COMP33711: Agile Software Engineering

TDD Worked Example: Converting Roman Numerals

A Worked Scenario for Red-Green-Go!

Suzanne M. Embury
School of Computer Science
University of Manchester

November 2015

Introduction

This document talks you through the development of a small and simple piece of code using the technique of test-driven development (TDD). We will create a unit of code that converts Arabic numbers (that is, the main number system used in the Western world today) into Roman numerals. Such code might be needed for a typesetting program, for example, which has to be able to format page numbers for the preface, table of contents, and other front-matter of a book using Roman numerals. We'll use JUnit/Java to implement the test/production code needed to meet this particular software requirement.

We will show how we apply the basic rules of TDD to "grow" the code we require, and its associated test suite, in small but meaningful increments. Recall that in TDD we divide the code we write into two distinct parts: the production code, which implements the generic behaviour required by the customer, and the test code, which gives specific concrete examples of the behaviour required of the production code unit. We then use a repeated cycle of the following three steps to add the necessary code of both kinds:

1. *Write a test that fails.* Ideally, this should be the simplest test we can think of that will fail—i.e., the simplest extension of the behaviour implemented so far, that is not supported by the production code implemented so far. Note that the test must fail; in TDD we are not allowed to write any production code unless we have a failing test to work towards.

2. *Write the production code that will make the test pass* (without breaking any of the existing tests). We should write the simplest possible implementation that will cause the test to pass, so that our production code really does implement only the behaviour that the tests are describing, and not the more general behaviour that we may have in our heads.

3. *Refactor the code to remove any design inelegancies* that may have been introduced by the most recent changes, while taking care not to break any of the existing set of test cases. Both test code and production code should be including in this refactoring step.

This cycle is often referred to as the test-code-refactor cycle, or the red-green-green cycle (indicating the test results we want to see from our test harness at the end of each step).

In this case study, we'll describe each step taken during the development of the Roman numeral number formatter, and give the reasons behind the decisions made. We'll use adversarial pairing to show how the work is divided up between the pair of developers. In this form of pairing, one of the pair writes a failing test, and then passes the keyboard to his/her partner. The partner writes the production code to make the test pass, and then writes the next failing test. At this point, the keyboard gets passed back to the original developer, to once again bring the tests into a passing state.

We're also going to show how the development proceeds when we work in very small increments. Of course, it is possible to arrive at a good solution to this problem using fewer steps than we show here, and probably in less time. But our aim in this document is to illustrate the most detailed form of TDD at work, that gives the best code coverage and the strongest guarantees of correctness. If you understand how TDD works at this fine-grained level, it is relatively easy to choose to work at a more coarse grain, where that is appropriate.

Step 1 (Red): Write the First Failing Test Case

After talking to the on-site customer or business analyst about the requirements for the story being implemented, our developer pair can begin to code. At this stage, both the production code and the test code for the unit under development are empty. Following the TDD rules, the only option at this stage is to write a failing test case.

Sam takes the driver role to begin with, and Suzanne navigates. They start by discussing the first test they will write. Sam explains that he likes to begin with a straightforward "happy path" test, to get things moving. So, their first test case will check that the code can convert the number 1 into the Roman numeral "I".

But, they have some design decisions to make before they can write the test case. How will the Roman numerals be represented in the code? Sam and Suzanne talk through the options. One obvious approach is to write a static converter method, such as:

```
String displayPageNo = RomanNumeralConverter.convert(page.getPageNumber());
```

An alternative is to create a domain type for Roman numerals, and to instantiate the class every time we need a new Roman Numeral quantity, using the `toString()` method to display the value in Roman numeral form:

```
RomanNumeral pageNo = new RomanNumeral(page.calculatePageNo());
…
String renderedPage = … + pageNo + …;
```

A third option is to create an interface for formatted numbers that all quantities that might require Roman numeral formatting must implement.

The static code option is the simplest and looks like it will require the least code and least run-time memory. But static code can suffer from testability limitations, and so should be used sparingly and with care on any project where test automation is a key practice.

The second option has the advantage of creating a domain type, and thus of including the concept of Roman Numerals explicitly within the domain model of the system being built. It would not require developers to remember to apply the Roman numeral formatter call for data values that must always be formulated as Roman numerals. We can say once, when we create them, that they are Roman numerals, and after that they will always be formatted appropriately.

But, in this application, we have page numbers which should sometimes be formatted as Arabic numbers and sometimes using Roman numerals, depending on their position in the book and on the configuration settings for the book (e.g. whether the front matter should have Roman numeral page numbers or not). This suggests that we need a way of switching between different formatting options for the same quantity, which would point to the interface solution.

After a quick discussion of these options, Sam and Suzanne decide that they can rule out the static code option, but don't yet have enough information about the design and behaviour of the rest of the system to know whether it is better to create a domain type or an interface for Roman numeral formatting. They decide to go with the simpler domain type option for now, confident that the comprehensive test suite they will create, and the high code quality they will maintain, will allow them to refactor easily and safely to the interface option in future, should later development indicate that they made the wrong choice. They don't need to spend lots of time making the "perfect" decision at this early stage.

With the basic design decided, Sam can go ahead and write the first test case. He quickly codes up the following JUnit test case:

RomanNumeralTest.java

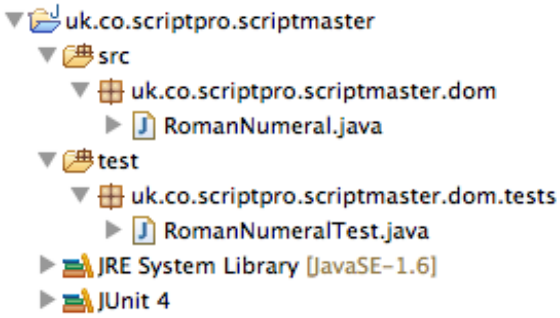
```
package uk.co.scriptpro.scriptmaster.dom.tests;

import static org.junit.Assert.*;

public class RomanNumeralTest {

    @Test
    public void shouldConvertArabicOne() {
        RomanNumeral rn = new RomanNumeral(1);
        assertEquals("I", rn.toString());
    }

}
```

He'd like to run this test, to make sure that it fails and that he sees the red bar needed to allow them to write some production code. But he has to fix the compile errors first, by creating a stub version of the production class mentioned in the test: `RomanNumeral`. Luckily, the IDE he is using provides a "quick fix" facility that can automatically create the needed stubs[1]:

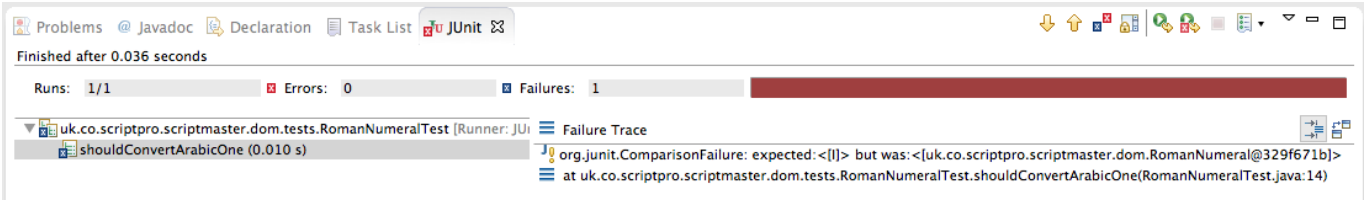
RomanNumeralTest.java | RomanNumeral.java

```
package uk.co.scriptpro.scriptmaster.dom;

public class RomanNumeral {

    public RomanNumeral(int i) {
        // TODO Auto-generated constructor stub
    }

}
```

Note the location for this new class, as shown in this view of the project structure. The new class is a production code class, so it is located in a production code source folder and a production code package, and is quite separate from the test code that Sam and Suzanne are creating for it:



The creation of the stub class and constructor fixes the compile errors in the test class, and Sam can now ask the IDE to run it. Unsurprisingly, it fails:



[1] In Eclipse, hover over the class name underlined in red, wait for the quick fix menu to appear, and select "Create class RomanNumeral". Once the class is created, you can use the same technique to ask Eclipse to create the missing constructor for you, too.

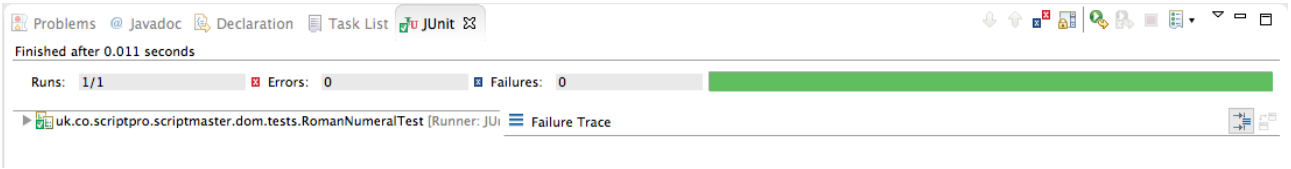
With the sight of this red bar from JUnit, the first step of the first red-green-green cycle is complete.

Step 2 (green): Make the Test Pass

Suzanne now takes the keyboard. Her task is to make the test Sam has just written pass, using the simplest, smallest amount of code she can get away with. The report on the test failure from JUnit tells her that the test fails because a `String` version of the object identifier is being returned, instead of the expected "I". This is because they haven't defined the `toString()` method on the `RomanNumeral` class yet, and so are inheriting the one defined on its superclass `Object`. She therefore makes the following changes to the `RomanNumeral` class stub:

```
public class RomanNumeral {

   public RomanNumeral(int i) {
   }

   public String toString() {
      return "I";
   }
}
```

She runs the test and it passes, giving the green bar she was looking for:



At first sight, the code that Suzanne wrote to get this test to pass looks wrong. Hard coding the expected result into the return value might make this one test pass, but it is definitely not a solution to the general problem we are trying to solve. But, in fact, Suzanne is following the TDD process carefully here, which requires that, when we have a failing test, we should make the simplest, most direct changes to the production code needed to make the test pass (while keeping all the existing tests passing, too). This simple, stupid-looking implementation is sufficient to get the project to a green bar state. Writing anything more complicated than this would be unnecessary gold-plating and wasteful.

This is an important part of TDD. We write the tests in advance of writing the code to force us to think about the behaviour we need to implement as a black box. It makes us think about what the code should do (because we need to understand that in order to write the test), while completely ignoring (for the moment) the issue of how the code should do it. The tests become a specification of the interface the class under development should present to the world, and the externally visible behaviour that the interface should provide. But if the tests are the specification, then *any* implementation that satisfies the tests is an acceptable implementation. If we are not happy with the simplistic implementation we have produced, then that indicates a problem not with the production code, but with the test cases. The solution is not to write production code that goes beyond what the tests say is needed, but to write new test cases that make clear that this simplistic implementation is not sufficient.

A further point to note is that the code Suzanne has written is not *all* wrong. An overriding of the `toString()` method *is* part of the final solution, and this method *will* need to return a `String` value. These parts of the production code won't be changed by the addition of new test cases. Only the code that computes the value to be returned will need to be changed, as Sam and Suzanne flesh out the specification for the behaviour that is required.

We will see this pattern occurring again and again as we work through the implementation of this simple functionality. With each new test, production code is added containing some elements that will be part of the final solution and some (concerning the behaviour that is not yet fully specified) that will appear in only an

approximated form. The closer we get to complete test coverage (i.e., a complete specification), the smaller and more limited these approximated elements will become, until we have eliminated them completely and the implementation is finished. Well, finished until the next requirements change or design improvement comes along, at least …

Step 3 (green): Refactor the Code to Keep the Quality High

Before moving on to the next failing test, Suzanne closes the current TDD cycle by looking over the code to see if the implementation can be improved in any way. At the moment, the code base is so small and simple, so we wouldn't expect to see many opportunities. The only thing that jumps out at Sam and Suzanne is the literal value "I" that is embedded in the production code. Literal values in code can be a potential source of future errors, since they can quickly become duplicated across the code base, and then become difficult to change consistently. It is a little early in the development of this code unit to know what the best solution is, but Suzanne decides to refactor the literal into a constant anyway, since it is cheap to both do and undo. She uses her IDE's refactoring menu[2], to extract a constant called `I`:

```
public class RomanNumeral {

   private static final String I = "I";

   public RomanNumeral(int i) {
   }

   public String toString() {
      return I;
   }
}
```

This done, she runs the tests, to check that it still passes after this change. It does, and she can move on to write the next failing test.

Step 4 (red): Write a New Failing Test Case

It's now Suzanne's turn to write a failing test. Ideally, at the beginning of each new TDD cycle, we should choose the simplest extension of the behaviour described by the existing tests that is not yet satisfied by the implemented production code. Sometimes there are several candidates for the next simplest test, when the behaviour we need is complex and can be extended in several ways. In this case, for example, we could choose to write a test describing the next happy path case (the next most simple Arabic number to translate to Roman numerals) or a sad path case (a test describing the simplest invalid case, for example). Sam started off with a happy path test, so Suzanne decides to continue in this line and writes the following failing test:

```
@Test
public void shouldConvertArabicTwo() {
   RomanNumeral rn = new RomanNumeral(2);
   assertEquals("II", rn.toString());
}
```

The failure message given by JUnit is: "`expected: <I[I]> but was: <I[]>`". Suzanne now passes the keyboard back to Sam, for him to continue as driver for the next stage of the work.

---

[2] To do this in Eclipse, select the literal value you want to convert into a constant, and right click. Choose Refactor → Extract Constant from the menu that appears. You can then indicate the name of the constant and the visibility you want it to have, in the dialogue box that appears.

Step 5 (green): Make the Test Pass

Sam quickly produces the following new version of `RomanNumeral`, as a simple way to get the two test cases to pass:

```
public class RomanNumeral {

   private static final String I = "I";
   private int value;

   public RomanNumeral(int i) {
      value = i;
   }

   public String toString() {
      if (value == 1)
         return I;
      return "II";
   }
}
```

After this change, he runs the tests and gets the green bar he expected.

"Triangulation"

Yet again, here, we see one of our developers writing production code that looks very strange when viewed from a traditional programming point of view. Just as Suzanne did when making the first test pass, Sam writes the simplest, stupidest piece of code he can that follows the behaviour specified by the tests *exactly*. Once again, the message is: if this (clearly incorrect) implementation gets the tests to pass, then we don't have enough tests. We need to write some more.

This process, of using lack of generality in our production code to drive the generation of useful test cases, is called "triangulation". This is a word more commonly used in map making and surveying[3] to describe a process whereby a space is divided up into triangles, so that heights, distances, etc., can be measured. In this process, the properties of two of the three points of the triangle are known, and from these are derived the properties of the third point.

In TDD, the analogy asks us to think of test cases as points in the desired solution space. We write production code that can satisfy two (or more) of our test case points, to indicate the need for a third test case point in the vicinity of the two we already have.

In fact, this version of the code brings out a very important aspect of the final implementation, that will not be removed by later TDD cycles: it tells us that the Roman numeral value we want to output must depend on the Arabic numeral value that is input to the constructor. Even with just two tests, there's no way to write the code so that the tests pass without having that condition on the `value` field. It took the second test to push out this aspect of the required behaviour. The next tests will help us to refine exactly what the relationship between the input `value` and the output Roman numeral string is.

Step 6 (green): Refactor the Code to Keep the Quality High

Before we can move on to those tests, however, we have to look for opportunities to improve what we have written so far. In this case, there is an obvious refactoring to perform. Suzanne made a constant of the

3 Hill walkers in the British Isles may sometimes encounter small white obelisks on the top of major hills, with metal loops stuck into their tops. These are "triangulation stations", used to fix surveying equipment in order to measure the height of surrounding hills by the process of triangulation.

previous literal, so for consistency, Sam can't leave this second literal lying around the code. He could create a second constant, but after a moment's thought he decides to try writing the code as follows:

```
public String toString() {
    if (value == 1)
       return I;
    return I + I;
}
```

The tests still pass, and this version seems to say something useful about how Roman numerals work. So between them, Suzanne and Sam decide to keep the code in this form.

A further refactoring is possible. The current version of the code puts the work of converting to Roman numerals inside the `toString()` method, while the Arabic value of the instance is stored in a field. This means that the work of converting to Roman numerals has to be done every time a RomanNumeral instance is accessed as a string. An alternative is to do the work of the conversion once, in the constructor, and to store the result to be returned on each call to `toString()`. Sam makes this change, and takes the chance to give the variables some more meaningful names, resulting in the following production code class:

```
public class RomanNumeral {

   private static final String I = "I";
   private int arabicValue;
   private String romanValue;

   public RomanNumeral(int i) {
       arabicValue = i;

       if (arabicValue == 1)
          romanValue = I;
       romanValue = I + I;
   }

   public String toString() {
       return romanValue;
   }
}
```

Looking at this together, Suzanne and Sam spot yet another chance to improve the code quality. The change makes the constructor look a little messy. It is doing two things (record the value of the RomanNumeral and convert the Arabic value to a Roman numeral value), but it's not really clear from the code where one of these tasks ends and the other begins. The level of abstraction of the code here is too low level. That makes the code less readable (i.e., less self-documenting) than it could be.

A decade or more ago, the solution to this problem would have been to add a comment before the if-statement, explaining the purpose of the statements that follow. Nowadays, however, we try to avoid using comments in code wherever possible, and try to make the actual code readable without comments. The way we do that in this case is to wrap up the problematic statements inside a method with a meaningful name. Sam makes this change, resulting in the following code:

```
public class RomanNumeral {

   private static final String I = "I";
   private int arabicValue;
   private String romanValue;
   public RomanNumeral(int i) {
       arabicValue = i;
       romanValue = convertToRomanNumeralForm(arabicValue);
   }
```

```
    public String toString() {
        return romanValue;
    }

    private String convertToRomanNumeralForm(int arabicValue) {
        if (arabicValue == 1)
            return I;
        return I + I;
    }
}
```

Creation of a method like this one, that is only called once and that is only needed to take the place of a comment, might seem strange. In fact, with modern compilers, there is no performance hit in declaring such methods. They exist in the source code, but are optimised away in the object code. Therefore, modern practice is to declare as many such helper methods as are needed to make the code readable and clear. It is far more important nowadays that the code be both correct and easy to change correctly, than it is to squeeze every last drop of performance out of the implementation.

After this, Sam and Suzanne are happy with the code in its current form. The tests all pass. They are now ready to move on to adding the next tiny increment of functionality.

Step 7 (red): Write a New Failing Test Case

Sam, as driver, writes the next failing test case. He thinks for a moment to identify the test case that will describe the smallest significant difference to the functionality described by the previous test cases. Then he writes the following test:

```
@Test
public void shouldConvertArabicThree() throws Exception {
    RomanNumeral rn = new RomanNumeral(3);
    assertEquals("III", rn.toString());
}
```

Of course, it fails, with the error report showing the problem: “expected: <II[I]> but was: <II[]>”

Step 8 (green): Make the Test Pass

Now Suzanne takes the keyboard once more, and modifies the `RomanNumeral` class, closely following the behaviour required by the tests:

```
private String convertToRomanNumeralForm(int arabicValue) {
    if (arabicValue == 1)
        return I;
    if (arabicValue == 2)
        return I + I;
    return I + I + I;
}
```

She runs the tests, and sees the green bar.

Step 9 (green): Refactor the Code to Keep the Quality High

The next step is to look for opportunities to refactor the code. So far we have seen Sam and Suzanne using the refactoring step to improve the quality of the code in general, spotting common design flaws and using

refactoring to address them. In this refactoring step, however, Suzanne is going to change the code for a different purpose: to move away from the sequence of individual cases produced by the triangulation process, and towards a more generic solution. She sees a pattern in the code that indicates that a looping behaviour is required, and refactors the code to make use of a loop to cover all three cases, rather than spelling out the behaviour of each separately:

```
private String convertToRomanNumeralForm(int arabicValue) {
   String romanNumeral = "";

   int numToAppend = arabicValue;
   while (numToAppend > 0) {
       romanNumeral += I;
       numToAppend--;
   }

   return romanNumeral;
}
```

After making this change, she runs the tests and is relieved to see the green bar signalling that the behaviour of the code relative to the tests so far has not been affected by what she has done here.

At this point, Suzanne begins to write the next failing test. But Sam suggests they refactor the test code first. He is getting uncomfortable about the amount of duplication that is appearing in the JUnit class. Suzanne sees his point, and writes a helper method for the tests that will deal with this problem:

```
public class RomanNumeralTest {

   @Test
   public void shouldConvertArabicOne() {
       assertRomanNumeralForm(1, "I");
   }

   @Test
   public void shouldConvertArabicTwo() {
       assertRomanNumeralForm(2, "II");
   }

   @Test
   public void shouldConvertArabicThree() {
       assertRomanNumeralForm(3, "III");
   }


   private void assertRomanNumeralForm(int arabicValue, String romanValue) {
       RomanNumeral rn = new RomanNumeral(arabicValue);
       assertEquals(romanValue, rn.toString());
   }
}
```

This refactoring is important, since it means that if the design of the production code changes, we will only have to update the helper method in the test class, and won't have to correct every single test case. Test maintenance can be an expensive business in real (large) software systems, so it is important that the quality of the test code be kept as high as the quality of the production code, to keep the maintenance costs down. All the same code quality principles apply to test code as well as to production code, just as here we applied the principle that duplication in code is often harmful and should be removed where possible.

Notice also that this refactoring makes our test cases more direct and readable. They merely assert the

equivalence between the Arabic and Roman numeral forms of the same quantity[4].

Of course, Suzanne has to run the tests after this refactoring, to check that nothing has been broken. This is just as important when we refactor test code as when we refactor production code. In this case, the tests do pass. So, the pair can move on to their next failing test.

Step 10 (red): Write a New Failing Test Case

Sam and Suzanne discuss what the next test should be. The last test described the Roman numeral equivalent for the number 3, so the obvious next test to try would be 4. But, would this give the smallest, simplest increment in the behaviour of code unit under development? Sam gets out the typesetting manual the on-site customer has given the pair, and searches for a description of how roman numerals should be formed. He finds it in an appendix[5]. Various rules are described, and divide into two broad classes: additive cases and subtractive cases.

In the additive cases, a numeral appearing to the right of a numeral of larger value indicates that the value of the combined number is the value of the individual numerals added together. Some numerals (e.g., "I" and "X") can be repeated up to 3 times. Other numerals (e.g. "V" and "L") can only appear next to numerals of a different value.

In the (much smaller number of) subtractive cases, a lower-valued numeral appears to the left of a higher-valued numeral, indicating that its value is to be subtracted from the value of the larger numeral, to get the value of the whole. "IV" is one of these subtractive cases. It has the value of 5 – 1, since "I" is of lower value than "V".

Jumping from the most basic additive case to a subtractive case doesn't sound like the most sensible increment of functionality to choose. Suzanne proposes that they complete the additive case behaviours before moving on to the subtractive cases. Sam agrees. He makes a note of their testing strategy:

- Additive cases
- Subtractive cases

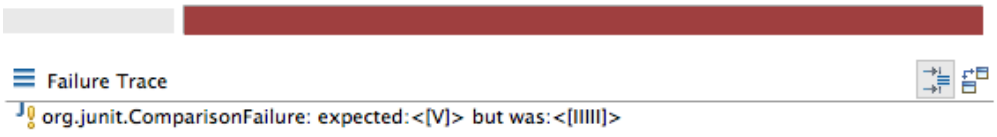
Suzanne suggests a third class of tests that they need to write, and Sam adds this to the list:

- Invalid roman numeral values

So Suzanne now presses on with the next additive case. Since repetitions of more than 3 I's are not allowed, any further additive cases must involve a new numeral that the tests so far don't cover. So, to pave the way for dealing with more additive cases, Suzanne writes a test that introduces another numeral:

```
@Test
public void shouldConvertArabicFive() {
   assertRomanNumeralForm(5, "V");
}
```

She runs the test, and (as expected) it fails.



[4] For even greater readability, we could have used the Hamcrest matcher library to write our assertions. This would have allowed us to convey the intention of the test in a way that reads like a sentence, but is still executable as a test. For example: `assertThat(theRomanNumeralFormOf(1), is("I"))`.

[5] It would be worthwhile looking up the rules of Roman numeral formation at this point, to help you follow along. The Wikipedia page gives a reasonable description, for example, in the section entitled "Reading Roman Numerals".

Step 11 (green): Make the Test Pass

Sam now takes up the driver role, and looks at the `RomanNumeral` code to see how this new test case can be accommodated. This test is the first of several that will help them to explore a new aspect of the required behaviour, so he sticks closely to the tests, to push out the generic pattern the code will eventually implement:

```
private String convertToRomanNumeralForm(int arabicValue) {
   String romanNumeral = "";

   if (arabicValue == 5)
       return "V";

   int numToAppend = arabicValue;
   while (numToAppend > 0) {
       romanNumeral += I;
       numToAppend--;
   }

   return romanNumeral;
}
```

This change makes all the tests pass.

Step 12 (green): Refactor the Code to Keep the Quality High

There's only one clear refactoring to be done, at this point. Sam turns the embedded literal he has introduced into a new constant.

```
public class RomanNumeral {

   private static final String I = "I";
   private static final String V = "V";

   …

   private String convertToRomanNumeralForm(int arabicValue) {
       String romanNumeral = "";

       if (arabicValue == 5)
          return V;

       int numToAppend = arabicValue;
       while (numToAppend > 0) {
          romanNumeral += I;
          numToAppend--;
       }

       return romanNumeral;
   }
}
```

Step 13 (red): Write a New Failing Test Case

The tests still pass after this simple refactoring, so Sam can write the next failing test. The next simplest extension to the behaviour would be to try some repeated "V" numerals. But this is not allowed in a valid Roman numeral, according to the typesetting manual. So, the next simplest case we can work on, with the

numerals we have so far, is to combine “V” and “I” in a simple additive case:

```
@Test
public void shouldConvertArabicSix() {
   assertRomanNumeralForm(6, "VI");
}
```

This test is the first that combines two different numerals in the same number. So we should start to see the pattern for handling multiple numerals begin to emerge in the code, as Sam and Suzanne use triangulation to work through the key cases that follow on from it.

The test fails, because “`expected: <[V]I> but was: <[IIIII]I>`”

Step 14 (green): Make the Test Pass

Suzanne takes back the keyboard, and sets about making the new test pass. The solution is obvious. The production code currently has a branch that says on the one side “convert 5 by adding V” and on the other side “convert 1-3 by adding I’s”. She needs to make both of these execute in sequence, so that V is added if the Arabic value is greater than or equal to 5, and as many I’s are added to the end as are needed to make up the total:

```
private String convertToRomanNumeralForm(int arabicValue) {
   String romanNumeral = "";
   int numToAppend = arabicValue;

   if (numToAppend >= 5) {
       romanNumeral += V;
       numToAppend -= 5;
   }

   while (numToAppend > 0) {
       romanNumeral += I;
       numToAppend--;
   }

   return romanNumeral;
}
```

All the tests pass after this change.

Step 15 (green): Refactor the Code to Keep the Quality High

The only bothersome aspect of the code at this point is the repeated 5 in the production code. Both our developers are uncomfortable with this duplicated literal, and feel that it should be removed. But, there is nothing in the code at the moment to suggest the appropriate refactoring. They could define a constant FIVE, but that seems somewhat vacuous. They decide to leave the code as it is for now, and to keep their eyes open for an opportunity to remove the duplication as they progress with the implementation.

Step 16 (red): Write a New Failing Test Case

Suzanne and Sam agree that their implementation will handle a few more cases than they have coded in the tests: “VII” and “VIII” to be precise. They decide they don’t need to include all these test cases in the suite. Although high code coverage is a good thing (because it provides good defect finding power), we have to keep an eye on the size of our test suites and to make sure we are not running redundant tests. We need our unit tests to run quickly, so that we can get fast feedback on the changes we make to the production code.

That means it is important that our unit tests don't waste effort testing things that don't need to be tested.

Just to make sure their expectations of the code are correct, Suzanne codes up the test for "VIII" (chosen because it's the boundary case), and confirms that it does indeed pass. She and Sam decide to leave the test in the test suite, because of its boundary case status, and because they don't expect the suite for this unit to be very large or slow. Then they move quickly on to the next uncovered part of the functionality. Suzanne really wants to write a test that involves a repeated numeral other than "I", but to do that she first needs to introduce another numeral that is allowed to repeat:

```
@Test
public void shouldConvertArabicTen() {
    assertRomanNumeralForm(10, "X");
}
```

The test fails as expected (with the message "`expected: <[X]> but was: <[VIIIII]>`").

Step 17 (green): Make the Test Pass

Sam now takes over as driver, and Suzanne resumes the navigator role. In choosing how to implement the new behaviour, Sam aims to follow the patterns and conventions we've been seeing in the code so far, while also making only changes that are required by the new test case:

```
private String convertToRomanNumeralForm(int arabicValue) {
   String romanNumeral = "";
   int numToAppend = arabicValue;

   if (numToAppend == 10)
       return "X";

   if (numToAppend >= 5) {
       romanNumeral += V;
       numToAppend -= 5;
   }

   while (numToAppend > 0) {
       romanNumeral += I;
       numToAppend--;
   }

   return romanNumeral;
}
```

He gets the desired green bar and moves quickly on to the next step, before the duplication and the increasing number of literals in this code distracts him too much from the patterns appearing in the code.

Step 18 (green): Refactor the Code to Keep the Quality High

The body of `convertToRomanNumeralForm()` is starting to look rather messy, and Sam is itching to put it right. But the pattern he is looking for in the code, indicating a suitable refactoring to sort out the mess, is not yet fully in place. It seems to be very close though, perhaps just one or two test cases away. So, Sam has to grit his teeth and content himself with just making a constant for the newly introduced numeral X in this refactoring step. The chance for more extensive improvements will hopefully come along soon.

Step 19 (red): Write a New Failing Test Case

Next, Sam wants to see what happens when he asks for an X numeral to be combined with an I. So he creates the following (failing) test:

```
@Test
public void shouldConvertArabicEleven() {
   assertRomanNumeralForm(11, "XI");
}
```

Step 20 (green): Make the Test Pass

It's Suzanne's turn at the keyboard again, and she just has to make a minor change to the handling of numeral "X", following the pattern that is already established for the other numerals:

```
private String convertToRomanNumeralForm(int arabicValue) {
   String romanNumeral = "";
   int numToAppend = arabicValue;

   if (numToAppend >= 10) {
      romanNumeral += X;
      numToAppend -= 10;
   }

   if (numToAppend >= 5) {
      romanNumeral += V;
      numToAppend -= 5;
   }

   while (numToAppend > 0) {
      romanNumeral += I;
      numToAppend--;
   }

   return romanNumeral;
}
```

This makes the tests pass.

Step 21 (green): Refactor the Code to Keep the Quality High

The last coding step caused the code to grow even longer and messier, with an additional literal duplication being introduced. Some refactoring is definitely needed, and the pattern is getting clearer and clearer in the code. The next step is to see what happens when they add the ability to convert numbers involving multiple "X" numerals. So, they decide to live with the messiness for just a little longer.

Step 22 (red): Write a New Failing Test Case

The pace is pretty quick now, as Sam and Suzanne step through the tiny changes needed to include the new test cases. A short discussion is enough to convince them that their code will cover a host of (no longer interesting) cases, such as "XV" and "XVIII". Suzanne codes up a couple of these, just to check, and they watch the bar stay green.

The next interesting case involves a repetition of a numeral other than "I". So Suzanne creates a test that extends the subset of the additive Roman numeral behaviour covered by another small, significant slice:

```
@Test
public void shouldConvertArabicTwenty() {
   assertRomanNumeralForm(20, "XX");
}
```

This fails, of course, with `toString()` returning the value "XVIIIII"!

Step 23 (green): Make the Test Pass

Sam, at the keyboard again as driver, has only another small change to make to get this test to pass. He changes the if-statement that checks whether a single "X" should be included in the output:

```
if (numToAppend >= 10) {
    romanNumeral += X;
    numToAppend -= 10;
}
```

into a while loop that adds as many X's as needed:

```
while (numToAppend >= 10) {
    romanNumeral += X;
    numToAppend -= 10;
}
```

That is all that is needed to make this test pass; just one keyword change.

Step 24 (green): Refactor the Code to Keep the Quality High

Sam and Suzanne have now pushed out enough of the behaviour of this additive case to start to see a general pattern emerging in the code. Some numerals can appear numerous times, and so are handled by a while loop; some can appear only once, and are handled by an if-statement. But the body of both of these ways of handling numerals is essentially the same. Driver Sam would like to move to a definition for this method that:

- has less duplication (both of code and of literals),
- doesn't require us to embed the equivalences between the roman numeral letters and their Arabic number equivalents into the middle of the code, and
- has all literals defined once somewhere as a clear and easily maintained constant, rather than just being buried in the code.

As a first step towards these goals, Sam points out that a while loop that executes either zero or one times has exactly the same semantics as an if-statement. The first while loop ensures that the variable `numToAppend` is less than 10 by the time it completes, we also know that we will be able to subtract 5 from it at most once in any run of the program (since the largest possible value for `numToAppend` on entry to the if-statement is 9, and 9-5 is always less than 5). Sam therefore replaces the if-statement with a while loop, giving the following method implementation:

```
private String convertToRomanNumeralForm(int arabicValue) {
    String romanNumeral = "";
    int numToAppend = arabicValue;
    while (numToAppend >= 10) {
        romanNumeral += X;
        numToAppend -= 10;
    }

    while (numToAppend >= 5) {
        romanNumeral += V;
        numToAppend -= 5;
    }

    while (numToAppend > 0) {
        romanNumeral += I;
        numToAppend--;
    }

    return romanNumeral;
}
```

He runs the test suite, and is relieved to see the friendly green bar at the end. All the test cases still pass with this new version of the code.

Now the repetitive nature of the code is very visible, and the next refactoring is obvious. For each numeral in the Roman numeral specification, in descending order of value, a while loop must be executed that adds numerals to the output string while counting down from the total by an amount equivalent to the value of each numeral added.

To make this happen, Sam must create a list of the supported Roman numerals and their values, in descending numerical order. Given the current test suite, this data structure will initially contain the pairs X-10, V-5 and I-1, though other values will obviously be included once more test cases are to be taken into account. Sam decides to use a Map structure to represent this information, and to pull a list of the descending numerical values to use to control the while loops from it:

```
public class RomanNumeral {

   private static final Map<Integer, String> numeralEquivalents =
                                                    new HashMap<Integer, String>();
   private static List<Integer> numeralValues;
   {
       numeralEquivalents.put(10, "X");
       numeralEquivalents.put(5, "V");
       numeralEquivalents.put(1, "I");

       numeralValues = new ArrayList<Integer>(numeralEquivalents.keySet());
       Collections.sort(numeralValues, Collections.reverseOrder());
   }

   private int arabicValue;
   private String romanValue;

   public RomanNumeral(int i) {
       arabicValue = i;
       romanValue = convertToRomanNumeralForm(arabicValue);
   }

    public String toString() {
       return romanValue;
   }


   // Helper methods

   private String convertToRomanNumeralForm(int arabicValue) {
       String romanNumeral = "";
       int numToAppend = arabicValue;

       for (int numeralValue : numeralValues)
          while (numToAppend >= numeralValue) {
              romanNumeral += numeralEquivalents.get(numeralValue);
              numToAppend -= numeralValue;
          }

       return romanNumeral;
   }
}
```

The production of this code involved a certain amount of discussion. Sam and Suzanne couldn't find names they were really happy with for the two new fields, but had to choose something. Maybe they'll get ideas for

better names later. Neither developer is very comfortable with the static initialiser block, being aware of the difficulties of testing static code. But, because the static code describes only reference information for use by the code unit under test, and not behaviour or dynamic data, they decide to live with this technical debt for now.

They also wonder whether to use a `LinkedHashMap` for the map of numeral equivalents, since this kind of map implementation allows iteration over its elements in the order in which they were added to the map. After a brief discussion, they decide on the implementation shown above, which requires more code and is probably less efficient, but which states clearly and explicitly the requirement that the list of boundaries be in descending numerical order. Later experience with the performance of this class may cause them to revisit this decision, but for now they have chosen to the more explicit and readable implementation over the more efficient but more opaque one.

Sam made quite substantial changes to the code in this refactoring step. He could do this with confidence, thanks to the safety net of test cases that the pair has built up so far, and by breaking the large overall change down into a sequence of smaller, safer changes. Notice also that the refactoring was not just about "tidying" of otherwise reasonable code; it was a vital part of a move towards a more general solution to a problem, from the patterns left by the individual test cases.

There is some tidying going on though. Sam's refactorings also removed the duplication and defined the literals in sensible places—just what he was hoping for at the start of this step. Plus, the test still all pass!

Step 25 (red): Write a New Failing Test Case

With our newly refactored code giving us a generic implementation of the additive Roman numeral behaviour, it is only necessary to add the numerals not covered by our test cases so far. We could add them one by one, but can probably speed up the process (and keep the number of pages in this already lengthy document down) by choosing our tests carefully. Sam chooses to write the following test as the focus of our next TDD cycle:

```
@Test
public void shouldConvertArabicOneThousandSixHundredAndSixtySix() {
   assertRomanNumeralForm(1666, "MDCLXVI");
}
```

This test contains all the legal Roman numeral letters, once, in sequence. It fails (with an amusing failure message).

Step 26 (green): Make the Test Pass

Sam passes the keyboard to Suzanne, who proceeds to add the new letters to the `numeralEquivalents` map, in the static initialiser block:

```
static {
   numeralEquivalents.put(1000, "M");
   numeralEquivalents.put(500, "D");
   numeralEquivalents.put(100, "C");
   numeralEquivalents.put(50, "L");
   numeralEquivalents.put(10, "X");
   numeralEquivalents.put(5, "V");
   numeralEquivalents.put(1, "I");

   numeralValues = new ArrayList<Integer>(numeralEquivalents.keySet());
   Collections.sort(numeralValues, Collections.reverseOrder());
}
```

These four additional lines cause the new test case to pass. (And don't break any of the old test cases.)

Step 27 (green): Refactor the Code to Keep the Quality High

No refactoring opportunities can be seen in the code at present.

Step 28 (red): Write a New Failing Test Case

Sam and Suzanne are pretty confident now that their implementation covers all the additive behaviour of roman numerals. To check this, Suzanne adds a new test in which all those Roman numerals that are allowed to repeat within a number do repeat:

```
@Test
public void shouldConvertArabicTwoThousandEightHundredAndSeventyEight() {
   assertRomanNumeralForm(2878, "MMDCCCLXXVIII");
}
```

As expected, it passes. Suzanne decides to leave the test in the suite, since it covers a useful boundary case, but now needs to write a failing test before the coding can progress. So she checks the Roman numeral specification for the next interesting test case. Since the additive behaviour is now complete, she crosses that category of tests off the list she and Sam created earlier, and starts to think about the subtractive cases. She begins by writing a test case for one of the simplest subtraction cases:

```
@Test
public void shouldConvertArabicFour() {
   assertRomanNumeralForm(4, "IV");
}
```

This test fails, since `toString()` returns “IIII” when given the value 4.

Step 29 (green): Make the Test Pass

With Sam driving, both developers look at the current production code to see how this new slice of behaviour can be incorporated. No immediately obvious options jump out, so they look at the conversion code prior to the refactoring step (i.e., step 23), to see how they could have fitted the new test case requirements into that version of the code.

Now an obvious option does appear. Sam realises that, after the if-statement that adds a single “V” if the value still to be converted is 5 or more, he could add another if statement checking whether the value is greater than or equal to 4. If it is, the code should add the string “IV” onto the end of the Roman numeral number being built up and subtract 4 from the value still to be converted. In other words, he proposes to handle the special subtractive case of “IV” exactly as they handled the special cases of the legal numerals. The only difference is that the code adds two numerals to the output string, instead of one.

If this is the case, Sam should be able to add in handling of “IV” to the refactored version simply by adding it to the list of numeralEquivalents:

```
static {
   numeralEquivalents.put(1000, "M");
   numeralEquivalents.put(500, "D");
   numeralEquivalents.put(100, "C");
   numeralEquivalents.put(50, "L");
   numeralEquivalents.put(10, "X");
   numeralEquivalents.put(5, "V");
   numeralEquivalents.put(4, "IV");
   numeralEquivalents.put(1, "I");
```

```
        numeralValues = new ArrayList<Integer>(numeralEquivalents.keySet());
        Collections.sort(numeralValues, Collections.reverseOrder());
    }
```

The highlighted line shows the only change Sam made to the production code in this step. It works, and when he runs the tests he gets the desired green bar again.

Step 30 (green): Refactor the Code to Keep the Quality High

No refactoring opportunities can be seen in the code at present.

Step 31 (red): Write a New Failing Test Case

Next, the remaining subtractive cases must be added. Sam therefore writes the following test case, which combines a subtractive special case with a single other literal in an additive role:

```
    @Test
    public void shouldConvertArabicFourteen() {
        assertRomanNumeralForm(14, "XIV");
    }
```

But this test passes (as Sam suspected it would). So Sam decides to delete this test and create another one for a number that includes as many of the special subtractive cases as he can fit into one example. The Roman numerals specification they are working from gives the following set of subtractive cases:

- 4 = “IV”
- 9 = “IX”
- 40 = “XL”
- 90 = “XC”
- 400 = “CD”
- 900 = “CM”

At best, he can get 3 of these into a single number. So, Sam quickly codes up the following two tests, covering all 6 special cases:

```
    @Test
    public void shouldConvertArabicOneThousandNineHundredAndFortyNine() {
        assertRomanNumeralForm(1949, "MCMXLIX");
    }

    @Test
    public void shouldConvertArabicOneThousandFourHundredAndNinetyFour() {
        assertRomanNumeralForm(1494, "MCDXCIV");
    }
```

Both the tests fail.

Step 32 (green): Make the Test Pass

Suzanne now has an easy task. As driver, she just has to add the special cases into the initialisation block:

```
static {
   numeralEquivalents.put(1000, "M");
   numeralEquivalents.put(900, "CM");
   numeralEquivalents.put(500, "D");
   numeralEquivalents.put(400, "CD");
   numeralEquivalents.put(100, "C");
   numeralEquivalents.put(90, "XC");
   numeralEquivalents.put(50, "L");
   numeralEquivalents.put(40, "XL");
   numeralEquivalents.put(10, "X");
   numeralEquivalents.put(9, "IX");
   numeralEquivalents.put(5, "V");
   numeralEquivalents.put(4, "IV");
   numeralEquivalents.put(1, "I");

   numeralValues = new ArrayList<Integer>(numeralEquivalents.keySet());
   Collections.sort(numeralValues, Collections.reverseOrder());
}
```

Now, all the tests pass.

Step 33 (green): Refactor the Code to Keep the Quality High

No refactoring opportunities can be seen in the code at present.

Step 34 (red): Write a New Failing Test Case

With the additive and subtractive functionality in place, the only thing that remains is to ensure that the code is robust and behaves gracefully when invalid input values are given. Sam and Suzanne discuss the boundaries of valid values for the Roman numeral conversion behaviour with the customer. She assures them that they will never be asked to format a page with number 0. She also cannot suggest any sensible strict upper limit. So, Suzanne decides to code a test to define the lower valid boundary for the code:

```
@Test(expected=CannotConvertValueToRomanNumeralException.class)
public void shouldComplainWhenAskedToConvertZero() {
    new RomanNumeral(0);
}
```

Note that she does not use the helper method to assert equality between an Arabic and a Roman numeral form of a number in this test. Instead, she merely attempts to create a Roman numeral with an invalid value. She extends the `@Test` annotation to state that the given exception should be thrown for this test case to pass.

Step 35 (green): Make the Test Pass

Suzanne now hands the keyboard to Sam, so that he can make this test pass. He makes the following change to the `RomanNumeral` constructor:

```
public RomanNumeral(int i) throws CannotConvertValueToRomanNumeralException {
   if (i == 0)
       throw new CannotConvertValueToRomanNumeralException();

   arabicValue = i;
   romanValue = convertToRomanNumeralForm(arabicValue);
}
```

He also has to add "throws" declarations to the test cases, now that this constructor throws an exception. Once all the compiler errors are fixed, Sam runs the tests and all pass.

Step 36 (green): Refactor the Code to Keep the Quality High

No refactoring opportunities can be seen in the code at present.

Step 37 (red): Write a New Failing Test Case

This "solution" indicates that our test cases describing invalid data values are incomplete, since the implementation that causes them to pass is so obviously incomplete. Sam writes the final test case needed to drive out the required behaviour:

```
@Test(expected=CannotConvertValueToRomanNumeralException.class)
public void shouldComplainWhenAskedToConvertNegativeNumber() throws Exception {
   new RomanNumeral(-1);
}
```

It fails.

Step 38 (green): Make the Test Pass

With Suzanne now driving, she can make the final tiny change needed to make this test pass. She changes this fragment of the `RomanNumeral` class constructor:

```
if (i == 0)
   throw new CannotConvertValueToRomanNumeralException();
```

to:

```
if (i <= 0)
   throw new CannotConvertValueToRomanNumeralException();
```

This causes the test to pass, and JUnit to give a green bar for the whole test suite.

Step 39 (green): Refactor the Code to Keep the Quality High

With all three groups of tests completed, Sam and Suzanne take a few moments to look at the class they have created as a whole, to see if there are further opportunities to improve the code. Does the class they have created describe a meaningful entity within the domain of the system, or some sensible unit of behaviour? Is the meaning of the class clear, from an examination of the public fields and records? Neither of our developers is quite happy with the class as it stands. Suzanne is worried about the name, `RomanNumeral`, which stands out in the domain model as being a different sort of thing from the other concepts they have (such as `Page`, `Preface` and `TableOfContents`). They discuss alternatives but agree to leave the name for now, until they can see the class placed in the context in which it will be used, and can judge the name from that standpoint. Sam doesn't like the unhelpful variable name "i" in the constructor and also notes that nothing meaningful is done with the arabicValue field.

Both problems are dealt with, and the tests still pass. But now the class looks more complicated than it need be – simply a way of translating between an Arabic value to a Roman value. Sam wonders whether they should have gone with the static code design in the first place, after all, while Suzanne increasingly feels that the interface design option might be best. But our developers agree that they won't make any changes now. They'll continue with the work on the story, and hope that seeing the class in use helps them to make the right decision. In any case, much of the work they have done will be retained: the reference information about valid Roman numerals and their values, plus the conversion algorithm, which turned out to be much smaller and simpler than they had expected when they started out.

Step 40: Linking to the Acceptance Tests

The next step is to link this new class into the existing system, so that page numbers in the front matter of a publication are displayed as roman numerals. Quite what this entails depends on how much of the needed functionality already exists. But, at some point, it will be necessary need to check that our `RomanNumeral` class operates correctly against the acceptance tests for this user story. When the code can be built successfully, with all tests passing, in the deployment environment, we can consider this story complete.

Summary

By working through this example in detail, we hope to have given you a flavour of how TDD helps us to control the conceptual complexity of programming tasks by using tests to break down the functionality that must be implemented into "wafer-thin" slices that are each very simple in themselves, but which together add up to a meaningful (and in some cases, complex) piece of functionality. In fact, this is just a continuation of an approach to functional decomposition that is one of the key points of the agile approach to software engineering in general. We start the process when we gather user stories, which describe thin end-to-end slices of functionality that have value in and of themselves. These thin slices are converted into sets of acceptance tests, which describe even thinner useful slices of behaviour. The acceptance tests point us to the code units that are required to implement them, and are further broken down into unit tests: very thin slices of the parts of the overall functionality that are provided by a single code unit. By the time we get to these unit tests, during TDD, we are thinking about the required functionality in chunks so small that many of them can be implemented (in ultra high quality code) in a matter of minutes or even seconds.

A major advantage of working in such small increments, protected by tests, is that it is much easier to write the correct code when we are only trying to change a couple of lines of code at a time. If we do make a mistake and the tests start to fail, we get an early warning of the error and can look for its cause before we have invested too much effort in the code as a whole. As likely as not, the problem will be with the code changes we have made in the current TDD cycle, so we can find and fix the problem quickly. This means that we rarely need to resort to using a debugger to step through the code when using TDD. And, more importantly, we rarely need to embark on day-long debugging sessions, without knowing how much more time we'll have to spend before we find the error. The unpredictable nature of debugging in conventional approaches to coding is one of the major causes of failure to meet planned delivery deadlines. TDD's ability to reduce the amount of debugging is one of its biggest advantages[6].

Of course, this Roman numeral converter is a small and simple example, that also happens to be well suited to showing how we can arrive at an implementation incrementally by following the lead set by the test cases closely. It also shows the importance of thinking carefully about which test to implement next. Poor test choices lead us to try to implement the code in unhelpful increments: either they are too big and we get confused and make mistakes, or else the order in which we tackle them means we don't see the patterns that we can generalise growing naturally out of the code. For example, if Sam and Suzanne had tried to implement the behaviour for "IV" (a subtractive special case) after the test for "III" passes, they would not have seen the pattern for the additive cases emerge so strongly, and could have been distracted by worrying about which letters were allowed to go in front of which others. These problems come from poor choice of test, although for beginning TDDers, they can feel as though they are caused by the TDD process itself. So, if you find yourself stuck, it may be worthwhile looking again at the groups of tests you need to cover, and trying to tackle them in a different order. With experience, you'll find yourself developing more of an eye for the extensions in behaviour that are likely to be helpful in growing the production code at a particular point, and those which may not be.

---

[6] A recent third year project student decided to use TDD for his project. He took to it quickly, and the work progressed well, with the student continually delivering code ahead of his planned schedule. So Suzanne was surprised when he lost a week of time trying to debug a part of his code. He confessed that the part of the code he was working on was a part he had written without using TDD; he had thought it was too simple to need TDD and that he would save time by coding it in the conventional manner. This was the only part of the code that needed extensive debugging work, and it was the only part of the code not written using TDD.

We also saw, in this example, the role of the refactoring steps in the derivation of the final implementation. When we refactor in TDD, we are not only concerned with writing beautiful code, but also with seeking out the general implementation that lies behind the concrete patterns that emerge in the code when we follow the tests to their logical conclusion. It is a key part of the development process, and not a "nice to have if time" optional extra. The refactoring step in TDD is in fact a reflection activity: after each wafer-thin slice is implemented, we stop to reflect on what we have done, looking for possible problems and possible opportunities for doing things better, but also improving our own personal processes, our technical knowledge and our approach to pairing with this particular partner, and our team members in general. This further shows the key role of feedback and reflection in agile development. Not only do we take time to reflect on what we are doing at the end of each iteration, but when we code using TDD we take time to reflect every few minutes. That is a lot of brain cycles being devoted to spotting mistakes and opportunities over the course of a one or two week iteration!

As with the choice of which test to write next, with experience, you will develop a gut feeling for when refactoring is going to help, and when it is better to wait until the current group of tests you are working on have revealed their secrets. Until then, the best approach is to follow the agile philosophy of failing fast: don't spend ages trying to decide whether now is a good time to refactor or not, just make a pragmatic decision now, and press on. You'll quickly see the effects of your decision to refactor, as patterns begin to emerge or as the code gets more and more disordered, and can learn to do better the next time as a result.

Finally, we note how, in addition to helping with the conceptual complexity of the coding task, TDD also helps us to write code that is itself agile; that is, code that is easy to change safely. TDD does this by:

- Removing the distinction between "writing" code and "changing" code. Before TDD, we typically wrote a single, complete implementation of a code unit in one initial step, and then moved to a pattern of applying changes to it (some smaller, some larger) over time. By contrast, in TDD, *all* implementation is done as a series of small scale changes. Because we are *always* changing code, aspects of the design that get in the way of change are spotted early, when they can be corrected easily.

- Providing comprehensive test suites as a safety net for refactoring. Code change is scary because of the near impossibility of predicting all the effects even a small change will have on a large software system. Having very high test coverage removes a significant element of this worry. If we make a change that causes a regression in some other part of the system that we had not expected, the tests will (in most cases) tell us.

- Giving us a way of describing code changes in small, manageable increments. Breaking down an implementation task by tests is a much more powerful and useful way of decomposing an implementation task than the traditional approach of converting it into a series of high-level steps, that can be tackled in turn. This stepwise approach may work well for the initial construction of code (when the requirements are clear and stable, and when sufficient time has been allocated to the task), but is much less effective when we move into the later "code change" phase. Changes to functionality can rarely be localised into a single step of an algorithm, but instead add new cases of the complete end-to-end behaviour. This is neatly encapsulated by test cases, in a form that shows clearly the parts of the code that need to change to make the new tests pass.

- Spreading the cost of keeping code quality high across the complete development. If we want to have code that is easy to change, we *must* keep its quality high, with a clean and simple design. Like writing comments and writing tests, refactoring to improve code quality is boring and time-consuming if you try to do it all in one go after the code is "finished". Much better to spread that work out across the development, so that keeping code quality high never becomes a chore, and you never have the excuse that you "just don't have time" to keep the code clean.

More Examples

There are now an increasing number of documented TDD examples available for study, many of them covering much more complex functionality than the simple roman numerals example we've looked at here. Some suggestions are:

- Engineer Notebook: an Extreme Programming Episode: a transcript of the conversation between Bob Martin and Bob Koss, and the code they produce, as they pair to develop a simple Ten-Pin Bowling Game using TDD principles. Available from:
  `http://www.objectmentor.com/resources/articles/xpepisode.htm`

- Lasse Koskela's book ("Test Driven: Practical TDD and Acceptance TDD for Java Developers") contains two early chapters which describe the development of a string template processor using the TDD approach. (Chapters 2 & 3, Manning Publications, October 2007, ISBN: 1932394850)

- Freeman and Pryce's book ("Growing Object-Oriented Software, Guided by Tests") describes the development of an Auction Sniper system, using the outside-in style of Acceptance Test Driven Development. This book is not focussed solely on TDD, but the red-green-green cycle is an intrinsic part of ATDD, so the book does contain a lot of TDD steps and coding. (Addison Wesley, October 2009, ISBN: 0321503626.)

Try it for Yourself

The next step, if you are interested in TDD, is to try it for yourself. Reading about other people doing TDD is a good way to get started, but you can only really see the benefits (and the challenges!) when you try to use TDD in your own code. During term-time, the weekly Agile 3rd Year Project Club is a good way to get help and advice on how to get started, or how to get unstuck. You are also welcome to contact Suzanne if you have any questions or need some suggestions for how to progress.