

Testing de aplicaciones en GeneXus

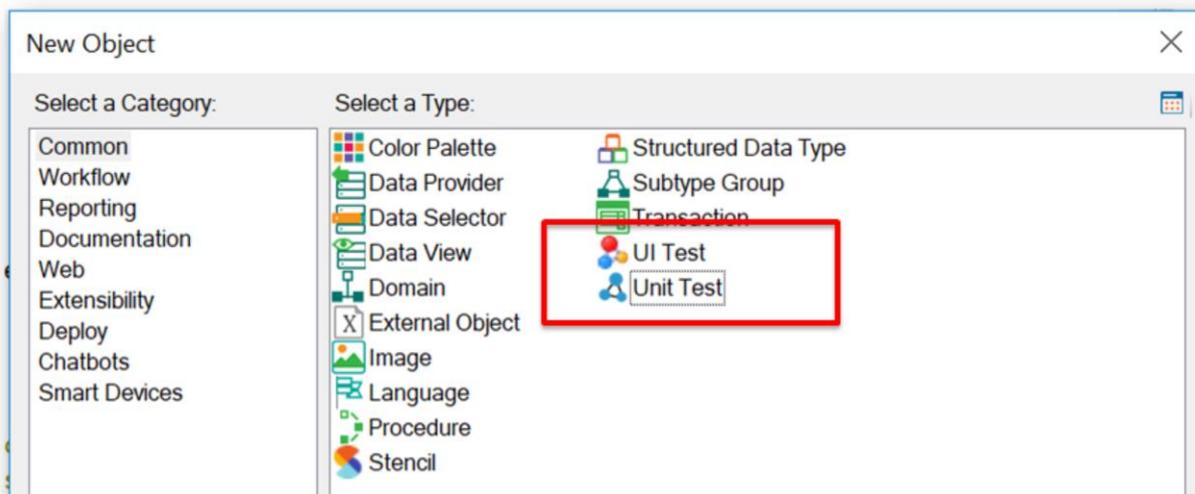
Test Unitario. Introducción.

GeneXus 16

Cuando desarrollamos una nueva funcionalidad en nuestra aplicación es necesario probar si lo que desarrollamos funciona de acuerdo a lo esperado, pero además es importante volver a probar toda la aplicación luego de ese cambio, para asegurarnos que lo que ya teníamos funcionando se siga comportando correctamente.

A medida que la aplicación crece este tipo de tareas se pueden ir volviendo mas tediosas ya que cada vez es mas lo que volver a probar y también mas costosas ya que cada vez consumen más tiempo

UITest & Unit Test

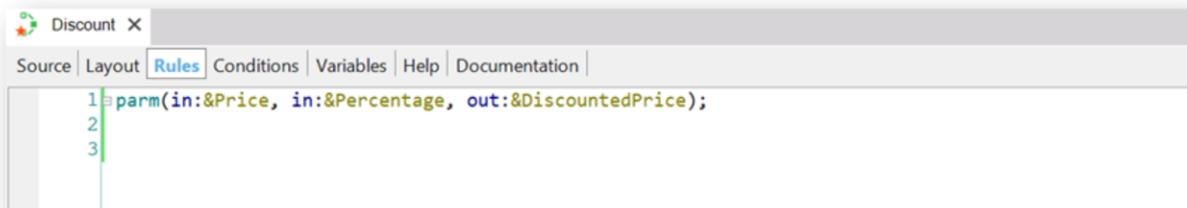


GeneXus nos ayuda en este frente dándonos funcionalidades para poder crear y ejecutar tests automáticos, tanto unitarios como de interfaz de forma de poder reducir parte del trabajo manual de verificación.

Lo tests Unitarios nos permiten testear una parte de la aplicación de forma aislada, esto aplica a pruebas sobre procedimientos, DataProviders y BusinessComponents. En definitiva aquellos componentes donde debería residir la lógica de negocios de nuestra aplicación.

El test de Interfaz –nos permite crear pruebas simulando las acciones de un usuario en el browser, de manera de poder testear flujos completos de la aplicación.

Unit Tests - Ejemplo



```
1 parm(in:&Price, in:&Percentage, out:&DiscountedPrice);  
2  
3
```

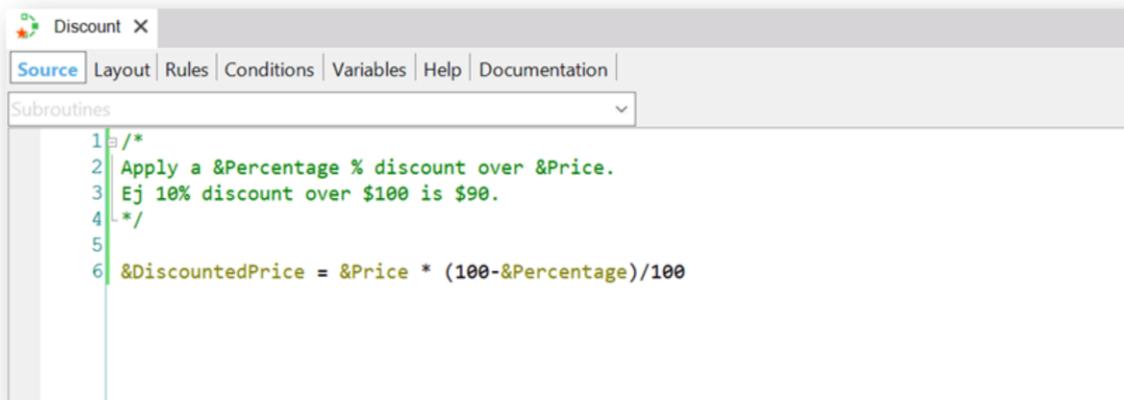
Vamos a enfocarnos en los tests unitarios , y lo haremos con un ejemplo muy básico.

Tenemos un procedimiento que calcula un **Descuento** – básicamente aplica un porcentaje de descuento a un precio – que se va a utilizar para calcular promociones en la agencia de viajes.

El procedimiento tiene dos parámetros de entrada: El Precio a descontar y el porcentaje de descuento. Y tiene un parámetro de salida que es el precio descontado.

Lo que haría este procedimiento por ejemplo es que si le pasamos un precio de \$100 y un descuento del 10%, nos devolvería el precio descontado que sería \$90.

Unit Tests - Ejemplo



```
Discount X
Source | Layout | Rules | Conditions | Variables | Help | Documentation
Subroutines
1 /*
2 Apply a &Percentage % discount over &Price.
3 Ej 10% discount over $100 is $90.
4 */
5
6 &DiscountedPrice = &Price * (100-&Percentage)/100
```

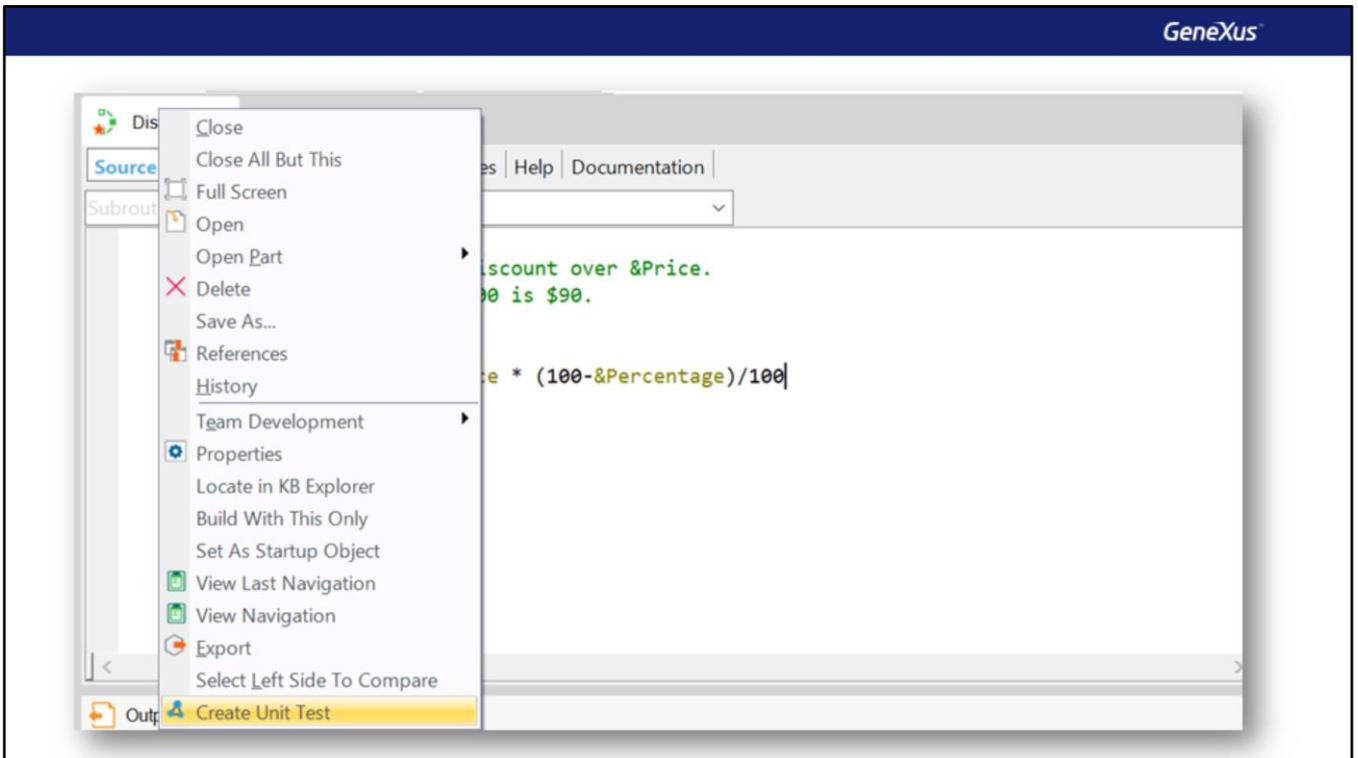
Acá podemos ver la implementación del procedimiento.

Ahora bien, cómo haríamos para probar que este procedimiento funcione de la manera esperada? Es decir, que dado un precio y un porcentaje me devuelva el precio ajustado correcto?

Podríamos hacer una pantalla que nos permita ingresar los valores del precio y el descuento, le ponemos un botón que llama al procedimiento y nos muestra en pantalla el valor del precio descontado. De esta manera podríamos ir probando de forma interactiva que el procedimiento se comporte como queremos. También podríamos si no, programar un procedimiento, en donde llamamos a este proc Discount con diferentes parámetros y vamos imprimiendo en consola el resultado, de forma también de poder verificar allí que el resultado es el esperado.

Ahora bien, esa forma de testear es costosa – ya que hay que crear una pantalla, ingresar los valores a mano y evaluar el resultado. Y esto lo haríamos en la máquina donde estamos desarrollando, pero luego si estamos satisfechos con que se comporta bien, e integramos nuestro trabajo con el del resto del equipo, hay que volver a testear para ver que quedó bien integrada.

La idea de los tests unitarios es poder automatizar este trabajo para evitarnos el retrabajo y simplificar la tarea



Vamos entonces a crear un test unitario sobre nuestro procedimiento, y para ello damos botón-derecho sobre el objeto y elegimos la opción "Create Unit Test"

Objects

- <Object>UnitTest
- <Object>UnitTestData
- <Object>UnitTestData

Al crear el test unitario se crean tres objetos, vamos a verlos en detalle..

<Object>UnitTestSDT

The screenshot displays the GeneXus IDE interface. At the top, there are several tabs: Discount, Navigation View, DiscountUnitTestSDT, DiscountUnitTestData, and DiscountUnitTest. Below the tabs, there are two main panels. The upper panel, titled 'Structure', shows a tree view of the 'DiscountUnitTestSDT' object. It lists the following properties:

Name	Type	Description	Is Collection
DiscountUnitTestSDT		Discount Unit Test SDT	<input checked="" type="checkbox"/>
Price	Price		<input type="checkbox"/>
Percentage	Percentage		<input type="checkbox"/>
ExpectedDiscountedPrice	Price		<input type="checkbox"/>
ErrorMsgDiscountedPrice	VarChar(100)	Error message to show in case assertion...	<input type="checkbox"/>

The lower panel, titled 'Source', shows the 'Rules' tab with the following code snippet:

```

1 parm(in:&Price, in:&Percentage, out:&DiscountedPrice);
2
3

```

Decíamos que si fuéramos a probar esto a mano quizá hubiéramos creado una pantalla donde pondríamos las dos variables de entrada del procedimiento, y alguna manera de ver el resultado de la variable de salida

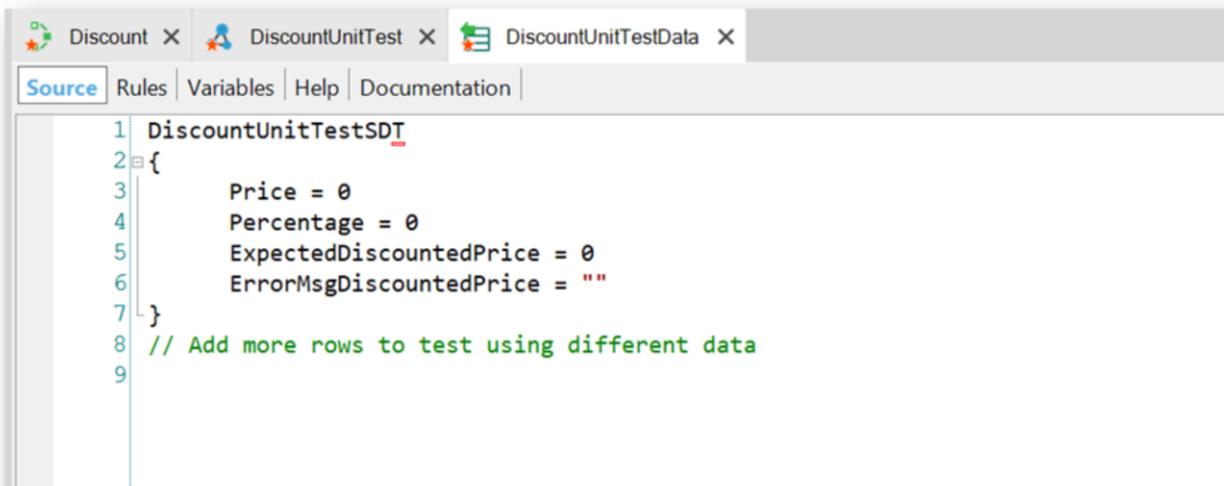
En definitiva alguna forma de validar que dados los valores de entrada, nos devuelva un resultado que esperamos.

Vamos a ver el DiscountUnitTestSDT que es uno de los objetos que se creó automáticamente al crear el test unitario del procedimiento Discount. Este SDT define la estructura de un caso de prueba concreto para el objeto que estamos testeando.

Vemos que – al igual que haríamos nosotros – define las dos variables de entrada – con el mismo nombre que los parámetros del procedimiento – y define también una variable ExpectedDiscountedPrice donde vamos a poder definir el valor del resultado que esperamos.

En definitiva, vamos a poder decir que para un precio de 100 y un porcentaje de descuento de 10 esperamos un resultado de 90.

También podemos asignar un mensaje que queremos que se muestre en el caso de que el resultado sea diferente de 90.



```
Discount x DiscountUnitTest x DiscountUnitTestData x
Source Rules Variables Help Documentation
1 DiscountUnitTestSDT
2 {
3     Price = 0
4     Percentage = 0
5     ExpectedDiscountedPrice = 0
6     ErrorMsgDiscountedPrice = ""
7 }
8 // Add more rows to test using different data
9
```

Ahora que vimos la estructura del caso de prueba para el procedimiento Discount, vamos a ver como definimos los juegos de datos.

Esto lo vamos a hacer en el DataProvider que se definió también de forma Automática.

Aquí vemos un grupo definido, con los elementos del SDT donde podemos instanciar los valores

The screenshot shows the GeneXus IDE interface. On the left, a code editor displays the following code for a Data Provider:

```

1 DiscountUnitTestSDT
2 {
3     Price = 100
4     Percentage = 10
5     ExpectedDiscountedPrice = 90
6     ErrorMessageDiscountedPrice = "10% OFF"
7 }
8 // Add more rows to test using different
9

```

On the right, the Properties window for the Data Provider "DiscountUnitTestData" is open. The "Output" section is highlighted with a red box, showing the following configuration:

Property	Value
Output	DiscountUnitTestSDT
Collection	True
Collection Name	DiscountUnitTestSDTCollection

...Y por ejemplo asignar los que veníamos hablando

También vamos a asignar el mensaje que queremos ver en caso que el resultado obtenido sea diferente de 90.

Este data-Provider nos devuelve una colección de datos de prueba, por lo tanto nos va a permitir definir de forma sencilla varias pruebas, o varios juegos de datos, que queramos ejecutar, pero por ahora nos alcanza con este, para poder ejecutar nuestra primera prueba.

```

Discount X DiscountUnitTest X
Source Layout Rules Conditions Variables Help Documentation
Subroutines
1 /* Autogenerated unit test code for Procedure 'Discount' */
2 For &TestCaseData in DiscountUnitTestData()
3
4     /* Act... */
5     Discount(&TestCaseData.Price, &TestCaseData.Percentage, &DiscountedPrice)
6
7     /* Assert... */
8     AssertNumericEquals(&TestCaseData.ExpectedDiscountedPrice, &DiscountedPrice, &TestCaseData.ErrorMsgDiscountedPrice)
9 endfor
10
11 |

```

Antes de ejecutarla vamos a ver el tercer objeto que se creó automáticamente, que es el unit-test propiamente dicho.

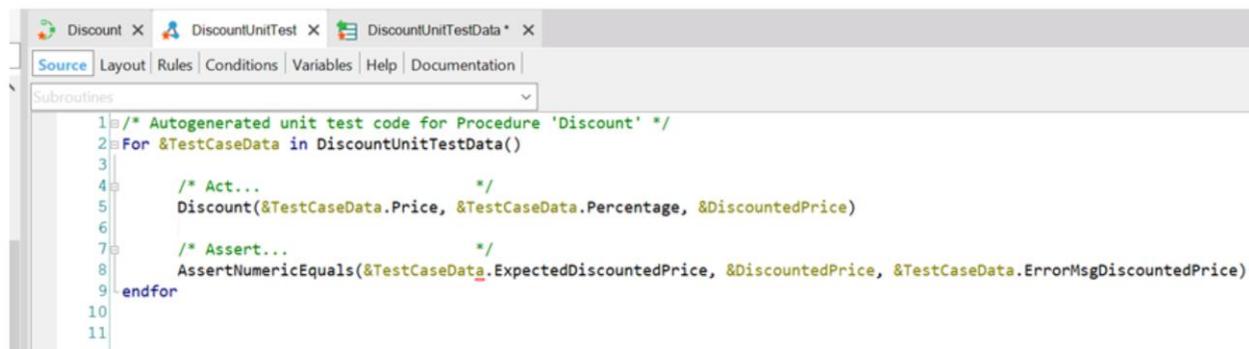
El objeto DiscountUnitTest es el que va a recorrer la colección de casos de prueba, y para cada uno de ellos va a invocar a nuestro procedimiento y validar si el resultado obtenido es igual al resultado esperado.

Este objeto es un procedimiento GeneXus y se programa como tal, por lo cual vamos a ver una notación que nos es muy familiar,

Es decir PARA CADA caso de prueba en la colección de Tests que definimos en el data Provider, se hace el call al procedimiento que estamos testeando con los parámetros de entrada definidos en el caso de prueba y una variable como valor de salida.

Lo que es nuevo en el test unitario es el comando ASSERT. Que básicamente compara un resultado esperado – definido como parte del caso de prueba – contra el resultado obtenido. Si el resultado esperado y el resultado obtenido son iguales, la prueba es exitosa y se dice que PASA o es un PASS, y si hay alguna diferencia la prueba falla o es un FAIL y se reporta que hubo un error mostrándose un mensaje asociado.

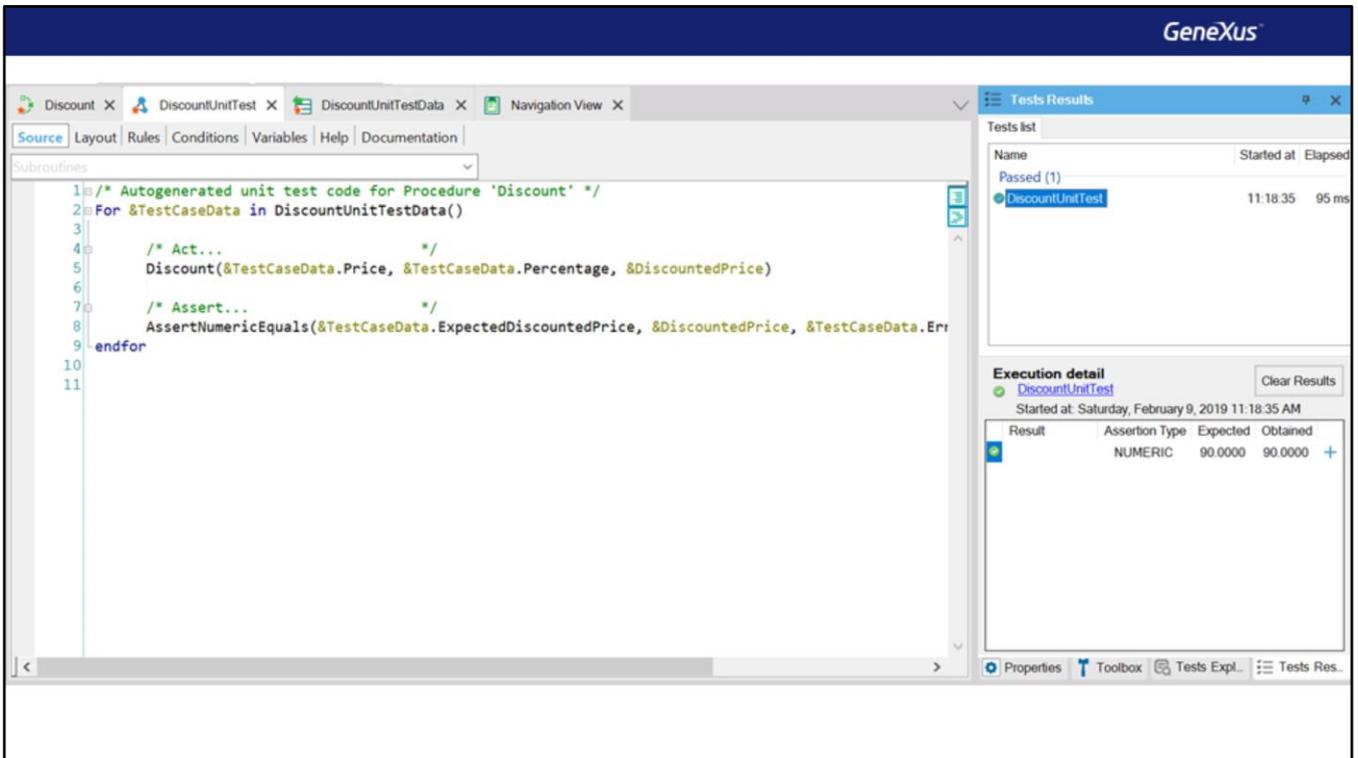
Aquí estamos utilizando la función AssertNumericEquals para validar el resultado ya que el precio descontado es un numérico pero también existe la posibilidad de utilizar AssertNumericEquals para comparar booleanos o AssertStringEquals que nos permite comparar texto y por lo tanto cualquier tipo de datos más complejo



The screenshot shows the GeneXus IDE interface with three open windows: 'Discount', 'DiscountUnitTest', and 'DiscountUnitTestData'. The 'DiscountUnitTest' window is active, displaying the source code for a unit test. The code is as follows:

```
1 /* Autogenerated unit test code for Procedure 'Discount' */
2 For &TestCaseData in DiscountUnitTestData()
3
4     /* Act... */
5     Discount(&TestCaseData.Price, &TestCaseData.Percentage, &DiscountedPrice)
6
7     /* Assert... */
8     AssertNumericEquals(&TestCaseData.ExpectedDiscountedPrice, &DiscountedPrice, &TestCaseData.ErrorMsgDiscountedPrice)
9 endfor
10
11
```

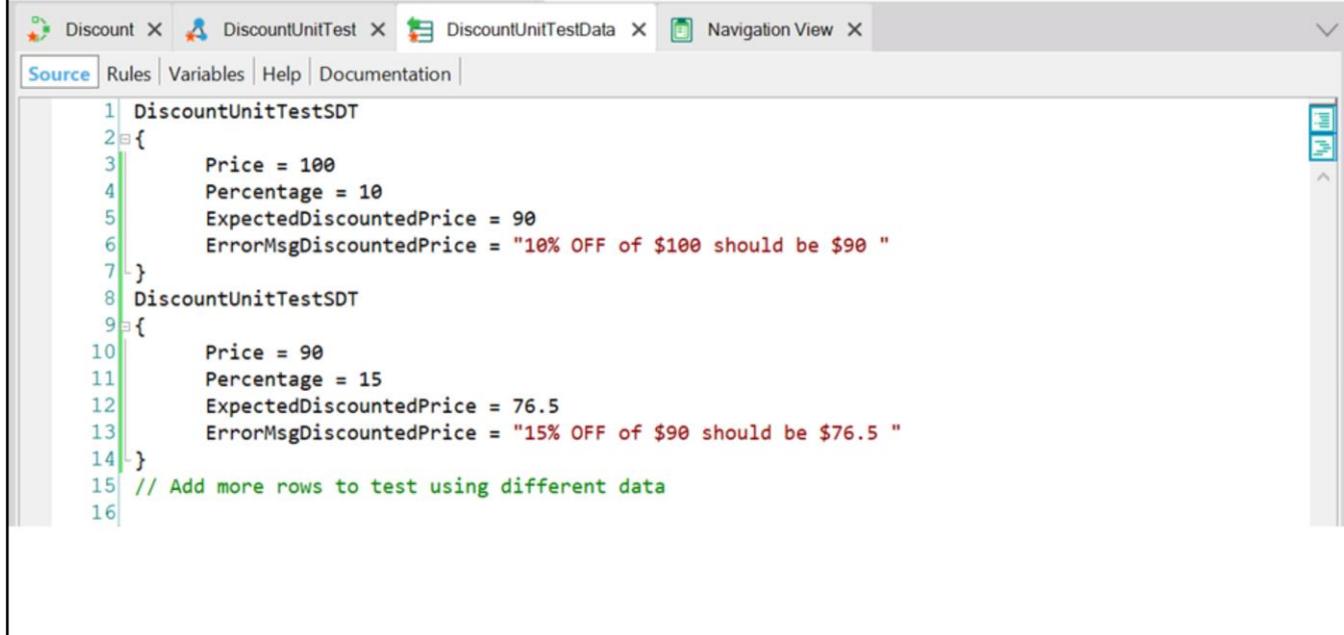
Ahora que vimos los tres objetos que se crearon de forma automática al crear nuestra prueba unitaria, que cargamos los datos para nuestro primer caso de prueba, vamos a ejecutar la prueba dando botón derecho y eligiendo la opción [Run This Test]



Una vez que la prueba completa la ejecución vamos a ver la nueva ventana - que se llama TEST-RESULTS - donde podemos ver que se ejecuto nuestra prueba (DiscountUnitTest) y que el resultado fue exitoso ya que esta marcado en verde.

También nos da información del tiempo de ejecución de la prueba

Abajo - en el sector Execution Detail - vamos a ver una línea por cada Assert que haya en nuestro test. Para cada uno podemos ver el resultado esperado, el resultado obtenido y también la marca verde o roja según si el Assert fallo o paso. En caso que el Assert falle vamos a ver el mensaje que definimos en nuestro caso de prueba.



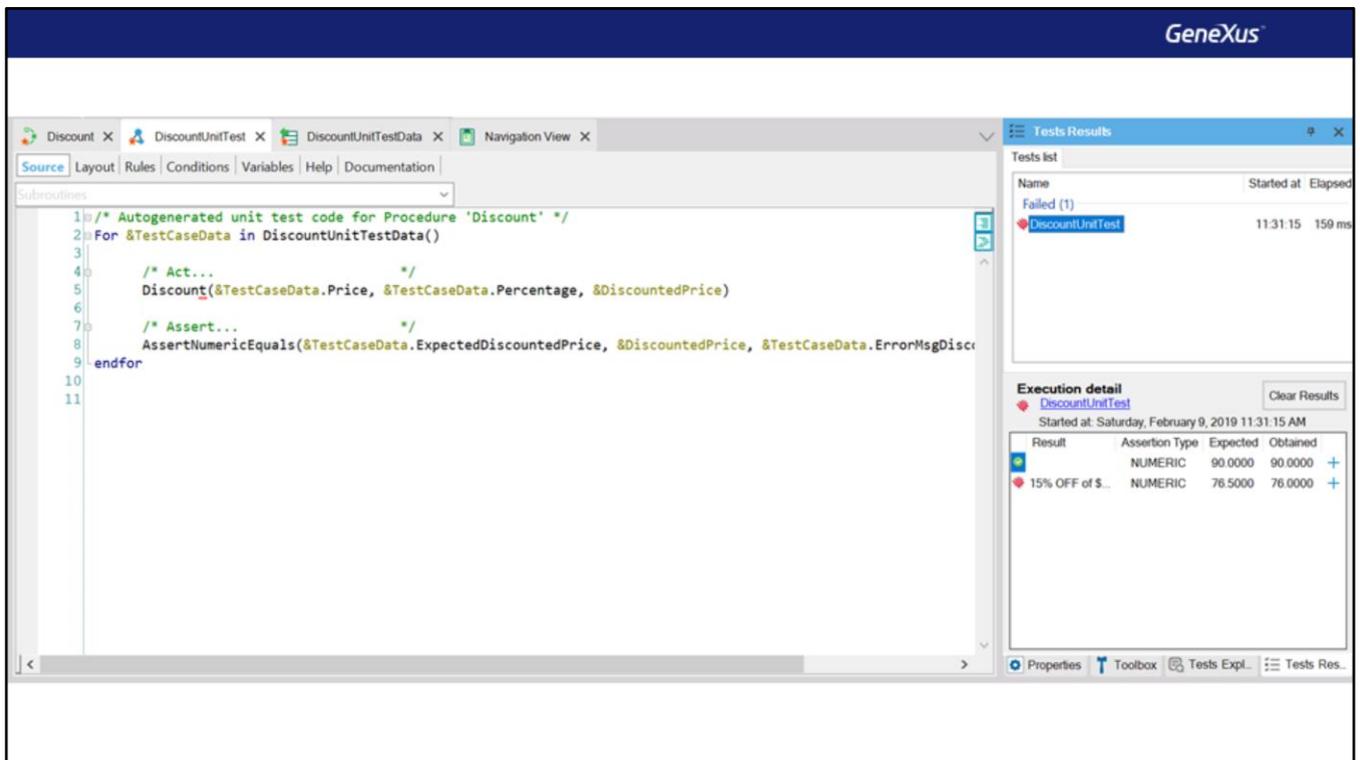
```
1 DiscountUnitTestSDT
2 {
3     Price = 100
4     Percentage = 10
5     ExpectedDiscountedPrice = 90
6     ErrorMsgDiscountedPrice = "10% OFF of $100 should be $90 "
7 }
8 DiscountUnitTestSDT
9 {
10    Price = 90
11    Percentage = 15
12    ExpectedDiscountedPrice = 76.5
13    ErrorMsgDiscountedPrice = "15% OFF of $90 should be $76.5 "
14 }
15 // Add more rows to test using different data
16
```

Ahora que tenemos nuestra primer prueba exitosa, y vimos que sencillo es definir un caso de prueba, vamos a definir otros casos mas en nuestro DataProvider

La selección de los datos a probar es una tarea importante, y una buena oportunidad para colaborar con el tester del equipo de forma de definir las pruebas que nos den mejor cobertura.

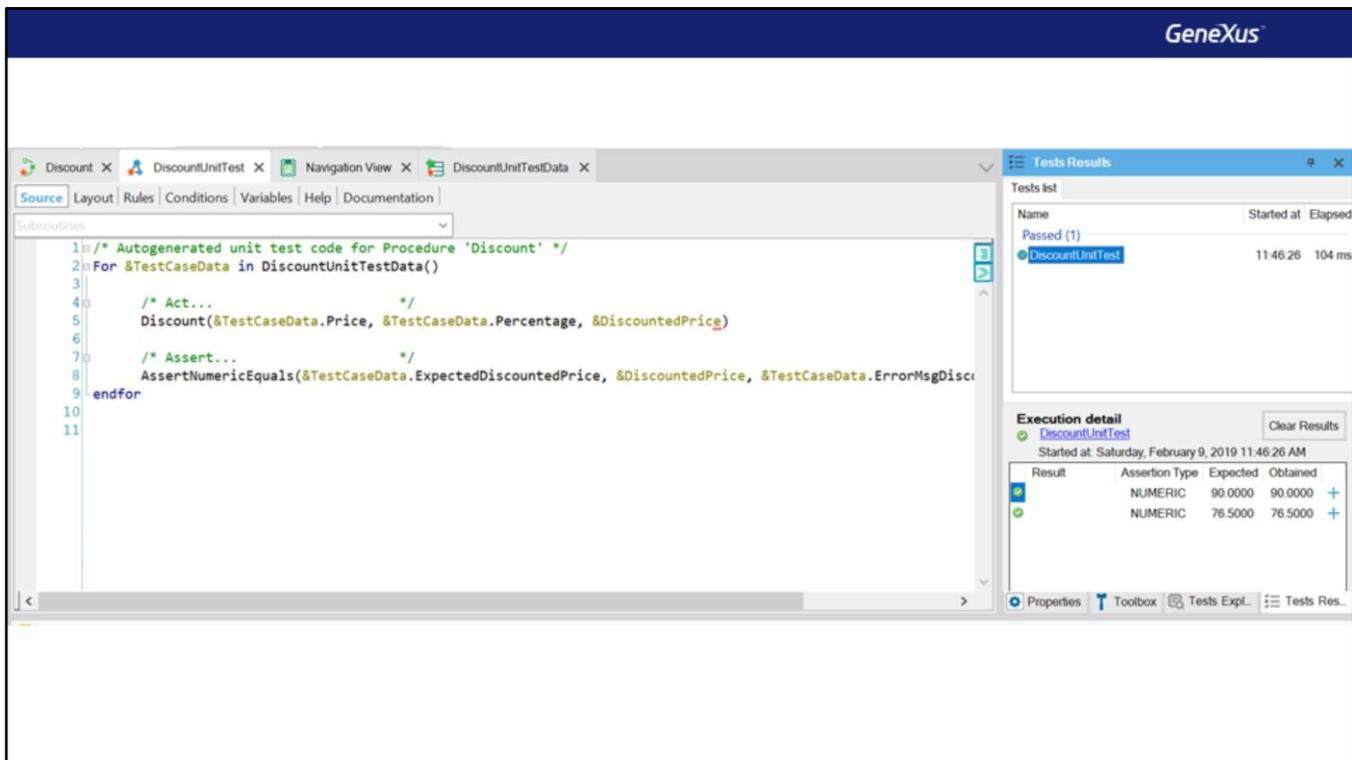
En este caso elegimos un caso cualquiera y simple (aplicar un 10% de descuento a un precio de \$100) y ahora vamos a agregar una prueba que valide que los decimales se manejan bien, por tanto elegimos números que nos den un resultado con decimales.

Agregamos entonces otro grupo y ahora decimos que dado un precio de 90 y un porcentaje de descuento de 15, el precio esperado es 76.5. Y nuevamente ejecutamos nuestra prueba



Vamos a ver que ahora vamos a tener 2 Assert, uno por cada caso de prueba que estamos ejecutando. Ahora vemos que para nuestra sorpresa que la prueba no es exitosa, como decíamos tenemos 2 Assert, el primero es el del primer caso de prueba que fue exitoso, y el segundo corresponde al 2º caso de prueba en donde el resultado obtenido fue 76 en vez de 76.5, es decir, nuestro procedimiento no esta considerando decimales.

Esto nos muestra que hay un error en el procedimiento y que la variable donde se calcula el precio descontado seguramente esta mal definida como un numero entero en vez de tener decimales. Acá vemos también que en caso de que la prueba falle, es decir que el Assert no de un resultado exitoso, vemos el mensaje que habíamos definido en el caso de error. Esto solo lo vemos en el caso en que el Assert falle, si la prueba es exitosa el mensaje no se muestra.



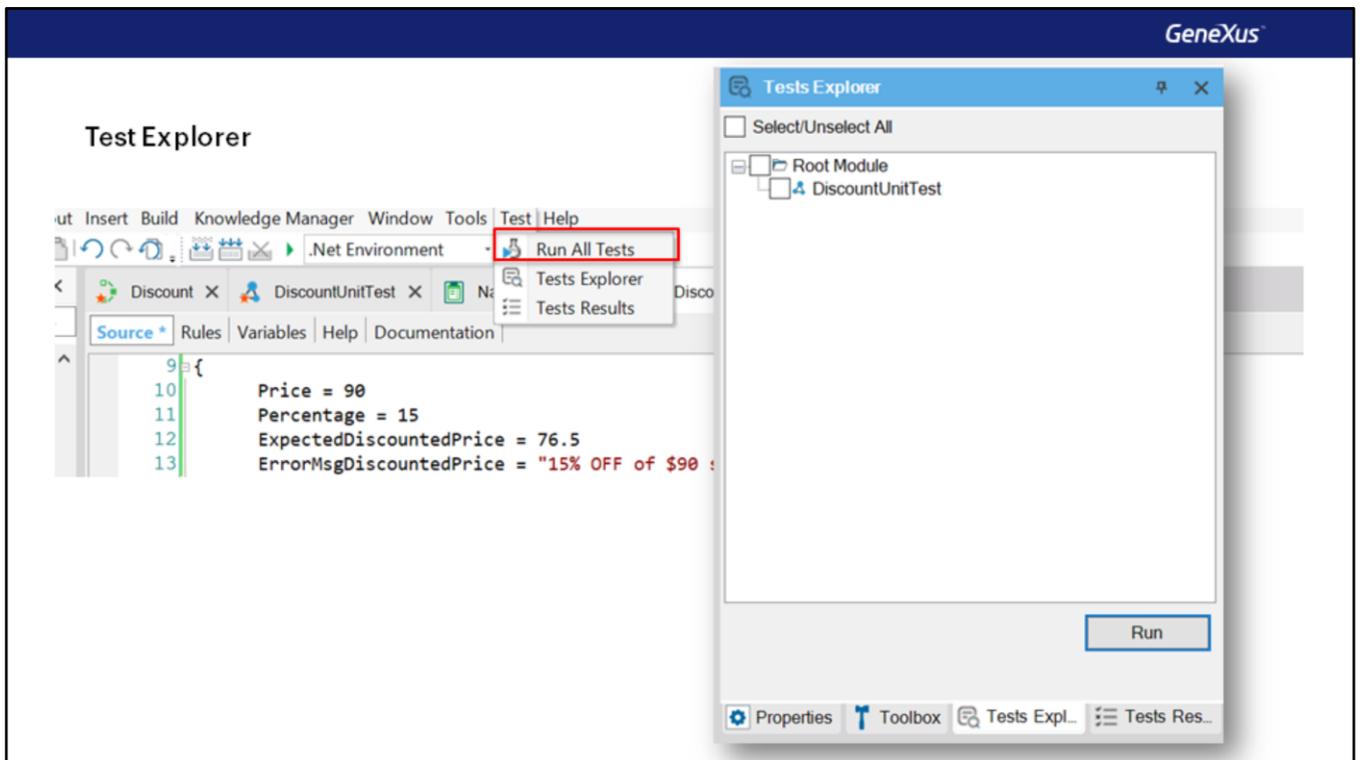
Sabiendo que tenemos un problema en la definición de la variable de salida de nuestro procedimiento vamos a editarlo, y vemos que efectivamente la variable había quedado sin definir. La definimos ahora correctamente en base al dominio Price. Luego de arreglar la variable volvemos a ejecutar la prueba y vamos a ver que ahora la misma es exitosa.

Para este ejemplo hice un poquito de trampa, porque cuando definimos el test unitario, los tipos de dato de los elementos del SDT se definen con el mismo tipo de dato que los del procedimiento a testear, por lo cual si no hubiera hecho trampa, el resultado del Assert hubiera sido un PASS ya que ambos valores hubieran sido enteros y el resultado esperado también hubiera sido 76. Si bien eso nos hubiera dado una pista de que había un problema, al mirar el resultado esperado y el resultado obtenido, no es la idea que uno descubra problemas analizando los resultados de una prueba en verde. Pero quería mostrar un ejemplo simple y en realidad este tipo de cosas podrían pasar en la vida real si alguien hubiera cambiado x equivocación la variable de salida del procedimiento Discount DESPUES de que el test ya hubiera estado creado, como fue lo que paso aquí.

```
2 {  
3     Price = 100  
4     Percentage = 10  
5     ExpectedDiscountedPrice = 90  
6     ErrorMsgDiscountedPrice = "10% OFF of $100 should be $90"  
7 }  
8 DiscountUnitTestSDT2  
9 {  
10     Price = 90  
11     Percentage = 15  
12     ExpectedDiscountedPrice  
13     ErrorMsgDiscountedPrice  
14 }  
15 DiscountUnitTestSDT2  
16 {  
17     Price = 0  
18     Percentage = 0  
19     ExpectedDiscountedPrice  
20     ErrorMsgDiscountedPrice = "0% OFF FROM $0 should be 0"  
21 }  
22 DiscountUnitTestSDT2  
23 {  
24     Price = 100  
25     Percentage = 200  
26     ExpectedDiscountedPrice = |  
27     ErrorMsgDiscountedPrice = ""  
28 }
```

Podemos seguir creciendo nuestras pruebas agregando casos de borde .. Algo que esta bueno de esta forma de testear es que nos facilita sentarnos a pensar sobre los casos... por ejemplo... esta claro que matemáticamente hablando el 200% de descuento sobre 100 seria -100 ahora bien... es un caso valido en nuestro contexto? Nuestro procedimiento Discount debería devolver un valor negativo o de alguna manera indicar un error?

Al pensar los casos de prueba nos surgen preguntas que nos ayudan a mejorar la definición de requerimientos y hacer nuestro sistema mas robusto.



Una vez que terminamos de implementar nuestro procedimiento y nuestras pruebas y compartimos – o integramos – nuestro trabajo con el del resto del equipo, es importante notar que las pruebas unitarias como son objetos GeneXus son parte de la base de conocimiento y los vamos a compartir de la misma manera que compartimos otros objetos.

En el TestExplorer podemos ver todos los objetos de tests definidos sobre nuestra KB, quizás algunos implementados por nosotros u otros por nuestros compañeros y podemos ejecutarlos todos o una selección de los mismos desde aquí haciendo solo un click.

Si el día de mañana nosotros, u otro desarrollador, tiene que modificar el procedimiento Discount – va a hacerlo mas tranquilo porque si rompe algo – es decir si el resultado obtenido para los mismos parámetros de entrada es diferente del anterior, la prueba va a fallar y se va a detectar el problema de forma temprana.

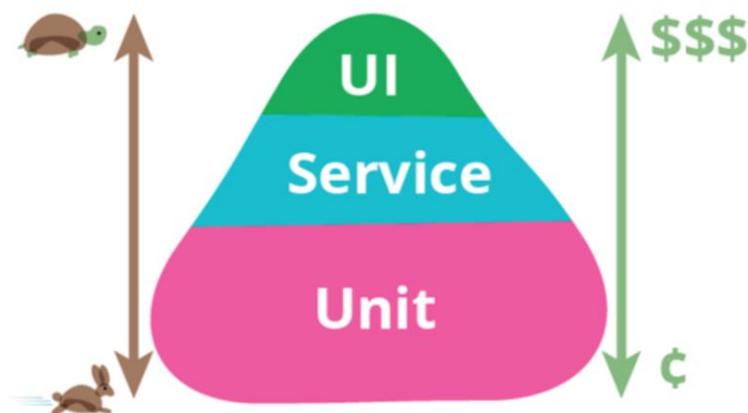
Unit Tests

- Rápidos
- Repetibles (Regresión)
- Mejoran la Testeabilidad de la aplicación

Si bien hemos visto un ejemplo muy básico, podemos testear todo tipo de procedimientos, DataProviders y BC de esta manera. Podemos realizar tests mucho más complejos – utilizando datos reales de nuestra aplicación (ya sea en una base de datos real o simulada) y cubrir la validación de una parte muy importante de nuestra aplicación.

Quizás, crear el test unitario nos da el mismo trabajo que crear un procedimiento de prueba que imprime valores en consola, pero tiene la gracia que la verificación la hace el propio test y no nosotros. Los tests unitarios deben ser rápidos de ejecutar – porque los vamos a querer usar no solo para probar mientras desarrollamos la funcionalidad, sino que también, vamos a querer repetir todas las pruebas que estén definidas en la KB fácilmente, luego de hacer cambios, para ver que no rompimos ninguna otra prueba al implementar nuestra funcionalidad. Es decir, el conjunto de tests unitarios que vamos construyendo para cada funcionalidad, nos automatizan parte de las pruebas de regresión.

Vimos también que al trabajar de esta manera, nos ayuda a desarrollar una aplicación más robusta.



<https://wiki.genexus.com/commwiki/servlet/wiki?38353,UI+Test+Automation>

Si bien con las pruebas unitarias vamos a cubrir una parte importante de la aplicación, no estamos cubriendo ninguna de las interacciones que suceden en pantalla ni los flujos completos de la aplicación.

Para esto no tenemos más remedio que ejecutar la aplicación utilizando la interfaz de la misma, es decir, navegando la pantallas de la aplicación como lo haría un usuario.

Para automatizar pruebas a nivel de Interfaz es que tenemos el objeto UI Test o User-Interface-Test

Estas pruebas son más complejas y llevan más tiempo de implementar y de ejecutar por eso la pirámide del testing nos recuerda que la mayoría de las pruebas automatizadas de nuestra aplicación deben ser unitarias – ya que son más rápidas y menos costosas – luego a nivel de servicios y reservar las pruebas de Interfaz solo para aquellos flujos que son críticos en nuestra aplicación.

Para saber más del Test de UI ir a

<https://wiki.genexus.com/commwiki/servlet/wiki?38353,UI+Test+Automation>,

GeneXus™

The power of doing.

Videos	training.genexus.com
Documentation	wiki.genexus.com
Certifications	training.genexus.com/certifications

Hemos visto entonces las facilidades de la versión 16 para crear y ejecutar pruebas automáticas, lo cual es un elemento clave en un buen proceso de ingeniería.