Test driven development in Mendix

As we all know, Mendix allows to really quickly build functionally rich applications. This is wonderful for our customers as they can quickly see their application come together and it allows them to ask for extensions and modifications as their requirements change. However there is a potential risk to this rapid building, delivering and modifying the applications and that is that the testing can't keep up with the pace of development. Especially allowing fast-paced modifications without full regression testing can be disastrous. One approach to combat this is to use Test Driven Development.

Test Driven Development (TDD) proposes that you first write a unit test based on the story that you are working on and then write the code to make the test pass. And the code is the simplest code that could possibly work. The tests stay with the code and every time some code is written or changed, ALL the tests are run to assure that the modifications didn't break any existing functionality.

Within Mendix these 'rules' need to be amended a bit as you cannot run a Mendix project before it compiles. And you can't run tests before the project is compiled, so within this project we will create the minimum necessary microflows (mf) to at least be able to compile the project. A MF only requires to have its signature (input parameters and output) to make it compile so that is typically the starting point of each microflow.

Within this tutorial we will build an application that understands Roman Numerals – most people know roman numerals and implementing the rules has the right level of complexity for this tutorial. The inspiration to use Roman Numerals came from the book "Good Math, A Geek's Guide to the Beauty of Numbers, Logic, and Computation by Mark C. Chu-Carroll, copyright 2013 The Pragmatic Programmers". Extract from the book are reproduced with permission of the publisher. Please visit www.pragprog.com for more information.

In the Mendix app store there is a module available for unit testing however to build a solid understanding of unit testing we will build our own basic Unit Testing framework with assertion microflows and a simple testrunner UI.

All the code for this tutorial can be found on Github - https://github.com/flyingdutch20/Mendix-TDD.

Unit Testing framework

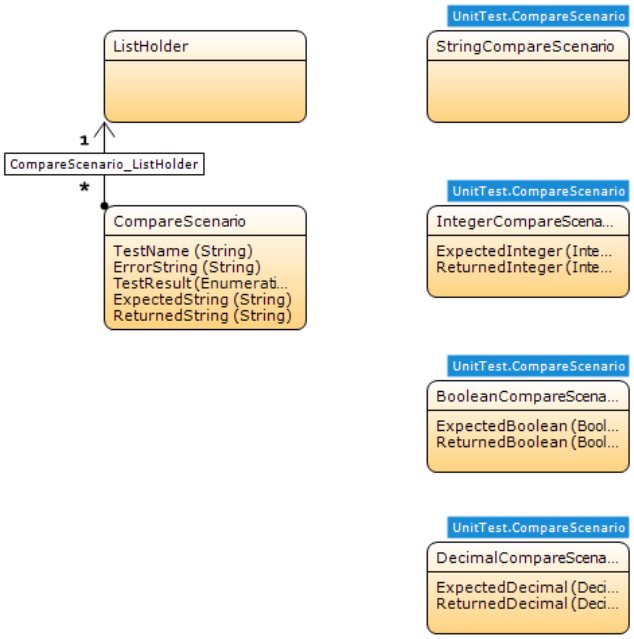
A unit test consists of a call to a piece of code with a set of input parameters, where the code returns something. The output of the call is then compared with the expected output (the assertion). The comparison is the result of the unit test.

The simplest way to achieve this in Mendix is to have assertions for the different types that need to be tested; assertBoolean, assertString, assertInteger and assertDecimal.

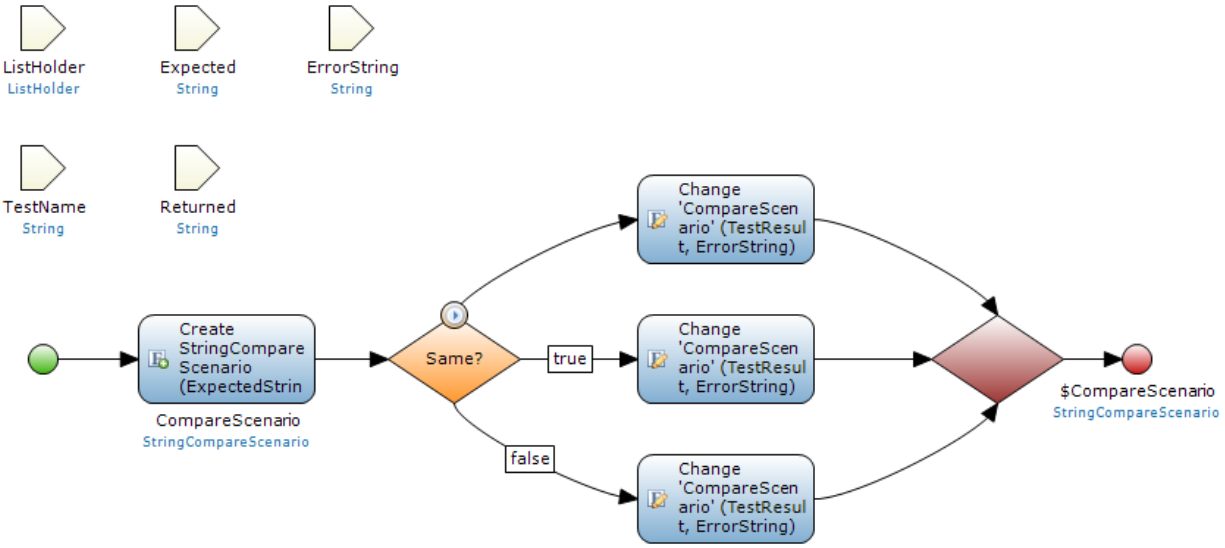
The assertions use a non-persistent object to contain the expected value, the returned value, the result of the test, the name of the test and an error message in case the assertion failed.

These objects are held together by a non-persistent list so they can be displayed in a list. Through generalisation the different types use the same parent object. For Integers, Decimals and Booleans their respective Expected and Returned values are parsed into the ExpectedString and ReturnedString for display purposes.

TestResult is an enumeration with the values Passed, Failed and Error. The values have icons attached to them so that you can quickly see if all is 'green'.

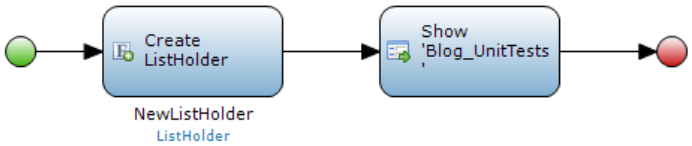


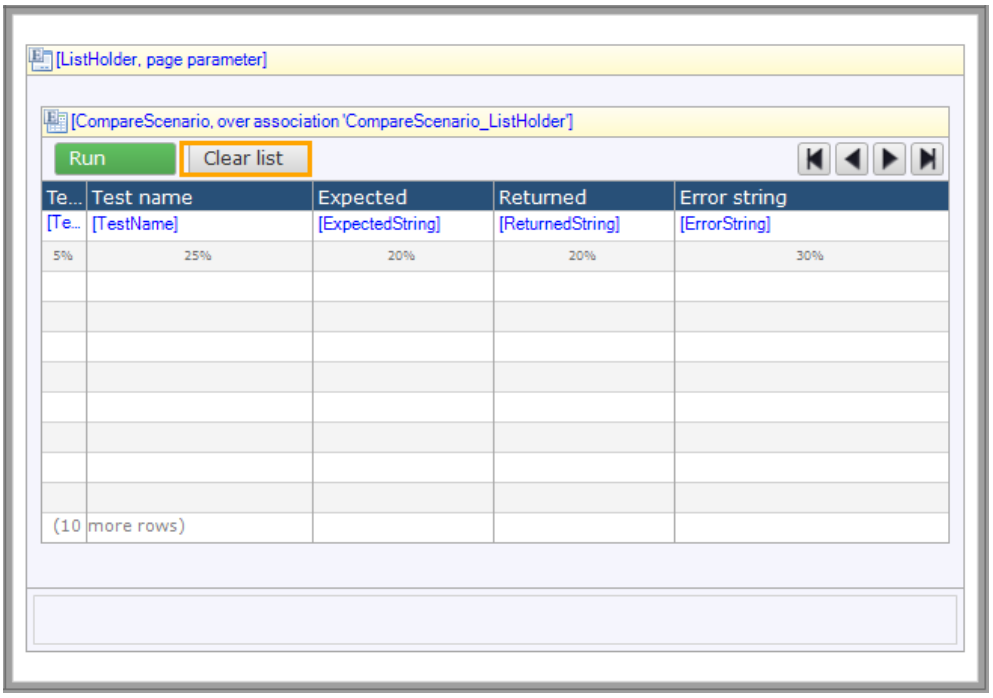
For each type there is an assert microflow that looks as follows:



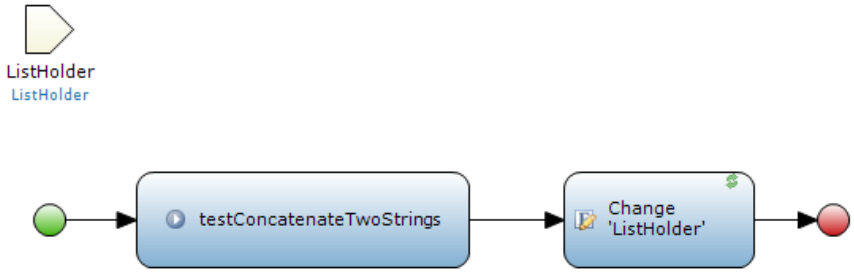
In case the expected and returned are the same, the testresult is set to ‘Passed’ and the error string is set to blank. If the expected and returned are not the same, the testresult is set to ‘Failed’ and the ErrorString is set to the ErrorString parameter. If the comparison results in a runtime error, then the TestResult is set to Error and the ErrorString is set to ‘Runtime error’. Note that this is different from a test failure, where the expected and returned values were different.

From the Admin homepage you would provide access to the UnitTest UI which shows an empty list and a button to run the tests.



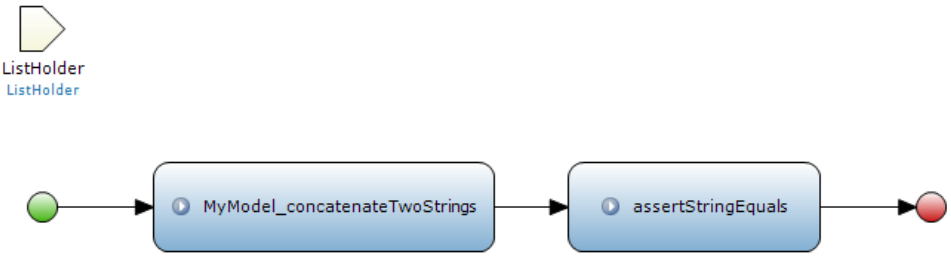


The Run button is connected to a microflow that calls the individual unit test microflows;

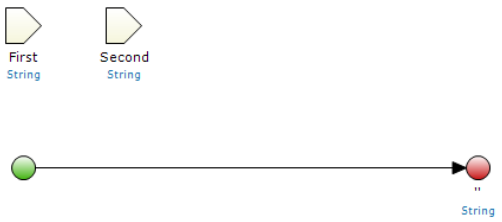


We will be building these unit tests as we go along but their format is always the same; call a microflow that returns something, then feed that into the assertion microflow where it is compared with the expected value.

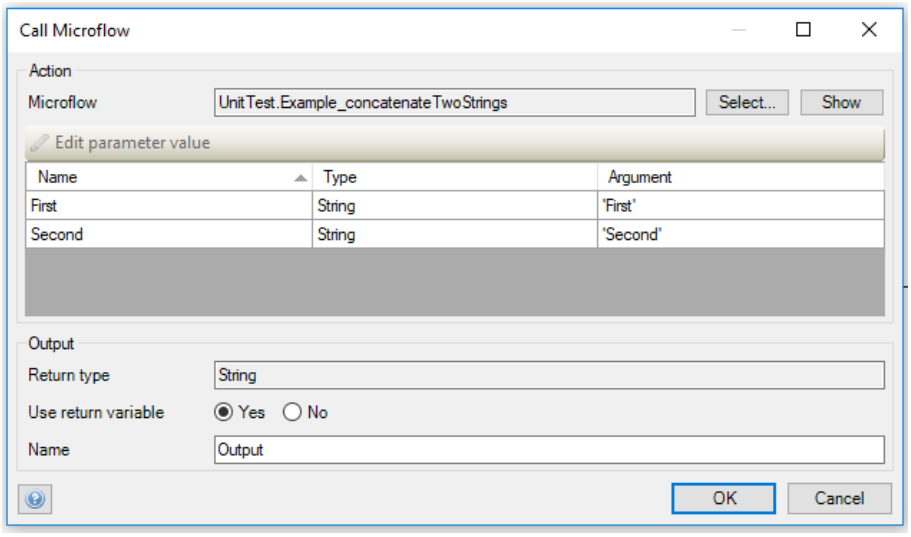
We could for example have a microflow where two words are concatenated. Our unit test for that MF would then be as follows:

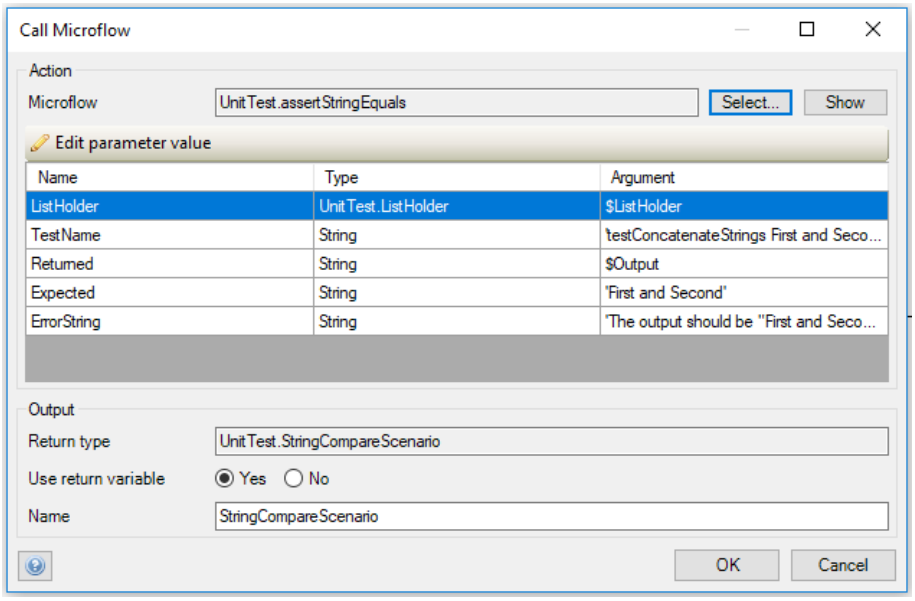


The microflow “MyModel_concatenateTwoStrings” doesn’t exist yet (we’re doing Test Driven Development!), so we create it while building the testMF. Our parameters are two input strings and we expect a string as output. The simplest starting point is just to return an empty string.



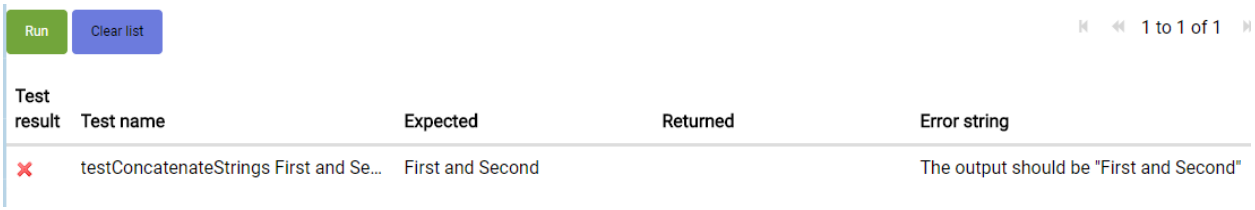
Setup the parameters in the microflow calls:



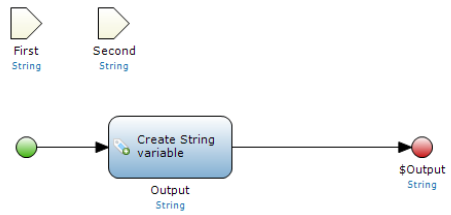


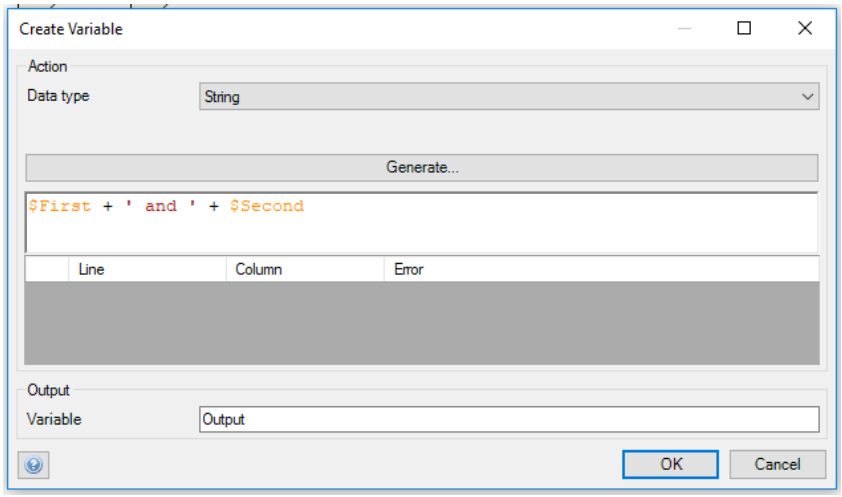
The ListHolder is the list that shows all the tests. The TestName is to identify the test. Returned is the output of the previous MF and expected is what you want the output to be. If it isn't then you want an error message to show the problem.

Now we actually have code that should compile, so F5 (run locally), log in as admin, go to the UnitTest UI and hit the Run button.

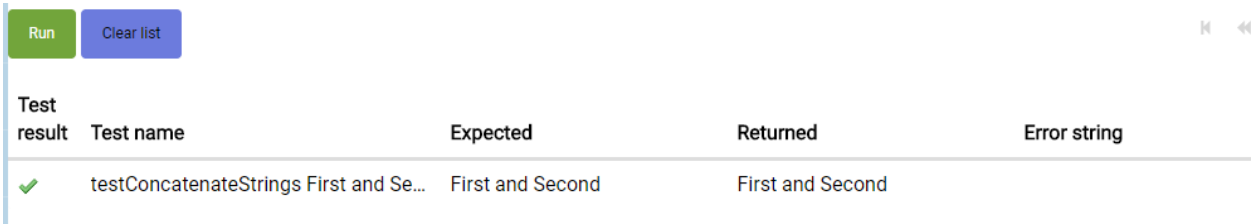


The test fails of course as we just returned an empty string. Now we have something to work on! Modify the “MyModel_concatenateTwoStrings” MF to actually return the concatenated string as specified.





Compile, login, go to the UnitTest UI and Run:



Congratulations, you have your very first passing Mendix Unit Test!

Roman Numerals User Stories

For Roman Numerals (RN) we can write the following stories:

1. As a user I want to enter a single Roman Numeral letter and get the equivalent decimal number, where “I” = 1, “V” = 5, “X” = 10, “L” = 50, “C” = 100, “D” = 500 and “M” = 1000. For now, any other character should return 0.
2. As a user I want to enter a string of Roman Numerals and get the decimal sum of those Roman Numerals. Again, we ignore the order of the characters and any invalid character just adds 0.
3. As a user I want to be able to use ‘subtractive prefixes’, so that when I enter a lower RN in front of a higher RN, it gets subtracted instead of added, for example ‘IV’ = 4 whereas ‘VI’ is 6. Note that in this case the order is important.
4. As a user I want to be able to add Roman Numerals together and have a Roman Numeral returned. Addition of Roman Numerals is basically concatenating and sorting, so ‘XII’ + ‘VI’ = ‘XVIII’.
5. As a user I want to be able to enter two Roman Numerals and get the sum as a Roman Numeral. While entering the Roman Numerals I want to see the numeric equivalent of both the entered values and the sum.

This will do to start with. Once we have these stories done you can create further stories like subtracting Roman Numerals, improving the UI, converting from decimal numbers to Roman Numerals and implement all this functionality through TDD.

Preparation

Before we start, we create a module for the Roman Numerals and a separate module for the Roman Numeral tests. Of course the test module would have dependencies with the main module, but the main module should have no dependencies to the test module. In that way we can deploy to production without the test module (although there is no real reason to strip them off).

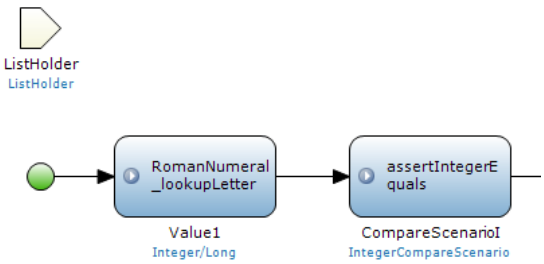
In the Roman Numerals test module we copy and amend the TestRunner UI and the MFs to open the UI and to run the tests from the UnitTest module. In the Administrator homepage we’ll add a button to call the mf to open the test runner.

Story 1.

As a user I want to enter a single Roman Numeral letter and get the equivalent decimal number, where “I” = 1, “V” = 5, “X” = 10, “L” = 50, “C” = 100, “D” = 500 and “M” = 1000. For now, any other character should return 0.

According to the story we provide a single character to a MF and get a number returned. We want to compare that number to what we expect, according to the list that we have from the story. So we need the mf ‘assertIntegerEquals’ and a mf that can lookup a RN letter.

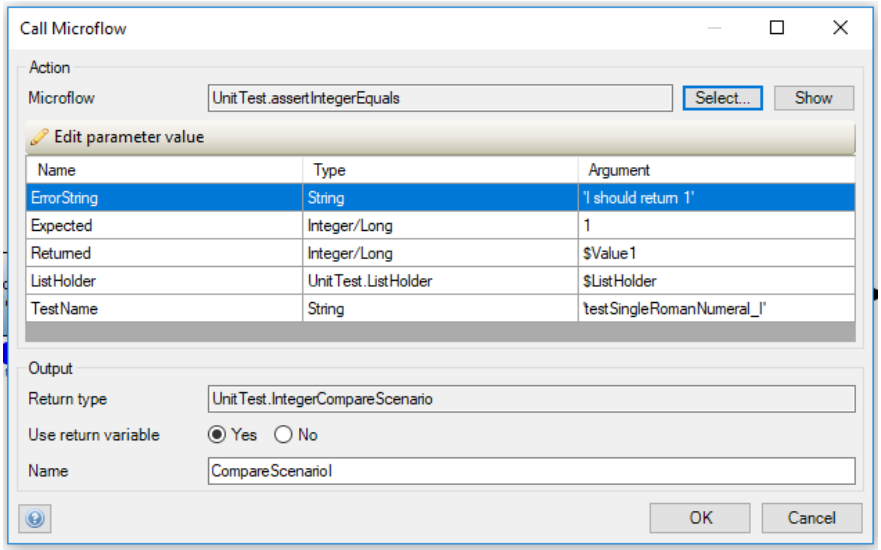
Let's start with setting up the test (testSingleRN):



Just like in the introduction, we start with creating a microflow with an input parameter of a 1-character string and as output an Integer. We'll call this microflow 'RomanNumeral_lookupLetter'. To make it compile we just set it to 0.

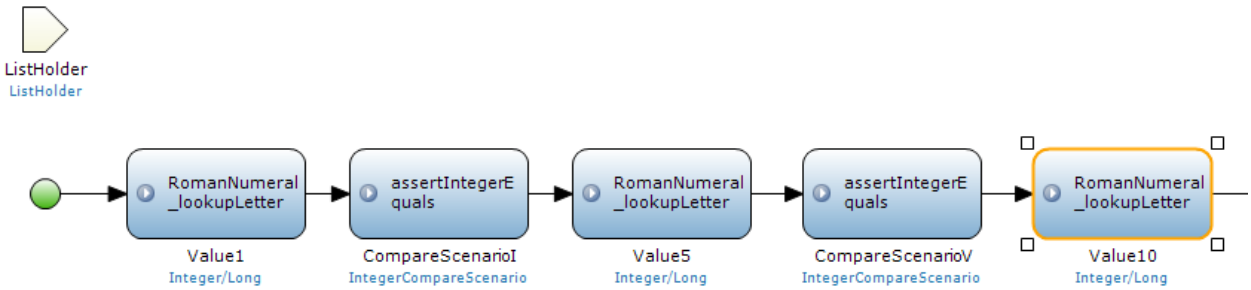
For the first test we set the input parameter of 'lookupLetter' to 'I'.

The assertIntegerEquals mf requires the following parameters:

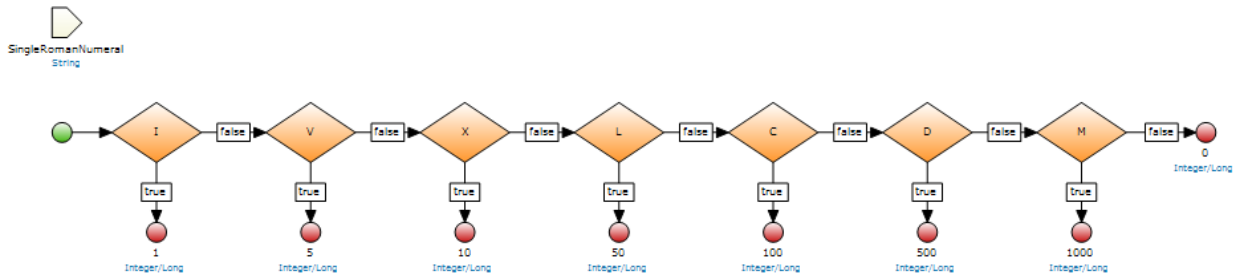


Add the 'testSingleRN' mf to the 'runTests' mf and we're ready to run the test. Compile, run and of course it gives a failure as the 'lookupLetter' mf returns 0 instead of 1. This is an important step though, as it proves that we can actually run the test and that the MF's all do their job. The next step is to fix it in the simplest way to make the test pass. Note the phrase 'simplest way'. This is very important. We just do what is needed to make it pass. When we come to subsequent tests we can make it more complicated to make those tests pass but at the same time make the current test pass. For experienced developers this might seem slow-going but the benefit of doing it this way is that you don't develop complicated solutions for simple requirements where the 'future-proof-ness' never actually comes off.

The simplest solution to make it pass is by changing the return value of the 'lookupLetter' mf to just return 1. Hardcoded. Yes. Compile, run, pass! Next test. Now we want to test whether it also works with the other valid and invalid letters. Add to the 'testSingleRN' mf to test the 'lookupLetter' mf with other letters;



Make sure you set the parameters in the lookup mfs and the assert mfs according to the values that you want to test. Include characters that are not used in roman numerals. According to the story they should return 0. Compile, run and of course you get failures on the subsequent tests. Amend lookupLetter to return the right values for the Roman characters (Simplest Solution!):

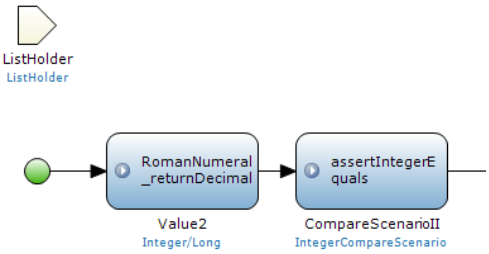


Compile, run and all the tests should pass. If not, use breakpoints in the failing test and debug until you find why it doesn't pass. Once they all pass, we've finished the first story!

Story 2.

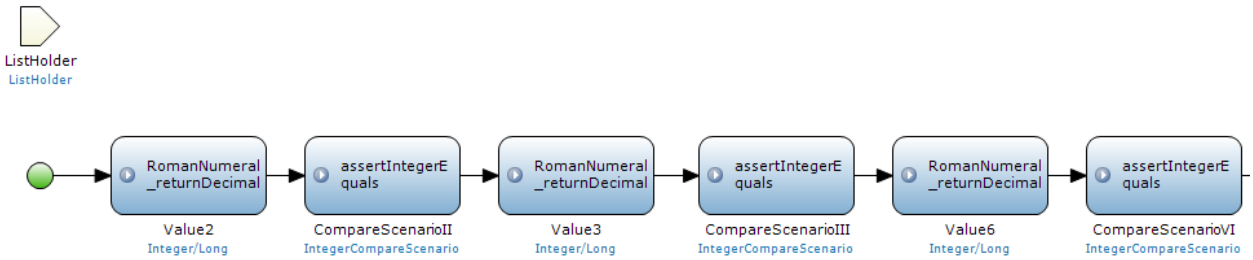
As a user I want to enter a string of Roman Numerals and get the decimal sum of those Roman Numerals. Again, we ignore the order of the characters and any invalid character just adds 0.

As always, we start with writing a test mf. This test drives us towards the clear naming of the actual mf and which parameters it will use. In this case we need a string of Roman Numerals and that should return a number.



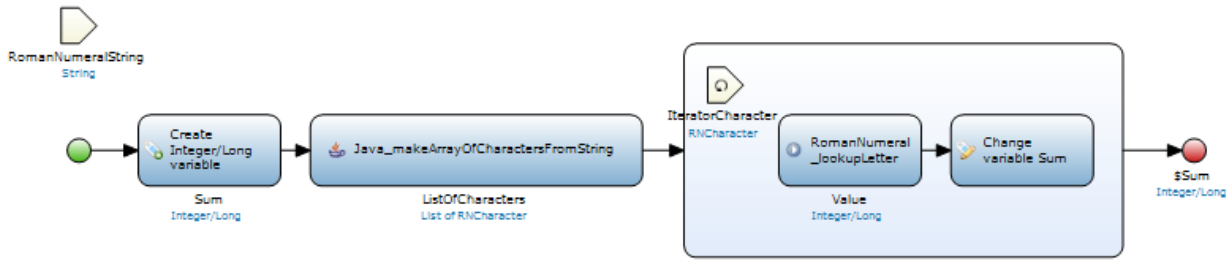
To make the test compile, we create the returnDecimal mf that just returns 0. Add the test mf to the runTest mf and compile, run. Red of course but it compiles. Note that we run the complete set, including the tests from Story 1. We want to assure that everything we do doesn't break what we've already built.

We hardcode the output to 2, run and pass. The next step is to extend the test with other roman numerals.

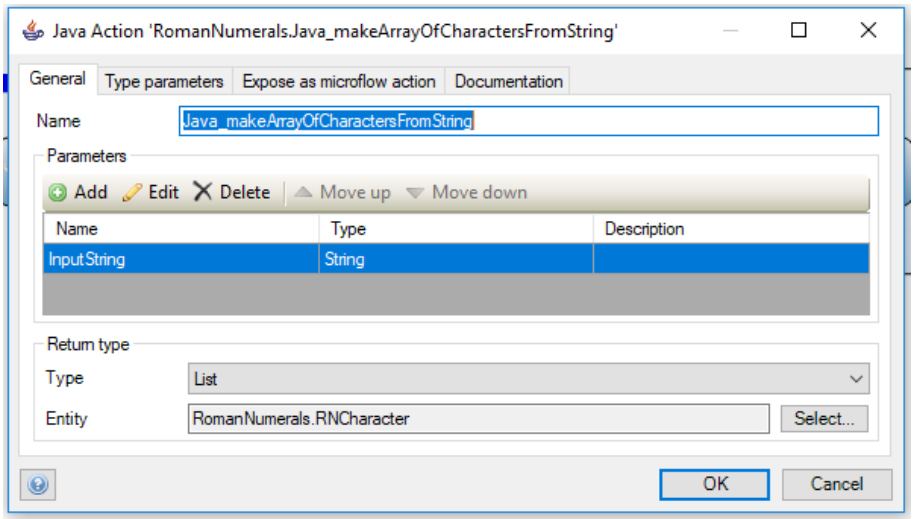


Now we have a challenge on our hands. As input we have a string of multiple characters. We want to walk through those characters and lookup what kind of value they represent. Once we have that value, we want to add it to the variable that we want as the return value.

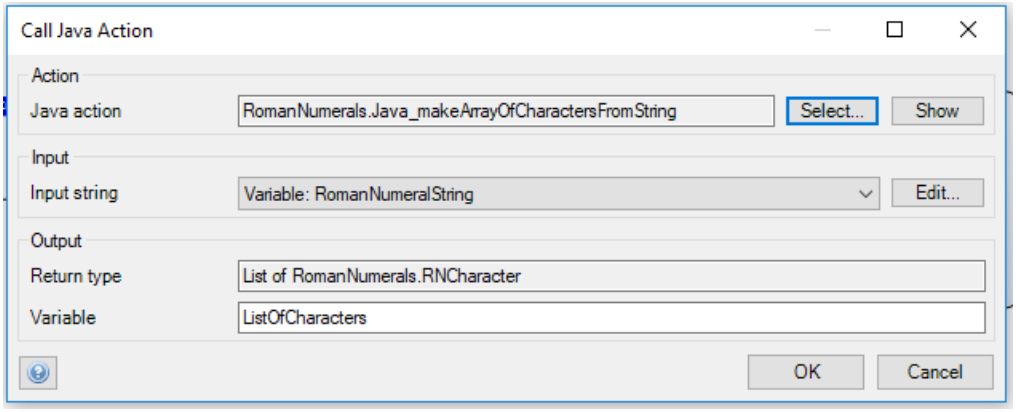
We could do this with the substring() function, a counter and a loop, but I personally found it easier to create a Java function for this. Send the Java function a string and receive a list of objects that have the individual character in. The objects only need to have a character in so we create a (non-persistable) entity with a single field of type String. I've called the entity 'RNCharacter' and the field 'Letter'. Note that these names need to correspond in the Java code below.



Define the Java action as follows:



The parameters for the Java call are as follows:



Then compile and run. Of course when you run the test you get an #ErrorNotImplemented back from Java in the console, but when you compile a project with a new Java action, then a template gets created in the deployment directory where you can easily enter the Java code.

Go to your Mendix project directory, in there go to javasource\romannumerals\actions. In that directory you will find a file Java_makeArrayOfCharactersFromString.java. Open this file in your favourite editor and amend it as follows:

…

```
import com.mendix.core.Core;
import romannumerals.proxies.RNCharacter;
import java.util.ArrayList;
…

// BEGIN USER CODE
ArrayList<IMendixObject> resultList = new ArrayList<IMendixObject>();
for(char c : InputString.toCharArray()) {
    IMendixObject myEntity = Core.instantiate(this.getContext(), RNCharacter.entityName.toString());

            myEntity.setValue(this.getContext(), "Letter", java.lang.Character.toString(c));
            resultList.add(myEntity);
}
return resultList;
// END USER CODE
```

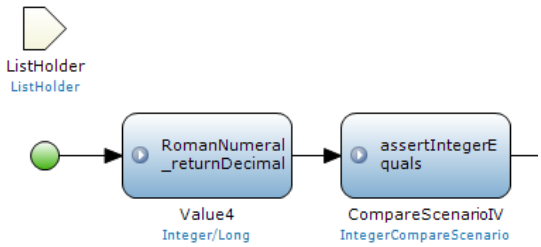
Once the Java code is in place, compile and run again. All green? On to the next story.

Story 3.

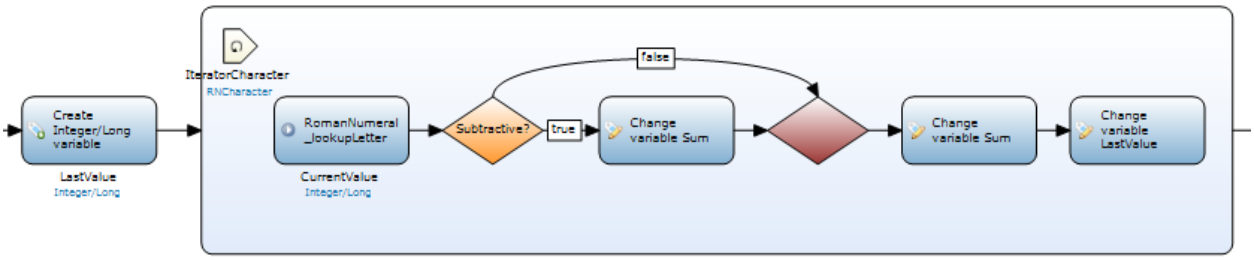
As a user I want to be able to use 'subtractive prefixes', so that when I enter a lower RN in front of a higher RN, it gets subtracted instead of added, for example 'IV' = 4 whereas 'VI' is 6. Note that in this case the order is important.

To shorten the number of characters needed in Roman Numerals, the Romans decided to use subtractive prefixes, which means that they can replace 4 characters with one by putting it in front of the next-higher character. Or if they put it in front of the two-next higher character, it replaces the next-higher character and the four characters that are the same. This trick can only be done with the characters “I” (1), “X” (10) and “C” (100). The “I” can reduce the “V” and the “X”, the “X” can reduce the “L” and the “C”, the “C” can reduce the “D” and the “M”. With these rules in place, we can start writing our first test. We will encapsulate this function within the returnDecimal mf, so all our existing tests for the returnDecimal function will ensure that we don’t break anything. This also means that we don’t have to add further tests for strings that don’t use subtractive prefixes.

In our first test we’ll use the “IV” string and assert that we get 4 back:



Compile, run and of course this test fails as it will return 6. So how are we going to fix this? At the moment we just get the value of the character and add it on to the total. What we need to do is check whether the current character has a bigger value than the previous, because then we know that we should have subtracted the previous character instead of adding it. Therefore change the loop in the returnDecimal mf to the following:



We need to have a variable that can store the last value and initialise it to 0. Then we check whether we should use Additive or Subtractive on the previous character by checking whether the value of the last character is smaller than the value of the current character. If that is the case we should have used subtractive in the previous iteration so we need to subtract the value of the previous character twice. First to adjust for what we did in the previous iteration and secondly because the character should reduce the total. This works in the first iteration as well of course as it will just subtract 0 twice from the total.

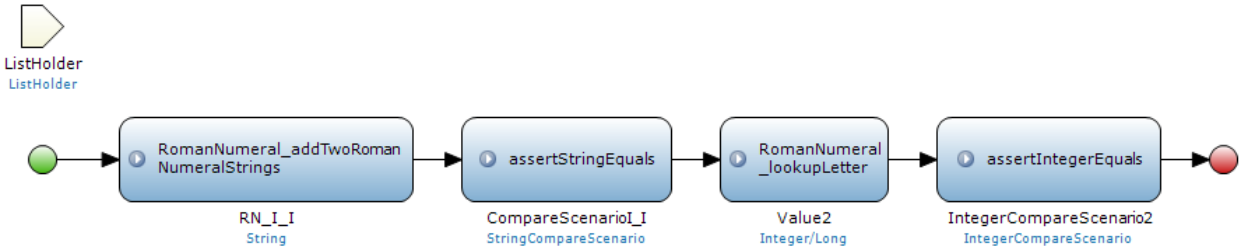
```
$LastValue < $CurrentValue

$Sum - $LastValue - $LastValue
```

Compile, run – all green. And more importantly, all previous tests where we used the same mf are still all green as well! Now you can add some further tests for other subtractive Roman Numerals, for example “IX” = 9, “XLIX” = 49, “MCMXCIX” = 1999. Make sure all is green before we start on the next story.

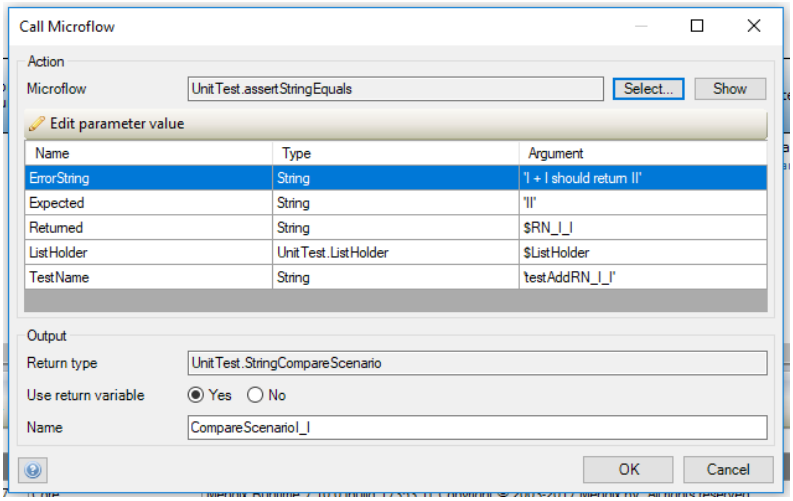
Story 4.

As a user I want to be able to add Roman Numerals together and have a Roman Numeral returned. Addition of Roman Numerals is basically concatenating and sorting, so ‘XII’ + ‘VI’ = ‘XVIII’.

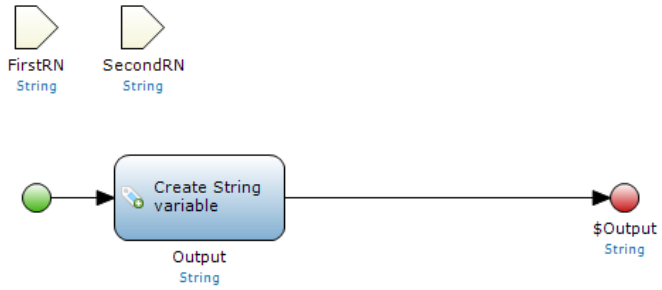
This sounds easy enough, just take two strings, concatenate them and sort them. Well let’s take it one step at a time. See what works already. We create a new microflow that has as input two strings and returns the two strings concatenated to a single string. From there we should be able to call the returnDecimal mf to find out what the number is. Step 1, write the test:



The assertStringEquals looks as follows:



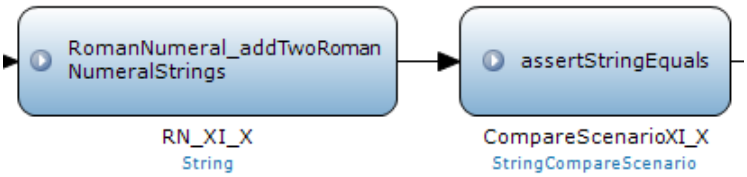
Do the simplest that could possibly work in the addTwoRomanNumeralStrings;

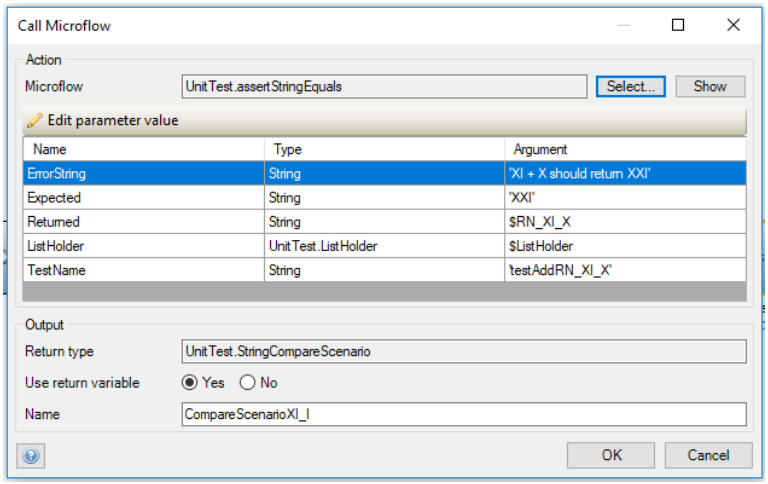


In the Output variable we just concatenate the two input strings;

```
$FirstRN + $SecondRN
```

Compile, run, green! Wow! Now of course this works in the simplest cases where the characters in the second string are all smaller than or equal to the rightmost character of the first string. As soon as a that is not the case, our subtractive code will mess the total up, for example “XI” + “X” would return “XIX” which is 19 where it should have returned “XXI” = 21. Now for this case we should sort the string and then it should work. We’ll add this as a test case and then we’ll make sure that it passes.

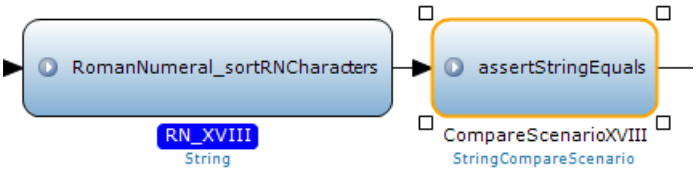




Compile, run should fail as we haven’t added the sorting yet. Let’s think what we need to do for sorting. As input we need a Roman Numeral string and as output we want a Roman Numeral string. I can see a new testcase bubbling up! Create a MF testSortRNString and add it to runRomanNumeralTests.

Subtask 1 - Sort

First we want to test that our sorting routine doesn't mess anything up that is already in the right order.



Here we pass "XVIII" to the new mf sortRNCharacters and we expect the return value to be the same. To make this test pass we of course create an output variable that we set to the input parameter.

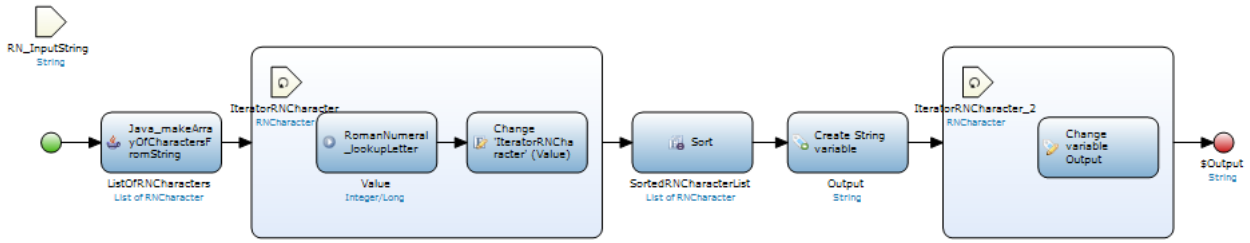
Next we extend the test where we do want the characters to get sorted, for example what we had above, "XIX" to be sorted as "XXI". Hold on, I hear you think, XIX is a valid Roman Numeral as well! Indeed it is, but as we're currently talking about the sorting of the string, let's just park this issue for now and we'll deal with it in a minute.

So how are we going to sort the characters? Well we know which number each character represents so if we make the characters into a list and just hang the number off each character. Then we can sort the list and concatenate the characters again.

In Story 2 we introduced the RNCharacter as objects in the outputlist of our Java action. It would make sense to extend that RNCharacter with the value so that we can attach it there and then just sort the list that we got back from the Java action. Let's give it a go!

1. Amend the entity RNCharacter with a field called Value and with type Integer.
2. In the mf sortRNCharacters, call our Java action to make it into a list of RNCharacters.
3. Loop through the list and retrieve the corresponding value. Change the Value field of the RNCharacter to the value that you've retrieved.
4. Once all the elements in the list have their corresponding value, sort the list descending.
5. Loop through the sorted list and attach each letter to the end of the output string.

Your mf should now look like this.



Time to test! Compile, run, Green! Feel free to add some more complicated strings and see if they sort in the correct way as well. Add some invalid characters and you'll see that they get pushed to the end as they have a value of 0. Note also that the sum test we did earlier, XI + X, has also turned green. Wonderful! Does that mean that we're done? Well as we mentioned before, our sorting routine messes up those Roman Numerals that use Subtractive Prefixes so let's see what happens if we use a number with subtractive prefix in an addition. Let's add a test "X" + "IX" which should of course return "XIX".

Compile, run gives a fail as we expected. So how can we fix this? Well the aforementioned book from where I got the inspiration to use Roman Numerals, actually set out a 'script' for addition;

1. Convert any subtractive prefixes to additive suffixes. So, for example, IX would be rewritten to VIIII.
2. Concatenate (or link together) the two numbers to add.
3. Sort the letters large to small.
4. Do internal sums (for example, replace IIIII with V)
5. Convert back to subtractive prefixes.

Aha, that makes sense! We've done number 2 and 3, but we should have first got rid of those subtractive prefixes.

Subtask 2 – Convert Subtractive Prefix to Additive Suffixes

Create the mf testConvertSubtractiveToAdditive and add it in the runRomanNumerals test. The function convertSubtractiveToAdditive should have an input parameter of a string and should output a string as well. Like we've done before, we first test where the conversion does nothing, calling convertSubtractiveToAdditive with input parameter "III" should output "III". Make that test pass like we did before. Next we want the simplest conversion to pass, where "IV" converts into "IIII".

We know from the previous story how we can find out whether a Roman Numeral uses Subtractive so maybe we can reuse some of that. What we did there was check whether the previous character was smaller than the current character. If that is the case, then we want to expand the current character into the same characters of the previous and subtract the previous character from it. This leads us to a new function; expandCharacter.

Subtask 3 – Expand Character

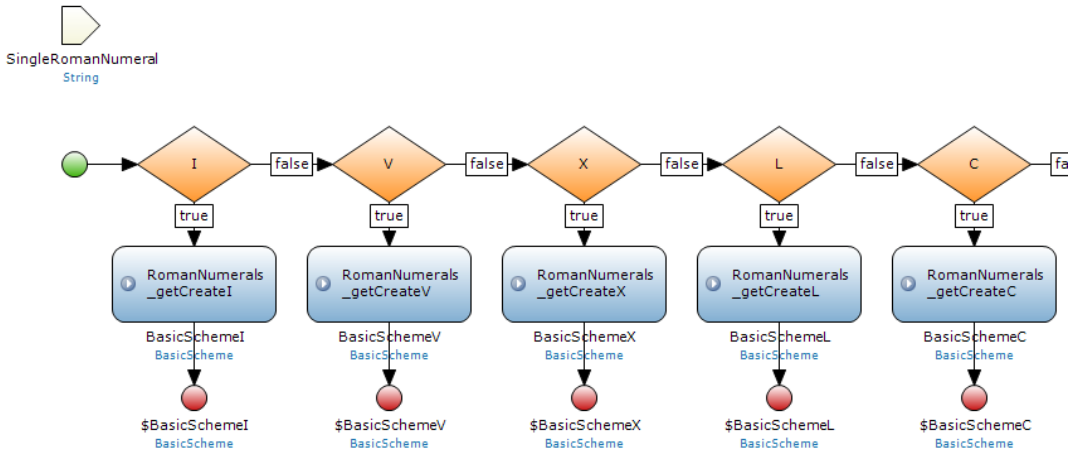
Of course we write a test for this where we can start with the character "I". If we throw the expandCharacter function to a character that can't be expanded, then we need to return the character itself. Easy peasy. Next we write a test to expand the character "V".

This requires the "V" to know that it can be expanded into "IIIII". That knowledge consists of two elements; which character precedes it and how many of that character are needed to be equivalent. It looks like we need to give the Roman Numeral letters some knowledge. Upto now when we did a lookupLetter, we just had a series of exclusive splits to return the equivalent number, however if we do lookupLetter, we should really return an object that can be asked to return the number, then we can extend that object with the knowledge which one is the following or preceding letter and how many of the preceding letters go into the following letter. This calls for some significant refactoring. Luckily we have tests to validate that everything we've done before still works.

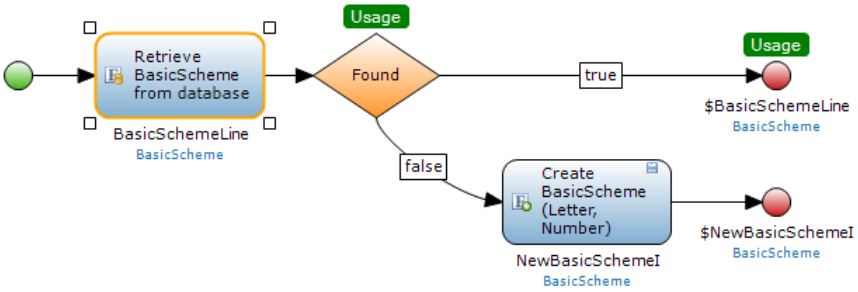
Our first step is to rename the lookupLetter into returnValueOfLetter because that better reveals its intention. As Mendix has numerous refactoring functions built-in, our renaming should still make all the tests pass. Then we duplicate that mf and rename the new mf into lookupBasicSchemeLine. This mf will lookup a letter and returns an object that has knowledge of the Roman Numeral scheme. I called the object BasicScheme which has two attributes, Letter (string) and Number (integer). The

returnValueOfLetter should call lookupBasicSchemeLine with the character and receive the BasicScheme line for that character. Then it can return the value of that BasicScheme object. We want to make sure that each valid character has exactly one line in the BasicScheme table. If it doesn't exist and it is a valid letter, we need to create it. To summarise we need to do the following:

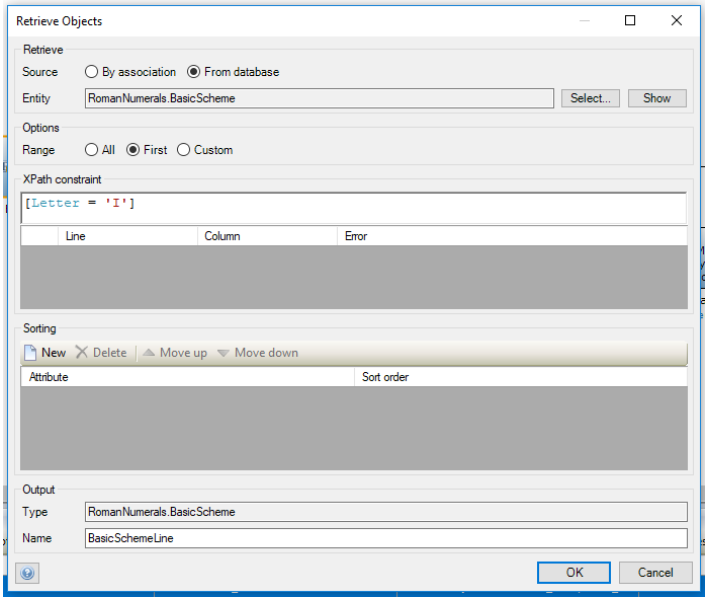
1. Create a new persistable entity called BasicScheme with two attributes;
   Letter – string(1)
   Number - integer
2. Rename lookupLetter into returnValueOfLetter – compile, run – all existing tests should still pass except the one that we're working on.
3. Duplicate returnValueOfLetter and rename it to lookupBasicSchemeLine.
4. In lookupBasicSchemeLine we call a specific mf for each character that checks whether the character exists. If it does, it returns it and if it doesn't, it creates it. If the parameter is an invalid character, we return 'empty'



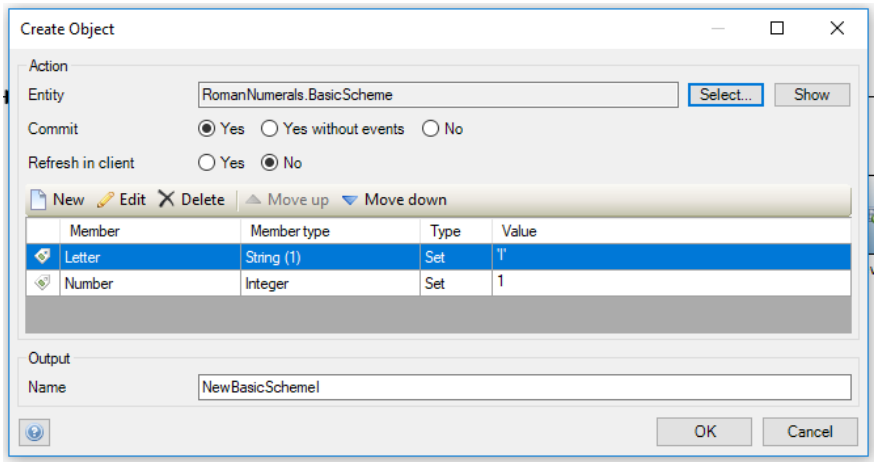
5. Each Roman Numeral character needs to have a 'getCreate' mf that looks as follows.



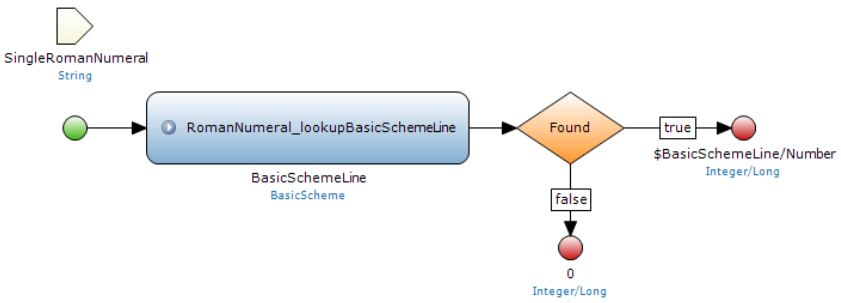
The retrieve looks as follows



And the create:

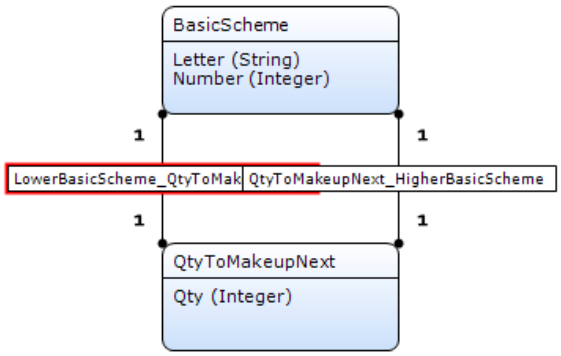


6. Change returnValueOfLetter to use lookupBasicSchemeLine:

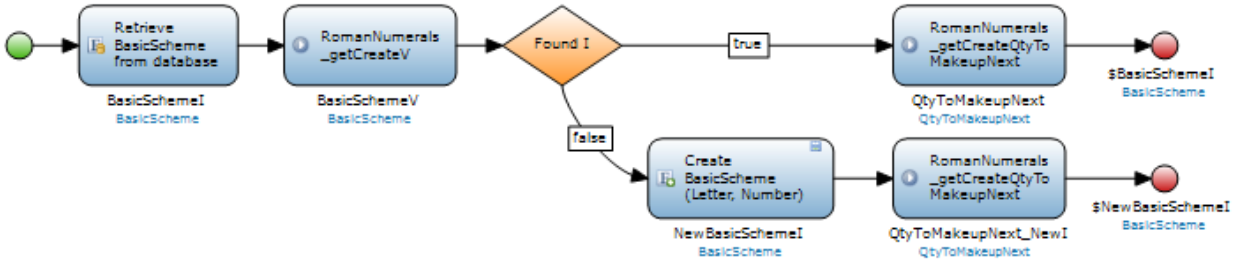


This is quite a significant change so we compile and run the tests. This should return passes for all the original tests – pfew. Next step is to make the 'expand' test on "V" work. For that the current character needs to know what the next smaller character is and how many of the next smaller character we need to make up the current character. For this we could make an association between the BasicScheme and

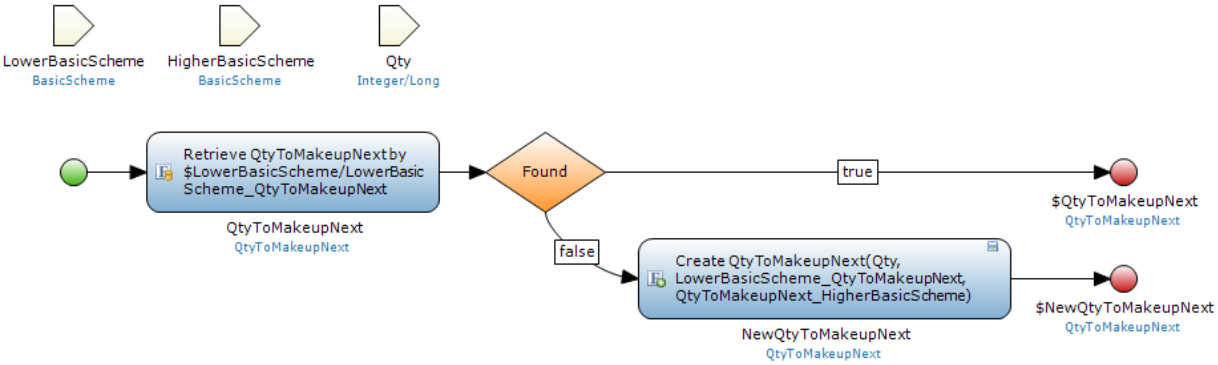
itself as a 'parent-child' relationship, however I've found that traversing that relationship doesn't always work as expected, so I prefer to use an intermediate object. In that intermediate object we can then also store how many of one we need for the other. We will call this intermediate entity 'QtyToMakeupNext' and it has one attribute; Qty – integer. It has two associations to BasicScheme, both of them 1 – 1 relationships. Make sure the naming of the associations are clear so we know which one we're talking about.



As we have a single point where we create the BasicScheme entities, we can just link into that single point to create these intermediate objects and associations. The character "I" needs to have a Qty of 5 and a link to the character "V", so we modify the 'getCreateI' as follows:

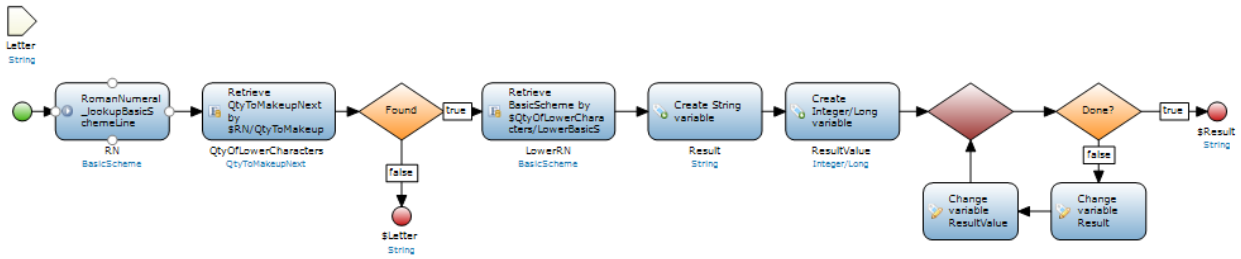


The getCreateQtyToMakeupNext needs the two BasicScheme lines and the quantity as parameters so that if it doesn't exist, it can be created.



For the highest letter, the "M", we of course need to do something different in that it should return 0 in the Qty and an empty HigherBasicScheme.

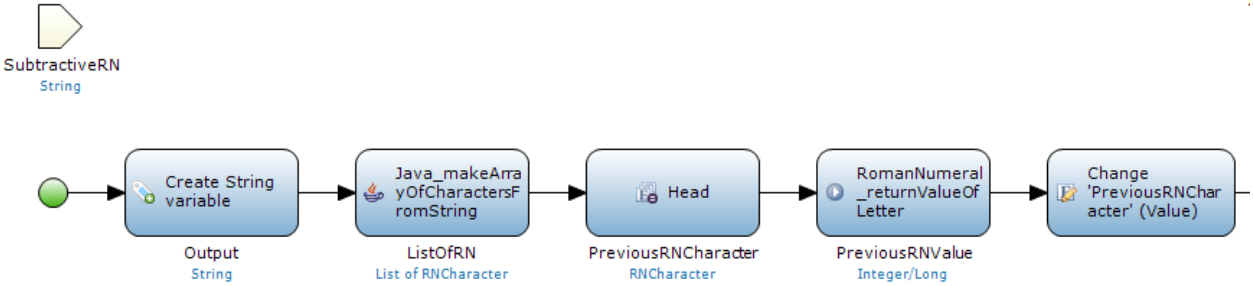
The advantage of creating the objects like this is that they automatically upgrade existing lines. Now we're finally ready to make the expandCharacter function work for the character "V".



We retrieve the RN by calling lookupBasicSchemeLine, then we retrieve the qtyToMakeupNext through the HigherBasicScheme association (we want to find the next one down). If there isn't one, we just return the character (so our first test should still work). Next we want to retrieve the character into which we want to expand, and then we loop until we have enough of the lower characters that makeup the higher character. Test with the "V", that should expand into "IIIII".

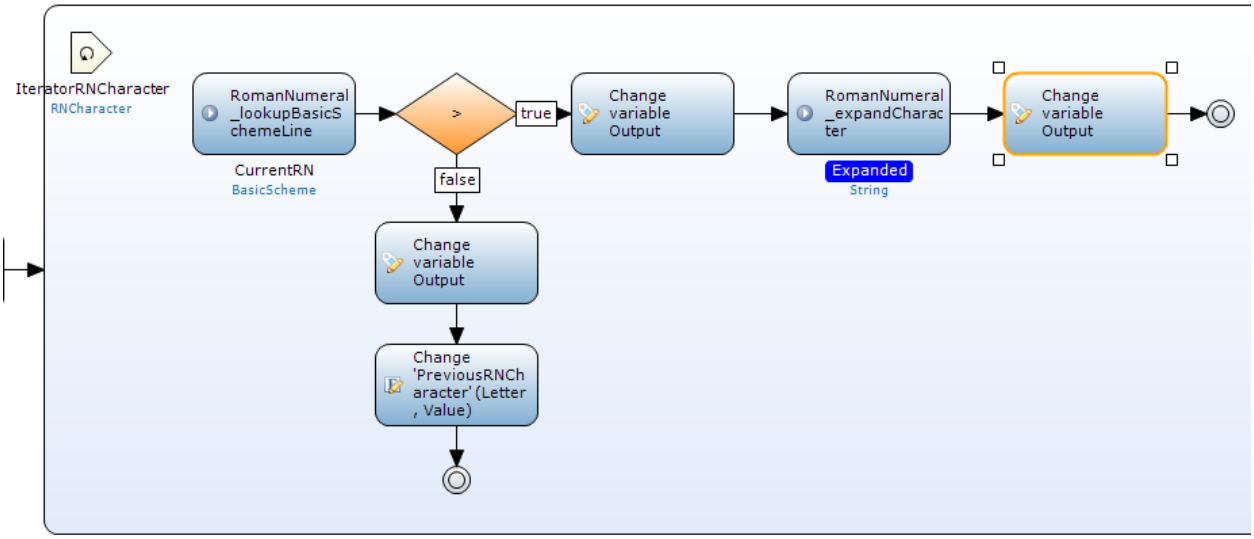
This all started with the fact that we wanted to convert "IV" into "IIII" in the mf convertSubtractiveToAdditive. I think we are ready for that now.

We want to walk through the list of characters and compare each character with the previous to see if we need to expand. So after we made the list through our Java action, we pick up the first Roman Numeral from the list with the Head function and find the value for that element with the returnValueOfLetter mf.



Next we loop through the list and compare the value of that RN with the previous. If it is equal or smaller, then we add the character to the output and set the current RN as the 'PreviousRN' for the next run of the loop.

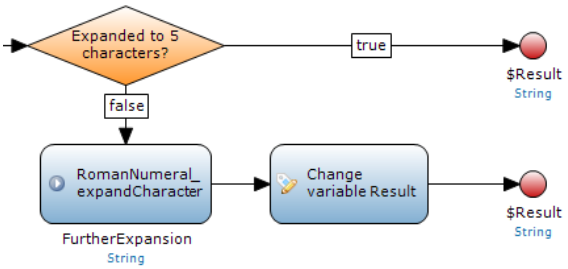
If the current character is higher than the previous, we're talking Subtractive so we need to do a bit more work. First we need to chop the last character from the output string as we shouldn't have put it there because it is just there to subtract from the current character. Then we call our expandCharacter mf which should return the current character as a string of the next one down characters. Then we want to chop the last character off the expanded string and stick the result to the end of the output string.

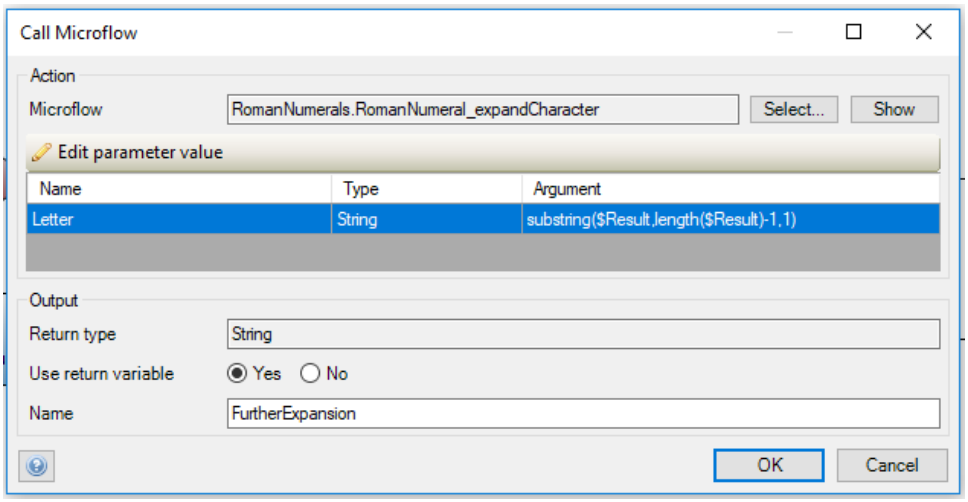


Compile, test and this should work for “IV”. However if we test it with some other numbers like “IX” then it fails, as our expand function expands “X” into “VV” so we need to do a bit more work on our expand function. We’ll start with adding a test that asserts that “X” expands into “VIIIII”. That test fails of course so now we are ready to amend the function.

Looking at the Roman Numerals, it appears that a number either expands into two characters or into 5 characters. (M into DD, D into CCCCC, C into LL, L into XXXXX, X into VV and V into IIIII). From here we can say that if the character was expanded into 2 characters, then the last of those two should again be expanded. Well, that shouldn’t be too difficult to do.

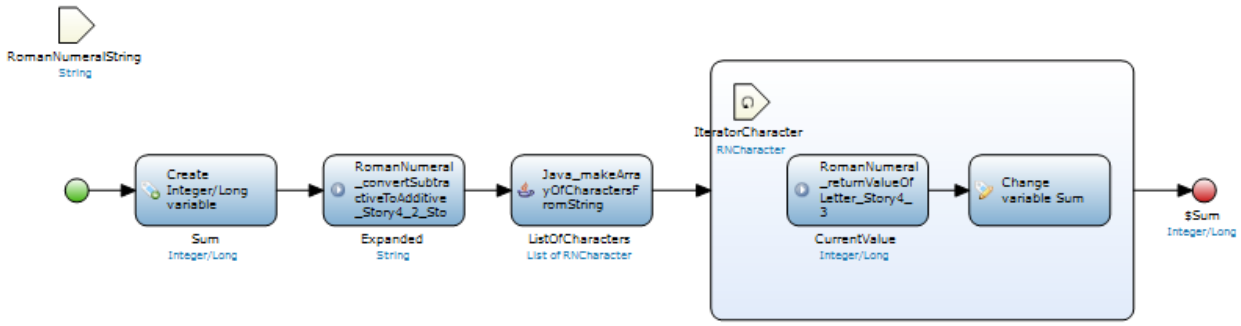
Stick the following to the end of expandCharacter where the parameter of the extra call to expandCharacter is the last character:





Compile, test, all green? Add some more tests, like “XLIX” = “XXXXVIIII and “MCMXCIX” = “MDCCCCLXXXXVIIII” and see if they work as well.

An important part of the Test Driven Development mantra is to refactor as soon as you see ‘code smell’ – ugly code, code duplication, unused functionality, code that does too much. As everything is captured within tests, you can change as much as you want. As long as everything continues to be green you stay on ‘terra firma’. With the last function that we built, we can actually improve some functionality that we built before. Remember that we had to cater for those Subtractive Prefixes before? We want that to have in one place only so that if we got it wrong or if the rules change, we only have to change it in one place. In our case it’s only a simple bit of code in returnDecimal, however if we expand the input string before we loop over it, then we don’t have to worry about Subtractive Prefixes in this code at all. With our tests we know that we can make our changes and don’t introduce new bugs.



As you can see this has significantly simplified the loop, no more temporary values apart from the total.

Where were we in the story again. Just to recap, our list of tasks was:

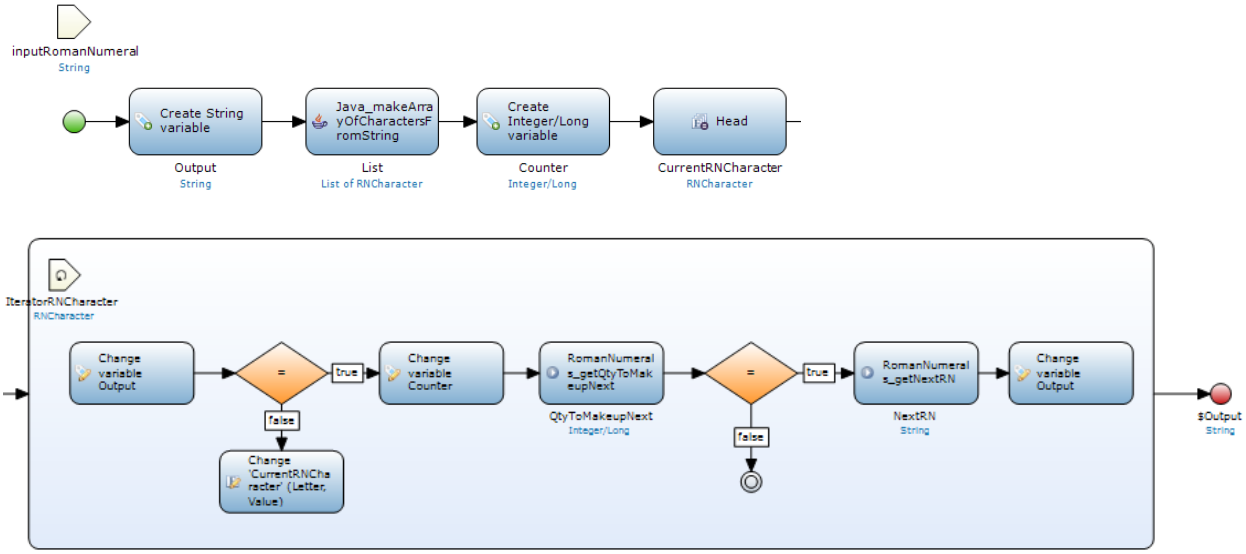
1. Convert any subtractive prefixes to additive suffixes. So, for example, IX would be rewritten to VIIII.
2. Concatenate (or link together) the two numbers to add.
3. Sort the letters large to small.
4. Do internal sums (for example, replace IIIII with V).
5. Convert back to subtractive prefixes.

We've done steps 1, 2 and 3 now so if we start with the string "XIX" and we add "XV" we should now get "XXVVIIII". In step 4 this should be converted to "XXXIIII" and in step 5 we should get it converted to "XXXIV"

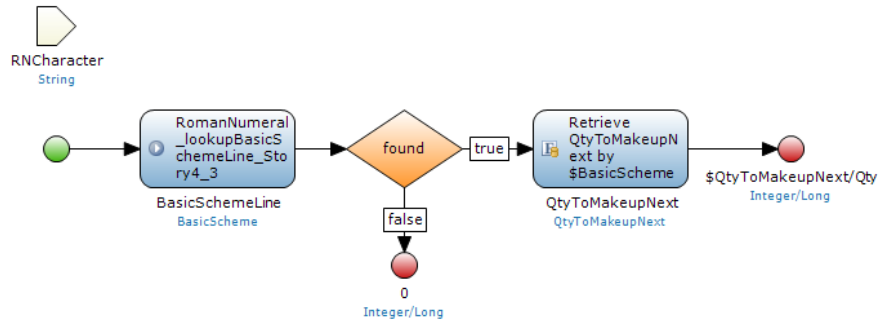
Subtask 4 – Do Internal Sums

As usual we start with writing the test, which also leads us to the clear naming of the microflow that we need. I think doInternalSums is intention revealing. The input parameter is a string and the output is a string as well. Our first test is to check that when we have a RN that doesn't need internal sums, we can call the mf and it will return the same RN. To make this test pass we just return the input string as output.

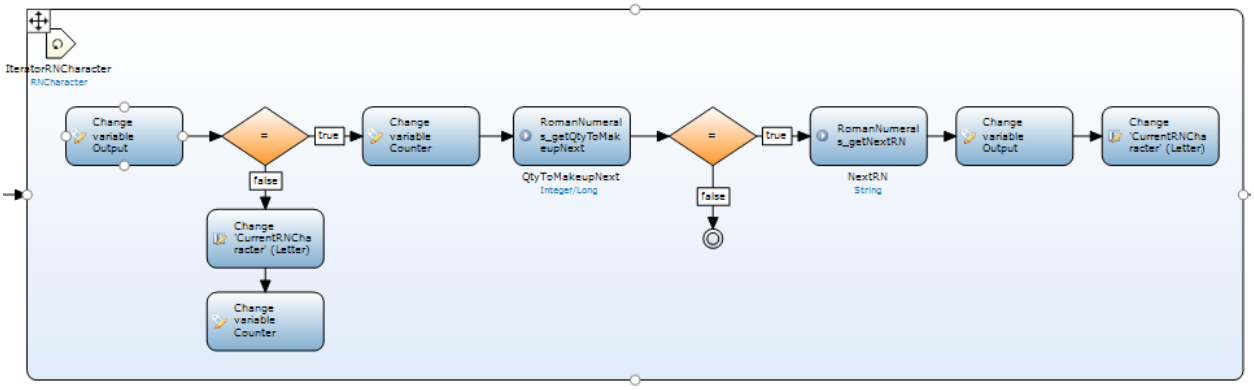
In our second test we will use 5 'I's as input and expect to get a 'V' back. What do we need to do to make that happen? We want to loop through the characters and check if the current one is the same as the previous. If that is the case, we want to count how may of these characters we've had and check how many we need to make up the next character. If we reach the number to make up the next, we have to find the next character and we have to replace the string of characters that made up the number with the new character.



We are using a microflow to get the quantity to make up the next number. That mf retrieves the BasicScheme line for that character and if found it retrieves the quantity.

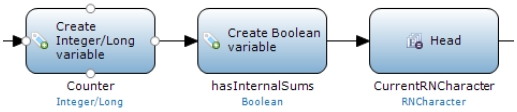


Compile, test, green. Time to put more complex scenarios in the test. What if there are characters in front of the sum, for example “XIIIIIII” = “XVII”. Green, good. What if the sum is in the middle, “XVVII” = “XXII”. Green, good. What if there are multiple sums, “CLLXVVIIIIIII” = “CCXXVII”. Red! What went wrong? Well, when the next character in the loop is different, we of course need to reset the counter, otherwise it just keeps adding up. We also need to reset the current character to the replacement character. This shows that this kind of testing makes debugging much easier. You can just focus into the area where the problem is. Solve it, retest the whole lot and assure that you haven’t broken anything else. In the same way, when you deploy the software to the functional testers (or even the end users) and they find a bug, then you can just add their scenario to the unit tests. Debug, fix, run all tests, assure nothing else broken, done.

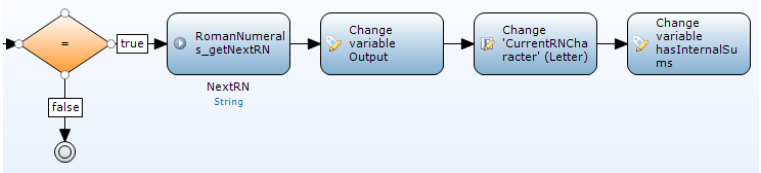


In the scenario where the output can be further reduced, we still get a red; “XVIIIIIII” = “XVVII” = “XXII”. If you write it out like this it automatically reveals the solution; when we do a replacement, we need to call our doInternalSums again to make sure it cannot be further reduced. So we add a flag that gets set when we do the replacement and we check if the flag is set. If it is set, we call it again. If it’s not set, we’re done.

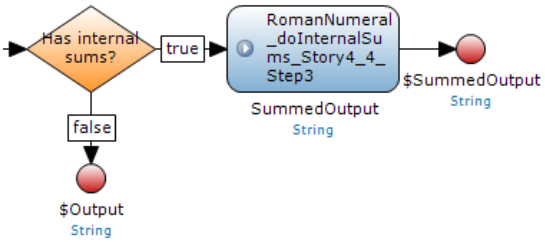
Initialise the variable:



Set the variable to true in the loop:



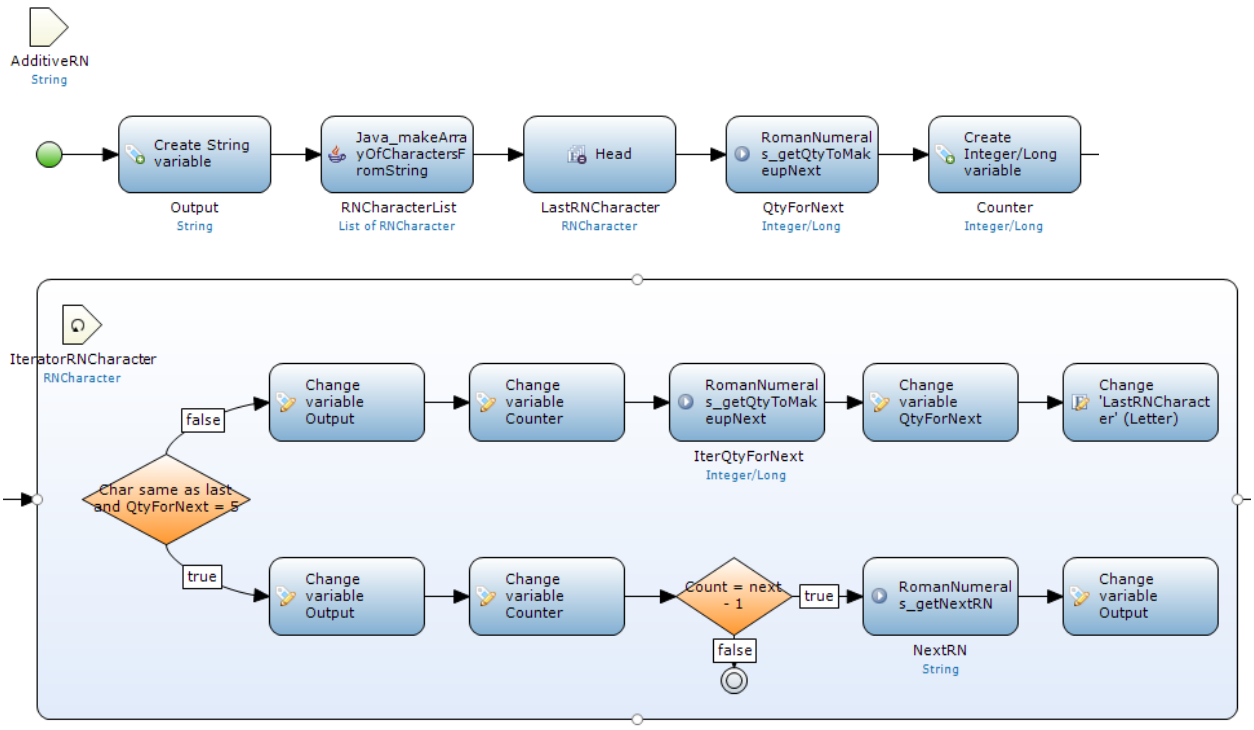
Check if it's set and rerun if necessary:



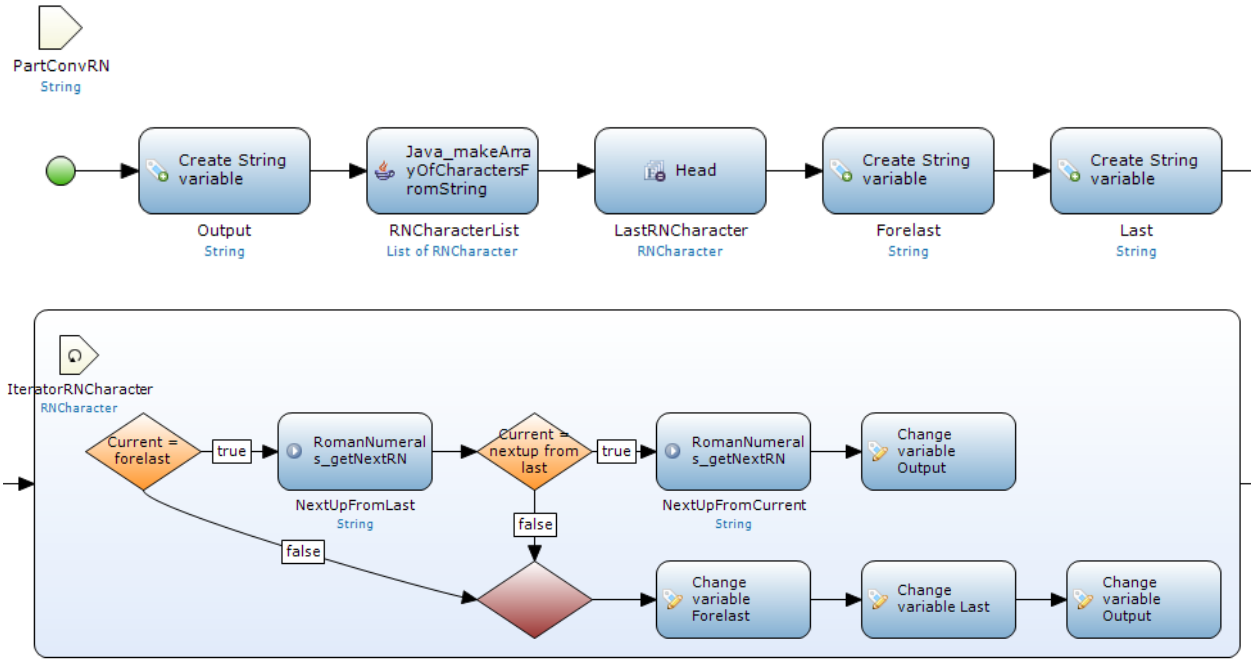
Compile, rerun the tests – all green. Add some more scenarios.

Subtask 5 – Convert back to Subtractive Prefixes

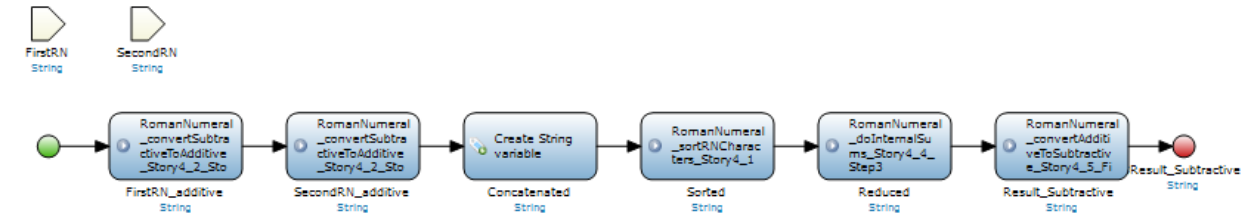
The last step in the addition story is to tidy up the Roman Numeral that we created by converting back to Subtractive Prefixes. Of course we start with writing a test that runs a mf that is supposed to convert a Roman Numeral to the equivalent with Subtractive Prefix. Well, we already have a mf that is called convertSubtractiveToAdditive so it makes sense to call this mf convertAdditiveToSubtractive. Our first scenario is a RN that doesn't need converting, like "XX" which should just stay the same. To make this test pass we just return the input string in convertAdditiveToSubtractive. In our second scenario we will do one of the simplest subtractive prefixes, the "IV", so our input is "IIII" and our expected output is "IV". Compile, run, red of course. So how are we going to fix this? Well, we loop through the characters and count how many we have (remember we have done the internal sums before this). If we have one less than how many makes up the next, we have ourselves a Subtractive (at least for this scenario). We're only interested in finding potential Subtractives if the QtyForNext is 5 so we can add that to our condition.



Compile, run, green. Now this covers the scenario where we have 4 of the characters where 5 make up the next RN character. Which other scenarios do we have? What about the scenario where we have one of the characters where 2 makeup the next and then 4 of the next one down, so for example “VIIII”. After the above code, this will have turned into “VIV”. So if we manage to convert “VIV” into “IX” then we might actually be ready. We could stick this on the end of the previous mf as it really belongs to the same function, however as it requires quite some variables it would get a bit lengthy so we will make it into its own mf and therefore have its own test.



Compile and run this test, then try some more Roman Numerals that should get converted. Once you’re happy, call this 2nd step mf from the convertAdditiveToSubtractive and add a call to convertAdditiveToSubtractive from the addTwoRomanNumeralStrings mf. This microflow should now look as follows:



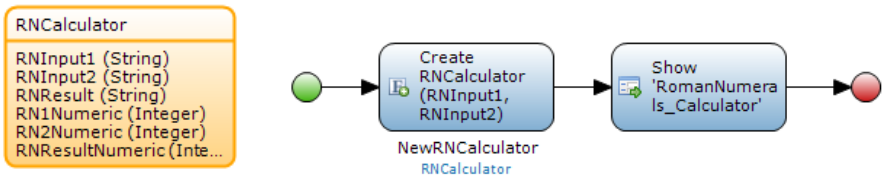
Compile and run the tests – now we should see all the tests green, including the ‘add X and IX’ that stayed red upto now.

Story 5.

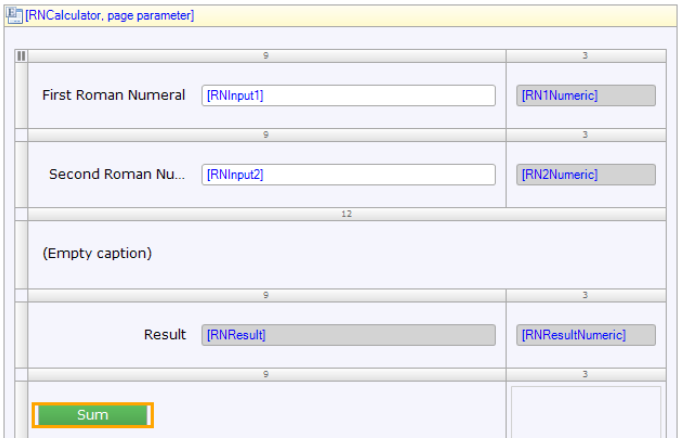
As a user I want to be able to enter two Roman Numerals and get the sum as a Roman Numeral. While entering the Roman Numerals I want to see the numeric equivalent of both the entered values and the sum.

This story is all about building a UI where the functions that we built actually get used. The UI's typically don't form part of Unit Testing, they typically get tested by the acceptance team. For UI testing there are very good testing tools available like Selenium. However if during UI testing bugs in the functionality are revealed then we add those scenarios to our Unit Tests and hence I've included this story.

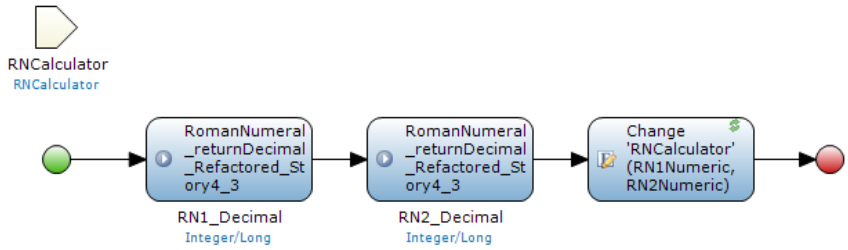
The easiest solution to build a UI for our application is to create a non-persistent object that has three string fields and three numeric fields (integers); two string fields for inputting the Roman Numerals and one to hold the sum. Once we have this entity we can build a form that uses that entity. To open the form we create a mf that creates the entity and opens the form. From the navigation screen we create a menu option for the microflow.



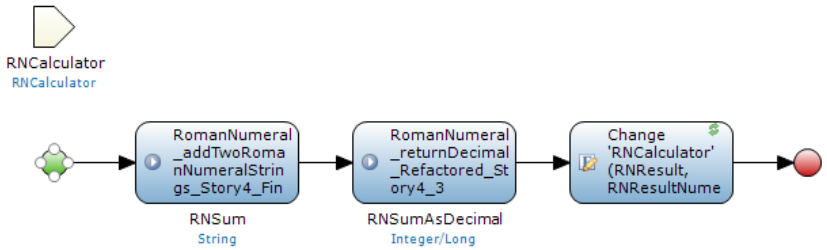
In the form we show the 6 fields where only the two input fields are editable. Set an event handler microflow on the On Leave event that updates the numeric equivalent fields of the input fields. Add a button that calls a mf that works out the sum and updates both the Roman Numeral sum field and the numeric sum field.



OC_InputRN:



IVK_CalculateRNSum:



Compile and open the form. Enter a Roman Numeral in the first field and exit the field. BOOM! What happened there? Have a look in the console. The mf convertSubtractiveToAdditive seems to have a problem. It appears that it tries to do Head on an empty list that came from our Java action. That head is of course empty and therefore we cannot set the value on it. So we clearly missed some scenarios where the input value is empty. We'd better add the empty string to our scenarios! Then change the convertSubtractiveToAdditive to check whether the list is empty in which case we just return an empty string without going into the loop. Compile, run, all green. Open the form again and enter some Roman Numerals in capitals and all works happily. However if you enter the Roman Numerals in lowercase, it all goes horribly wrong again! Same mf, same point. In this case the list wasn't empty so our solution didn't work. Of course we could capture the input and make them capitals before we start our process (a good idea anyway), however then we still have the issue where an invalid character returns an empty object and we therefore can't set the value.

In good TDD fashion, we'll do things one at a time; first we'll make sure we can handle lowercase characters so we add a test to the testStory2_SimpleMultipleRN as that is where we started working with the full Roman Numeral. As input we'll set a valid Roman Numeral, however in lowercase. And based on that we expect to get back the correct number. Now where would be the best place to convert from lowercase to uppercase? Well, maybe the best way is not to do that at all and just allow the user to enter either lowercase or uppercase or even a mixture. This is where the product owner should really give clarification. But our safest bet is to just allow both so in our lookupBasicSchemeLine we will allow for either in our comparisons.

The remaining issue is that empty object for an invalid character. Again this is a question for the product owner, whether they would allow the user to enter invalid characters and what to do with them. However in our case we do not want it to break our system so we want to ensure a test with invalid characters does not cause an error. At the moment the faulty character causes an error when we call convertSubtractiveToAdditive so that is where we need to add a test for the invalid character. When we

run the test we get the error that we saw before so we know that we are in the right place. Debugging shows that in the loop we are asking the CurrentRN for its value. The CurrentRN is a BasicScheme object, which was the lookup of the IteratorRNCharacter. However, as the character is invalid, we return 'empty' and we can't ask 'empty' for its value. So the easiest solution is to actually initialise an object in lookupBasicSchemeLine for invalid characters, so that we can ask it for its value, which is 0 of course. In that object we want to set the character to empty so that when we add it to our output, we add an empty string.

Of course we can tidy the UI further by converting everything to capitals but that is left to the reader. The main point of this story is that when you build the UI you are likely to come across scenarios that you didn't cover in the original story so you feed them back into the original tests.

Final thoughts

Have you noticed that almost all our microflows are relatively simple and relatively small? The reason for that is that most of the mfs only do one thing and then return whatever they've done, because in our tests we setup certain input and expected certain output. Another advantage of this approach is that the microflows that we created have clear intention revealing names.

We also concentrated our interaction with databases and other input/output devices to a specific set of mfs. Of course in this problem domain there is little requirement for databases and I/O devices, but by isolating these functions from the functions that calculate and make decisions, you get clearer programs that are easier to maintain. Many people currently talk about functional programming, where this separation is a clear aim. By separating the I/O and database interactions from the other functions, you get 'pure' functions, functions without side effects. The advantage of those functions is that they are easily testable as they will always give the same result given a set of input parameters. When functions interact with mutable data, the outcome cannot be predicted as the data might have changed in between two test runs.

Note that we did not pay any attention to performance, even though in the end the test runs did start to take a noticeable amount of time. However the mantra should always be to first make it work and then start focusing on performance. Once the tests are in place we can start profiling to find out where the bottlenecks are and then refactor those. Having the tests in place we are always sure that the functionality doesn't change.