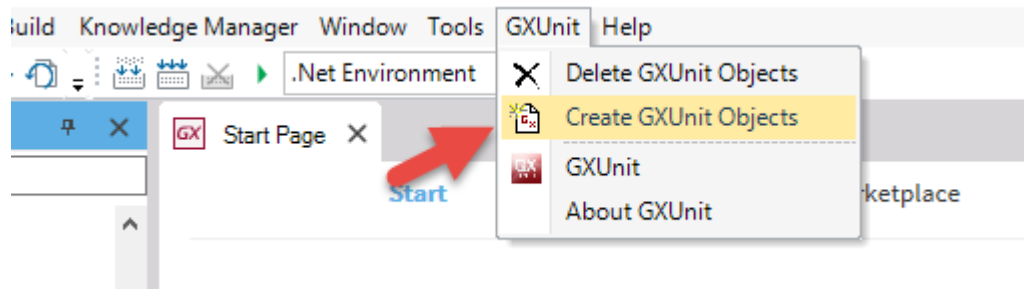


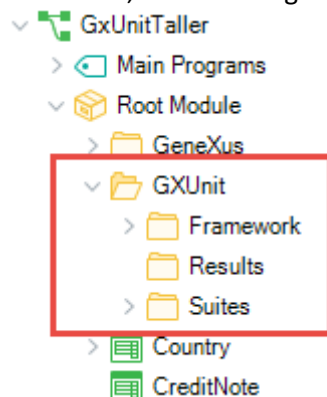
Practical Unit Testing with GXUnit

What is the correct way to define unit tests?

First a unit test folder should be created in the project. To do this select the GXUnit -> Create GXUnit Objects option from the top menu.

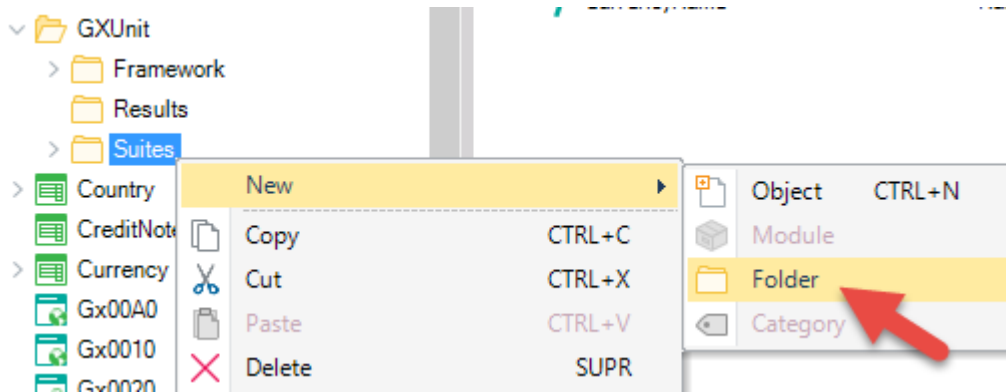


As a result, the following structure will be created within the KB.

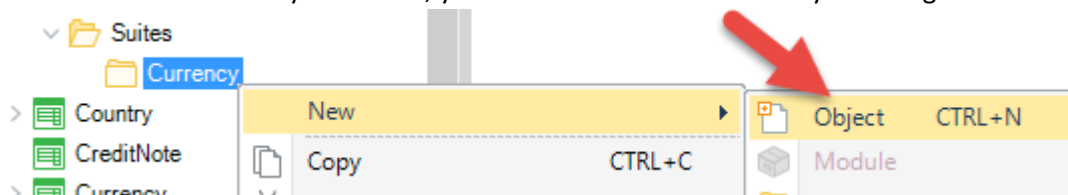


Suppose we are asked to add the attribute Counter to the Currency transaction, to take a counter of how many times it was used, by default this attribute must be initialized to 0.

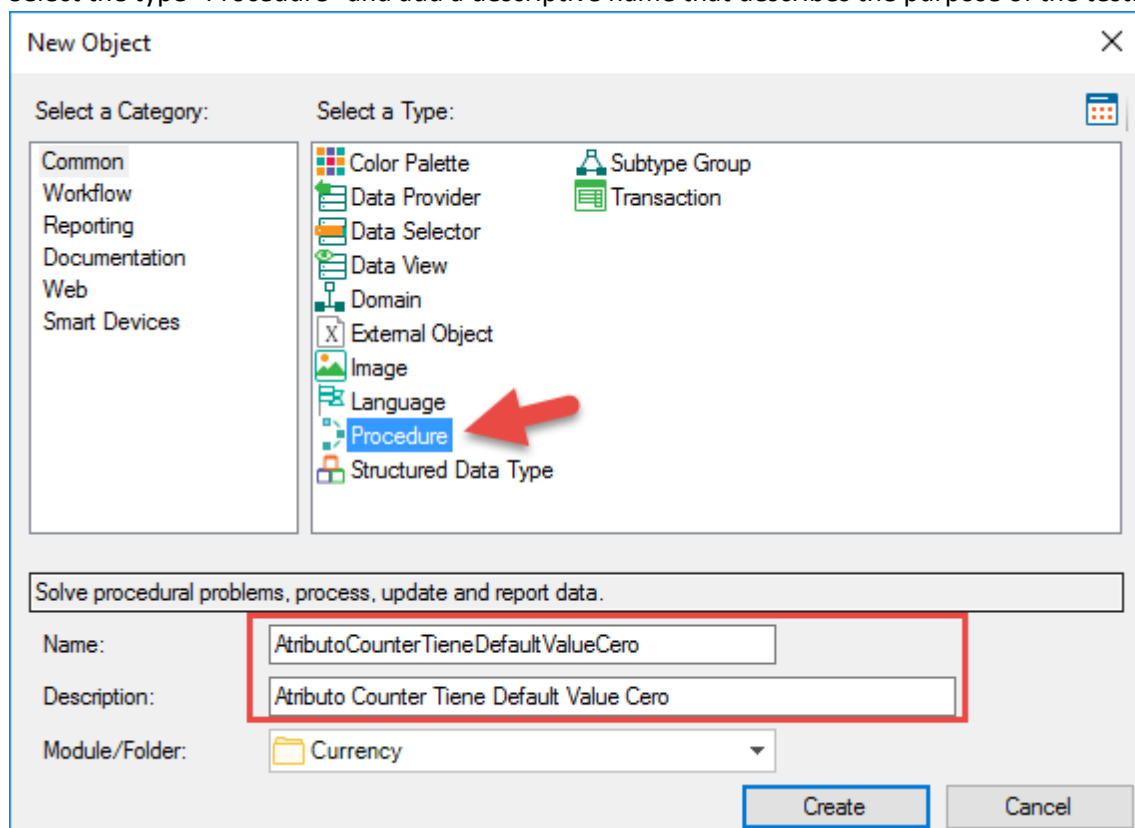
As a first step you should create a test suite for the Currency transaction. Inside the folder Suites the folder Currency is created.



Now inside the Currency Test Suite, you must create the new test by selecting New -> Object.



Select the type "Procedure" and add a descriptive name that describes the purpose of the test.



Now you must write a test to check that Counter is initialized to 0. First generate the variable & Currency and add it to the procedure with right click -> add variable, Genexus will already take the Currency type for the variable. Then run the save function to the & Currency variable
testing@genexusconsulting.com

and use the GXUnit AssertNumericEquals function to check the default value of the counter attribute. **NOTA:** GXUnit cuenta con dos procedimientos para comparar, AssertNumericEquals el cual compara numéricos y AssertStringEquals el cual compara Strings.

The case should be as follows:

```
1 &Currency.Save()
2
3 AssertNumericEquals(&Currency.Counter, 0)
```

When you try to save the test, the following error is displayed.

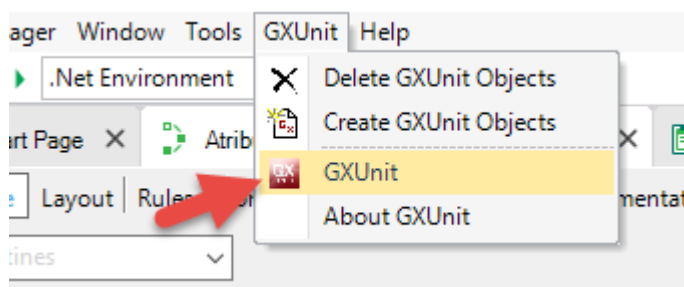
error src0216: 'Counter' invalid property.

Now we add the code to run our test successfully. First add the Counter attribute in Currency.

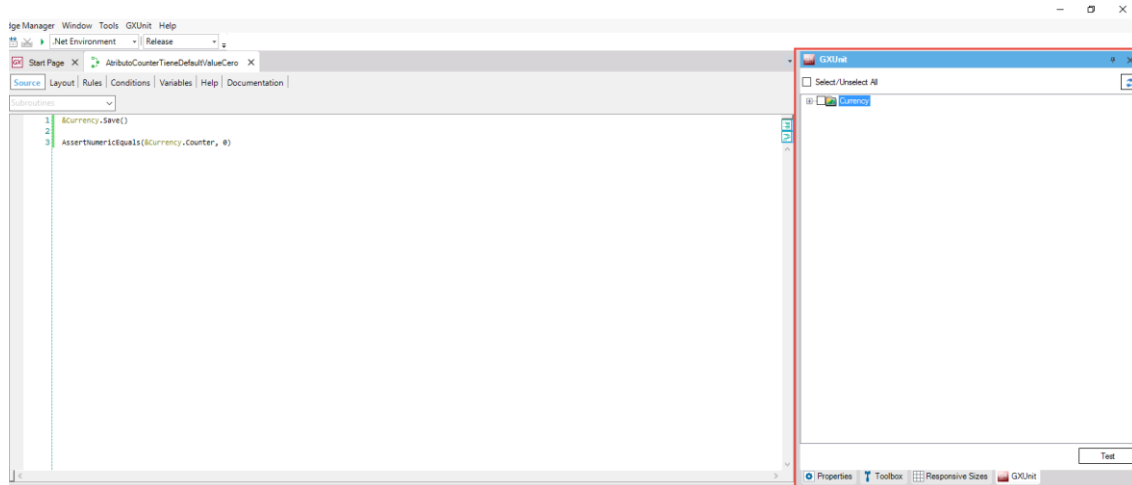
Name	Type	Description
Currency	Currency	Currency
CurrencyCode	Character(3)	Currency Code
CurrencyName	Name	Currency Name
Counter	Numeric(4,0)	Counter

A rule for default value 0 will not be added since, for Numeric types, the default value is already used.

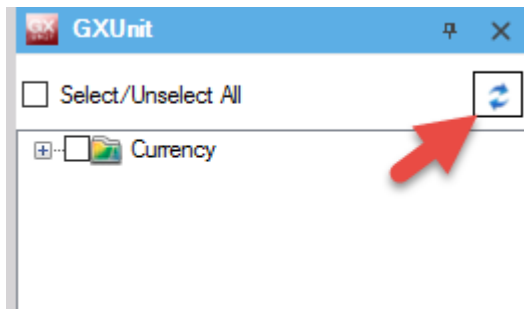
Now the test itself is executed. To do this go to: GXUnit -> GXUnit from the top menu.



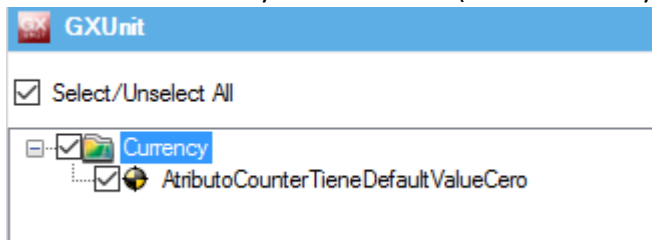
This will open the GXUnit interface in the right sector of the screen.



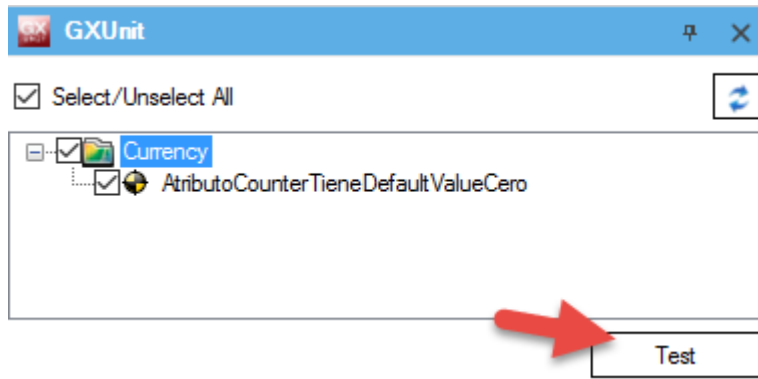
To see the added test, press the refresh button.



Then select the tests you want to run (choose the only one available).



And press the Test button.



For the case of the example, if it is visualized through some navigator it will be possible to verify that the check passes the test.

```

===== Web config update started =====
Updating web config ...
Web config update Success
===== Execution started =====
"C:\Models\GXUnitTaller\CSharpModel\web\bin\arunnerprocedure.exe"
Execution Success
GXUnit_OnAfterBuild- Results located at C:\Models\GXUnitTaller\CSharpModel\web\GXUnitResults\R_20171004_142429.xml
Procedure Object RunnerProcedure deleted!
Run RunnerProcedure Success

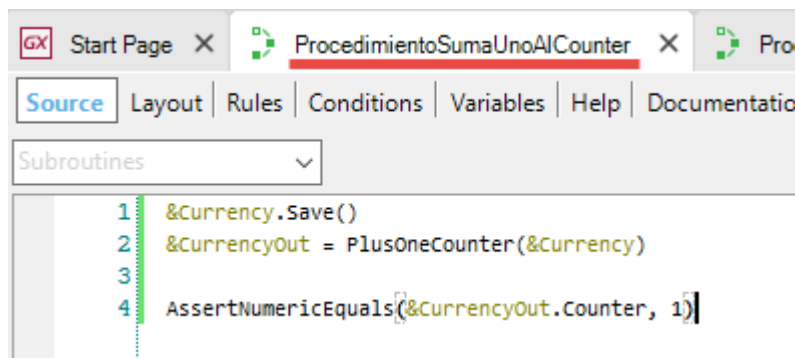
-<testsuites>
- <testsuite name="Currency" tests="1" failures="0" errors="0" time="0.865">
  <testcase name="AtributoCounterTieneDefaultValueCero" time="0.865" classname="Currency.AtributoCounterTieneDefaultValueCero"/>
</testsuite>
</testsuites>

```

Why is it necessary to do this?

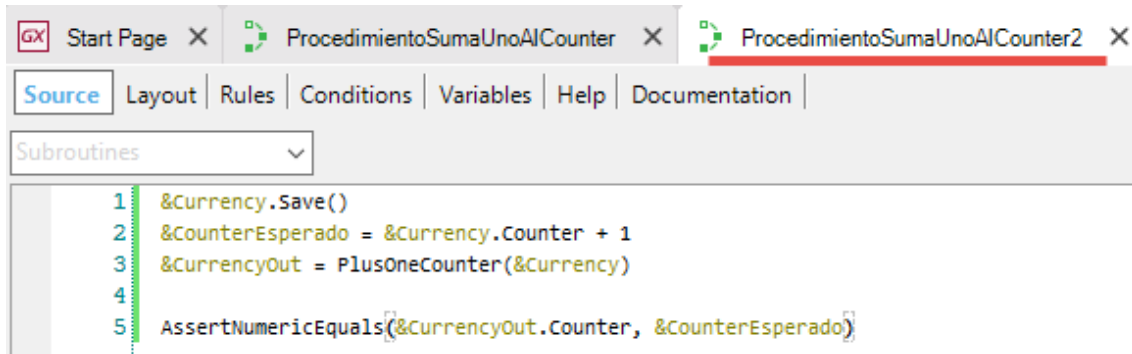
For this workshop we have a beta version of the tool which does not allow us to access a results viewer. It is for this reason that it is necessary to consume the generated XML.

Now you have a new requirement that is generate a procedure that receives a Currency and increase its counter by 1, finally returning the modified Currency. Let's start by creating our test to test this procedure. The first test we will write is the following.



At first sight the test does not seem wrong, however it is important to remember at the time of writing this test, it should also be considered that the test is maintainable, so it will be necessary to write it with the greatest possible intelligence. For the example then, it should be noted that the counter magnification value is `&Currency.Counter + 1` which may not be the value 1 specifically.

The optimized test should look like this:

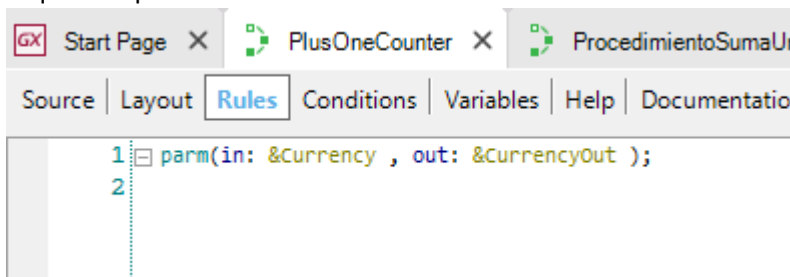


```

1  &Currency.Save()
2  &CounterEsperado = &Currency.Counter + 1
3  &CurrencyOut = PlusOneCounter(&Currency)
4
5  AssertNumericEquals(&CurrencyOut.Counter, &CounterEsperado)

```

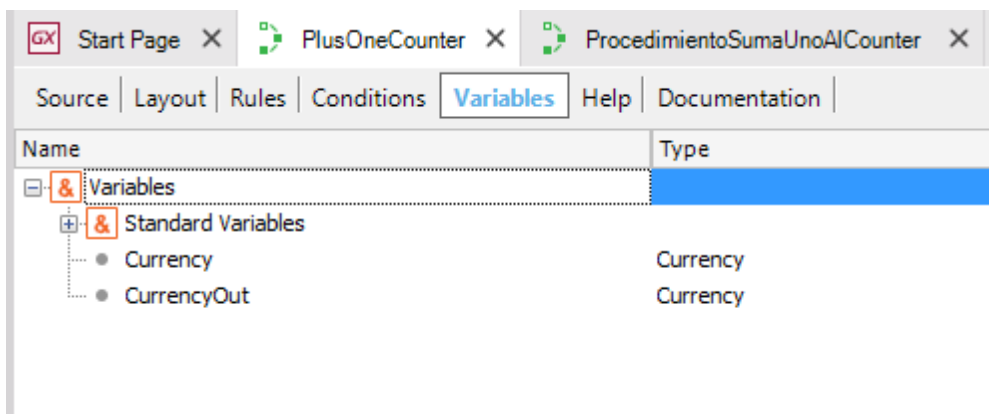
Testing will not be allowed until the PlusOneCounter procedure is implemented. Create the requested procedure as follows.



```

1  parm(in: &Currency , out: &CurrencyOut );
2

```

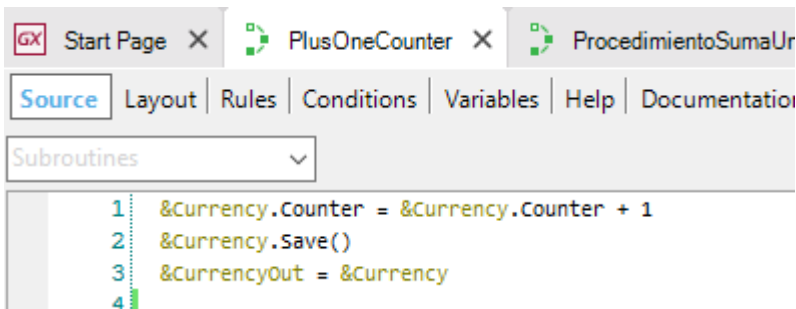


Name	Type
& Variables	
& Standard Variables	
• Currency	Currency
• CurrencyOut	Currency

Execute the two new test, the result of both should fail.

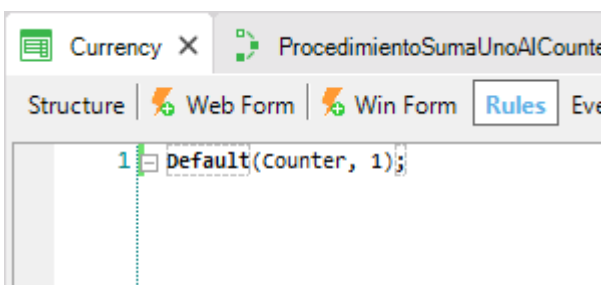
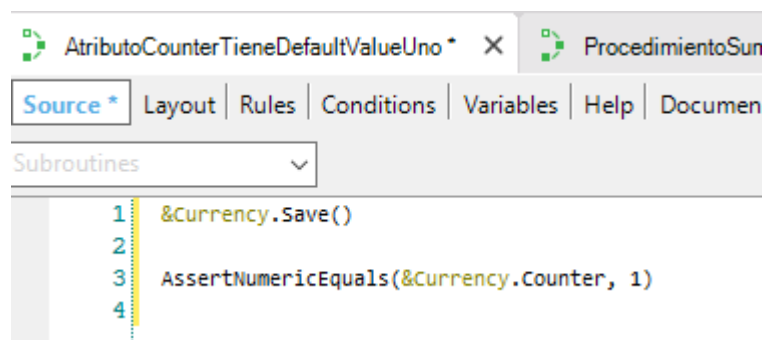
```
<testsuites>
  <testsuite name="Currency" tests="2" failures="2" errors="0" time="0.823">
    <testcase name="ProcedimientoSumaUnoAlCounter" time="0.625" classname="Currency.ProcedimientoSumaUnoAlCounter">
      <failure message="Assertion FAILED">Expected: 1.0000, Obtained: 0.0000</failure>
    </testcase>
    <testcase name="ProcedimientoSumaUnoAlCounter2" time="0.198" classname="Currency.ProcedimientoSumaUnoAlCounter2">
      <failure message="Assertion FAILED">Expected: 1.0000, Obtained: 0.0000</failure>
    </testcase>
  </testsuite>
</testsuites>
```

In both tests 1 was expected but 0 was obtained as a result, now add code to the procedure for both tests to be successful.



```
<testsuites>
  <testsuite name="Currency" tests="2" failures="0" errors="0" time="0.698">
    <testcase name="ProcedimientoSumaUnoAlCounter" time="0.657" classname="Currency.ProcedimientoSumaUnoAlCounter"/>
    <testcase name="ProcedimientoSumaUnoAlCounter2" time="0.041" classname="Currency.ProcedimientoSumaUnoAlCounter2"/>
  </testsuite>
</testsuites>
```

However, it again changes the definition of the attribute Counter and now must be initialized with the value 1, so it is necessary to create a new unit test and the implementation of a new rule for the default value



Run the complete suite so far. The result should be the following:

```
-<testsuites>
-  <testsuite name="Currency" tests="4" failures="2" errors="0" time="0.427">
-    <testcase name="AtributoCounterTieneDefaultValueCero" time="0.366" classname="Currency.AtributoCounterTieneDefaultValueCero">
-      <failure message="Assertion FAILED">Expected: 0.0000, Obtained: 1.0000</failure>
-    </testcase>
-    <testcase name="AtributoCounterTieneDefaultValueUno" time="0.044" classname="Currency.AtributoCounterTieneDefaultValueUno">
-    <testcase name="ProcedimientoSumaUnoAlCounter" time="0.009" classname="Currency.ProcedimientoSumaUnoAlCounter">
-      <failure message="Assertion FAILED">Expected: 1.0000, Obtained: 2.0000</failure>
-    </testcase>
-    <testcase name="ProcedimientoSumaUnoAlCounter2" time="0.008" classname="Currency.ProcedimientoSumaUnoAlCounter2"/>
-  </testsuite>
</testsuites>
```

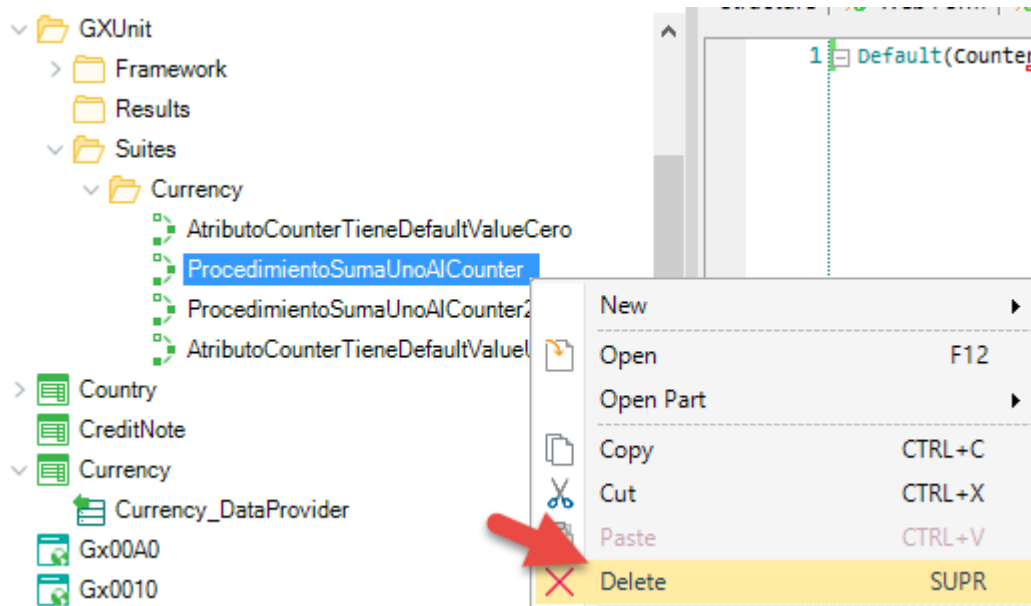
There are 2 tests that fail

AttributeCounterTieneDefaultValueCero which is deprecated

AND

ProcedureOneUnoAlCounter, which fails because it always expects 1 as a result instead of the value of Counter +1. This case demonstrates how to define more abstractly a test is not affected by changes in other modules, as is the case of ProcedureOneUnoAlCounter2 which, when not having a static value of comparison was coupled to the change and verified correctly.

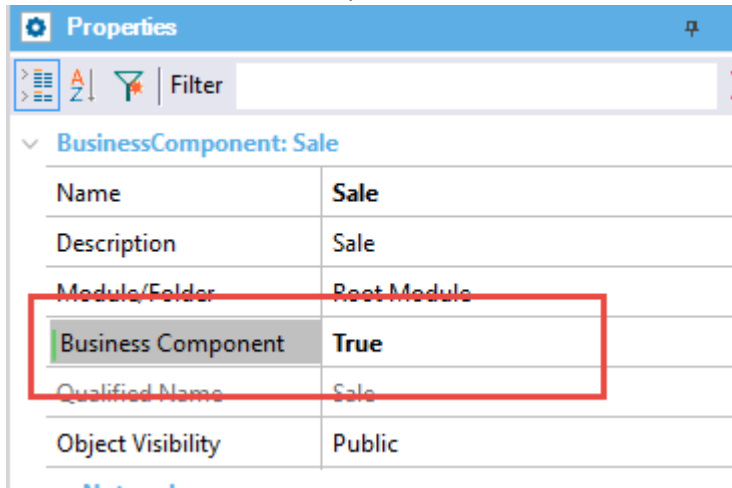
Now we will remove both obsolete tests.



What is the advantage of using unit tests?

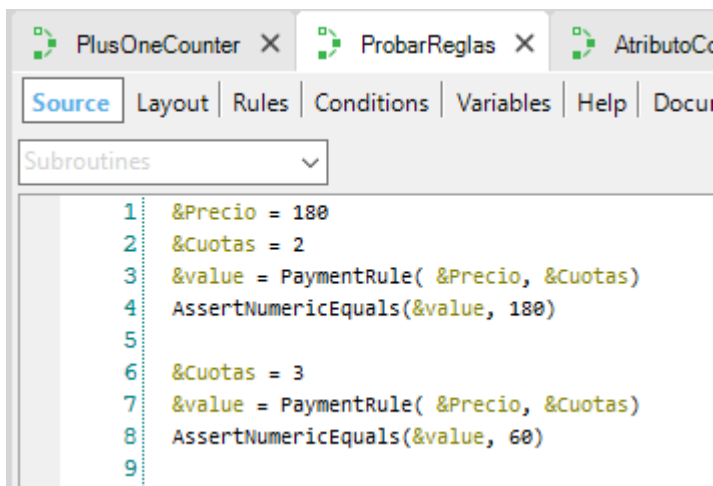
Then analyze the importance of a high% of code coverage in a simple example. It is requested to perform a procedure that takes as a parameter a Sale Transaction and the quantity of quotas, apply a procedure with the negotiation logic for the calculation of quotas and then return the final value of the quota. For the calculation logic of quota, it is known that the quotas can't be less than 3 so that any smaller amount of quotas must return the total value of the sale.

We set Sale as Business Component



BusinessComponent: Sale	
Name	Sale
Description	Sale
Module/Folder	Root Module
Business Component	True
Qualified Name	Sale
Object Visibility	Public

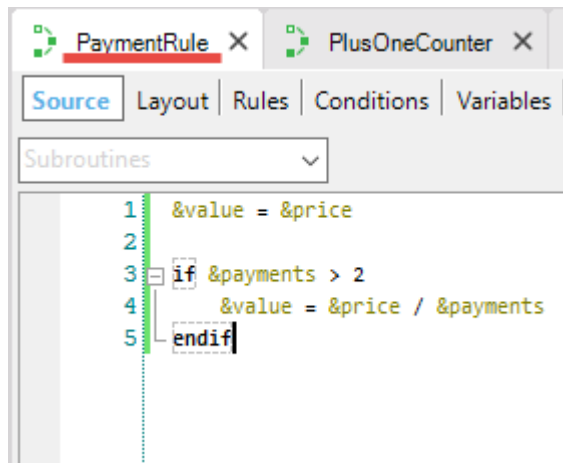
First we will carry out the procedure of calculation of quota and its test.



```

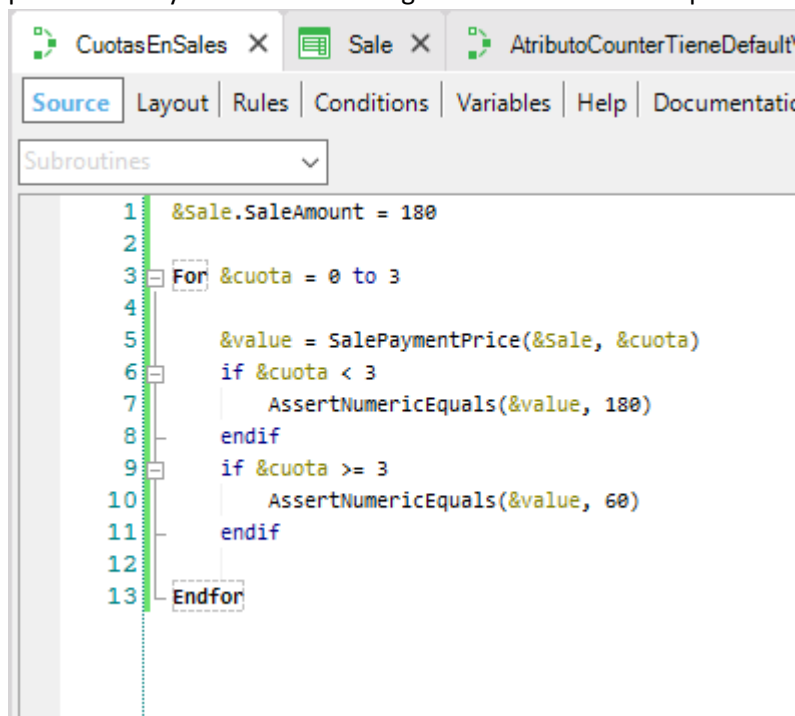
1  &Precio = 180
2  &Cuotas = 2
3  &value = PaymentRule( &Precio, &Cuotas)
4  AssertNumericEquals(&value, 180)
5
6  &Cuotas = 3
7  &value = PaymentRule( &Precio, &Cuotas)
8  AssertNumericEquals(&value, 60)
9

```

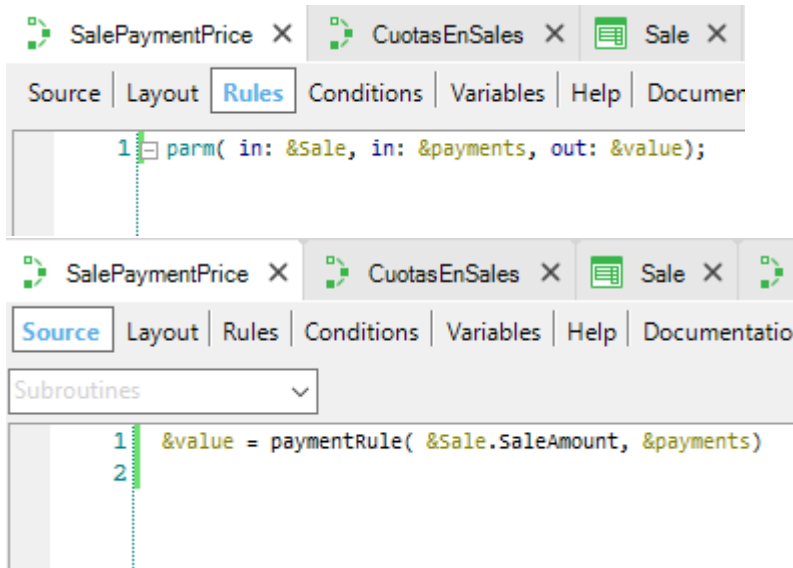


```
1  &value = &price
2
3  if &payments > 2
4      &value = &price / &payments
5  endif
```

Then we perform the procedure that receives a Sale and the amount of quotas and applies the procedure PaymentRule returning its value. The test and procedure should be as follows.

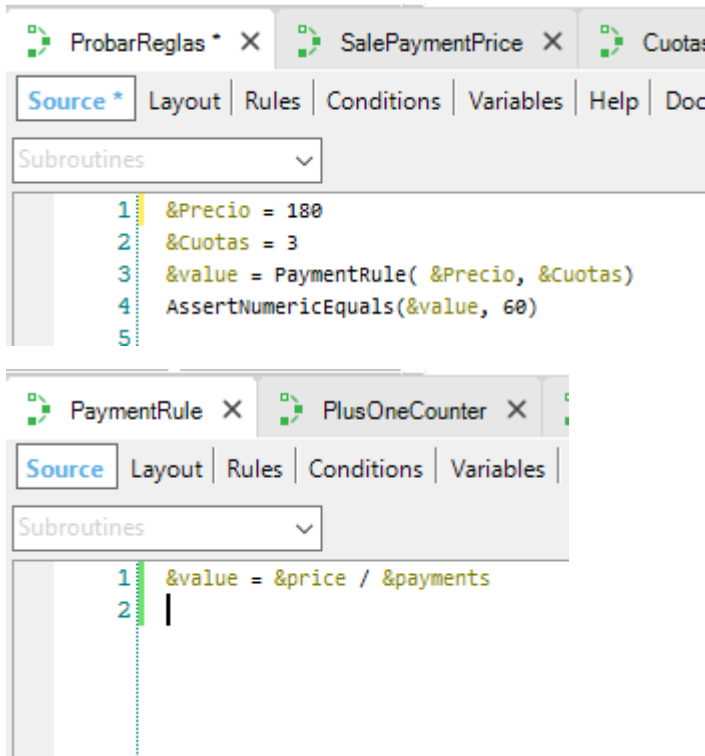


```
1  &Sale.SaleAmount = 180
2
3  For &cuota = 0 to 3
4
5      &value = SalePaymentPrice(&Sale, &cuota)
6      if &cuota < 3
7          AssertNumericEquals(&value, 180)
8      endif
9      if &cuota >= 3
10         AssertNumericEquals(&value, 60)
11     endif
12
13 Endfor
```



After implementing the procedures all the tests should be executed correctly.

The change of implementation is requested to accept any amount of payments and is not limited to 3. The test and procedure are changed as follows.



We run all the tests and as a result you should get.

```
"C:\Models\GxUnitTaller\CSharpModel\web\bin\arunnerprocedure.exe"
System.DivideByZeroException: Attempted to divide by zero.
```

This error is caused by the unit test QuotasEnSales, as you can see the unit tests prevent cases not contemplated, in this case the division between 0.

Having unit tests well defined and with good coverage decreases the comings and goings with the test sector and the use of the debugger.