Colin Vipurs

Tests Need Love Too

Unit Testing Mistakes and How to Avoid Them

developer .press

by Colin Vipurs

Tests Need Love Too

Unit Testing Mistakes and How to Avoid Them

ISBN: 978-1-909264-18-2

An Imprint created by Software & Support Media Ltd.

Cover: ©istockphoto.com/kycstudio

Introduction

In 1994 I started my undergraduate degree and received the marks for my first programming assignment. I remember it vividly as the comments said "One of the best implementations I have seen", yet I only scored a barely passing 44%. The reason for this was the lack of evidence of testing, which was unsurprising given that my code had bugs and the calculations it was performing were incorrect. This should have been the time I realised the importance of testing, but unfortunately this didn't come until a few years later when I started my first programming job.

I was working on a system that would take TV listings in a raw, custom format and convert them, according to various rules, into the format that our many customers wanted[1]. I had a very simple change to make and knew that it would only affect a few customers so I made it and pushed it to production only to find that the change I made affected everybody! Vowing to never make this mistake again I wrote a Perl script that would exercise the existing production version of the code and a release candidate over all data for every customer. It would take 5 hours to run but at the end of the run you were guaranteed you hadn't broken anything. Little did I know that this simple mistake and script would lead me to a lifetime of being test infected[2].

When I started working with Java I stumbled upon the concept of unit testing and immediately fell in love with the ideology, which has been a core part of my development process ever since. Over the years I have re-defined how I view a "unit" and have made many, many mistakes in writing tests. I have trained less experienced developers in the art of unit testing and noticed that they would make the same mistakes I did. As part of my job I have also interviewed lots of developers and saw that they were making the same mistakes too.

For Devoxx UK 2013 I put together a presentation on what I experienced as the most common mistakes and how to avoid making them. After the event I was asked lots of questions on certain parts and it became obvious it needed expanding upon and so the idea for this book was born. The material contained herein will attempt to address what I see as the most painful mistakes developers make when writing unit tests- the kinds of mistakes that make unit testing a painful and slow process and lead to testing costing more time than it saves.

The Gilded Rose

Throughout the book I will be using, where possible, the same piece of code. For this I have chosen to use the "GildedRose"[3] kata, which was designed as a refactoring exercise. The GildedRose is a fictional system simulating a small inn that sells goods and has the following rules:

- All items have a SellIn value which denotes the number of days we have to sell the item
- All items have a Quality value which denotes how valuable the item is
- At the end of each day our system lowers both values for every item
- Once the sell by date has passed, Quality degrades twice as fast
- The Quality of an item is never negative
- "Aged Brie" actually increases in Quality the older it gets
- The Quality of an item is never more than 50
- "Sulfuras", being a legendary item, never has to be sold or decreases in Quality
- "Backstage passes", like aged brie, increases in Quality as its SellIn value approaches; Quality increases by 2 when there are 10 days or less and by 3 when there are 5 days or less but Quality drops to 0 after the concert

Full source code for the Java version of this kata can be found in the appendix, and it has been ported to many languages, which can be found online. Due to the limitations of the book width, it is highly recommended to view the source code online or in your favourite IDE.

1 If you've ever read TV or radio listings in a newspaper, magazine or online there's a good chance what you've read has been through some of my software
2 *http://c2.com/cgi/wiki?TestInfected*
3 *http://craftsmanship.sv.cmu.edu/exercises/gilded-rose-kata*

1 Naming

"There are only two hard problems in Computer Science: cache invalidation and naming things."

- Phil Karlton

In software development, naming should be easy, it's something that must be done all the time; files, classes, methods, variables, so why is it so hard to get right? At the time of writing this book, a Google search for "public void test1() filetype:java" yields about 323,000 results! This is publicly available source code, yet each and every developer could not think of a better name for these tests than "test1". It is highly unlikely that a class would be named "Class1" or a method "method1"[4] and for the same reason you shouldn't name a test "test1".

Early on in a developer's education or career it is taught that something is named so that anyone working on the system can understand its purpose without having to read every line of code. A quick look at some examples from the Java standard classes shows examples like:

```
Integer::parseInt
StringBuilder::append
```

Or in standard JavaScript there are examples like:

```
Array::forEach
JSON::parse
```

Even without exposure to either language it is easy to understand what the classes or methods above do, but with a test named "test1", there is no choice but to read the source to understand what's being tested.

It is often said that unit tests can act as executable documentation for a system, but by naming tests poorly this documentation loses most of its usefulness.

Imagine the GildedRose code introduced in the introduction has a suite of tests all named very badly as shown in listing 1-1.

```
public class GildedRose {
   @Test
   public void test1() { … }

   @Test
   public void test2() { … }

   @Test
   public void test3() { … }
}
```

Listing 1.1 Unnamed tests

In order for anyone to be able to tell what each of these tests is doing requires reading of each method body, which inhibits the ability to quickly identify the test required. Tests with these names do exist in the wild, but they are quite extreme. What's more common is to see tests named for the function they are executing as can be seen in listing 1-2.

```
public class GildedRose {
   @Test
   public void updateQuality() { … }

   @Test
```

```
    public void updateQuality_2() { … }

    @Test
    public void updateQuality_AgedBrie() { … }
}
```

Listing 1.2 Badly named tests

These are a lot better than the previous examples but they still don't carry all of the information required to pinpoint the behaviour being tested. In addition, code that has a variety of tests for the same method ends up acquiring meaningless labels at the end of the test name to differentiate them from previous tests.

When production code fails, this will usually manifest itself as a bug report and at this point the test suite should already be passing[5]. This means there is either an incorrect test or no test for the feature that has a bug. In either of these cases there will be a need to quickly identify where the test is (or isn't) and naming can be an invaluable tool. When a test itself fails, it is typically a sign that the code under test is incorrect[6] and again, a well-named test will help identify what the feature is instead of having to read through the test code to identify what exactly is being tested.

A good naming strategy should allow reading the test name as a complete sentence focussing on **behaviour**, as seen in listing 1-3.

```
public class UpdatingQuality {
    @Test
    public void reducesTheQualityOfAStandardItem() { … }

    @Test
    public void reducesTheSellInValue() { … }

    @Test
    public void raisesTheQualityOfAgedBrie() { … }
}
```

Listing 1.3 Test names focussing on behaviour

Because this example is in Java, a little imagination is needed to visualise the sentences, which become:

"Updating quality reduces the quality of a standard item"

"Updating quality reduces the sell in value"

"Updating quality raises the quality of Aged Brie"

Languages that support anonymous functions do not suffer from the restrictions of requiring the method name to carry information like they do with tools like JUnit or phpUnit, but the recommendations still hold true in these languages. One of the popular unit testing frameworks for JavaScript is Mocha, and if the above tests were written using that, they might look like something like the code shown in listing 1-4

```
describe('Updating Quality', function() {
    it('reduces the quality of a standard item', function() { … })
    it('reduces the sell in value', function() { … })
    it('raises the quality of aged brie', function() { … })
})
```

Listing 1.4 Well named tests using Mocha

The use of the word "should" is popular but something I personally do not like as it suggests to me that the behaviour is optional, not mandatory. The IETF RFC guidelines state that "[should], or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course" (http://www.ietf.org/rfc/rfc2119.txt)

A lot of tutorials and documentation indicate that tests be named for the class and method they are testing. One of the biggest problems with this is that these schemes are more tied to implementation details rather

than focussing on behaviour and as such are not resilient to renaming and could eventually do more harm than good if the test name is not kept up to date.

4 I did once work on a system that had a method called "bodge_proc
5 If your test suite is failing and you've pushed to production anyway, you've got much bigger problems!
6 A common sight for those practicing Test-Driven Development

2 Verification

One of the reasons many developers adhere to strict automated testing practises, in particular unit testing, is that it is a tool that allows changes to be made to code and get near instant feedback on changes. Without automation this becomes a slow, manual and tedious process that must be repeated over and over and over.

Some systems have an extremely high level of test coverage, i.e. some level of automated testing covered all functionality. Working on such a system can enable the delivery of functionality[7] extremely quickly as within a matter of minutes of making a change the test suite has verified everything still works as expected. Some teams are brave enough to push this to extremes and automatically deploy code to production after a commit, a process known as continuous deployment[8]. In order to do this the level of confidence that the code works must be very high, which is where the tests come in. Something that can be easily overlooked though is verifying that the **tests** also work correctly.

Practitioners of TDD follow the red-green-refactor cycle[9], where the red part is writing a **failing** test. When writing tests after the production code this is a lot more difficult as the production code is already in place and is assumed to work, but seeing tests fail is just as important as it is when writing tests first.

```
@Test
public void neverAllowsQualityToBeNegative() {
  GildedRose gildedRose = new GildedRose(items);
  gildedRose.updateQuality();
}
```

Listing 2.1: Test that will never fail

Listing 2.1 shows a test that might be written for the GildedRose system. To some people it will be obvious that this test will virtually **never fail**. The only time this test can fail is if the code under test throws an exception, in fact all of the production logic could be removed and this test will still pass because it never verifies anything. Unless the test can be seen to fail there should be no confidence that it actually works.

I've always considered tests and production code to have a symbiotic-like relationship. Once you have a verified test and the production code to make that test pass, the two should almost be "locked in" and a change in either should make the test fail.

As well as seeing the test fail, it is important to see the test fail for the right reason. A typical mistake is to check for the presence of the red bar only[10]. This can be problematic when the problem lies within the test itself as shown in listing 2.2.

```
@Test
public void neverAllowsQualityToBeNegative() {
  GildedRose gildedRose = null;
  gildedRose.updateQuality();
}
```

Listing 2.2: A test that fails for the wrong reason

When just looking for the red bar and nothing more, the usual cycle to follow is:

- See the test fail
- Implement or fix the production code
- See the test is incorrect
- Fix the test and see it pass

By doing this, the test has still never been verified and again there should be no confidence that the test is valid.

> **TIP:** Some test runners, e.g. JUnit differentiate between test **errors** and test **failures**. A test error will let you know that there was a problem instantiating and running your test and that it never even got as far as verifying your assertions.

The key to avoiding this is to inspect the test diagnostics during the failure phase. The output from listing 2.2 will show that a NullPointerExpection has been thrown and that the test never actually executed correctly.

Not only will this guarantee that the test is correct but when used in conjunction with a good naming strategy outlined in chapter 1 it can help isolate problems with production code when bugs occur. Every test must include the "assert" part of "arrange-act-assert"[11], and this is where the diagnostics can be checked and improved upon. At the very least the assertion output should include both the expected and actual values.

The GildedRose has a rule that states:

The Quality of an item is never negative

When testing this the diagnostics should state exactly why this test failed. A naïve solution to this problem is to write the error diagnostics text by hand, as shown in listing 2.3.

```
@Test
public void neverAllowsQualityToBeNegative() {
  Item itemAtMinQuality = itemWithQualityAt(0);
  updateQualityOf(itemAtMinQuality);
  assertTrue("Item quality should not be negative", itemAtMinQuality.getQuality() >= 0);
}
```

Listing 2.3: Manual test diagnostics

Not only is this time consuming, and like all documentation, prone to becoming out of date, it is duplication as the test already carries all of the information to produce these diagnostics.

This test can be rewritten to automatically produce the diagnostics required and has the added benefit that the diagnostics will change if the rules do.

```
@Test
public void neverAllowsQualityToBeNegative() {
  Item itemAtMinQuality = itemWithQualityAt(0);
  updateQualityOf(itemAtMinQuality);
  assertEquals(0, itemAtMinQuality.getQuality());
}


java.lang.AssertionError:
  Expected :0
  Actual  :-1
```

Listing 2.4: Automatic test diagnostics

Taking the test naming and diagnostics into account, the failure can be read as:

Updating quality never allows quality to be negative

```
Expected 0, actual -1
```

Without having to read the test code it should be obvious that there is a missing boundary check in the production code that allows the quality to drop below zero.

Tip: The use of a boolean assertion, e.g. JUnit's assertTrue() should be limited only to methods that actually return a boolean as it provides the least amount of diagnostic information. The overuse of this assertion is the biggest contributor to hand-written diagnostic messages

Diagnostics can be improved even further though by introducing a matching library. There are a variety available but the most widely available one is Hamcrest[12], which is available for a variety of frameworks and languages. These libraries are designed to match two values and produce rich diagnostics about a mismatch. They are typically shipped with common matchers, can be extended to match against custom objects. Rewriting the negative quality test in Hamcrest allows the test to fully embody what is to be verified and output much nicer diagnostics.

```
@Test
public void public void neverAllowsQualityToBeNegative() {
  Item itemAtMinQuality = itemWithQualityAt(0);
  updateQualityOf(itemAtMinQuality);
  assertThat(itemAtMinQuality.getQuality(), greaterThanOrEqualTo(0));
}


java.lang.AssertionError:
  Expected: a value equal to or greater than <0>
  but: <-1> was less than <0>
```

Listing 2.5: Diagnostics using Hamcrest

The use of matching libraries also allows the tests to be less strict while maintaining good diagnostics. Listing 2.5 asserts the value is greater than or equal to zero instead of the previous test, which ensured the value is exactly zero.

2.1 Verifying Collections

An area that can cause a lot of grief for producing good diagnostics is when working with and attempting to verify collections, especially unordered collections. By far and away the biggest violation here is to merely assert that the size of the collection is correct. In some cases this may be the correct thing to do but usually what is in the collection at the end of the test is what is interesting.

Within the GildedRose system, assume that items that have passed their sell-by date are removed from the store at the end of the day. A testing approach for this might be to add items, some of which should expire at the end of the day and then verify that the number of items left has been reduced by the correct amount. The problem here is that there is no guarantee that the **correct** items have been removed. Another issue is that the diagnostics will not give useful information when the code does not work correctly.

```
@Test
public void public void removesItemsPassedTheirSellByDate() {
  Item itemtoBeRemoved = itemWithSellIn(0);
  Item itemtoBeRetained = itemWithSellIn(1);
  List<Item> remainingItems =
   updateQualityOf(itemtoBeRemoved, itemtoBeRetained);
  assertEquals(1, remainingItems.size());
}


java.lang.AssertionError:
  Expected :1
  Actual  :2
```

Listing 2.6: Poor diagnostics when working with collections

To implement this code there would probably be a simple statement to check the current sellIn value, but getting this if statement wrong would let this test pass as there would still only be a single value in the collection. Matching libraries have powerful mechanisms designed for working with collections and giving the power to match exactly what is required and no more.

```
@Test
public void public void removesItemsPassedTheirSellByDate() {
```

```
    Item itemtoBeRemoved = itemWithSellIn(0);
    Item itemtoBeRetained = itemWithSellIn(1);
    List<Item> remainingItems =
     updateQualityOf(itemtoBeRemoved, itemtoBeRetained);
    assertThat(remainingItems, containsInAnyOrder(itemtoBeRetained));
  }


  java.lang.AssertionError:
    Expected: iterable over [<Item selling in 1>] in any order
    but: Not matched: <Item selling in 0>
```

Listing 2.7: Useful diagnostics when working with collections

Again, the rich diagnostics produced can help pinpoint the problem in the production code without having to read the test first.

7 I've worked on such systems and going back to anything less is like going back to the dark ages
8 For more information see *http://en.wikipedia.org/wiki/Continuous_delivery*
9 For more information on the red-green-refactor cycle see *http://www.jamesshore.com/Blog/Red-Green-Refactor.html*
10 This isn't limited to test-after, I have also seen TDD practitioners make this mistake
11 *http://c2.com/cgi/wiki?ArrangeActAssert*
12 *http://hamcrest.org*

3 Test Coverage

The previous chapter outlined how a comprehensive test suite can enable you to deliver software faster by allowing production code to be automatically regression tested. This fast-feedback cycle can mean the difference between finding a bug within a few seconds on a development machine or waiting until it is released and having users find it. Depending on the type of software being delivered, the identification of bugs in production can be massively expensive[13].

3.1 Not testing enough

One of the questions for someone new to automated testing is "what should I write tests for?". This can be answered naively by stating "everything", but testing everything can all too easily lead to wasting time writing tests that aren't needed. A much better answer to this question is "test anything that can reasonably break"[14]. A common mistake seen (typically among more senior developers) is that the simpler the code, the less likely it is for it to break, but experience has shown the more complicated code is less likely to break purely because its complexity leads to taking more care.

The code in listing 3.1 is a fairly typical example of code that might be deemed too simple to break[15].

```
public static String join(String[] input) {
   String result = "";
   for (String value : input) {
      result += value + ",";
   }
   return result.substring(0, result.length());
}
```

Listing 3.1: Code considered too simple to break

As it stands this code is functionally correct and poses no problems. Unfortunately in Java this code will perform very badly given a sufficiently large input array. Modifying this code to a more performant version is a simple change by anyone's standard and the resulting code is shown in listing 3.2.

```
public static String join(String[] input) {
   StringBuilder result = new StringBuilder();
   for (String value : input) {
      result.append(value).append(",");
   }
   return result.toString();
}
```

Listing 3.2: Performant but broken code

At first glance the code seems to be ok, but leaves a trailing comma at the end of the result. This is an all-too-easy mistake to make, especially when in a rush or when tired[16]. Even a single happy day scenario would be enough to catch this regression, but without such a test, the error will not come to light until later on in the development cycle. Because this code is utility code, it is likely to be used in many places and finding the bug could be time consuming.

Chapter 1 discussed how well named tests can act as documentation for a system, describing how code should work, but in this case the presence of additional tests to the happy day scenario help describe how code performs when unexpected inputs are provided, for example how an empty or null is handled or how empty strings within the array will affect the output.

3.2 Testing Too Much

The previous section showed how even the simplest of code should have unit tests, but sometimes this can be taken too far and tests are written for code that **can never break**. The worst violations of this are tests that verify the runtime itself. For many years the FizzBuzz[17] test has been used to screen candidates for recruitment and it is not uncommon to see tests that verify that the JVM can perform math or that object allocation itself works!

```
@Test
public void fizzBuzzWorks() {
   assertTrue(15 % 3 == 0);
}


@Test
public void fizzBuzzCanBeInstantiated() {
   assertNotNull(new FizzBuzz());
}
```

Listing 3.3: Tests that can never fail

The tests shown in listing 3.3 can never ever fail[18]. With the first test something has to be wrong with the fundamentals of basic arithmetic and you have much bigger problems. With the second test an OutOfMemoryError will be raised and the runtime will be borked. Notice how it is not possible to give this test a name describing the behaviour under test- this can be a good indicator that the test is useless. Another indicator is that you are trying to test code you didn't write, or that no code has to even exist for the test to pass.

One area of code that unit tests should be skipped for is properties in languages that do not support them natively, i.e. field assignment that has to be done by hand. Although you do have to write code to support this it really is too simple to break and a failure to perform this assignment properly should manifest itself by breaking other tests that actually use the data. Modern IDEs can automate the assignment of fields in constructors and creation of getters/setters anyway.

```
@Test
public void public void constructsItemCorrectly() {
   Item item = new Item("Item Name", 5, 10);
   assertEquals("Item Name", item.getName());
   assertEquals(5, item.getSellIn());
   assertEquals(10, item.getQuality());

   item.setQuality(15);
   assertEquals(15, item.getQuality());
}
```

Listing 3.4: Tests for code that is too simple to break

To implement the code behind the test in listing 3.4 it is possible to not manually write a single line of code, it can all be handled by the IDEs refactoring support. There are tools available to automate the testing of field assignment, but I would question the value of automating both the testing and production code!

Writing boilerplate code can be a killer to creativity and productivity, and this is true in unit tests as well as production code. An area this often manifests itself is testing that exceptions are raised. A quick google for how to test exceptions will show many examples like that shown in listing 3.5.

```
@Test
public void willRaiseException() {
   try {
      codeThatRaisesException();
      fail("exception not raised");
   } catch (ExceptionToBeRaised e) {
      // success!
   }
}
```

Listing 3.5: Unnecessary boilerplate in a test

In some of the older unit testing frameworks this was indeed the only way to handle exceptions, but most frameworks now provide built-in mechanisms for this. In JUnit this is provided via the "exception" parameter to @Test.

```
@Test(expected = ExceptionToBeRaised.class)
public void willRaiseException() throws Exception {
   codeThatRaisesException();
}
```

Listing 3.6: Removal of boilerplate

There is a caveat to using this approach over the previous version, this test will pass if the expected exception is raised anywhere within the test body and not just from the line that you're expected it to be raised from.

3.3 Coverage

A popular technique for identifying how much code has been tested is the use of a code coverage tool. These work by reporting which lines of code are executed while tests are running. The more advanced tools will also report branch coverage (how many branches in an if-statement are executed) and calculate cyclomatic complexity.

When used correctly, coverage tools can be an invaluable asset but unfortunately my experience has shown they are regularly used incorrectly. Chapter 2 showed how tests need to be verified to ensure they work correctly and how if this step is omitted it is possible to have tests that do not actually work. Because broken tests will still execute a production code, a coverage report will produce a higher percentage than is strictly correct. The tests shown in listing 3.7 when run with a coverage tool report that the FizzBuzz class has 100% line and branch coverage, when in fact the production code can be changed in any way with no test breakages! The production code has been *covered* but not actually *tested*.

```
@Test
public void byNotAssertingTheMultiplesOfThreeBranch() {
   FizzBuzz.of(3);
}

@Test
public void byNotAssertingTheMultiplesOfFiveBranch () {
   FizzBuzz.of(5);
}

@Test
public void byNotAssertingTheMultiplesOfFifteenBranch () {
   FizzBuzz.of(15);
}

@Test
public void byNotAssertingTheDefaultBranch () {
   FizzBuzz.of(1);
}
```

Listing 3.7: Tests that will produce 100% code coverage but not verify the code under test works

When used in conjunction with test verification and either code reviews or pair programming, coverage is a highly effective way of identifying areas of code that are missing tests[9]. In a system where you have a high level of confidence in your tests, coverage can also be employed to identify dead code, i.e. code that will no longer be executed.

When teams first start to use a coverage tool on an existing yet poorly tested codebase, it is all too easy to start "gaming the system" to try and get numbers up. This becomes worse when a target coverage rate is mandated. Testing complex code will be time consuming for little improvement in numbers so in order to raise numbers faster, the easier targets are picked. I have heard stories of teams doing tricks such as automating

the testing of getters and setters of all objects because this can give a massive jump in coverage numbers. Unfortunately this can give a false sense of security as the coverage numbers are high, but the actual quantity of code tested is still very low. In this situation, high value code[20] should be identified and coverage used to ensure tests are put in place and remain in place.

13 *http://bit.ly/costofbugs*
14 *http://junit.sourceforge.net/doc/faq/faq.htm#best_3*
15 This is an example from a real system that I worked on in 2009 that had no unit tests around this code
16 This really happened and the code was being patched at 2am!
17 *http://c2.com/cgi/wiki?FizzBuzzTest*
18 I have used FizzBuzz as a screening test previously and seen these kinds of tests pop up many times
19 I would argue that if you are a TDD practitioner, coverage tools are of limited use as you should never be writing a line of code unless a test has forced it into existence
20 Code that is mission critical or that changes often

4 Test Cleanliness

The presence of unit tests can be essential for performing worry free refactoring on production code. The ability to completely change production code and nearly immediately know that it still works correctly is a huge boost to ensuring code can be kept clean and reduce future maintenance costs. For those who use a TDD development methodology, the refactoring step is a mandatory part of the process. Refactoring and the principles of clean code are not only applicable to production code though, it is just as important to keep test code clean and maintainable. Test code will live as long as production code so should be treated with just as much care, if not more[21].

One of the cornerstones of clean code are the SOLID[22] principles, of which the first is the "Single Responsibility Principle". This states that a class should have one and only one reason to change. When applied further, this means that a method of that class should perform a single subset of the responsibility of the class. When applied to production code this generally leads to small classes composed of small methods. When applied to automated tests this means that a test should verify a single piece of behaviour only, but all too often I see this principle violated heavily.

Although I have no evidence, I believe this to be because behaviour is, or at least should be defined differently for production code and test code. For example, the GildedRose code has the method updateQuality() which ages the items currently held. It could easily be argued that this is a single piece of behaviour, at least from the public API point of view, but the tests to back this up will be verifying all of the different code paths, leading to multiple "behaviours". Because the public API exposes this as a single behaviour, and indeed a single method, a common approach is to test this from within a single test method[23]. Taking this approach with the GildedRose would yield a test method looking like that in listing 4.1.

```
public class GildedRoseWill {
   @Test
   public void ageItems() {
      List<Item> items = new ArrayList<Item>();
      items.add(new Item("+5 Dexterity Vest", 10, 20));
      items.add(new Item("Aged Brie", 2, 0));
      items.add(new Item("Elixir of the Mongoose", 5, 7));
      items.add(new Item("Sulfuras, Hand of Ragnaros", 0, 80));
      items.add(new Item("Backstage passes to a TAFKAL80ETC concert", 15, 20));
      items.add(new Item("Conjured Mana Cake", 3, 6));
      GildedRose gildedRose = new GildedRose(items);
      gildedRose.updateQuality();
      assertEquals(19, items.get(0).getQuality());
      assertEquals(9, items.get(0).getSellIn());
      // "Aged Brie" increases in Quality the older it gets
      assertEquals(1, items.get(1).getQuality());
      assertEquals(1, items.get(1).getSellIn());
      assertEquals(6, items.get(2).getQuality());
      assertEquals(4, items.get(2).getSellIn());
      // "Sulfuras" never has to be sold or decreases in Quality
      assertEquals(0, items.get(3).getSellIn());
      //  ... or decreases in Quality
      assertEquals(80, items.get(3).getQuality());
      // backstage passes increase in quality
      assertEquals(21, items.get(4).getQuality());
      assertEquals(14, items.get(4).getSellIn());
      // conjured items degrade twice as fast
      assertEquals(4, items.get(5).getQuality());
      assertEquals(2, items.get(5).getSellIn());


      items = new ArrayList<Item>();
      items.add(new Item("+5 Dexterity Vest", 10, 0));
```

```
        gildedRose = new GildedRose(items);
        gildedRose.updateQuality();
        assertEquals(0, items.get(0).getQuality());


        items = new ArrayList<Item>();
        items.add(new Item("Aged Brie", 2, 50));
        gildedRose = new GildedRose(items);
        gildedRose.updateQuality();
        assertEquals(50, items.get(0).getQuality());
    }
}
```

Listing 4.1: A test verifying the updateQuality() method on GildedRose.

In order for this to completely cover all use cases, it would be much bigger than it is here[24], but already this test case is much bigger than it should be. The comments added should be a clue that multiple behaviours are being tested at once and that this should in fact be split up into multiple tests, all living as part of the same suite. In Chapter 1, the importance of naming was explained, and in Chapter 2 the importance of verification. With tests that are like this, both of these principals are difficult to adhere to. The verification is especially troublesome, especially when a test fails. With unexpected test failures this becomes even more so due to the fact that a failure in an earlier part of the test will hide failures later on. Say the production code has a change so that the rules for Aged Brie are broken, the test will fail very early on. Other failures inside this test could give you vital clues about why this has failed, but this will not be apparent unless the failing tests are removed which slows down the feedback cycle. Tests like these also violate the O in SOLID – the Open/Closed Principle, which states that code should be closed for modification but open for extension. In the example above, when an ageing rule is added or removed the single large test method must be modified, and with any code that is modified, it should be retested, or in this case re-verified.

The key to avoiding large tests is to break them up into small, single, discrete tests focussing on a single piece of testable behaviour as shown in listing 4.2.

```
public class AgingAnItemWill {
    @Test
    public void increaseTheQualityOfAgedBrie() {
        List<Item> items = new ArrayList<Item>();
        items.add(new Item("Aged Brie", 2, 0));
        GildedRose gildedRose = new GildedRose(items);
        gildedRose.updateQuality();
        assertEquals(1, items.get(0).getQuality());
    }
}
```

Listing 4.2: A small test focussing on a single piece of behaviour

The naming of this test can help form part of the executable documentation and explains exactly what the action and behaviour is. Notice how the check for the sellIn value has been removed. This is intentionally done to show how this test only cares about the quality of Aged Brie and nothing else.

It is possible to take this principle too far, and a common misconception is that a test should only have a single assertion. Multiple assertions are fine as long as they are verifying a single outcome. As the number of individual asserts grows though, this is a smell that perhaps the test isn't focussed enough or that custom matchers should be employed.

4.1 Duplication

Breaking large tests into smaller tests will undoubtedly lead to a certain level of duplication within the test code and the normal rules of clean code apply here. All testing frameworks I have worked with have the notion of 'before' and 'after' constructs. These are places to execute code that will run before and after each test executes. Some frameworks also have the ability to execute code before and after all tests within a class.

```
public class AgingAgedBrie {
   @Test
   public void increaseTheQuality() {
      List<Item> items = new ArrayList<Item>();
      items.add(new Item("Aged Brie", 2, 0));
      GildedRose gildedRose = new GildedRose(items);
      gildedRose.updateQuality();
      assertEquals(1, items.get(0).getQuality());
   }


   @Test
   public void decreasesTheSellIn () {
      List<Item> items = new ArrayList<Item>();
      items.add(new Item("Aged Brie", 2, 0));
      GildedRose gildedRose = new GildedRose(items);
      gildedRose.updateQuality();
      assertEquals(1, items.get(0). getSellIn ());
   }
}
```

Listing 4.3: Tests with duplication

Code that is executed before every test method should always be moved into the common setup block

```
public class AgingAgedBrie {
   private List<Item> items;

   @Before
   public void setUp() throws Exception {
   items = new ArrayList<Item>(); gildedRose.updateQuality();
     items.add(new Item("Aged Brie", 2, 0)); gildedRose.updateQuality();
}

   @Test
   public void increaseTheQuality() {
      assertEquals(1, items.get(0).getQuality());
   }

   @Test
   public void decreasesTheSellIn() {
      assertEquals(1, items.get(0).getSellIn());
   }
}
```

Listing 4.4: Common setup for tests

The code in listing 4.4 uses the setup method to initialize the GildedRose with the required item, Aged Brie. Not only has this removed duplication within the tests but because each individual test no longer has this it is much easier to focus on exactly what they are focussed on.

This technique works perfectly for tests that require the exact same set-up for each fixture, but more often than not this is not the case and will require slightly different setup. The easiest way to handle this is to do whatever common setup can be done in 'before' code and perform ad-hoc initialization per test, but again following the rules of clean code. Typically this is handled with the use of syntactic-sugar, i.e. helper methods, as shown in Listing 4.5.

```
public class AgingAgedBrie {
   private List<Item> items;

   @Before
   public void setUp() throws Exception {
     items = new ArrayList<Item>();
   }
```

```
    @Test
    public void increaseTheQuality() {
       havingUpdatedGildedRoseWithAgedBrie(2, 0);
       assertEquals(1, items.get(0).getQuality());
    }

    @Test
    public void decreasesTheSellIn() {
       havingUpdatedGildedRoseWithAgedBrie(6, 4);
       assertEquals(5, items.get(0).getSellIn());
    }

    private void havingUpdatedGildedRoseWithAgedBrie(int sellIn, int quality) {
       items.add(new Item("Aged Brie", sellIn, quality));
       GildedRose gildedRose = new GildedRose(items);
       gildedRose.updateQuality();
    }
}
```

Listing 4.5: Setup per test

The key here is to perform merciless refactoring on test code to remove duplication. In Listing 4.5 the code is mostly clean, but both tests still access the list item at element zero. This might not jump out immediately as duplication[25], but as your test suite grows, refactoring this will pay dividends when things inevitably change.

```
    @Test
    public void increaseTheQuality() {
       havingUpdatedGildedRoseWithAgedBrie(2, 0);
       assertEquals(1, agedBrie().getQuality());
    }

    @Test
    public void decreasesTheSellIn() {
       havingUpdatedGildedRoseWithAgedBrie(6, 4);
       assertEquals(5, agedBrie().getSellIn());
    }

    private Item agedBrie() {
       return items.get(0);
    }
```

Listing 4.6: Removal of duplication

Listing 4.6 is an example of how the list access duplication can be removed. A change to how to access the Aged Brie is now confined to a single place so if this changes it is not necessary to update several tests. The individual tests know nothing about how to instantiate the GildedRose or how to inspect it to get at the element they want. This is nice because it makes refactoring a lot easier and leads to easier to read tests as the language used can focus on behaviour, not implementation.

TIP: One of the most obvious signs that tests contain duplication is the "ripple effect" – a change to either production or test code causes multiple tests or suites to not compile or fail. It is all too easy to patch up the tests, but this should be taken as a perfect opportunity to remove that duplication

---

Duplication across tests can be a useful tool for identifying if the fixtures are cohesive or not. Within a single test class, slightly different setup per test is a sign that slight variations on the same thing are being tested. Wildly different setup or setup that is common among a few tests but not others is an indicator that the suite is not cohesive and should be broken up further. There is a belief there should be one-to-one mapping between test classes and a production class or method, but this is not the case at all[26]. Given Listing 4.6, if another test were added that constructs the GildedRose in a different way or used an object that is not AgedBrie, it should not be in that suite, but in a suite of its own. If tests are named well, the names themselves might be a clue, i.e. a prefix common among some tests but not others are candidates for a separate test suite.

Duplication is not confined to the test code written, or how to use it but also using production code within tests, for example accessing the quality of a GildedRose item. As the number of tests grow, so does the use of production code and again introduces a high level of duplication. Take the code in Listing 4.7 as an example.

```
    @Test
    public void increasesTheQualityOfBackstagePasses() {
      assertEquals(21, backstagePass().getQuality());
    }
    @Test
    public void increaseTheQualityOfAgedBrie() {
       assertEquals(1, agedBrie().getQuality());
    }

    @Test
    public void willNotAffectTheQualityOfSulfuras() {
       assertEquals(80, sulfuras().getQuality());
    }

    @Test
    public void decreasesTheQualityOfConjuredItemsTwiceAsFast () {
       assertEquals(4, conjuredItem().getQuality());
    }
}
```

Listing 4.7: Multiple tests accessing production code in the same way

These tests are all concerned with the quality of an item and hence call the .getQuality() method in each test. It is easy to argue that this is not duplication and should not be removed, especially in languages where refactoring can be applied with an IDE. Problems arise though when there is a change that cannot be automatically applied, or the API changes in such a way that it ripples out into the tests. Duplication within tests can be easily solved with more syntactic sugar.

```
    @Test
    public void increaseTheQualityOfAgedBrie() {
       assertEquals(1, qualityOf(agedBrie()));
    }
```

Listing 4.8: Removed of production code logic

Splitting tests up into multiple classes or performing the same production logic in different tests is harder to solve. The most common way I have seen of doing this is to introduce "helper methods". All testing frameworks I have used have a much more flexible solution to this problem, matchers, which were introduced in Chapter 2.

The Hamcrest library[27] has a concept known as a "Feature Matcher", which allows the combining of field extraction from an object and applying that to a chained matcher. An example of this is shown in listing 4.9.

```
    @Test
    public void increaseTheQualityOfAgedBrie() {
       assertThat(agedBrie(), hasQuality(equalTo(2)));
    }

    private Matcher<? super Item> hasQuality(Matcher<Integer> matcher) {
       return new FeatureMatcher<Item, Integer>(matcher, "quality", "quality") {
          @Override
          protected Integer featureValueOf(Item item) {
             return item.getQuality();
          }
       }
    }
```

Listing 4.9: Removal of duplication using a feature matcher

The duplication of the getQuality() method has now been removed and centralised within a single test, but by extracting this to another class it is reusable across as many tests as need it. Feature matchers require the use of a submatcher which it delegates the work to, in this case equalTo. This mechanism allows a nice combination of feature extraction and matching without having to duplicate work. When this test fails the diagnostics produced are nicer than those produced with a raw equality check.

```
Expected: quality <2>
   but: quality was <1>
```

Listing 4.10: Diagnostics produced by a feature matcher

In verbose languages such as Java, feature matchers make sense when the effort of removing duplication dictates it, but other languages solve the problem in more succinct ways thus should be used more aggressively. For example using Scala in conjunction with the Specs2 testing library yields the code shown in Listing 4.11.

```
  "aging items" should {
     "increase the quality of aged brie" in {
        agedBrie must haveQuality(===(2))
     }

     def haveQuality(matcher: Matcher[Int]): Matcher[Item] = {
        matchA[Item].quality(matcher)
     }
  }
```

Listing 4.11: Feature matcher using Scala and Specs2

4.2 Duplicating Production Logic

There is another case of duplication that is often overlooked- duplicating production logic within tests. This occurs when an attempt is made to cover all possible inputs instead of choosing meaningful and interesting examples. A classic example of this is the FizzBuzz problem. Trying to test all 100 outputs can lead to performing the same looping and logic within the test as shown in Listing 4.12.

```
@Test
public void calculatesAllValuesCorrectly() {
   for (int i=1 ; i<101 ; i++) {
      if (i % 3 == 0) {
         assertEquals("Fizz", FizzBuzz.of(i));
      } else if (i % 5 == 0) {
         assertEquals("Buzz", FizzBuzz.of(i));
      } else if (i % 15 == 0) {
         assertEquals("FizzBuzz", FizzBuzz.of(i));
      } else {
         assertEquals(Integer.toString(i), FizzBuzz.of(i));
      }
   }
}
```

Listing 4.12: Test duplicating production logic

By writing these kinds of tests it is difficult to give them meaningful names as the test becomes a test of all cases, not discrete pieces of behaviour. These tests are also near impossible to verify as there are in reality 100 test cases wrapped until a single test. In addition to this, a production code failure is also hard to track down as the first failure will mask all others.

The biggest issue by far in duplicating production logic into tests is that a mistake made in the test can easily be made in the production logic too. Without the ability to see the test failing it is possible for this to go completely unnoticed. The previous test is invalid as the "FizzBuzz" branch can never be hit (the "Fizz" and "Buzz" branches mean it is dead code). The worst case scenario is for this to be copied and pasted into pro-

duction code and have the assertions replaced with returns! The golden rule to follow is that tests should use concrete data, pre-calculated and hard-coded. When this is used in conjunction with test isolation and good naming the test would resemble something more like that shown in listing 4.13.

```
@Test
public void fizzForMuliplesOfThree() {
   assertEquals("Fizz", fizzBuzzOf(3));
}

@Test
public void buzzForMuliplesOfFive() {
   assertEquals("Buzz", fizzBuzzOf(5));
}

@Test
public void fizzBuzzForMuliplesOfThree() {
   assertEquals("FizzBuzz", fizzBuzzOf(15));
}

@Test
public void theInputForEverythingElse() {
   assertEquals("7", fizzBuzzOf(7));
}
```

Listing 4.13: FizzBuzz tests with rules of isolation, no-duplication, naming and verification applied

Ignoring boilerplate, these tests now have less actual lines of code and apply several of the techniques outlined elsewhere in this book. For those interested in code complexity, the V(G) value has dropped from 5 to 1.

The duplication inherent here can be further reduced by the use of examples[28], which allow the same test to be executed multiple times for a given set of inputs. Whereas these are useful for cutting down on duplication they can lose some of the richness of naming tests discretely when the behaviour needs to be called out explicitly. An example for FizzBuzz using Specs2 is showing in listing 4.14.

```
class FizzBuzzSpec extends Specification with DataTables {
   "fizz buzz should return the correct value" ^ {
      "input"   | "output"      |
        3       !  "Fizz"       |
        5       !  "Buzz"       |
        15      !  "FizzBuzz"   |
        7       !  "7"          |>
      { (input, output) => FizzBuzz.of(input) must_== output }
   }
```

Listing 4.14: A Specs2 test using a data table

A special case of production logic duplication is that where a test makes use of a production method but that method can later be changed to mean something else. Because tests concern themselves with asserting things that are equal this problem usually manifests itself by overriding the meaning of "equals" in languages that support it as a native construct[29]. With the GildedRose Item class, there may be tests that want to verify all fields are set correctly after some logic has been performed- or more likely only a subsets of fields need to be checked but using equality is quicker and easier than writing a custom matcher[30].

```
public boolean equals(Object o) {
   if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
   Item item = (Item) o;

   if (quality != item.quality) return false;
   if (sellIn != item.sellIn) return false;
   if (name != null ? !name.equals(item.name) : item.name != null) return false;
   return true;
}
```

Listing 4.15: IntelliJ autogenerated equals() method for the Item class

The problem here will not immediately manifest itself as the equality method and the tests are in sync. The only issue at first may be poor diagnostics in older test frameworks. In the future, the equality method may change to have business meaning, and only include a subset of fields in the check. With the Item class this may become a check purely on the item name and tests that expect all fields to be the same will carry on working perfectly as the constraints are still met. If a bug is introduced in production logic these tests may not fail due to the fact the assertion contract is now weaker than it originally was.

21 You can rewrite your entire production code and let the tests verify the new code is correct, but you cannot throw away your test code and do the same!
22 *http://bit.ly/whatissolid*
23 For an example of this in the wild, check out the CookieTest.java in the GWT codebase
24 Fully implementing the all the tests was sapping my will to live
25 I've had a lot of arguments over the years on removing this level of duplication!
26 I believe this stems from naming test classes the same as the production classes with "Test" at the end, a throwback to the early test runners
27 Available for all major languages
28 Also know as data tables or data points
29 In Java this would be overriding the equals() method; Scala and Ruby allow the overriding of the '==' operator.
30 Especially when using an IDE that allows the generation of such methods

5 Isolation

No matter if you believe a unit test to be a single method/class or several classes working together, tests must run fast and be predictable. In order to achieve this it is necessary to isolate both the test and production code from the outside world, e.g. file systems and databases. It is also necessary to control dependencies that have natural variance such as a random number generator. A sign that unit tests or code are not properly isolated are tests that fail, but when re-run pass without any code being changed ("flaky"), trying to force your tests to run in a particular order or running them X number of times and ensuring a certain percentage pass. Slow and/or flaky tests can lead to the sort of pain that stops tests being run, either on a developer's machine or in a continuous integration environment.

One area this rule is usually violated is in modifying system state, that is, state that is maintained system wide and not confined to the local object graph. Prime examples of this are code that use system time[31], singletons[32] or use an external resource like a database. The fact that these things are hard to test or are not completely deterministic is the codes way of signifying that dependencies are not explicit and that production code is too tightly coupled. Running unit tests in parallel is a sure fire way to catch these kind of isolation problems early on.

When I first started with unit testing, following these guidelines seemed like overhead and introducing unnecessary abstractions. For example, code that would read data from a file had to be written in such a way that it knew nothing of files and needed to accept data in a different way. In Java this meant using an InputStream[33] so that it could be replaced for testing. This of course meant that some other piece of code was responsible for the opening of the file and creating the InputStream and doing all the necessary checks that came with it. Although unit testing forced the separation, the end result was two classes that much better adhered to the Single Responsibility Principle – a class that would handle file semantics, which was reusable anywhere this would happen, and a class that could read its data from any source that implements InputStream.

5.1 Controlling Collaborators

The GildedRose has the notion of ageing items of which some can ultimately end up with a value of zero. In the real world (or at least a computer simulated game world) the GildedRose would need to replenish its stock in order to keep on trading. Imagine a 'merchant' travels to the GildedRose with items to sell and there is some business logic to decide which items to buy (if any). Within the tests this would be a dependency needing to be controlled, as presumably the stock of a merchant would vary and there would be a need to verify all scenarios.

Any developer writing unit tests, at some point or another starts using a mocking framework for isolating dependencies. These are very clever and flexible libraries that allow the programmatic control of dependencies at runtime and are typically the go-to choice for providing a controllable merchant. To see an example, the requirements for GildedRose have changed and in order to replenish sold or out of date stock a merchant will visit with items available to purchase. The GildedRose must ask the merchant what items are for sale and make purchasing decisions based on the list returned as shown in listing 5.1.

```
public void acceptVisitFrom(Merchant merchant) {
   for (Item item : merchant.itemsForSale()) {
      addItemToStock(item);
   }
}
```

Listing 5.1: GildedRose adding purchased items to stock

For this example, assume that the GildedRose will purchase all items the merchant has for sale and payment is handled elsewhere.

Using a mocking framework like jMock the test to verify that items available have been added to the stock would look something like that shown in listing 5.2.

```
public class GildedRoseWillNotBuyFromMerchant {
   @Rule public final JUnitRuleMockery context = new JUnitRuleMockery();
   @Mock private Merchant merchant;

   @Test
   public void whenThereAreItemsToBuy() {
      final List<Item> itemsForSale = noItems();
      context.checking(new Expectations() {{
         oneOf(merchant).itemsForSale();
            will(returnValue(noItems));
      }});
      GildedRose gildedRose = gildedRoseWith(startingItems());
      gildedRose.acceptVisitFrom(merchant);

      // verify no items added to stock
}}
```

Listing 5.2: Using mocks to verify interactions between the GildedRose and Merchant

The mocks here state that when the acceptVisitFrom(merchant) method is called there must be one and only one call to merchant.itemsForSale() and when that call happens the specified list of items will be returned. To any developer experienced with mocking frameworks this is standard fare, but what is less common is the knowledge that there are other kinds of test doubles[34] available. The most useful of these are the stub and fake. Stubs are implementations of a class that provide nothing but canned responses. The classic xkcd comic about random numbers[35] is a perfect example of a stub implementation. For the GildedRose Merchant interaction a stub would always return the same list of items no matter where or when it is called.

```
public class GildedRoseWillNotBuyFromMerchant {
   @Test
   public void whenThereAreNoItemsToBuy() {
      GildedRose gildedRose = gildedRoseWith(startingItems());
      gildedRose.acceptVisitFrom(new MerchantWithNoItems());

      // verify no items added to stock
   }

   private class MerchantWithNoItems implements Merchant {
      public List<Item> itemsForSale() {
         return Collections.EMPTY_LIST;
      }
   }
}
```

Listing 5.3 Using a stub to verify interactions between the GildedRose and Merchant

Due to the nature of always doing the same thing, the use of stubs is fairly limited, although even in this simple example it can be seen that the actual code in the test itself is a lot cleaner. One area they are really useful is when testing exceptions, as method implementations can just raise an exception and nothing more- so for the Merchant an ExceptionRaisingMerchant stub would be effective.

A more flexible approach is to use a fake, which is an implementation that short circuits the real logic but can still make some decisions. The easiest fake for the Merchant is one that can be programmed to return a list of items at runtime.

```
public class GildedRoseWillNotBuyFromMerchant {
   @Test
   public void whenThereAreNoItemsToBuy() {
      GildedRose gildedRose = gildedRoseWith(startingItems());
```

```
        gildedRose.acceptVisitFrom(
           new MerchantWithItems(itemsForSale())

        // verify no items added to stock
     }

     private class MerchantWithItems implements Merchant {
        private final List<Item> items;

        public MerchantWithItems (List<Item> items) {
           this.items = items;
        }

        public List<Item> itemsForSale() {
           return items;
        }
     }
  }
```

Listing 5.4: Using a fake to verify interactions between the GildedRose and Merchant

With a single test the use of mocks might seem the more attractive, especially when using Java where the verbosity of the language can lead to excessive boilerplate. Mocking frameworks are extremely useful in some scenarios but more often than not the choice of a stub or fake would be more prudent.

Because mocking frameworks typically require the setting up of expectations and returning of values before code is executed and use their own language, mocking can start to get ugly and detract from behavioural concerns in a test very quickly. This can easily be handled by applying some syntactic sugar, i.e. extracting it to well named methods as showing in listing 5.5.

```
public class GildedRoseWillNotBuyFromMerchant {
   @Rule public final JUnitRuleMockery context = new JUnitRuleMockery();
   @Mock private Merchant merchant;

   @Test
   public void whenThereAreNoItemsToBuy() {
      GildedRose gildedRose = gildedRoseWith(startingItems());
      gildedRose.acceptVisitFrom(
         merchantWith(itemsForSale()));

      // verify no items added to stock
   }

   private Merchant merchantWith(final List<Item> itemsForSale) {
      context.checking(new Expectations() {{
         oneOf(merchant).itemsForSale();
            will(returnValue(itemsForSale));
         }});
      }
   }
}
```

Listing 5.5: Extracting mock setup into a private method

Not only does this remove detail from the test and put it back to reading like a spec, but as per the usual rules of removing duplication, the decision to use a fake or mock can be changed easily without the actual tests having to know.

By far and away the biggest problem with the overuse of mocks is the fact that they implicitly tie the test code to the implementation of the production code, and this is something that should be avoided wherever possible. With the GildedRose example it is easily to fall into the trap of believing that the interaction with the merchant is something that needs to be verified, but it isn't! In this case the interaction is incidental and the behaviour to be verified is actually that the stock has been updated correctly. This is something that can be difficult to get right and takes practise, but the key is to think about what is being verified within the test.

If there are expectations on the mock and assertions done on the resulting state or return value that might be an indicator that the interaction with the collaborator is not the important part of the test and a candidate for replacing with a different type of test double.

Changing how the production code interacts with collaborators is a time at which the correct choice of test double becomes most important. To illustrate this, the Merchant API will change to accept a "shopping list" when returning the list of items for sale. This is just a list of item names[36] that the GildedRose is interested in and only items within that list should be returned by the merchant, as shown in Listing 5.6.

```
public void acceptVisitFrom(Merchant merchant) {
   List<Item> itemsForSale =
      merchant.itemsForSale(outOfStockItems());
   for (Item item : itemsForSale) {
      addItemToStock(item);


   }
}
```

Listing 5.6: GildedRose passing a shopping list to the merchant

In terms of code and test changes this should be extremely simple and something that should only take a couple of minutes to do, but when mocks are used to control collaborators, the updating of tests can take orders of magnitude more time, especially when removal of duplication hasn't been applied. This is a time when developers (and managers) can become frustrated with unit testing due to the fact that simple code changes end up taking much longer than they should and ends up with lots of complaints about the time taken for test maintenance.

Because the mocks are very strict in what they expect, everywhere an expectation is placed on the itemsForSale() method call it must be updated to reflect the change in API, even if that change is to instruct the mock to expect "any" value. With the use of stubs or fakes the API change is isolated to a single place- and with intelligent IDE refactoring you might not even have to do anything by hand.

```
private class MerchantWithItems implements Merchant {
      private final List<Item> items;

      public MerchantWithItems (List<Item> items) {
         this.items = items;
      }

      public List<Item> itemsForSale(List<String> items) {
         return items;
      }
   }
}
```

Listing 5.7: Refactored fake Merchant that ignores the shopping list

Using Java and IntelliJ, the refactoring applied to add the shopping list of the merchant API resulted in only the tests using mocks having a compilation failure. The tests making use of a stub and a fake not only compiled but executed correctly with no manual work to be done.

The argument against this kind of flexibility is that you want code not to compile and tests to fail when an API changes, and to some degree this is true, but only for a limited set of tests. With the merchant API change there only really needs to be a single test that verifies the correct shopping list is passed to the merchant with the rest of the tests focussing on the return value. This test can still be written with a fake that provides default behaviour in the event no expected shopping list has been provided. One very important rule to adhere to though is that fakes should not get complicated. Some simple logic is acceptable, but if they grow to a point where there is a need to test the fakes then it is a sign that the interactions are complicated enough to warrant using a mock or refactoring the production code to remove complexity.

One place to absolutely never use a mock is for value objects/structs. These are objects that typically provide no behaviour and thus can just be used directly within tests. Remember that isolation does not have to mean

replacing every dependency with a test double, sometimes it is more than acceptable to use a real dependency if it gets the job done. If the real implementation of the GildedRose Merchant only has a list of items and returns that list, there is no point implementing a fake version to do exactly the same thing!

There are some circumstances when using a mock is the correct choice, which is usually when a call to a collaborator absolutely must happen. This typically takes the form of an event that changes the world, e.g. saving state to a database. To illustrate this the GildedRose will again have the requirements changed so that when some items have passed the sell by date they must be recycled. For this a collaborator will be introduced that handles this- a RecyclingCentre that expects a list of items that are to be recycled. Recycling is done at the end of every day so it forms part of the updateQuality() method on the GildedRose.

```
public class GildedRose {
      private List<Item> items;
      private RecyclingCentre recyclingCentre;

      public GildedRose(RecyclingCentre recyclingCentre, List<Item> items) {
         this.recyclingCentre = recyclingCentre;
         this.items = items;
      }

      public void updateQuality() {
         // existing updating quality code

         recyclingCentre.recycle(itemsToBeRecycled());
      }
}
```

Listing 5.8: The GildedRose interacting with the RecyclingCentre

Not only is a mock required because we have to ensure this happens, the fact that this method has no return value means that there is no state to verify within the GildedRose itself. The interaction with the RecyclingCentre is no longer incidental, resulting in a state change on the GildedRose, this is an explicit interaction and hence must be verified. Testing this could be handled with a fake, but the amount of work required to implement the fake outweighs the use of a mock. Also, if the interaction requires that this happens once and only once the amount of logic in the fake increases – this is one of the reasons that mock frameworks were invented in the first place[37].

The introduction of the recycling centre modifies existing logic – that of the updateQuality method, and all existing code will fail to compile because of the need to introduce the RecyclingCentre into the constructor of the GildedRose. The temptation here might be to introduce the mock into an existing set of tests and verify the logic there, but this lead back on to brittle tests. The best approach here is for the existing quality updating tests to use a stub RecyclingCentre as they are focussed on the ageing behaviour, not the recycling behaviour. This should be a few seconds worth of coding to get the tests back to green. A new set of tests should be introduced purely for verifying the RecyclingCentre interaction which can use a mock. Although both sets of tests are calling the same method on the GildedRose they are testing different behaviours and should be isolated as such.

5.2 3rd Party Interfaces

A golden rule that should be followed is to never, ever mock an interface or class that you don't own. As well as suffering from the problem of tying your tests to the implementation as previously discussed, you are at the mercy of the API owner as to what the method signatures are. The upgrade of a library could change the API and leave you with tests that no longer compile. As well as this, mocking a 3rd party API only verifies that you use API the way you think it should be used which itself could be wrong, but your tests will give you the false confidence that your code is correct.

There are two solutions to this problem and the correct one to choose is context dependent. The first instance is where an acceptable real class exists that can be substituted. In Java there is a class called BufferedReader that *"Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of charac-*

ters, arrays, and lines"[38]. It is possible to mock out the use of this class, but instead there are implementations that exist that can be used directly within the test. Both solutions are shown in listing 5.9.

```
@Test
public void shouldNotBeMocked() throws Exception {
   final BufferedReader reader =
      context.mock(BufferedReader.class);
      context.checking(new Expectations() {{
         atLeast(1).of(reader).readLine();
            will(onConsecutiveCalls(
               returnValue("first line"),
               returnValue("second line")));
      }});
      // execute code under test
   }
}


@Test
public void shouldUseARealReader() throws Exception {
   BufferedReader reader = new BufferedReader(
      new StringReader("first line\nsecond line"));
   // execute code under test
}
```

Listing 5.9: Alternative to mocking 3rd party classes

In this example, not only is the code cleaner and more concise but does not dictate how the code under test interacts with the BufferedReader and is hence less brittle.

The other example of when not to mock 3rd party interfaces is when a unit test is the wrong kind of test to perform. This is best exemplified when interacting with 3rd party systems, e.g. a database. Not only is it nearly impossible to get the order of execution correct with a mocking framework[39] but again does not verify the code is doing what you want it to do, only what you think it should do. For these kinds of tests the best thing to do is to actually write an integration test that interacts with the service in question. For a database this would be round-tripping to an in-memory database or a real database. The only way to be sure this code performs correctly is to exercise it.

31 Most languages have libraries available for mutating system time, please don't use them
32 This is the one of my many reasons for hating the Singleton pattern
33 This was back before Java had a rich streams API
34 *http://xunitpatterns.com/Test%20Double.html*
35 *http://xkcd.com/221/*
36 Obviously in real-life the use of such primitives should be avoided, but we'll keep it as strings for the sake of brevity
37 Steve Freeman, one of the authors of jMock explained that they wrote it because they spent so much time writing and maintaining fakes
38 *http://docs.oracle.com/javase/8/docs/api/java/io/BufferedReader.html*
39 I've tried this with JDBC and it's a complete nightmare

6 Miscellaneous

This chapter will cover other random pieces of advice that don't cohesively fit into the other chapters yet are still important

6.1 Test dependency order

An often-requested ability is to specify the running order of tests. When writing higher level tests this is fine as these are more stories, but unit tests, as has already been discussed, are supposed to fast running isolated tests. The need to specify running order usually signifies there is a shared state or previous state changing dependency happening. Shared state should be isolated using the techniques shown in Chapter 5. If there is a dependency on previous tests having been run it could be a sign that the test suite isn't cohesive enough and the rules for using common setup outlined in Chapter 4 should be applied.

6.2 Sleeping in tests

A guaranteed way to slow down a test run is to include sleep statements within the test itself. Typically this happens because the code under test is doing something asynchronously. A sleep is the fastest way to solve this issue but will cost in the long run. In addition to slowing down the test run, this technique often suffers from environmental issues, e.g. a 1 second sleep might be fine on a development machine but fail on a CI server. It can also lead to random failures depending on factors such as how much load the machine the test is being run on is under at the time.

The ideal solution to this is to separate out threading and business logic into separate concerns so that business logic can be tested in isolation. For example, with the GildedRose, this class knows nothing about how often to update the quality of items, only how to do it. It is assumed that other code will be in control of the actual scheduling and testing separately. Imagine having to wait 24 hours for the production code to run and test!

Sometimes the two concerns cannot be separate so other techniques must be used. The idea is to make the test pass as quickly as possible. If a sleep absolutely must be done, putting this into a tight loop with much shorter sleep durations is much more preferable than a single long-delay sleep. Remember that this is test code so a tight polling loop is more acceptable that it would be in production code. Ideally a control flow construct such as a semaphore should be used that allows the blocking of the test but will release the moment the production code is done.

6.3 Not all duplication is bad

Chapter 5 discussed the concepts of merciless refactoring and removal of duplication, but not all duplication should be removed. An often seen place the duplication should remain is sharing string values where the values actually have business meaning, for example, adding a HTTP header to a HTTP response as shown in Listing 6.1

```
public class HeaderAddingFilter implements ContainerResponseFilter {
  @Override
  public ContainerResponse filter(ContainerRequest request, ContainerResponse response)
  {
   response.getHttpHeaders().add("X-SomeHeader", "SomeValue");
   return response;
```

```
    }
}
```

Listing 6.1: Code to add a HTTP header to a HTTP response

The temptation here is to move the header name to a variable that the test can access under the misguided belief that if the header value changes it should only change in a single place. The problem here is that the header should have a very specific name and the test **should** fail if the name changes.

6.4 Testing private methods

The simple advice here is "don't". Feeling the need to do this is the tests way of signifying a problem with code design. Private methods are an implementation detail that should be able to change without breaking any tests. All tests should go through the public API of a class as that is what the production code uses. One way of avoiding doing this is to move the private logic into its own class and execute the tests via the public API of the new class, with the old class delegating. Doing this not only allows the new class tests to be focussed but opens up the old class for injecting a test double and controlling the functionality better with the overall effect being a more flexible design. When working with legacy code this may not be immediately possible and other techniques have to be applied. Enumerating the possibilities is outside the scope of this book, but the excellent "Working Effectively With Legacy Code" book by Michael Feathers is the bible for getting legacy code under test.

6.5 Strict assertions

As has been shown throughout this book, assertions are the backbone of any test suite, whether that be checking return values, state changes or within test doubles. Assertions should be made as flexible as possible while still verifying behaviour is correct. With mocking this can manifest itself as setting the cardinality of expectations correctly, for example, should a collaborator really only be called once or can it be called multiple times without a change to behaviour?

When checking values, the key is to only look for what you really want. With numbers this could be the difference between equalTo(10) and greaterThan(10). The former would fail for any value other than 10 whereas the latter will ensure the value is above 10. With strings it's important to know if the entire string is important or just a portion[40].

When working with object graphs the principles apply – only assert on what is important for that specific test. Checking an entire object graph can lead to the dreaded multiple test failure scenarios when the structure changes, e.g. adding a new field which no tests are expecting to be there. Feature Matchers (covered in Chapter 4) are an ideal way to avoid this problem.

40 Remember to use your language/assertion framework support for looking in strings rather than using indexOf checks

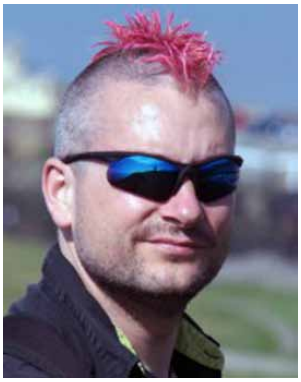
Appendix

The GildedRose Java Source Code

```
import java.util.ArrayList;
import java.util.List;

public class GildedRose {

    private static List<Item> items = null;

    /**
     * @param args
     */
    public static void main(String[] args) {

        System.out.println("OMGHAI!");

        items = new ArrayList<Item>();
        items.add(new Item("+5 Dexterity Vest", 10, 20));
        items.add(new Item("Aged Brie", 2, 0));
        items.add(new Item("Elixir of the Mongoose", 5, 7));
        items.add(new Item("Sulfuras, Hand of Ragnaros", 0, 80));
        items.add(new Item("Backstage passes to a TAFKAL80ETC concert", 15, 20));
        items.add(new Item("Conjured Mana Cake", 3, 6));

        updateQuality();
}


    public static void updateQuality()
    {
        for (int i = 0; i < items.size(); i++)
        {
            if ((!"Aged Brie".equals(items.get(i).getName())) && !"Backstage passes to a
                                        TAFKAL80ETC concert".equals(items.get(i).getName()))
            {
                if (items.get(i).getQuality() > 0)
                {
                    if (!"Sulfuras, Hand of Ragnaros".equals(items.get(i).getName()))
                    {
                        items.get(i).setQuality(items.get(i).getQuality() - 1);
                    }
                }
            }
            else
            {
                if (items.get(i).getQuality() < 50)
                {
                    items.get(i).setQuality(items.get(i).getQuality() + 1);

                    if ("Backstage passes to a TAFKAL80ETC concert".equals(items.get(i).
                                                                        getName()))
                    {
                        if (items.get(i).getSellIn() < 11)
                        {
                           if (items.get(i).getQuality() < 50)
                            {
                                items.get(i).setQuality(items.get(i).getQuality() + 1);
                            }
                        }
```

```
                    if (items.get(i).getSellIn() < 6)
                    {
                        if (items.get(i).getQuality() < 50)
                        {
                            items.get(i).setQuality(items.get(i).getQuality() + 1);
                        }
                    }
                }
            }
        }

        if (!"Sulfuras, Hand of Ragnaros".equals(items.get(i).getName()))
        {
            items.get(i).setSellIn(items.get(i).getSellIn() - 1);
        }

        if (items.get(i).getSellIn() < 0)
        {
            if (!"Aged Brie".equals(items.get(i).getName()))
            {
                if (!"Backstage passes to a TAFKAL80ETC concert".equals(items.get(i).
                                                                        getName()))
                {
                    if (items.get(i).getQuality() > 0)
                    {
                        if (!"Sulfuras, Hand of Ragnaros".equals(items.get(i).getName()))
                        {
                            items.get(i).setQuality(items.get(i).getQuality() - 1);
                        }
                    }
                }
                else
                {
                    items.get(i).setQuality(items.get(i).getQuality() - items.get(i).
                                                                        getQuality());
                }
            }
            else
            {
                if (items.get(i).getQuality() < 50)
                {
                    items.get(i).setQuality(items.get(i).getQuality() + 1);
                }
            }
        }
    }
  }
}
```

About the Author



Colin Vipurs started professional software development in 1998 and released his first production bug shortly after. He has spent his career working in a variety of industries using a wide range of technologies always attempting to release bug-free code. He holds a MSc from Liverpool University and current works at Shazam as a Developer/Evangelist. He has spoken at numerous conferences worldwide.