Roman Numerals Kata

Here are instructions on how to set-up and run a Roman Numerals Kata that uses BDD and TDD.

Setup Steps:

* Create simple Maven project based on zip file
* Start with the partially refactored .feature file that has only the first feature that needs to be refactored

Kata Steps – setup to run feature

1. These will be shortened to keep the whole presentation less than 1 hour – it will be a lite version
2. Add the RunCukesTest.java file in ~test/java/cucumber/kata
3. Run in Intellij and see TODO
4. Based on result, decide to refactor the first scenario, adding Outline:, Examples: and the column names at the end of the when and then. Remove the Given as it is redundant.
5. Run tests in IT and see the difference

Kata steps to get first coding to pass

1. Add the class TestConvFromRomanNumeral to ~cucumber/kata/junit
2. Create first test to convert I and V to Arabic

@Test

public void testArabicToRoman() {

assertEquals("1 should be the outcome",1, RomanNumbers.toArabic("I"));

assertEquals("5 should be the outcome",5, RomanNumbers.toArabic("V"));

}

1. Now we can see the tests fail – they won’t even compile.
   1. The Class RomanNumbers does not exist yet – that is the application class that I will use
   2. This will be class based behavior vs instance
   3. Need to add imports for the jUnit behavior
2. Now to start coding in the application, add the class ~src/main/java/cucumber/kata RomanNumbers
   1. Put this class method in:

public static long toArabic(String aRomanNumber) throws IllegalArgumentException{

RomanNumeral aRomanNumeral = RomanNumeral.valueOf(aRomanNumber);

switch (aRomanNumeral) {

case I:

return RomanNumeral.I.getArabic();

case V:

return RomanNumeral.V.getArabic();

default:

throw new IllegalArgumentException("Invalid Digit " + aRomanNumber);

}

}

1. I decided to use an enum as part of my design – thus the “Design” aspect of TDD
2. Add java class file called /src/main/java/cucumber/kata/RomanNumeral
3. Implement the I and V definitions to make the code complile:

public enum RomanNumeral {

V(5,"V"),

I(1,"I");

private int arabic;

private String name;

private RomanNumeral(int arabic, String name){

this.arabic = arabic;

this.name = name;

}

public int getArabic() {

return arabic;

}

public String getName() {

return name;

}

1. Now lets get a feature to pass. We created a step def, now to put some real code in it to run our application and hopefully pass.
2. In the RomanNumeralStepDef, lets look at some of the code generated for us by intelliJ for each of the feature details. The regular expressions are generated for us, so all we need to do is implement the code to reach into the application and request the service we want.

@When("^I ask for a translation of a \"([^\"]\*)\"$")

public void I\_ask\_for\_a\_translation\_of\_a(String romanNumeral) throws Throwable {

// Express the Regexp above with the code you wish you had

throw new PendingException(); }

@Then("^I get the correct \"([^\"]\*)\"$")

public void I\_get\_the\_correct(int anArabicNumber) throws Throwable {

// Express the Regexp above with the code you wish you had

throw new PendingException();

}

1. Create the code in the step to start the test process.

aRomanConversion = new ConversionEntry();

aRomanConversion.romanNumeral = romanNumeral;

1. Because we are using an Example: with amounts to run “across” two steps, this would need to have a way for the “code” to interact together. Generally, that is not a good thing to do, but I think it is safe in this instance. Lets create a class (2) to hold values from one step to another. Two is to minimize the chance of a collision if the tests are run in parallel:

private ConversionEntry aRomanConversion;

private ConversionEntry aComplexConversion;

public class ConversionEntry {

String romanNumeral;

Integer arabicConversion;

public void setRomanNumeral(String romanNumeral) {

this.romanNumeral = romanNumeral;

}

}

1. Now lets a step to work by using our helper class and calling our application:

@When("^I ask for a translation of a \"([^\"]\*)\"$")

public void I\_ask\_for\_a\_translation\_of\_a(String romanNumeral) throws Throwable {

aRomanConversion = new ConversionEntry();

aRomanConversion.setRomanNumeral(romanNumeral);

//no test done, this is setup for the next step

}

@Then("^I get the correct \"([^\"]\*)\"$")

public void I\_get\_the\_correct(int anArabicNumber) throws Throwable {

assertEquals(anArabicNumber, RomanNumbers.toArabic(aRomanConversion.romanNumeral));

}

1. Now when we run the feature tests, a few pass now! Lets put in the remaining code to convert the single digits:

case X:

return RomanNumeral.X.getArabic();

case C:

return RomanNumeral.C.getArabic();

case D:

return RomanNumeral.D.getArabic();

case M:

return RomanNumeral.M.getArabic();

case L:

return RomanNumeral.L.getArabic();

1. When we run the tests, now the first scenario now works! I decided to not write another unit test as my feature tests covered everything I was expecting. This really validates the first ½ of my design, going from Roman to Arabic for simple numbers.
2. Lets look at finishing the translation from Roman to Arabic with more than just the simple characters. I don’t like the case statement, but that was just an initial step to verify my enum definition was working. Lets get into the refactor step while we take on the next challenge.
3. The next step is to continue working from the Outside In, getting the next step to fail. Lets generate the stub for more complex translations. Here is the feature:

Scenario Outline: translate common subtraction number combinations

Given a Roman Number simple combination

When I ask for a translation a "<Roman Numeral>"

Then I get the correct combination translation "<Arabic Number>"

Examples:

| Roman Numeral | Arabic Number |

| IV | 4 |

| IX | 9 |

| XC | 90 |

| XL | 40 |

| CD | 400 |

| CM | 900 |

| CCLVI | 256 |

1. We don’t really need the Given part of the statement, so we can remove it. Lets also adjust the working so we can make use of the other helper class to keep the steps less likely to step on each other.

Scenario Outline: translate common subtraction number combinations

When I ask for a translation a combination "<Roman Numeral>"

Then I get the correct combination translation "<Arabic Number>"

Examples:

1. Lets generate and then code the steps. The key difference is to use another instance of the helper class to avoid cross-talk between the steps.

@When("^I ask for a translation a combination \"([^\"]\*)\"$")

public void I\_ask\_for\_a\_translation\_a\_combination(String romanNumeral) throws Throwable {

aComplexConversion = new ConversionEntry();

aComplexConversion.setRomanNumeral(romanNumeral);

//no test done, this is setup for the next step

}

@Then("^I get the correct combination translation \"([^\"]\*)\"$")

public void I\_get\_the\_correct\_combination\_translation(int anArabicNumber) throws Throwable {

assertEquals(anArabicNumber, RomanNumbers.toArabic(aComplexConversion.romanNumeral));

}

1. Now when I run the step, the “setup” works, but I’m not getting the write value. Now I should write another unit test to drive the next part of my coding.

assertEquals("256 should be the outcome",256, RomanNumbers.toArabic("CCLVI"));

1. When I run this test, it now fails – there is not a enum of CCLVI. Now back to refactoring and getting rid of my case statement:

public static long toArabic(String aRomanNumber)throws IllegalArgumentException {

int arabic = 0;

String romanName;

for (RomanNumeral currentRomanNumber : RomanNumeral.values()) {

romanName = currentRomanNumber.getName();

while (aRomanNumber.startsWith(romanName)) {

arabic += currentRomanNumber.getArabic();

aRomanNumber = aRomanNumber.substring(romanName.length());

}

}

if (aRomanNumber.length() > 0) {

throw new IllegalArgumentException("Invalid Digit " + aRomanNumber);

}

return arabic;

}

1. Now my unit tests pass. Let see about the acceptance criteria.
   1. Now all of the simple tests pass.
   2. The negative simple tests have not been implemented yet
   3. All of the complex tests pass.
   4. The malformed complex numbers negative tests have not been implemented yet
   5. The Arabic to Roman tests have not been implemented yet
2. Before we finish up the error handling, lets make sure the Arabic to Roman testing works. We could have done this earlier to make sure our enum approach works both ways. Lets put in a test and see it fail:

@Test

public void testRomanToArabic() {

assertEquals("I should be the outcome", "I", RomanNumbers.toRoman(1));

assertEquals("V should be the outcome", "V", RomanNumbers.toRoman(5));

}

1. Now lets put in the code into RomanNumbers make this test pass.

public static String toRoman(int arabic) {

StringBuilder result = new StringBuilder();

for (RomanNumeral numeral : RomanNumeral.values()) {

while (arabic >= numeral.getArabic()) {

arabic -= numeral.getArabic();

result.append(numeral.name());

}

}

return result.toString();

}

1. The unit tests now pass. Lets write some glue code in the step defs.. Need to adjust my helper class:

public class ConversionEntry {

String romanNumeral;

Integer arabicNumber;

public void setRomanNumeral(String romanNumeral) {

this.romanNumeral = romanNumeral;

}

public void setArabicNumber(int anArabicNumber) {this.arabicNumber = anArabicNumber;}

public String getRomanNumeral() {

return romanNumeral;

}

public int getArabicNumber() {

return arabicNumber;

}

}

1. Now we can write the step defs:

@When("^I ask for a Roman translation of a \"([^\"]\*)\"$")

public void I\_ask\_for\_a\_Roman\_translation\_of\_a(int anArabicNumber) throws Throwable {

anArabicConversion = new ConversionEntry();

anArabicConversion.setArabicNumber(anArabicNumber);

}

@Then("^I get the correct Roman Numeral \"([^\"]\*)\"$")

public void I\_get\_the\_correct\_Roman\_Numeral(String aRomanNumber) throws Throwable {

assertEquals(aRomanNumber, RomanNumbers.toRoman(anArabicConversion.getArabicNumber()));

}

1. Hmm. Now we have many of the AC tests pass, but one fails:
   1. Examples:
   2. | Arabic Number | Roman Numeral |
   3. | 1993 | MVIIM |
2. We have an example where the acceptance criteria is wrong. Lets go ahead and fix that to be MCMXCIII. Now those tests pass.
3. So now lets to after the exeception handling. Lets put in the failing unit test:

@Test(expected=IllegalArgumentException.class)

public void testInvalidRomanNumeralB() {

RomanNumbers.toArabic("B");

}

1. That test passed, so now lets add another:

@Test(expected=IllegalArgumentException.class)

public void testInvalidRomanNumeralSequence() {

RomanNumbers.toArabic("VC");

}

1. Hmmm. That one passed as well. Lets go the other way toRoman:

@Test(expected=IllegalArgumentException.class)

public void testInvalidRomanNumeralOne() {

RomanNumbers.toRoman(0);

}

1. This test fails, we should have seen an exception. So lets add the code to get the correct exception:

private static final int LOWER\_NUMERAL\_LIMIT = 1;

private static final int UPPER\_NUMERAL\_LIMIT = 3999;

if (arabic < LOWER\_NUMERAL\_LIMIT || arabic > UPPER\_NUMERAL\_LIMIT) {

throw new IllegalArgumentException(String.format("Only numbers between %s and %s is supported.", LOWER\_NUMERAL\_LIMIT, UPPER\_NUMERAL\_LIMIT));

}

1. Application should now be ready, let see the glue code in the step def pass now.
2. Generate the step def place holders
3. Put this code in to look for an exception as a good thing for bad Roman input:

@Then("^I get the invalid numeral error$")

public void I\_get\_the\_invalid\_numeral\_error() throws Throwable {

try {

RomanNumbers.toArabic(aRomanConversion.romanNumeral);

fail("Invalid RomanNumeral");

}

catch (IllegalArgumentException e) {

//do nothing here as the exception was expected

}

}

@Then("^I get the invalid arabic error$")

public void I\_get\_the\_invalid\_arabic\_error() throws Throwable {

try {

RomanNumbers.toRoman(anArabicConversion.getArabicNumber());

fail("Invalid ArabicNumeral");

}

catch (IllegalArgumentException e) {

//do nothing here as the exception was expected

}

}

1. We run test tests and all pass again with one exception. It appears the spec is incorrect again, where CM is not an error.

Code Coverage

When the code coverage is turned on inside of Intellij, here is what we get:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Feature Testing | cucumber.kata.RomanNumbers | 100% (1/1) | 66% (2/3) | 94% (17/18) | | Feature Testing | cucumber.kata.RomanNumeral | 100% (1/1) | 100% (5/5) | 100% (20/20) | | Unit Testing | cucumber.kata.RomanNumbers | 100% (1/1) | 66% (2/3) | 94% (17/18) | | Unit Testing | cucumber.kata.RomanNumeral | 100% (1/1) | 100% (5/5) | 100% (19/19) | |  |  |  |
|  |  |  |  |