down 90%? If nothing else, it will make life a lot easier for other teams who are hoping to try TDD.

This is why I also strongly recommend, as your TDD confidence grows, you keep track of key measures like defect counts, cycle times, and cost of change, so you can make your case when the time comes.

IT'S EASIER TO APOLOGISE THAN GET PERMISSION

What's your company's policy on FOR loops? Did you need permission to use them?

To an experienced TDD practitioner, writing failing tests first is just how they write code. It doesn't take them any longer, typically, and often saves time. It doesn't require investment in software licenses or other purchases. It doesn't change the flow of the user experience or alter the application's UI design. Arguably, TDD-ing code is a developer's decision, just like using FOR loops.

It only becomes a business decision when TDD-ing your code is going to cost more money, or take more time, or fundamentally alter the business-eye view of the software in some way.

But if you ask…

All too often, managers will say "no". Which is why, as an *experienced* test-driven developer, I just don't ask.

PRACTICE, PRACTICE, PRACTICE!

I’m going to finish this book where I started. To learn TDD, you need to *do* TDD. *Lots of TDD.*

Use the exercises in this book. Seek out TDD project ideas on the Web (searching for “TDD katas” is a good place to start). Come up with your own ideas for projects you could test-drive.

And when you’ve completed an exercise or a project. Do it again, *better*. I must have done the Fibonacci Numbers TDD kata 100 times, and I’m still discovering new ways of doing it.

Practice alone. Pair program with friends. Do a screencast of you TDD-ing and watch it back (you’d be amazed the stuff we do that we didn’t know we were doing). Grab a space, a laptop and projector and “mob program”. Watch other developers doing TDD. YouTube is crammed full of screencasts (though not all of them setting great examples – but even TDD done badly can be educational.)

Don’t be afraid to try. Don’t be afraid to fail.

Good luck!

SUGGESTED FURTHER READING

Test Driven Development: By Example (Addison-Wesley, 2002) – Kent Beck

Growing Object-Oriented Software, Guided by Tests (Addison-Wesley, 2009) – Steve Freeman & Nat Pryce

Refactoring: Improving the Design of Existing Code (Addison-Wesley, 1999) – Martin Fowler

Working Effectively with Legacy Code (Prentice Hall, 2004) – Michael Feathers

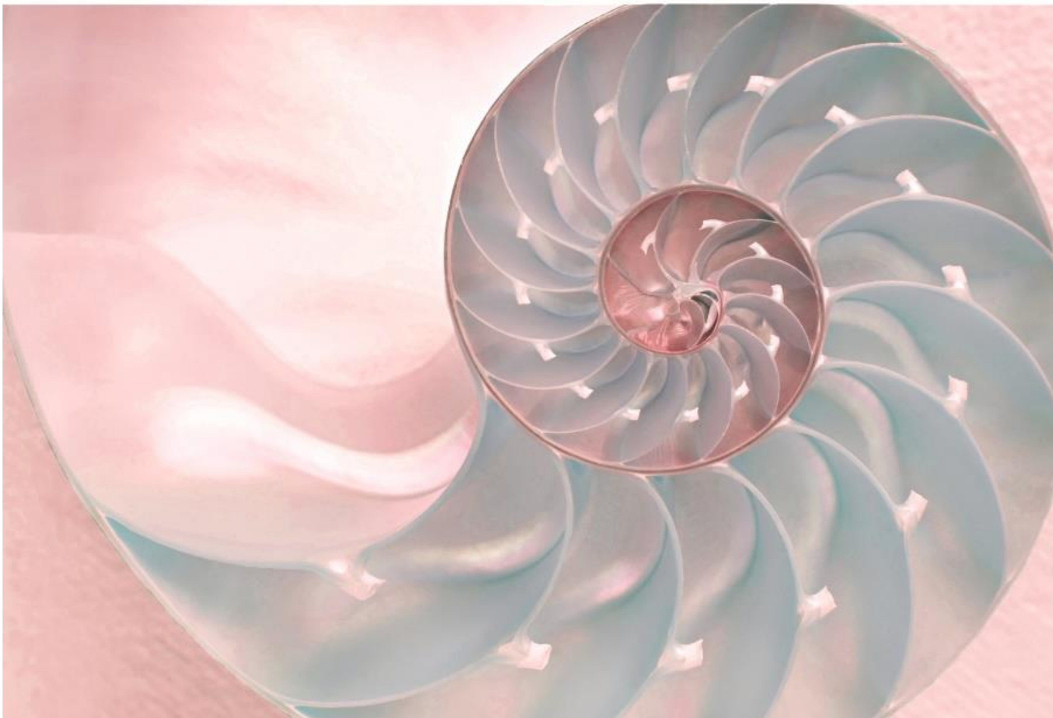
Clean Code: A Handbook of Agile Software Craftsmanship (Prentice Hall, 2008) – Robert C Martin

Specification by Example: How Successful Teams Deliver the Right Software (Manning, 2011) – Gojko Adzic

xUnit Test Patterns: Refactoring Test Code (Addison Wesley, 2007) – Gerard Meszaros

Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation (Addison Wesley, 2010) – Jez Humble & David Farley

codemanship



NOT FOR RESALE