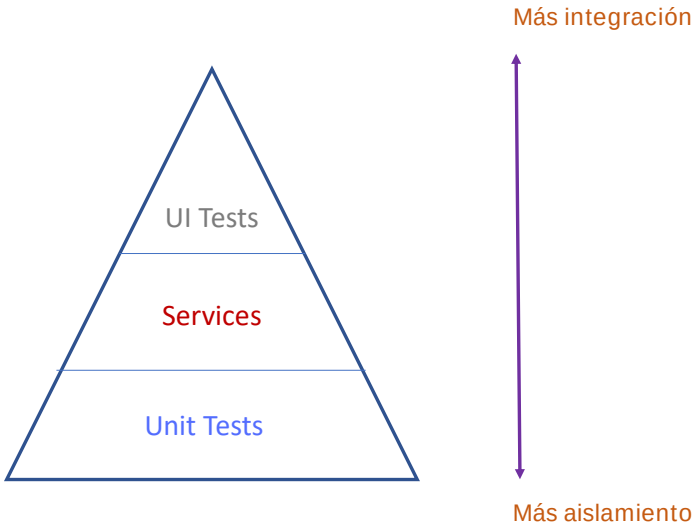
Tests unitarios

Introducción

GeneXus™

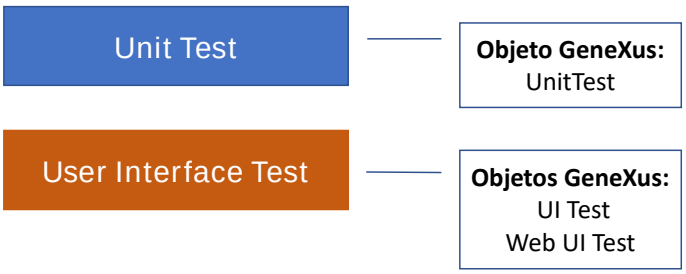
A lo largo del desarrollo de nuestra aplicación para la Agencia de viajes, hemos mencionado la importancia de testear en forma aislada las nuevas funcionalidades que vamos desarrollando, y de probar luego la aplicación entera para asegurarnos de que el comportamiento sea correcto.

Tests automáticos



A medida que la aplicación crece este tipo de tareas se puede ir volviendo más tediosas, y es entonces que GeneXus nos ayuda dándonos funcionalidades para poder crear y ejecutar tests automáticos, de forma de poder reducir parte del trabajo manual de verificación.

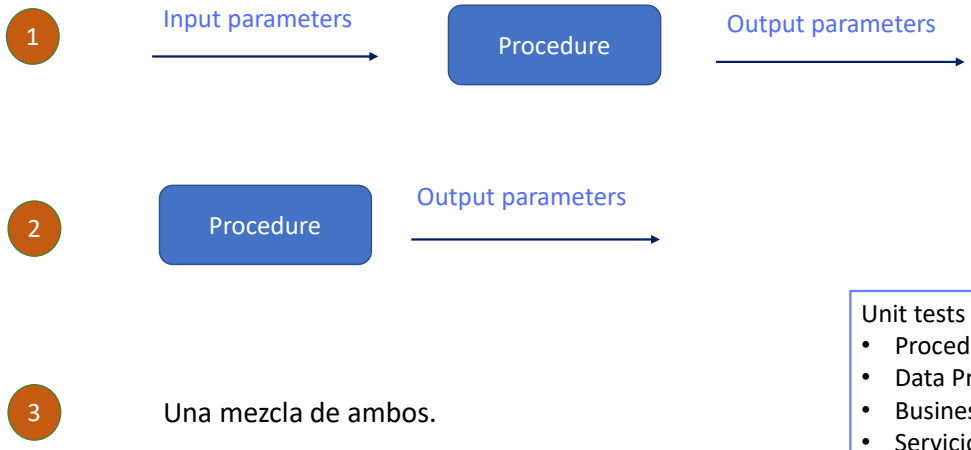
Tests automáticos



- Los **tests Unitarios** nos permiten testear una parte de la aplicación de forma aislada. En GeneXus, los tests unitarios sin interfaz aplican a pruebas sobre procedimientos, DataProviders y BusinessComponents. En definitiva, sobre aquellos componentes donde debería residir la lógica de negocios de nuestra aplicación, y para eso existe el objeto Unit Test.

- 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. Para eso existen los objetos UITest para interfaz mobile, y Web UITest para interfaz web.

¿Cuándo interesa testear un procedimiento GeneXus? (lógica de GeneXus)



Entonces, ¿Cuándo interesa testear un procedimiento GeneXus?

- Cuando dados algunos parámetros de entrada, se desea probar la salida del Procedimiento comparando los resultados esperados (usando Aserciones) con los resultados del proceso (ya sea variables de salida, archivos o registros de la base de datos). Para esto se utilizan aserciones, o sea aseveraciones o enunciados incluidos en el procedimiento.
- Cuando no hay parámetros de entrada, pero a partir de parámetros o configuraciones de la base de datos, se espera que el Procedimiento devuelva algunos resultados esperados (por ejemplo, variables o registros de la base de datos).
- Y cuando se tiene una mezcla de ambos escenarios.

Esto significa que cualquier Procedimiento GeneXus puede ser probado para verificar que funcione de acuerdo con las especificaciones definidas. Pero además de los Procedimientos, también se pueden definir test unitarios asociados a Data Providers, Business Components y Servicios Rest.

Beneficios:

Detectar errores en el código en forma temprana.

Dar feedback inmediato a los desarrolladores.

Son rápidas y se comparten en toda la Base de Conocimiento si se está usando GeneXus Server.

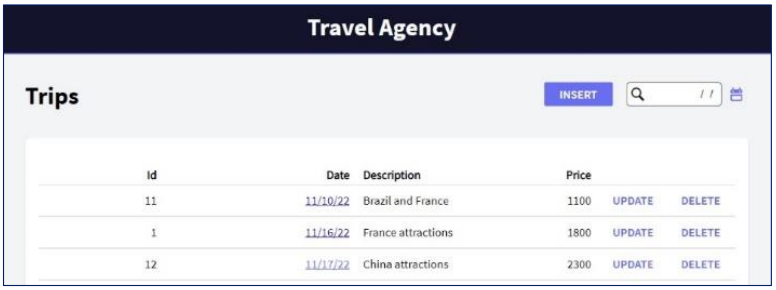
Los desarrolladores pueden ejecutar sus pruebas desde el propio IDE de GeneXus sin necesidad de otras herramientas.

Los principales beneficios de realizar pruebas unitarias son los siguientes:

- Detectar errores en el código en forma temprana.
- Dar feedback inmediato a los desarrolladores.
- Son rápidas y se comparten en toda la Base de Conocimiento si se está usando GeneXus Server.
- Los desarrolladores pueden ejecutar sus pruebas desde el propio IDE de GeneXus sin necesidad de otras herramientas

Ejemplo: UpdateTripPrice

Procedimiento que actualiza el precio de un viaje de acuerdo a un porcentaje recibido por parámetro.



Bien, vamos a ver entonces un ejemplo.

Desde el launchpad, ejecutamos Work with Trip, y vemos que tenemos tres viajes registrados con sus costos actuales.

1100, 1800 y 2300

Ejemplo: UpdateTripPrice

```
Parm(in:&TripId,in: &Percentage, out: &UpdateResult);


For each Trip
     Where TripId = &TripId
     &NewTripPrice = TripPrice*(1+&Percentage/100)
     if &NewTripPrice > 2500
          &UpdateResult = &NewTripPrice.ToString() + " - Too expensive"
     else
          TripPrice = &NewTripPrice
          &UpdateResult = &NewTripPrice.ToString() + " - The Trip price was updated"
     endif
when none
     &UpdateResult = "The TripId = " + &TripId.ToString() + " is not registered"
endfor
```

Hemos creado un procedimiento, de nombre UpdateTripPrice, que calcula el aumento de precio de un determinado viaje, de acuerdo con un porcentaje recibido por parámetro. Para la Agencia de viajes, el precio final de un viaje no puede superar el valor de 2500.

Vemos entonces que el procedimiento recibe dos parámetros de entrada:

La variable &TripId y el porcentaje de aumento

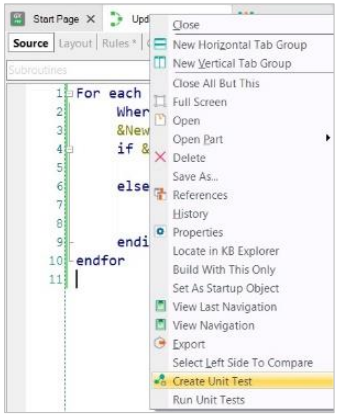
y tiene un parámetro de salida que devuelve el valor obtenido por el aumento junto con un comentario que dice si se realizó la actualización o se superó el monto máximo indicado.

Entonces, para el TripId recibido, el procedimiento calcula el valor de acuerdo con el porcentaje también recibido, y si ese valor es superior a 2500 no actualiza y devuelve el correspondiente mensaje.

Si el valor es menor o igual a 2500 entonces sí actualiza el valor del atributo TripPrice y también devuelve al mensaje correspondiente.

En caso de no existir un viaje con el valor de TripId recibido por parámetro, se devolverá un mensaje indicando que el viaje no está registrado.

Creación del Unit test asociado



Objeto: UpdateTripPriceTestSDT

Name	Type	Description	Is Collection
UpdateTripPriceTestSDT		Update Trip Price Test SDT	☐
TestCaseId	VarChar(40)	Test case identifier	☐
TripId	Attribute:TripId		☐
Percentage	Numeric(4.0)		☐
UpdateResult	Character(30)		☐
ExpectedUpdateResult	Character(30)		☐
MsgUpdateResult	VarChar(100)	Message to show for assertion over field Upd...	☐

Para testear este procedimiento vamos a crear el correspondiente Unit test.

Para eso, sobre la solapa del procedimiento, damos click derecho y elegimos Create Unit Test.

GeneXus crea entonces tres objetos, que podemos ver bajo el nuevo nodo Tests en la ventana KBExplorer:

El objeto UpdateTripPriceTestSDT,

que 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 ExpectedUpdateResult donde vamos a poder definir el valor del resultado que esperamos.

En definitiva, vamos a poder decir, por ejemplo, que, para el viaje, TripId=11, con un costo actual de 1100, si indicamos un porcentaje de aumento de 10% esperamos un resultado de 1210 – con un mensaje indicando que el precio se actualizó correctamente.

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

Creación del Unit test asociado

Objeto: **UpdateTripPriceTestData**

```
UpdateTripPriceTestSDT
{
    TestCaseId = '1'
    TripId = 11
    Percentage = 10
    ExpectedUpdateResult = '1210 - The Trip price was updated'
    MsgUpdateResult = ''
}

UpdateTripPriceTestSDT
{
    TestCaseId = '2'
    TripId = 1
    Percentage = 10
    ExpectedUpdateResult = '1980 - The Trip price was updated'
    MsgUpdateResult = ''
}

UpdateTripPriceTestSDT
{
    TestCaseId = '3'
    TripId = 12
    Percentage = 10
    ExpectedUpdateResult = '2530 - The Trip price was updated'
    MsgUpdateResult = ''
}

UpdateTripPriceTestSDT
{
    TestCaseId = '4'
    TripId = 8
    Percentage = 10
    ExpectedUpdateResult = 'The TripOd = 8 is not registered'
    MsgUpdateResult = ''
}
```

Pasemos ahora el **objeto UpdateTripPriceTestData**, también creado automáticamente por GeneXus.

Este Data Provider está basado en el SDT que vimos anteriormente, y nos permite definir un juego de datos. Por defecto se crean 5 casos de prueba, pero podemos modificarlo.

Tenemos tres viajes registrados, así que vamos a definir tres pruebas con un porcentaje de aumento del 10%. En cada caso indicamos el valor esperado devuelto por el procedimiento.

Pero definimos también un cuarto juego de prueba para el TripId= 8 que no existe actualmente en nuestra base de datos.

Creación del Unit test asociado

Objeto: **UpdateTripPriceTest**

```
/* Autogenerated unit test code for Procedure 'UpdateTripPrice' */

For &TestCaseData in UpdateTripPriceTestData()

    /* Act...                       */
    &TestCaseData.UpdateResult = UpdateTripPrice(&TestCaseData.TripId, &TestCaseData.Percentage)

    /* Assert...                    */
    AssertStringEquals(&TestCaseData.ExpectedUpdateResult, &TestCaseData.UpdateResult, format(!'%1.ExpectedUpdateResult: %2', &TestCa
endfor
```

Comando Assert para comparar un resultado esperado con un resultado obtenido

- AssertStringEquals – para comparar textos
- AssertBoolEquals – para comparar valores booleanos
- AssertNumericEquals – para comparar valores numéricos

Finalmente, el **objeto UpdateTripPriceTest**, 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.

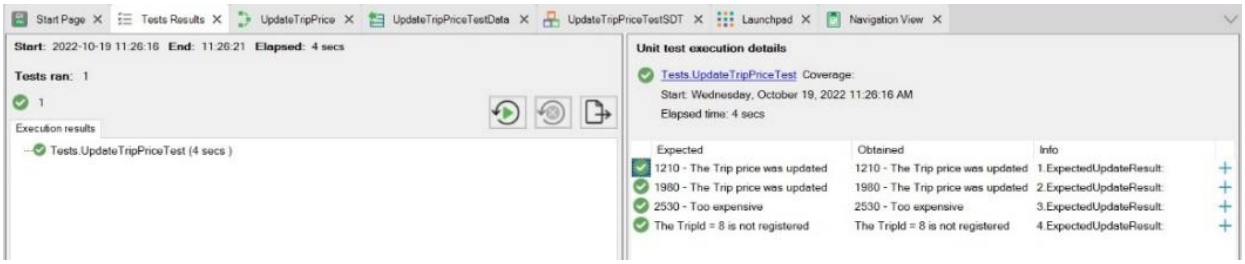
Este objeto es un procedimiento GeneXus y se programa como tal, por lo cual vamos a ver una notación que nos es muy familiar,

Es decir PARA CADA caso de prueba en la colección de Tests que definimos en el data Provider, se hace la llamada 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 AssertStringEquals para validar el resultado ya que el resultado es un texto, pero también existe la posibilidad de utilizar AssertBoolEquals para comparar booleanos o AssertNumericEquals que nos permite comparar valores numéricos.

Ejecución del Unit test



Bien. Para ejecutar, click derecho, Run Test

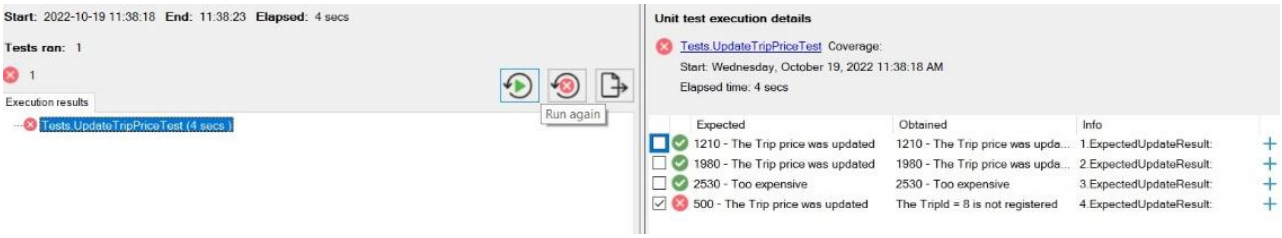
Una vez que la prueba completa la ejecución vamos a ver la nueva ventana – de nombre TEST-RESULTS - donde comprobamos que se ejecutó nuestra prueba (UpdateTripPriceTest) y que el resultado fue exitoso ya que está marcado en verde.

También nos da información del tiempo de ejecución de la prueba
Vemos una línea por cada Assert que hemos definido 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 falló o pasó.

Ejecución del Unit test

Se modifica el Data Provider para generar una falla.

```
UpdateTripPriceTestSDT
{
     TestCaseId = '4'
     TripId = 8
     Percentage = 10
     ExpectedUpdateResult = '500 - The Trip price was updated'
     MsgUpdateResult = ''
}
```



Vamos a correr nuevamente el test, pero antes vamos a volver los costos al estado inicial, y modificar el Data Provider asumiendo que el viae con identificador TripId = 8 existe en nuestra base de datos y le asignamos un costo determinado. La idea es generar una falla en este juego de prueba ya que nuestro procedimiento indicará que el viaje no está registrado.

Corremos nuevamente el test presionando este botón, y vemos que un juego de pruebas falló ya que el valor esperado y el obtenido son diferentes:

Pdemos ver la comparación entre ambos resultados, y podemos volver a ejecutar los juegos de prueba que fallaron.

También podemos exportar el resultado del test a HTML

Los Tests unitarios automatizan parte de las pruebas de software y nos ayudan a desarrollar aplicaciones más robustas.

Si bien hemos visto un ejemplo sencillo, como ya hemos dicho, podemos testear todo tipo de procedimientos, DataProviders y Business Component, 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.

El conjunto de tests unitarios que vamos construyendo para cada funcionalidad, nos automatiza parte de las pruebas de regresión, y nos ayuda a desarrollar una aplicación más robusta.

GeneXus™

training.genexus.com
wiki.genexus.com