

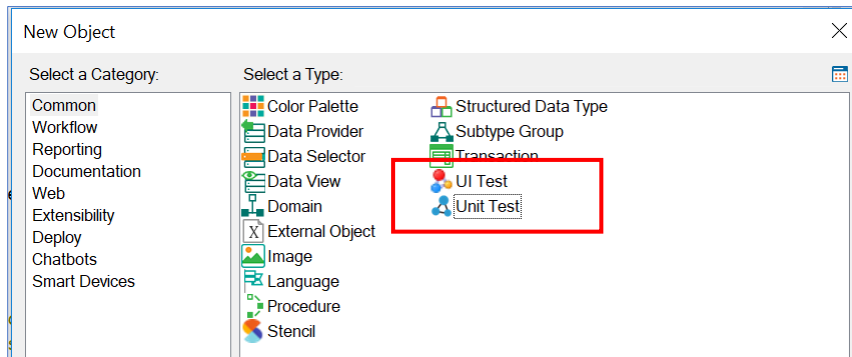
## Unit Test. Introduction.

GeneXus™

When we develop a new functionality in our app, we must test what we develop to verify that it functions as expected. It is also important to test the whole application again after that change to ensure that what was already functioning continues to behave correctly.

As the app grows, these tasks may become increasingly tedious because we must test over and over again. There is also a higher cost of testing because more time becomes necessary each time.

## UITest &amp; Unit Test

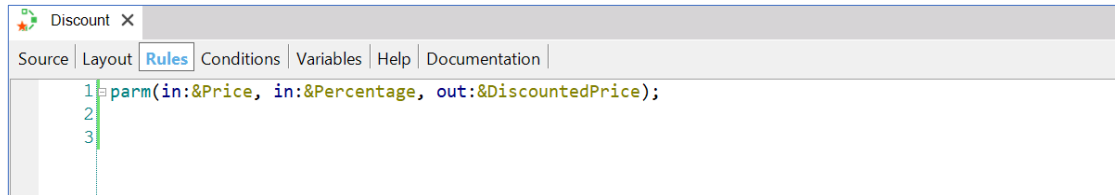


GeneXus aids us with this by providing us with functionalities to create and execute automatic tests, both unit tests and interface tests that will reduce part of the manual work required for verification.

Unit tests enable us to test one part of the app in an isolated manner. This applies to tests for procedures, DataProviders, and BusinessComponents. In sum, all those components where the business logic of our app lies.

The Interface tests allows us to create tests by simulating the actions of a user on the browser, so as to test full flows in the app.

## Unit Tests - Example



```
Discount X
Source | Layout | Rules | Conditions | Variables | Help | Documentation |
1 | parm(in:&Price, in:&Percentage, out:&DiscountedPrice);
2 |
3 |
```

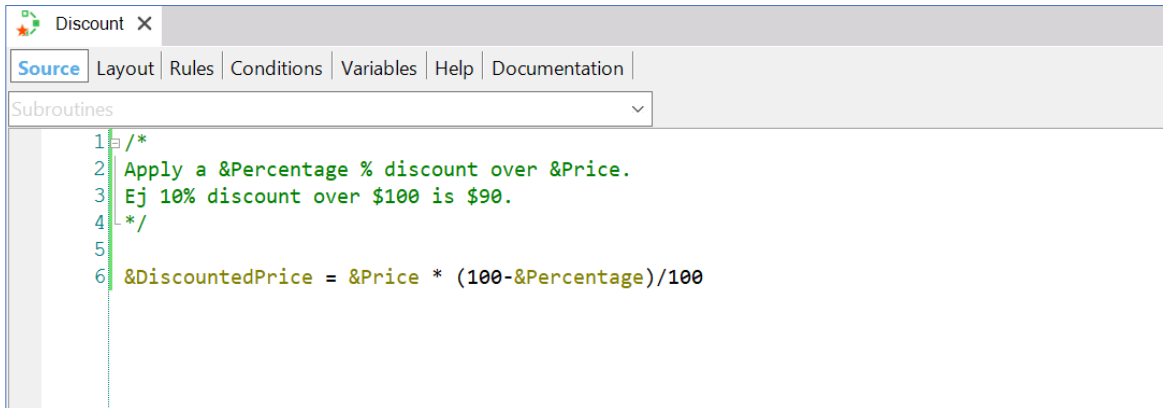
We will be focusing on unit tests, and we will do it with a very basic example.

We have a procedure that calculates a Discount – it basically applies a discount percentage to a price – which will be used for calculating promotions at the travel agency.

The procedure has two input parameters: the Price to be discounted, and the discount percentage. And it has an output parameter that is the discounted price.

What this procedure would do, for example, is, when we pass a price of \$100 and a discount of 10%, it will return the discounted price of \$90.

## Unit Tests - Example



```
1 /*  
2 Apply a &Percentage % discount over &Price.  
3 Ej 10% discount over $100 is $90.  
4 */  
5  
6 &DiscountedPrice = &Price * (100-&Percentage)/100
```

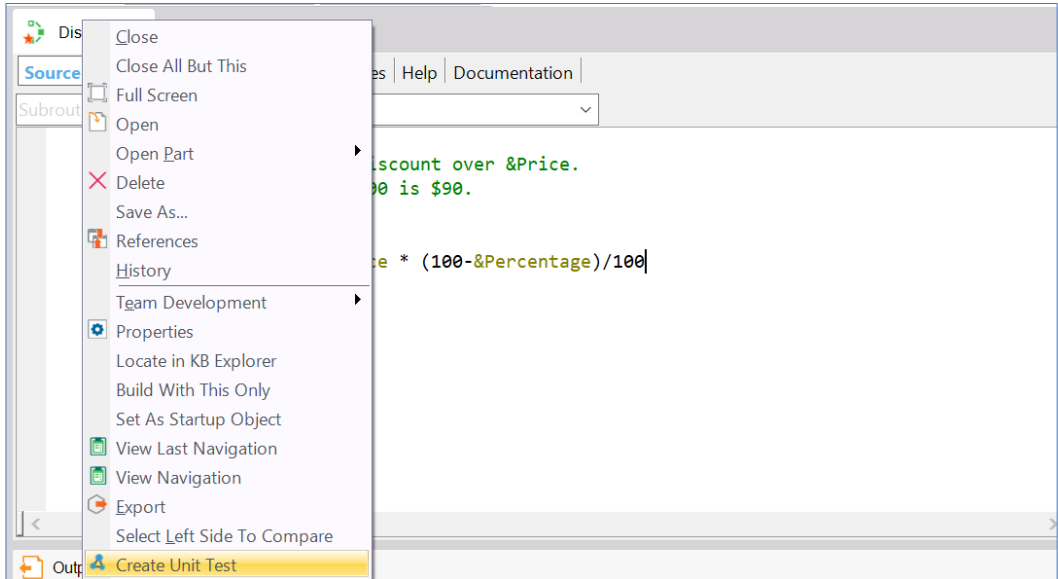
Here we will see how the procedure is implemented.

So, how could we test the functioning of this procedure as we expect it to function? That is, upon a given price and a percentage, it must return the correct adjusted price.

We could build a screen to enter values of price and discount, with a button calling the procedure and showing the value of the discounted price on screen. And we could test in an interactive manner that the procedure behaves as expected. We could also program a procedure called proc Discount with different parameters to start printing the results on the console, so as to verify that there the result is what we expect.

But that testing method is costly, because we must create a screen, enter the values manually and evaluate results. And we would be doing this on the computer that we use for developing. But later, when we are satisfied with the behavior, after sharing our work with the rest of the team, we must go on testing to verify that what we shared was well integrated.

The idea behind unit tests is to have the possibility of automating these tasks to avoid doing repeated work and to simplify the task.



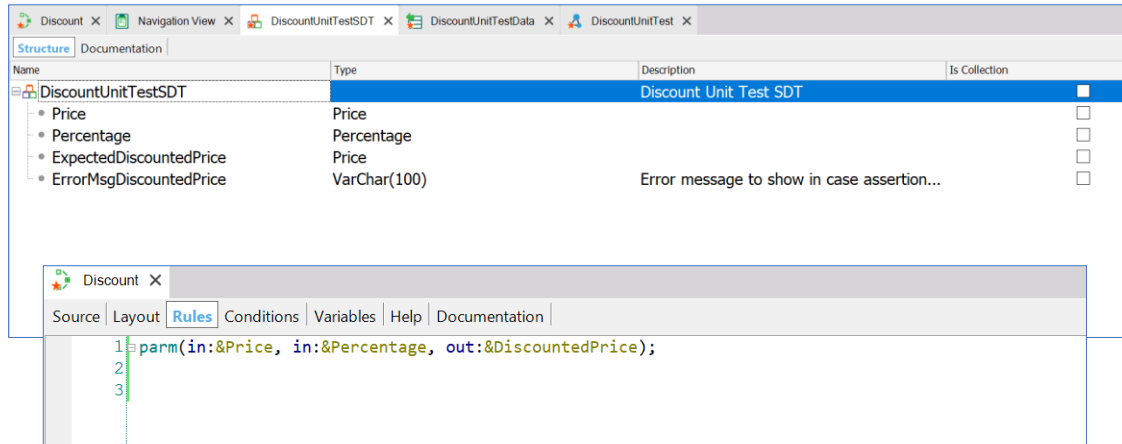
So, let's create a unit test for our procedure. To do that we right click on the object and we select the "Create Unit Test" option.

## Objects

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

When we create the unit test, three objects will be created. Let's see them in detail ...

<Object>UnitTestSDT



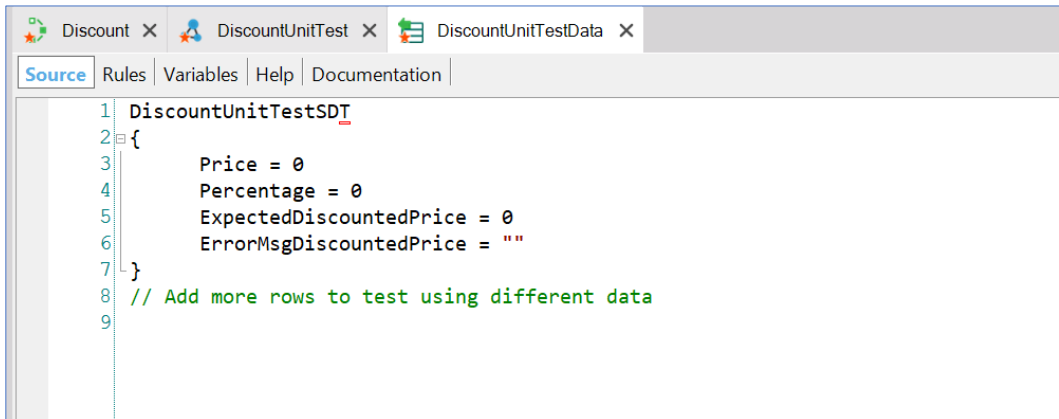
As mentioned, if we were to test this manually, we might have created a screen with the two input variables for the procedure and some way to view results of the output variable. In sum, a way of validating that upon the input values, we will get back an expected result.

We will see the DiscountUnitTestSDT, which is one of the objects created automatically when the unit test of the Discount procedure was created. This SDT defines the structure of a specific test case for the object we are testing.

We can see that, as we would do ourselves, it defines the two input variables, with the same name as the procedure's parameters, and it also defines a variable called ExpectedDiscountedPrice where we will be able to define the value of the result we expect to have.

We will actually be able to say that, for a price of 100, and a discount percentage of 10 we expect the result to be 90.

We can also assign a message stating that, for the case where the result is different from 90, we want it to be shown.



```
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
```

Now that we saw the structure of the test case for the Discount procedure, we will see how to define the data sets. We will do this in the DataProvider that was also defined automatically. Here we can see a defined group, with the elements of the SDT where we can instance the values...



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

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

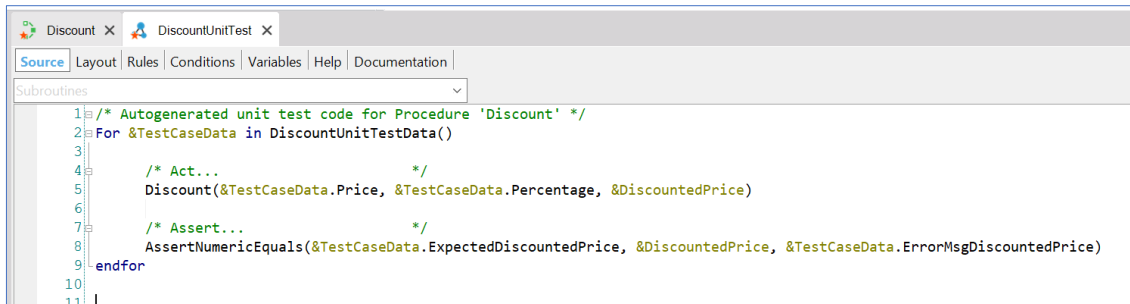
On the right, the Properties window for the Data Provider 'DiscountUnitTestData' is open. The 'Output' section is highlighted with a red box, showing the following configuration:

Data Provider: DiscountUnitTestData	
Name	DiscountUnitTestData
Description	Discount Unit Test Data
Expose as Web Service	False
Main program	False
Call protocol	Internal
Qualified Name	DiscountUnitTestData
Object Visibility	Public
Output	
Infer Structure	No
Output	DiscountUnitTestSDT
Collection	True
Collection Name	DiscountUnitTestSDTCollection
Network	
Connectivity Support	Inherit
Warning messages	
Disabled warnings	spc0096 spc0107 spc0142
Miscellaneous	
Generate Object	True

... and, for example, assign the ones we mentioned before.

We will also assign the message that we want to see for the case of obtaining a result different from 90.

This data-Provider returns a collection of test data, so it will allow us to define several tests, or several data sets in a simple manner for us to execute. But, for the time being, this one will suffice for executing our first test.



```
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.ErrorMessageDiscountedPrice)
9 endfor
10
11
```

Before we execute it we will see the third object that was created automatically, that is, the unit-test in itself.

The DiscountUnitTest object will go over the collection of test cases, and for each of them it will invoke our procedure and validate whether the result obtained is the same as the result expected.

This object is a GeneXus procedure that is programmed as such, so we will see an annotation with which we are quite familiar.

This means that FOR EACH test case in the collection of Tests that we define in the data Provider, a call is made to the procedure that we are testing with the input parameters defined in the test case, and a variable as output value.

The new thing in the unit test is the ASSERT command, which will basically compare an expected result – defined as part of the test case– against the result actually obtained. When the expected and the obtained results match, then the test is deemed successful, and we say that it has PASSED, or we call it a PASS. When there are differences, the test is failing so we call it a FAIL, and a report is made indicating that there was an error, showing an associated message.

Here, we are using the `AssertNumericEquals` function to validate the result because the discounted price is numeric, but there is also the possibility of using `AssertBoolEquals` to compare Booleans or `AssertStringEquals` that enables us to compare texts, and therefore any data type of greater complexity.

Now that we saw the three objects that were created automatically when we created our unit test, and after loading the data for our first test case, we will execute the test by right clicking and selecting the [Run This Test] option.

The screenshot displays the GeneXus IDE interface. The main editor shows the following code:

```

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.Err)
9
10 endfor
11

```

The 'Tests Results' window on the right shows the following information:

Tests list

Name	Started at	Elapsed
Passed (1)		
DiscountUnitTest	11:18:35	95 ms

Execution detail

DiscountUnitTest

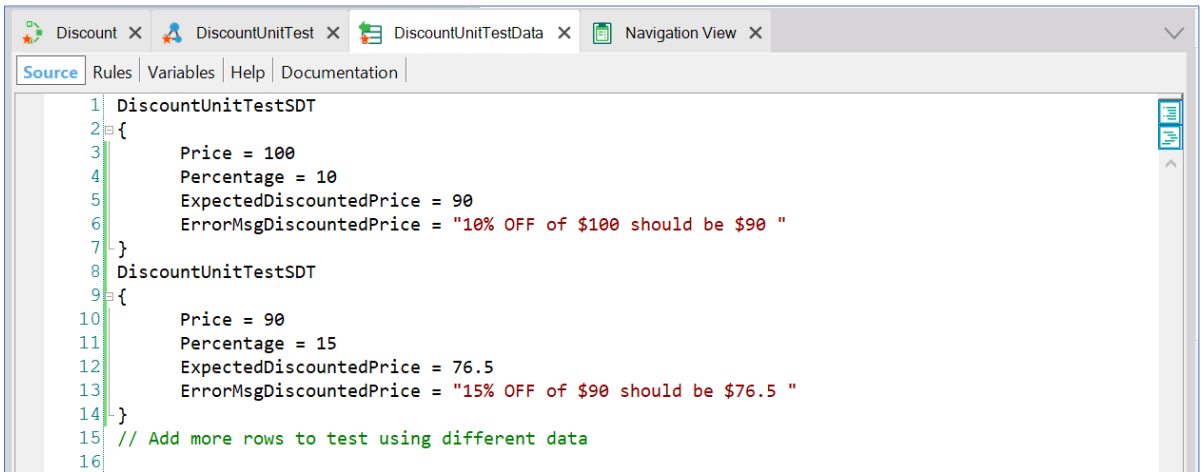
Started at: Saturday, February 9, 2019 11:18:35 AM

Result	Assertion Type	Expected	Obtained
+	NUMERIC	90.0000	90.0000

Once the test's execution is completed, we will see the new window – called TEST-RESULTS- where we can see that our test was executed (DiscountUnitTest) and that the result was successful because it is marked in green.

It also provides us with information of the time of execution for the test.

Below –in the Execution Detail area- we will see a line for each Assert found in our test. For each one of them we can see the result expected, the result obtained, and also a green or a red mark depending on whether the Assert has failed or passed. In the event that the Assert fails, we will also see the message that we defined in our test case.

A screenshot of a code editor window with four tabs: 'Discount', 'DiscountUnitTest', 'DiscountUnitTestData', and 'Navigation View'. The 'DiscountUnitTestData' tab is active, showing a source code file with two test cases. The first test case (lines 2-7) sets Price = 100, Percentage = 10, ExpectedDiscountedPrice = 90, and ErrorMessageDiscountedPrice = "10% OFF of \$100 should be \$90 ". The second test case (lines 8-14) sets Price = 90, Percentage = 15, ExpectedDiscountedPrice = 76.5, and ErrorMessageDiscountedPrice = "15% OFF of \$90 should be \$76.5 ". Line 15 contains a comment: "// Add more rows to test using different data".

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

Following our first successful test and knowing how simple it is to define a test case, we will now define other cases in our DataProvider.

The selection of data to be tested implies an important task and a good opportunity for cooperating with the team's tester in defining the tests that provide us with the best coverage possible.

This time we select any simple case (applying 10% discount to a price of \$100), and now we will add a test to validate that the decimal values are well managed. So we select numbers that will result in decimal numbers.

Then, we add another group, and now we decide that, upon a given price of 90 and a discount percentage of 15, the expected price is 76.5. And, again, we execute our test.

```

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.ErrorMessageDisc)
9
10 endfor
11

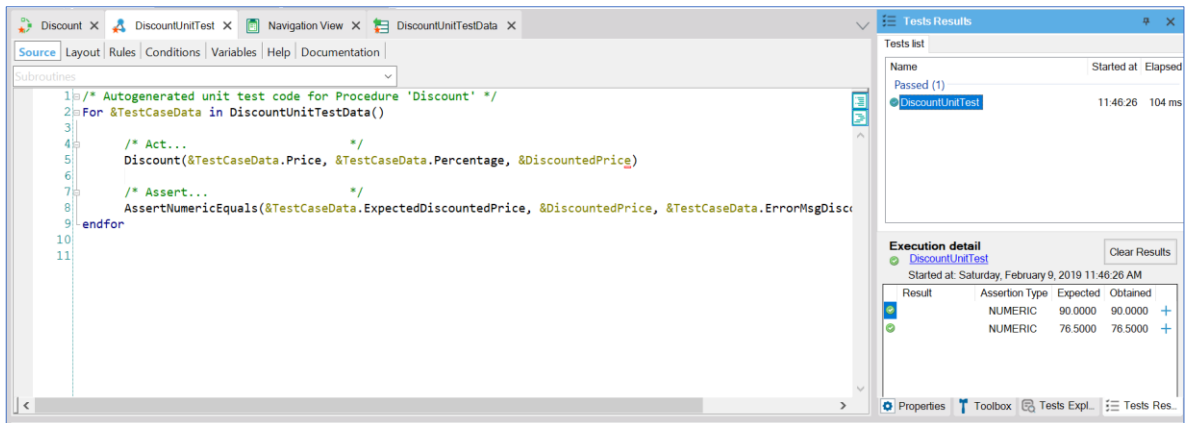
```

Result	Assertion Type	Expected	Obtained	
	NUMERIC	90.0000	90.0000	+
15% OFF of \$...	NUMERIC	76.5000	76.0000	+

We will see that, now, we will have two Asserts, one for each test case we execute.

Now we are surprised by the fact that the test is not successful. As we mentioned, we have two Asserts. The first one corresponds to the first test case, that turned out to be successful, and the second one corresponds to the second test case, where the result obtained was 76 instead of 76.5. This means that our procedure is not considering decimal values.

This is showing us that there is an error in the procedure and that the variable where the discounted price is calculated is most probably wrongly defined as a whole number instead of having decimal values. We can also see here that, in the event that the test fails –meaning that the Assert does not show a successful result–, we see the message we had defined in the error case. We will only see this in the event where the Assert fails. When the test is successful, the message is not shown.



```
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.ErrorMessageDisc
9 endfor
10
11
```

**Tests Results**

Name	Started at	Elapsed
DiscountUnitTest	11:46:26	104 ms

**Execution detail**

DiscountUnitTest

Started at: Saturday, February 9, 2019 11:46:26 AM

Result	Assertion Type	Expected	Obtained
+	NUMERIC	90.0000	90.0000
+	NUMERIC	76.5000	76.5000

If we know that we have a problem in the definition of the output variable of our procedure, we proceed to editing it. If we see that, in fact, the variable was left undefined, we now define it correctly based on the Price domain. After fixing the variable and executing the test again, we will see that the test now proves successful.

For this example, I introduced a little trick when we defined the unit test. The data types of the elements in the SDT are defined with the same data type as that of the procedure to be tested. So if we had not introduced that trick, the result of the Assert would have been a PASS because both values would have been whole numbers and the expected result would also have been 76. Even when this would have been a clue as to the existence of a problem, when we look at the expected result and the result obtained, the idea is not for us to discover problems by analyzing the results of a test that has been marked with green. But I wanted to show you an simple example. This sort of things could happen in reality if someone had mistakenly changed the output variable of the Discount procedure AFTER the test was already created, as it was the case here.

```

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 = 76.5
13     ErrorMsgDiscountedPrice = "15% OFF of
14 }
15 DiscountUnitTestSDT2
16 {
17     Price = 0
18     Percentage = 0
19     ExpectedDiscountedPrice = 0
20     ErrorMsgDiscountedPrice = "0% OFF from $0 Should be 0"
21 }

22 DiscountUnitTestSDT2
23 {
24     Price = 100
25     Percentage = 200
26     ExpectedDiscountedPrice = |
27     ErrorMsgDiscountedPrice = ""
28 }

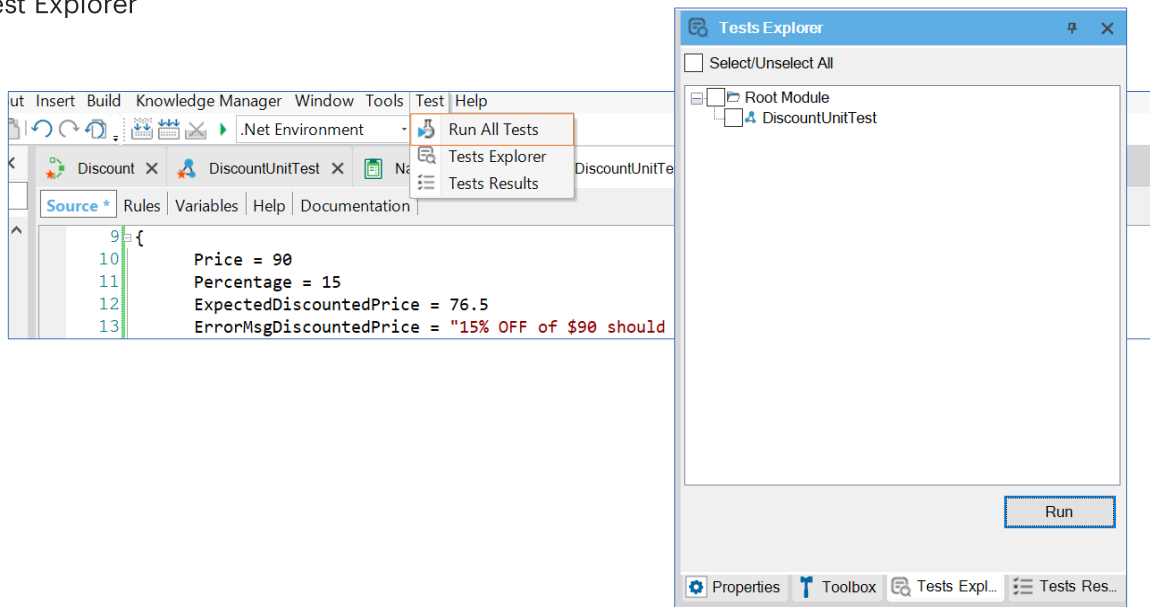
```

We can go on increasing our tests by adding borderline cases. Something interesting in this testing method is that it facilitates our thinking about cases. For example, it is clear that, mathematically speaking, a 200% discount on 100 would be -100. But, ... is this a valid case in our context? Should our Discount procedure return a negative value, or it should somehow point out an error?

When we think of test cases, we come up with questions that help us in improving the definition of requirements, thus making our system more robust.



## Test Explorer



Once we complete the implementation of our procedure and tests, and we then share –or integrate– our work with the work of the rest of the team, it is important to take into account that, since unit tests are GeneXus objects, they will be part of the knowledge base and we will be sharing them in the same way that we share other objects.

In the Test Explorer we can see all the test objects defined on our KB. Perhaps we implemented some of them, and others were implemented by our fellow team-members. And we may execute all of them or we may choose to execute a selection of them from here, with just one click.

If, in the future, we –or other developers– need to modify the Discount procedure, we will be more at ease in doing so, because if something is broken, that is, if the result obtained for the same input parameters differs from the previous result, then the test will be a Fail, and the problem will be detected at an early stage.

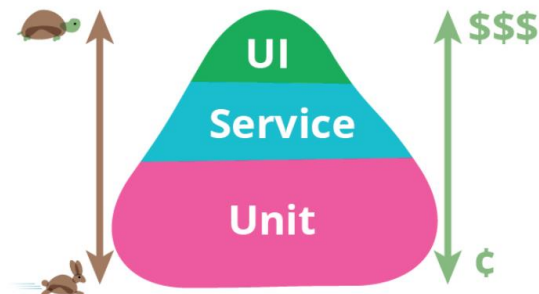
## Unit Tests

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

Even when we have analyzed a very basic example, we may test all sorts of procedures, DataProviders and BCs in the same manner. We may carry out much more complex tests using actual data from our app (either on an actual database or on a simulated one), thus covering the validation of a very significant part of our application.

Perhaps, we have the same work in creating the unit test as in creating a testing procedure that prints out values on the console. However, the interesting thing here is that the verification is carried out by the test itself and not by us. The execution of unit test must be fast because we will want to use them not only to test as we develop the functionality. After introducing changes we will also want to easily repeat all tests defined in the KB to verify that we have not broken any other test when we implemented our functionality. This means that the set of unit tests that we build for each functionality will be automating part of the regression tests.

We also saw that when we work this way, it helps us in developing more robust applications.



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

Even though unit tests will allow us to cover an important part of the app, we are not covering any of the interactions that occur on screen, nor the full flows of the application.

To that end, we have no other option but to execute the app using its interface, that is, browsing through the application's screens as a user would do.

For automating tests at the interface level, we have the UI Test object – User-Interface-Test.

These tests are more complex and their implementation and execution take longer. Therefore the testing pyramid reminds us that most automated tests in our app should be unit test, because they are faster and less costly. Then, at the service level, we save the Interface tests only for the flows that prove critical in our application.

To learn more about the UI Test, go to <https://wiki.genexus.com/commwiki/servlet/wiki?38353,UI+Test+Automation>

*GeneXus*<sup>™</sup>

[training.genexus.com](http://training.genexus.com)  
[wiki.genexus.com](http://wiki.genexus.com)