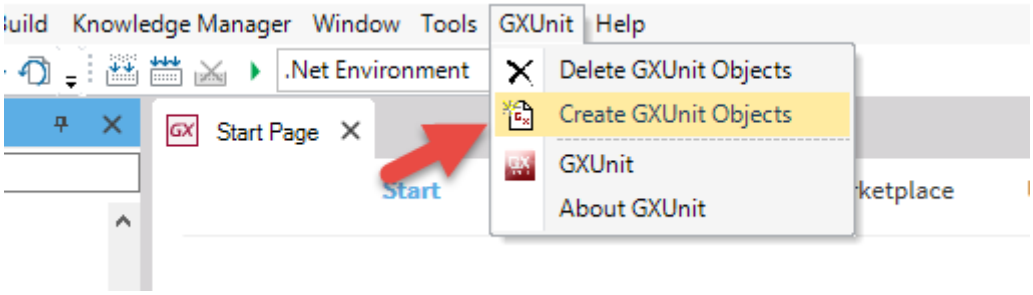
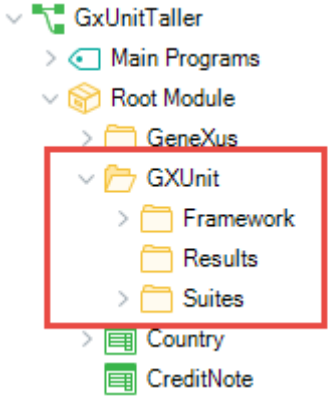
# Práctico de Pruebas unitarias con GXUnit

## ¿Cuál es la forma correcta de definir los test unitarios?

Primero se deberá crear una carpeta de pruebas unitarias en el proyecto. Para eso seleccionar la opción *GXUnit -> Create GXUnit Objects* del menú superior.

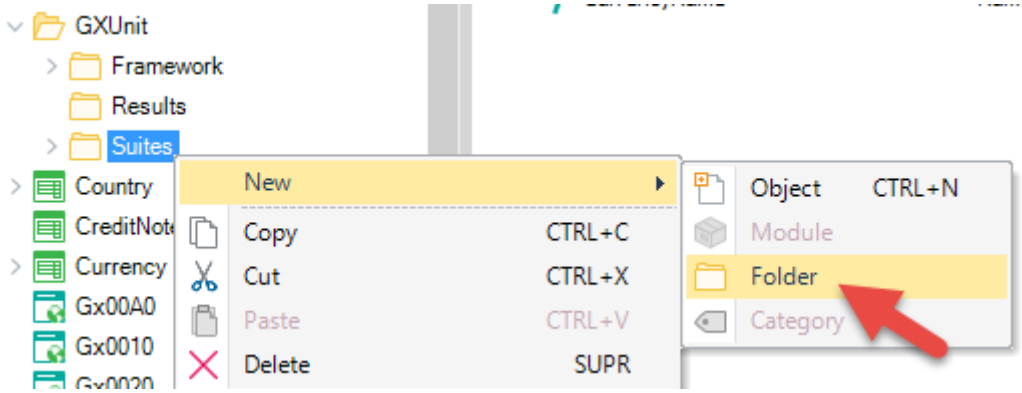



Como resultado se creará la siguiente estructura dentro de la KB.

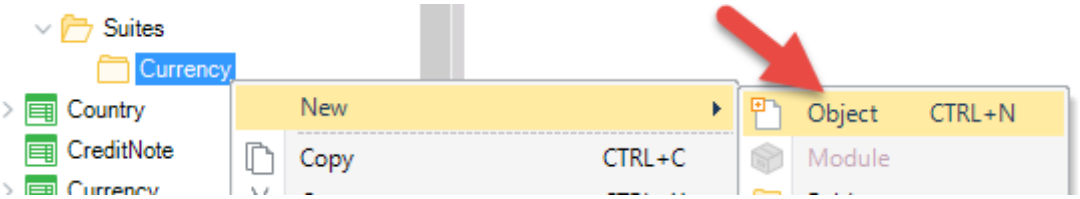



Supongamos que se solicita agregar un atributo Counter a la transacción Currency, para llevar un contador de cuantas veces esta fue utilizada, por defecto este atributo deberá inicializarse en 0.

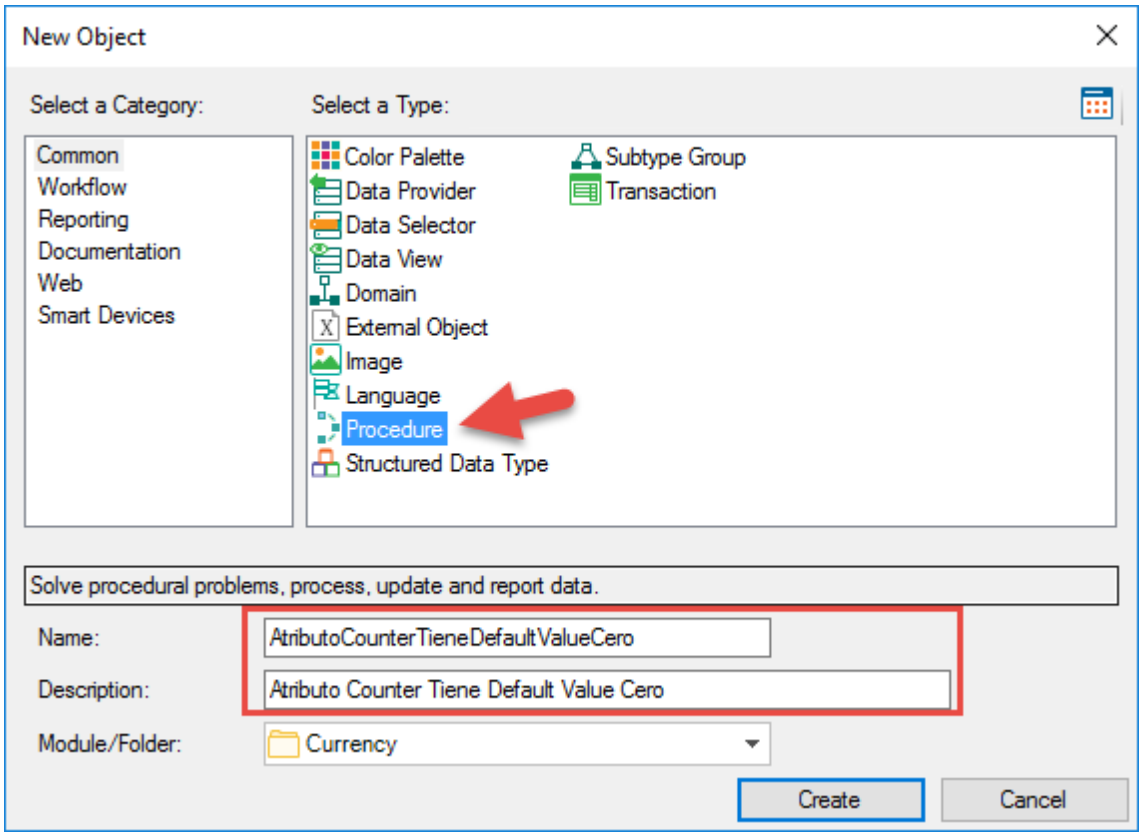
Como primer paso se deberá crear una suite de pruebas para la transacción Currency. Dentro del folder Suites se crea el folder Currency.



Ahora dentro de la Suite de pruebas Currency, se deberá crear el nuevo test, seleccionando *New -> Object*.




Seleccionar el tipo “Procedure” y agregar un nombre descriptivo que describa el objetivo del test.




Ahora se deberá escribir una prueba para comprobar que Counter se inicializa en 0. Primero generar la variable &Currency y agregarla al procedure con clic *derecho -> add variable*, Genexus ya tomara el tipo Currency para la variable. Luego ejecutar la función save a la variable &Currency y utilizar la función AssertNumericEquals de GXUnit para comprobar el valor por defecto del atributo counter.

**NOTA:** GXUnit cuenta con dos procedimientos para comparar, AssertNumericEquals el cual compara numéricos y AssertStringEquals el cual compara Strings.

El caso debería quedar de la siguiente forma:

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

Al intentar guardar el test, se mostrará el siguiente error.

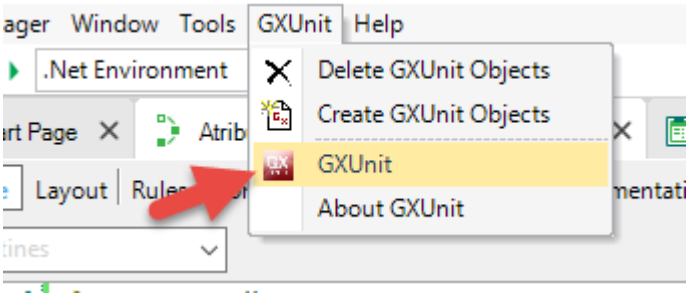
```
error src0216: 'Counter' invalid property.
```

Ahora agregamos el código para poder correr nuestro test de forma exitosa. Primero agregar el atributo Counter en Currency.

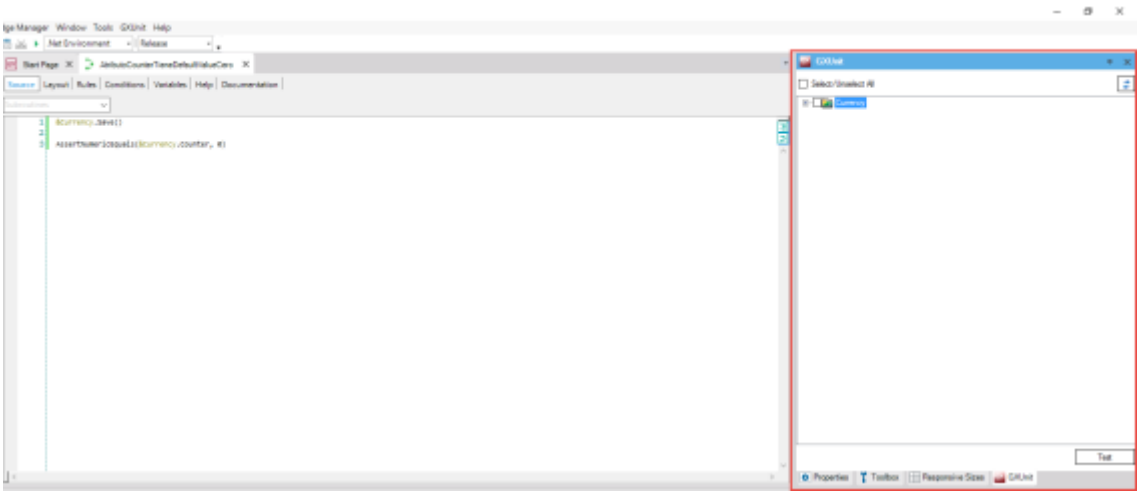
| Name | Type | Description |
|---|---|---|
| Currency | Currency | Currency |
| CurrencyCode | Character(3) | Currency Code |
| CurrencyName | Name | Currency Name |
| Counter | Numeric(4.0) | Counter |

No se agregará una rule para default value 0 ya que, para los tipos Numeric ya se utiliza dicho valor por defecto.

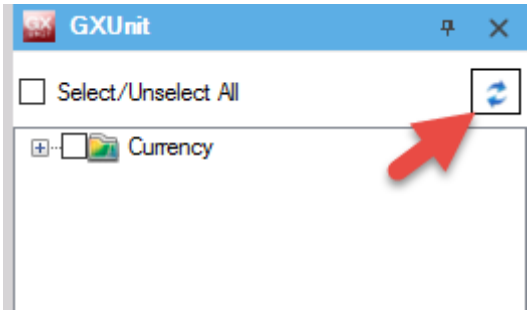
Ahora se ejecuta el test propiamente dicho. Para ello ir a: *GXUnit -> GXUnit* del menú superior.



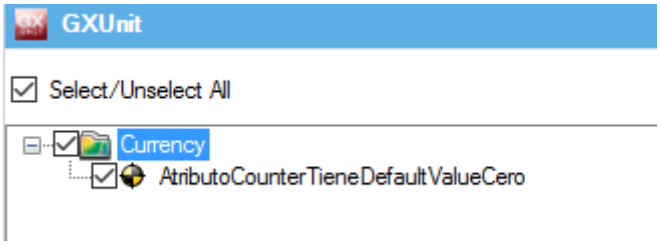

Lo cual abrirá la interfaz de GXUnit en el sector derecho de la pantalla.



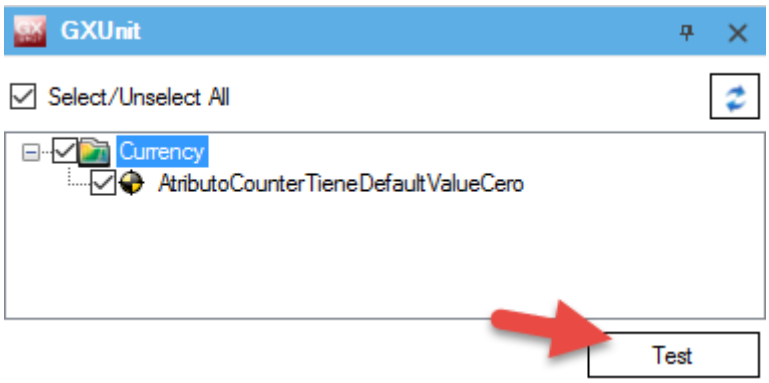
Para ver el test agregado se deberá presionar el botón de refresh.




Luego seleccionar los test que se desean correr (elegir el único disponible).




Y presionar el botón Test.




Al finalizar la corrida se mostrará la ruta en donde se encuentra el xml de resultados.

Para el caso de ejemplo, si se visualiza a través de algún navegador se podrá comprobar que el test pasa la prueba.

```
========== 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
```

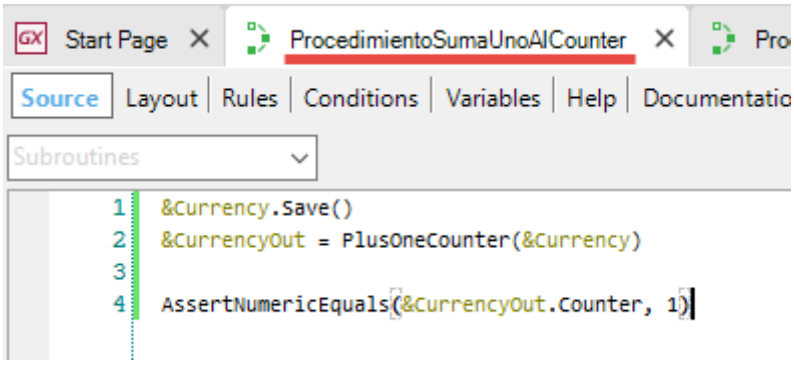
```
- <testsuites>
  - <testsuite name="Currency" tests="1" failures="0" errors="0" time="0.865">
      <testcase name="AtributoCounterTieneDefaultValueCero" time="0.865" classname="Currency.AtributoCounterTieneDefaultValueCero"/>
    </testsuite>
  </testsuites>
```

¿Por qué es necesario hacer esto?

Para este taller tenemos una versión beta de la herramienta la cual no nos permite acceder a un visualizador de resultados. Es por dicha razón que es necesario consumir el XML generado.

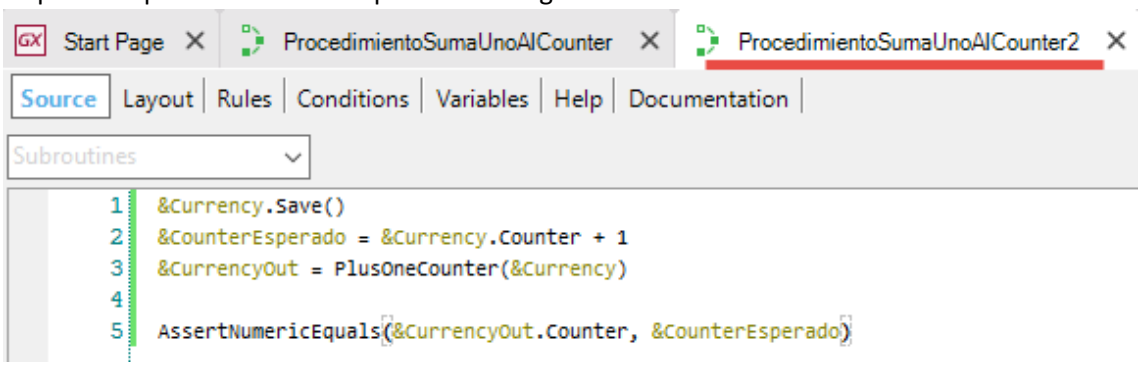
Ahora se solicita generar un procedimiento que reciba una Currency y aumente su contador en 1, devolviendo finalmente la Currency modificada. Comencemos creando nuestro test para probar dicho procedimiento.

La primer prueba que escribiremos es la siguiente.

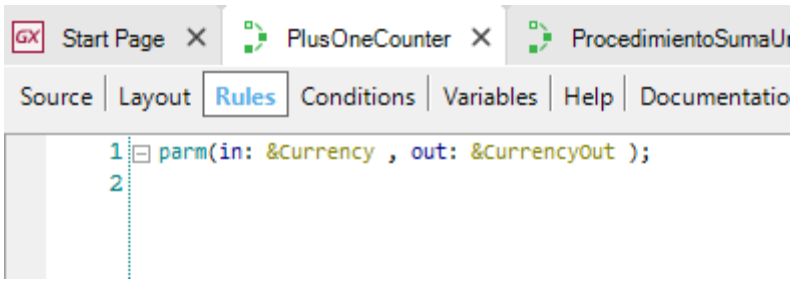


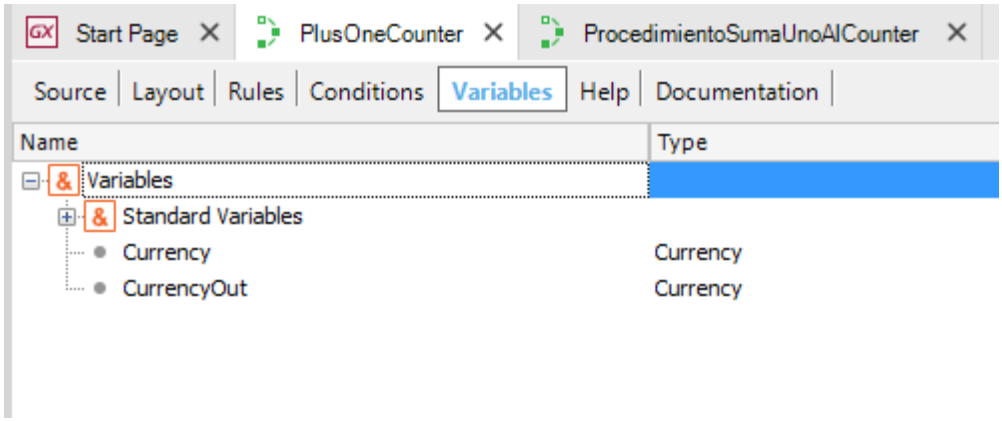

A simple vista la prueba no parece estar mal, sin embargo es importante recordar al momento de escribir dicha prueba, también se deberá considerar que dicha prueba sea “mantenible”, por lo que será necesario escribirla con la mayor inteligencia posible. Para el ejemplo entonces, se deberá tener en cuenta que el valor de aumento de Counter es &Currency.Counter + 1 lo cual puede no llegar a ser el valor 1 específicamente.

La prueba optimizada debería quedar de la siguiente forma:



Ahora no se permitirá correr los test hasta implementar el procedimiento PlusOneCounter. Crear el procedimiento solicitado de la siguiente forma.

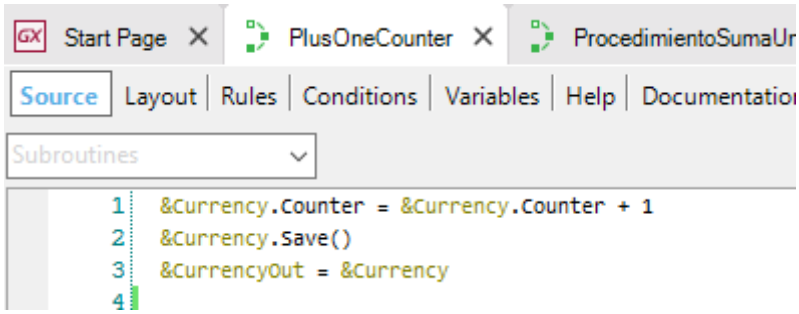



Ejecutar los dos nuevos test, el resultado de ambos deberá fallar.

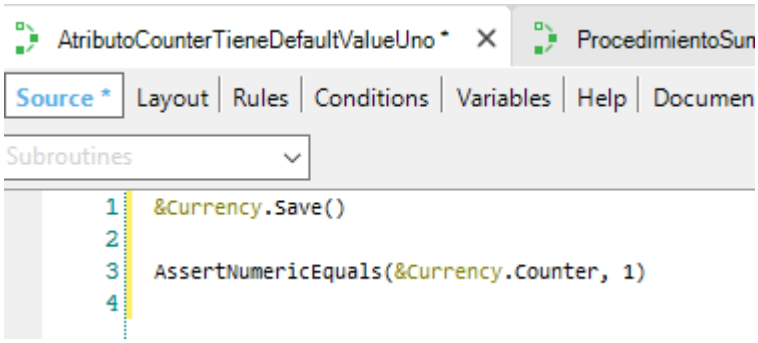
```
- <testsuites>
  - <testsuite name="Currency" tests="2" failures="2" errors="0" time="0.823">
    - <testcase name="ProcedimientoSumaUnoAlCounter" time="0.625" classname="Currency.ProcedimientoSumaUnoAlCounter">
        <failure message="Assertion FAILED">Expected: 1.0000, Obtained: 0.0000</failure>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

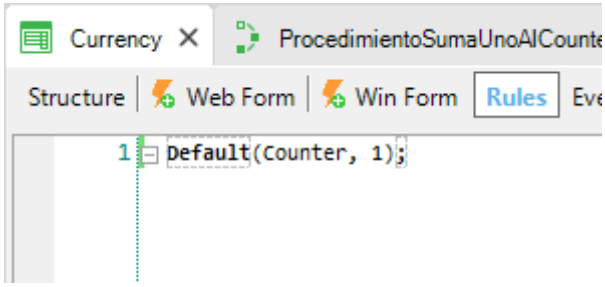
En ambos test se esperaba 1 pero se obtuvo 0 como resultado, ahora agregue código al procedimiento para que ambos test sean exitosos.



```
- <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>
```

Sin embargo, nuevamente cambia la definición del atributo Counter y ahora debe inicializarse con el valor 1, por lo cual es necesario crear un nuevo test unitario y la implementación de una nueva rule para el default value.




Ejecutar la suite complete existente hasta el momento. El resultado deberá ser el siguiente:

```
- <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>
```

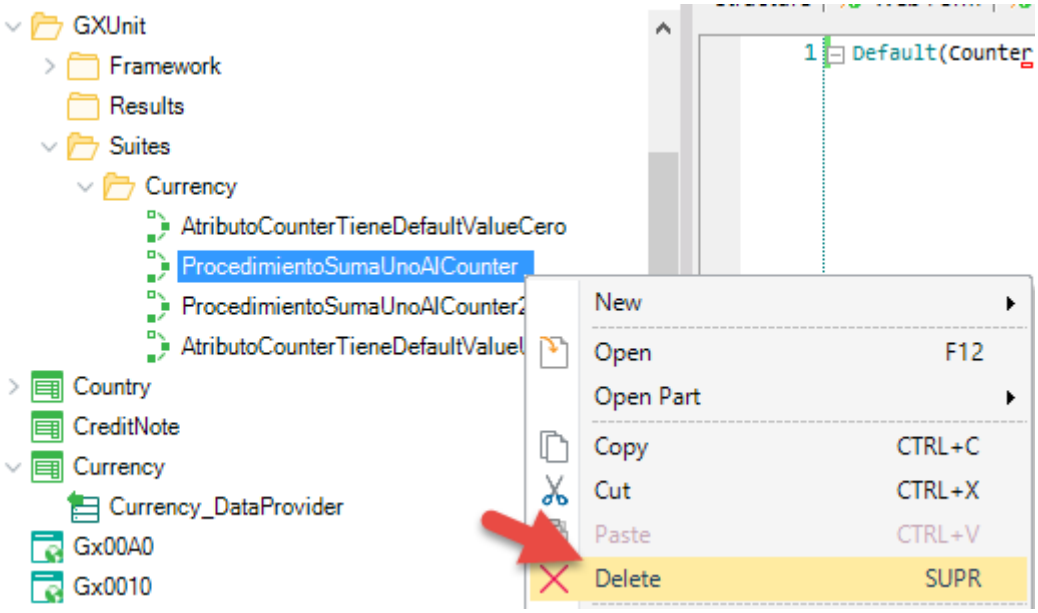
Existen 2 pruebas que fallan

*AtributoCounterTieneDefaultValueCero* la cual quedo obsoleta

Y

*ProcedimientoSumaUnoAlCounter*, el cual falla puesto que siempre espera 1 como resultado en lugar del valor de Counter +1. Dicho caso evidencia como al definir de forma más abstracta un test este no se ve afectado por cambios en otros módulos, como es el caso de *ProcedimientoSumaUnoAlCounter2 el cual al no tener un valor estático de comparación se acopló al cambio y lo verificó correctamente*.

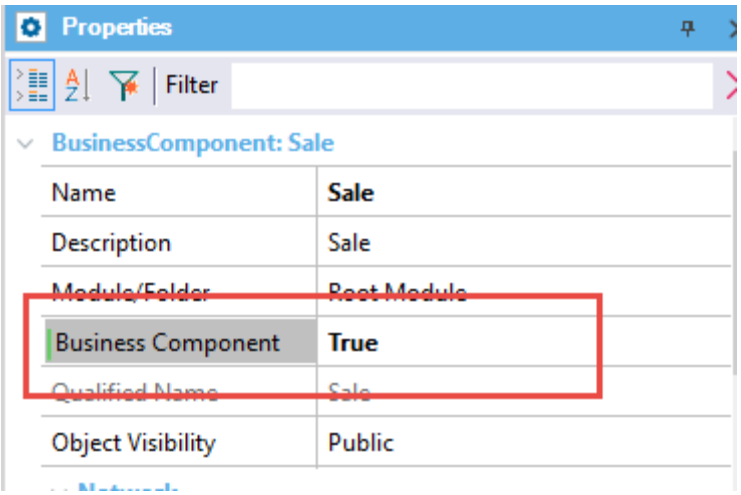
Ahora eliminaremos ambos tests obsoletos.



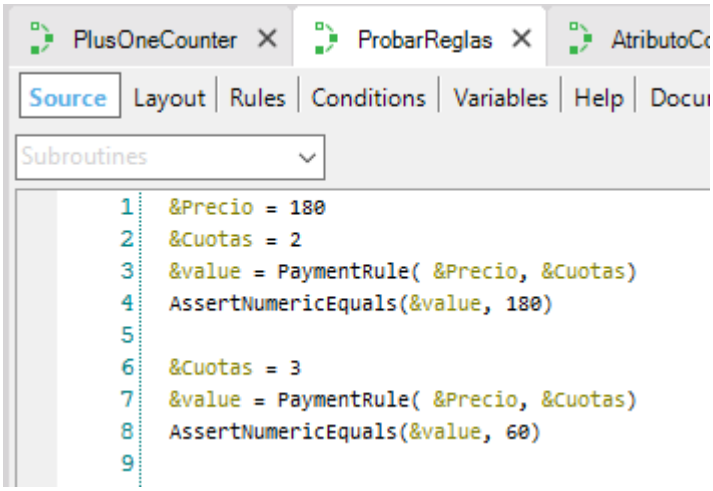

## ¿Cuál es la ventaja de utilizar test unitarios?

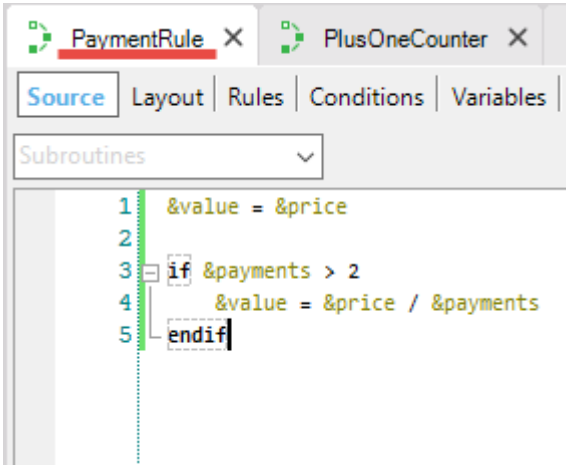
A continuación analizar la importancia de un alto porcentaje de code coverage en un sencillo ejemplo. Se solicita realizar un procedimiento que tome como parámetro una Transacción del tipo Sale y la cantidad de cuotas, le aplique un procedimiento con la lógica de negoció para el cálculo de cuotas y luego devuelva el valor final de la cuota. Para la lógica de cálculo de cuota, se sabe que las cuotas no pueden ser menores a 3 por lo cual, cualquier cantidad menor de cuotas debe devolver el valor total de la venta.

Seteamos Sale como Business Component.

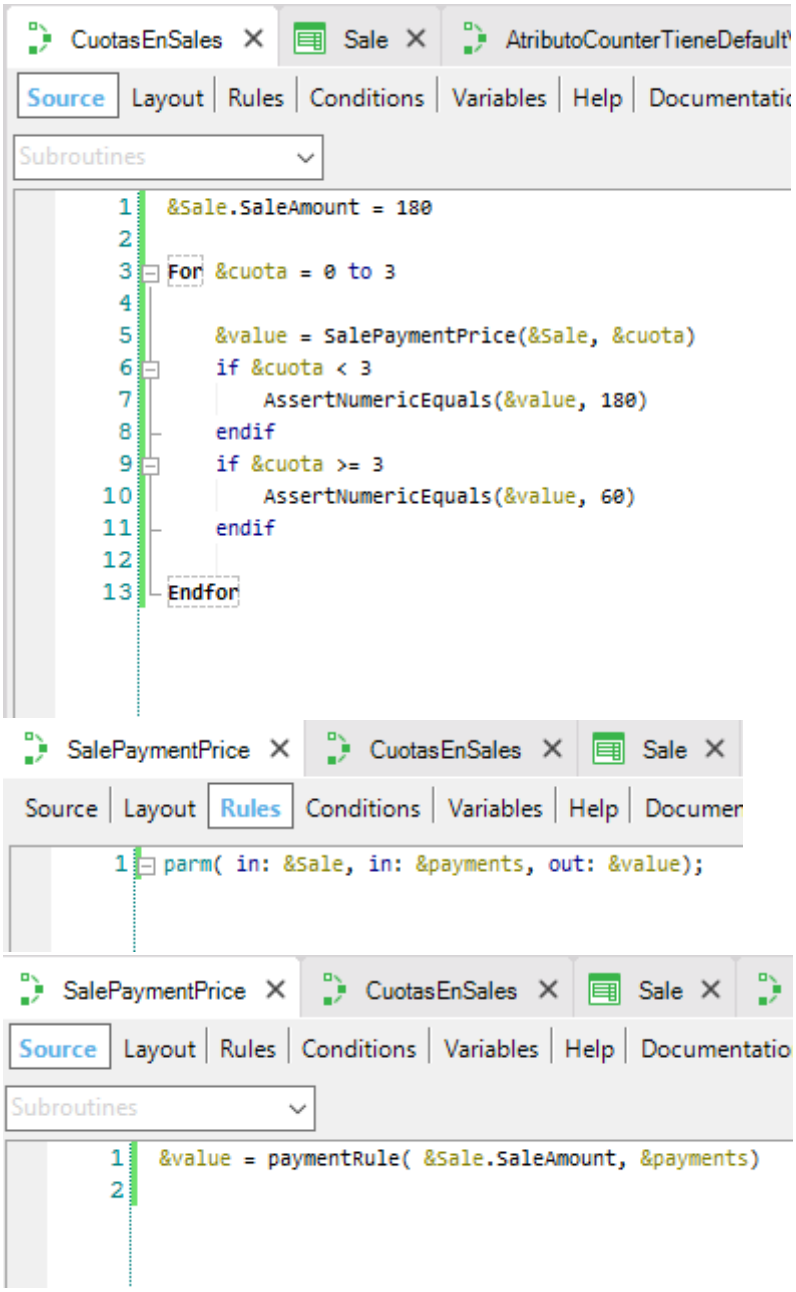



Primero realizaremos el procedimiento de cálculo de cuota y su test.



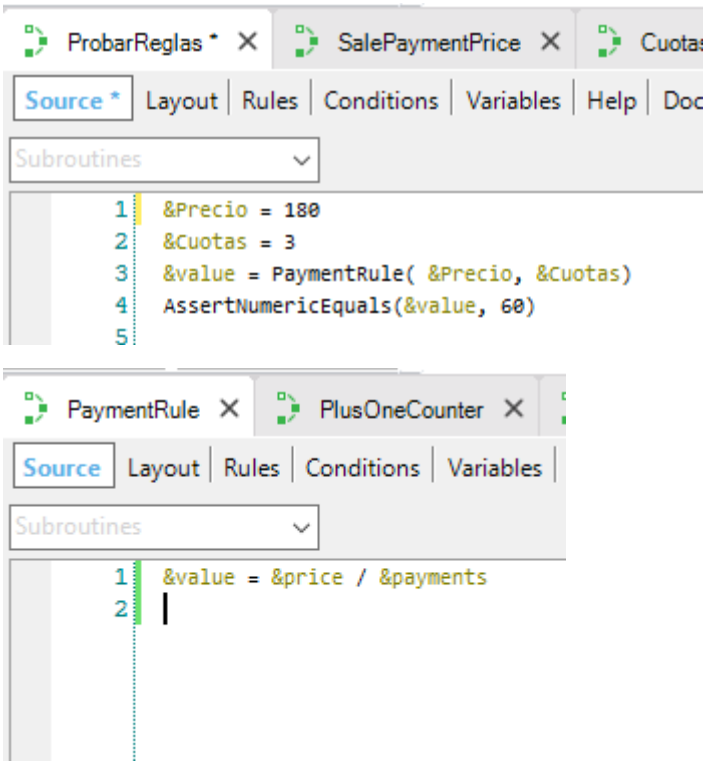




Luego realizamos el procedimiento que recibe una Sale y la cantidad de cuotas y le aplica el procedimiento PaymentRule devolviendo su valor. El test y el procedimiento deberían quedar de la siguiente forma.




Luego de implementar los procedimientos todos los test deberían de ejecutarse de forma correcta.

Se pide el cambio de la implementación para que acepte cualquier cantidad de pagos y no este limitado a 3. Se cambia el test y el procedimiento de la siguiente forma.




Corremos todos los tests y como resultado se deberá obtener.

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

Este error es provocado por el test unitario *CuotasEnSales,* como se puede apreciar los test unitarios previenen casos no contemplados, en este caso la división entre 0.

Tener test unitarios bien definidos y con un buen coverage disminuye las idas y vueltas con el equipo de test y la necesidad de realizar debugging.