

# Teste de aplicações em GeneXus

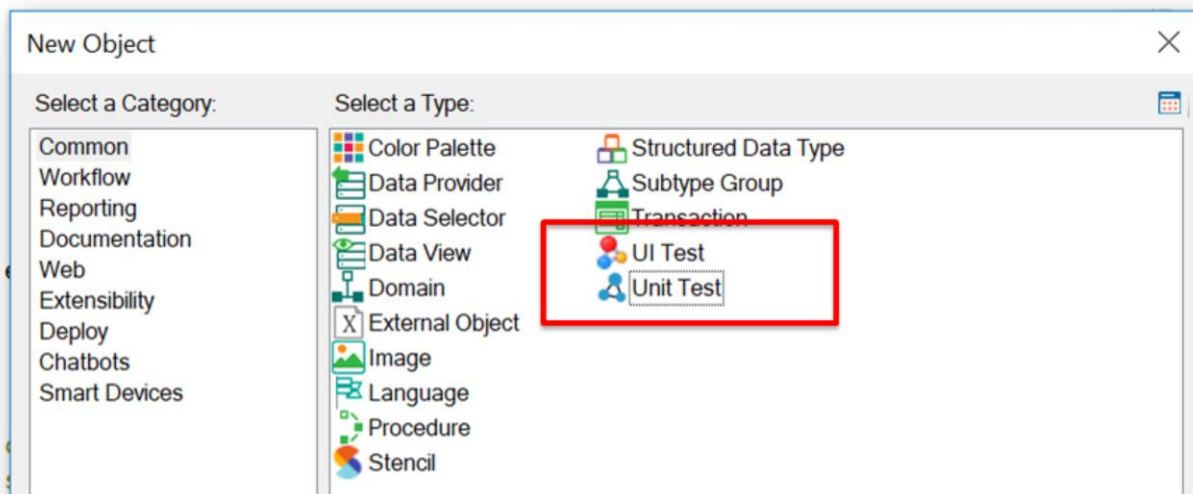
Teste Unitário. Introdução.

*GeneXus 16*

Quando desenvolvemos uma nova funcionalidade em nossa aplicação, é necessário testar se o que desenvolvemos funciona de acordo com o esperado, mas também é importante testar novamente a aplicação inteira após essa alteração, para garantir que o que já tínhamos funcionando continua se comportando corretamente.

À medida que a aplicação cresce, este tipo de tarefa pode se tornar mais tediosa, já que é cada vez mais o que é necessário testar novamente, e também mais caro, já que cada vez consomem mais tempo.

## UITest & Unit Test

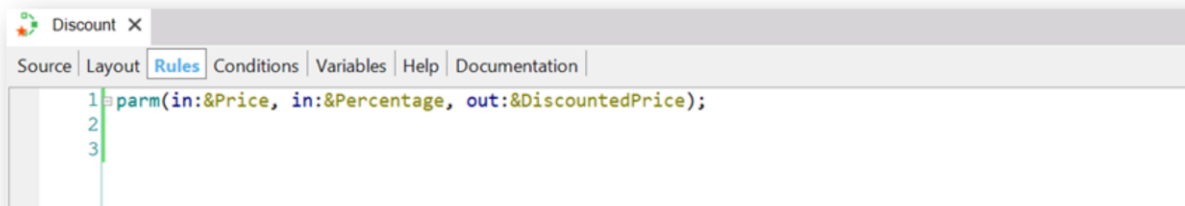


GeneXus nos ajuda nesta frente, dando-nos funcionalidades para poder criar e executar testes automáticos, tanto unitários quanto de interface, de forma a ser possível reduzir parte do trabalho manual de verificação.

Os testes Unitários nos permitem testar uma parte da aplicação isoladamente, isto se aplica a testes em procedimentos, DataProviders e BusinessComponents. Em suma, os componentes em que deveria residir a lógica de negócios de nossa aplicação.

O teste de interface nos permite criar testes simulando as ações de um usuário no browser, de maneira a poder testar fluxos completos da aplicação.

## Unit Tests - Exemplo



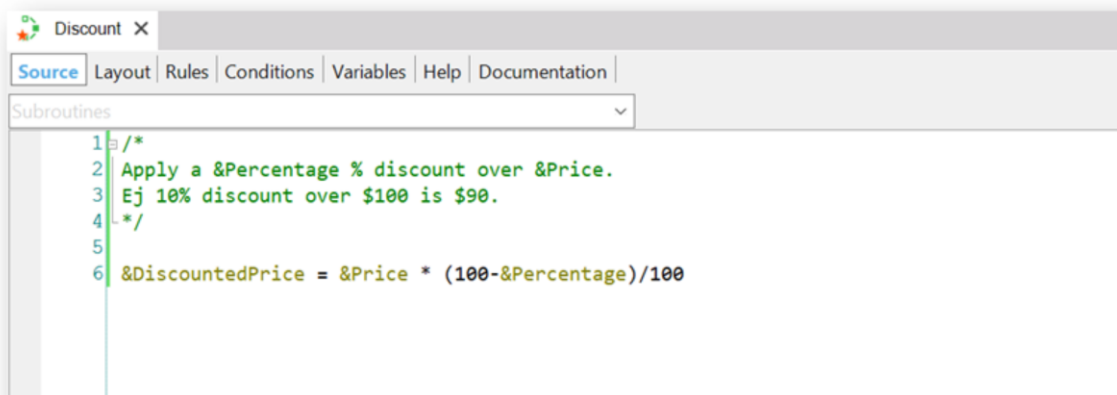
Vamos nos concentrar nos testes unitários, e vamos fazê-lo com um exemplo muito básico.

Temos um procedimento que calcula um **Desconto** – basicamente aplica uma porcentagem de desconto a um preço – que será utilizado para calcular promoções na agência de viagens.

O procedimento tem dois parâmetros de entrada: O Preço a descontar e a porcentagem de desconto. E tem um parâmetro de saída que é o preço com desconto.

O que este procedimento faria, por exemplo, é que, se passarmos um preço de \$100 e um desconto de 10%, nos devolveria o preço com desconto que seria de \$90.

## Unit Tests - Exemplo



The screenshot shows the GeneXus IDE interface with a window titled 'Discount'. The 'Source' tab is active, displaying the following code:

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

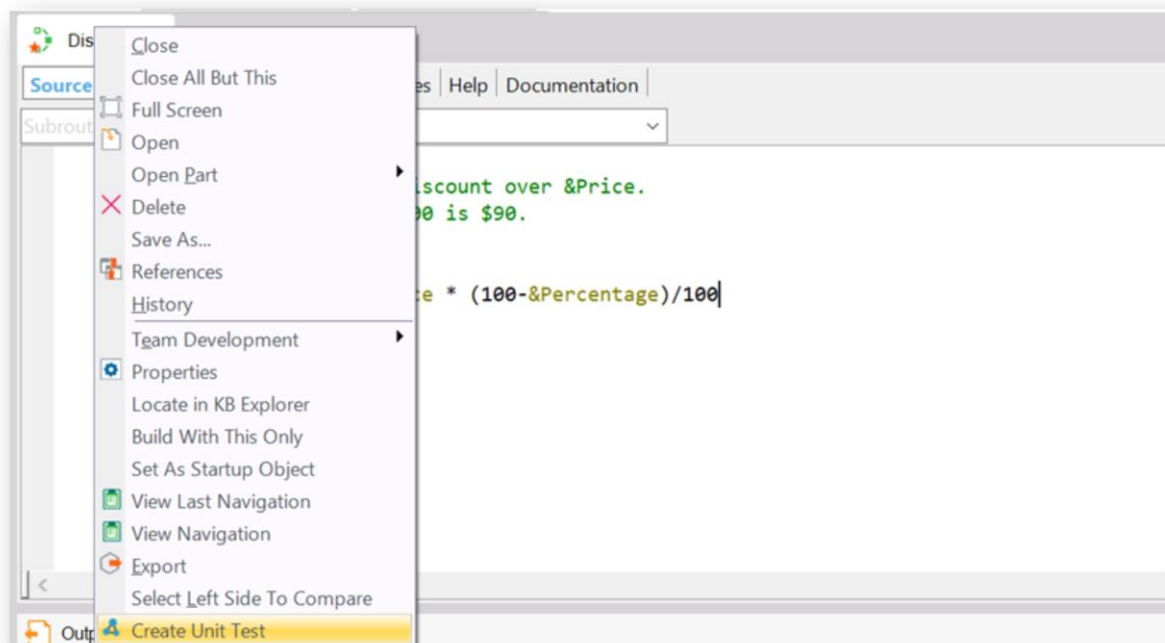
Aqui podemos ver a implementação do procedimento.

Agora, como faríamos para testar que este procedimento funciona da maneira esperada? Ou seja, que dado um preço e uma porcentagem me devolva o preço ajustado correto?

Poderíamos fazer uma tela que nos permita inserir os valores do preço e o desconto, colocamos um botão que chama o procedimento e nos mostra o valor do preço com desconto na tela. Desta forma, poderíamos testar de forma interativa que o procedimento se comporta como queremos. Poderíamos também, se não, programar um procedimento, onde chamamos este proc Discount com diferentes parâmetros e vamos imprimindo no console o resultado, de forma também a poder verificar ali que o resultado é o esperado.

Agora bem, essa forma de testar é cara – já que tem que criar uma tela, inserir os valores manualmente e avaliar o resultado. E isto faríamos na máquina onde estamos desenvolvendo, mas então, se estivermos satisfeitos por se comportar bem e integrarmos nosso trabalho ao do restante da equipe, teremos que voltar a testar para ver que ficou bem integrada.

A ideia dos testes unitários é poder automatizar este trabalho para evitar o retrabalho e simplificar a tarefa



Vamos então criar um teste unitário sobre nosso procedimento e, para isso, daremos um clique com o botão direito do mouse sobre o objeto e escolheremos a opção "Create Unit Test"

**Objects**

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

Ao criar o teste unitário, são criados três objetos, os veremos em detalhes.

## &lt;Object&gt;UnitTestSDT

The screenshot displays the GeneXus IDE interface. At the top, there are tabs for 'Discount', 'Navigation View', 'DiscountUnitTestSDT', 'DiscountUnitTestData', and 'DiscountUnitTest'. Below the tabs, the 'Structure' view shows a tree of objects. The 'DiscountUnitTestSDT' object is expanded, revealing its properties: 'Price' (Type: Price), 'Percentage' (Type: Percentage), 'ExpectedDiscountedPrice' (Type: Price), and 'ErrMsgDiscountedPrice' (Type: VarChar(100)). The 'Description' column for 'ErrMsgDiscountedPrice' contains the text 'Error message to show in case assertion...'. Below the structure view, a 'Rules' tab is open, showing a snippet of code: `1 parm(in:&Price, in:&Percentage, out:&DiscountedPrice);`, `2`, and `3`.

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"/>
• ErrMsgDiscountedPrice	VarChar(100)	Error message to show in case assertion...	<input type="checkbox"/>

```

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

```

Dissemos que, se fossemos testar isto manualmente, talvez tivéssemos criado uma tela onde colocaríamos as duas variáveis de entrada do procedimento e alguma forma de ver o resultado da variável de saída.

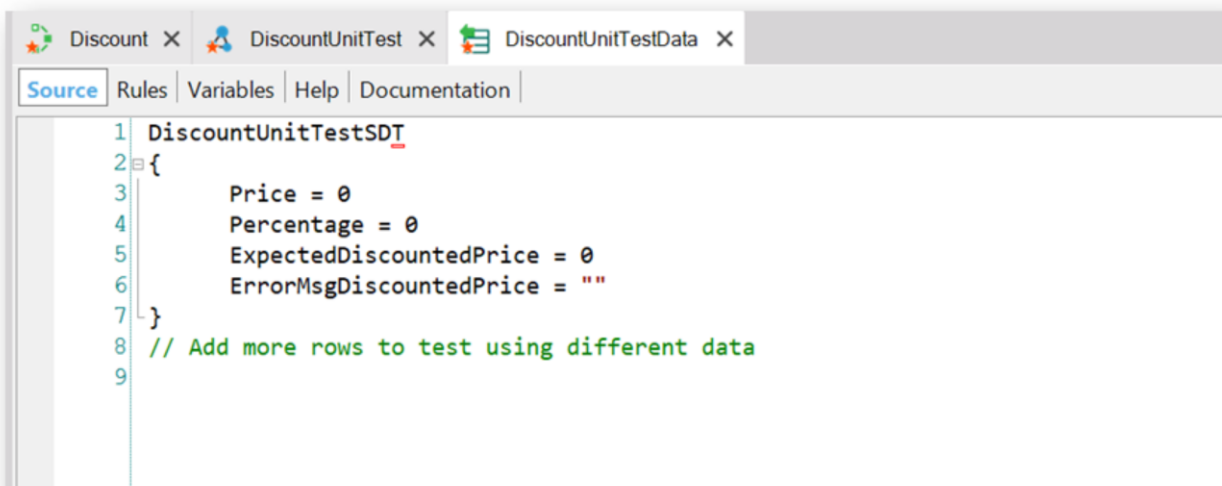
Em suma, alguma maneira de validar que, dados os valores de entrada, nos retorne um resultado que esperamos.

Vamos ver o DiscountUnitTestSDT, que é um dos objetos criados automaticamente ao criar o teste unitário do procedimento Discount. Este SDT define a estrutura de um caso de teste específico para o objeto que estamos testando.

Vemos que – como nós faríamos – define as duas variáveis de entrada – com o mesmo nome que os parâmetros de procedimento – e também define uma variável ExpectedDiscountedPrice onde poderemos definir o valor do resultado esperado.

Em resumo, podemos dizer que, para um preço de 100 e uma porcentagem de desconto de 10, esperamos um resultado de 90.

Também podemos atribuir uma mensagem que queremos que seja exibida, caso o resultado seja diferente de 90.



The screenshot shows the GeneXus IDE interface with three tabs: Discount, DiscountUnitTest, and DiscountUnitTestData. The DiscountUnitTestData tab is active, displaying the source code for the DiscountUnitTestDataSDT. The code is as follows:

```
1 DiscountUnitTestDataSDT
2 {
3     Price = 0
4     Percentage = 0
5     ExpectedDiscountedPrice = 0
6     ErrorMsgDiscountedPrice = ""
7 }
8 // Add more rows to test using different data
9
```

Agora que vimos a estrutura do caso de teste para o procedimento Discount, veremos como definimos os conjuntos de dados.

Faremos isto no DataProvider que também foi definido de forma Automática.

Aqui vemos um grupo definido, com os elementos do SDT onde podemos instanciar os valores ....



The screenshot shows the GeneXus IDE with a code editor on the left and a configuration window on the right.

**Code Editor:**

```

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

```

**Configuration Window (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
<b>Output</b>	
Infer Structure	No
Output	DiscountUnitTestSDT
Collection	True
Collection Name	DiscountUnitTestSDTCollection
<b>Network</b>	
Connectivity Support	Inherit
<b>Warning messages</b>	
Disabled warnings	spc0096 spc0107 spc0142
<b>Miscellaneous</b>	
Generate Object	True

... E por exemplo, atribuir os que vínhamos falando

Também atribuiremos a mensagem que queremos ver, caso o resultado obtido seja diferente de 90.

Este data-Provider nos devolve uma coleção de dados de teste, portanto, nos permitirá definir de forma simples vários testes, ou vários conjuntos de dados, que queremos executar, mas por enquanto temos o suficiente para poder executar nosso primeiro teste.

```

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 executá-lo, veremos o terceiro objeto que foi criado automaticamente, que é o unit-test propriamente dito.

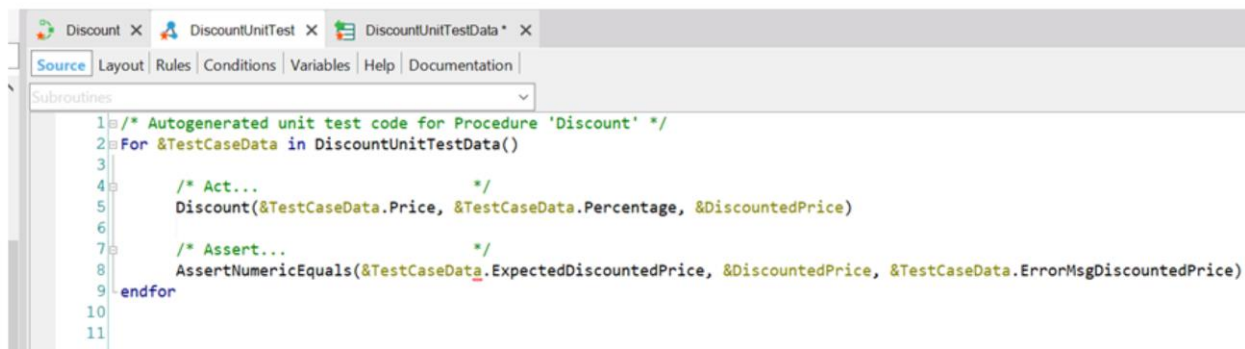
O objeto DiscountUnitTest é o que irá percorrer a coleção de casos de teste, e para cada um deles Invocará nosso procedimento e validará se o resultado obtido é igual ao resultado esperado.

Este objeto é um procedimento GeneXus e se programa como tal, para o qual veremos uma notação que nos é muito familiar,

Ou seja, PARA CADA caso de teste na coleção de Tests que definimos no data Provider, é feita a call para o procedimento que estamos testando com os parâmetros de entrada definidos no caso de teste e uma variável como valor de saída.

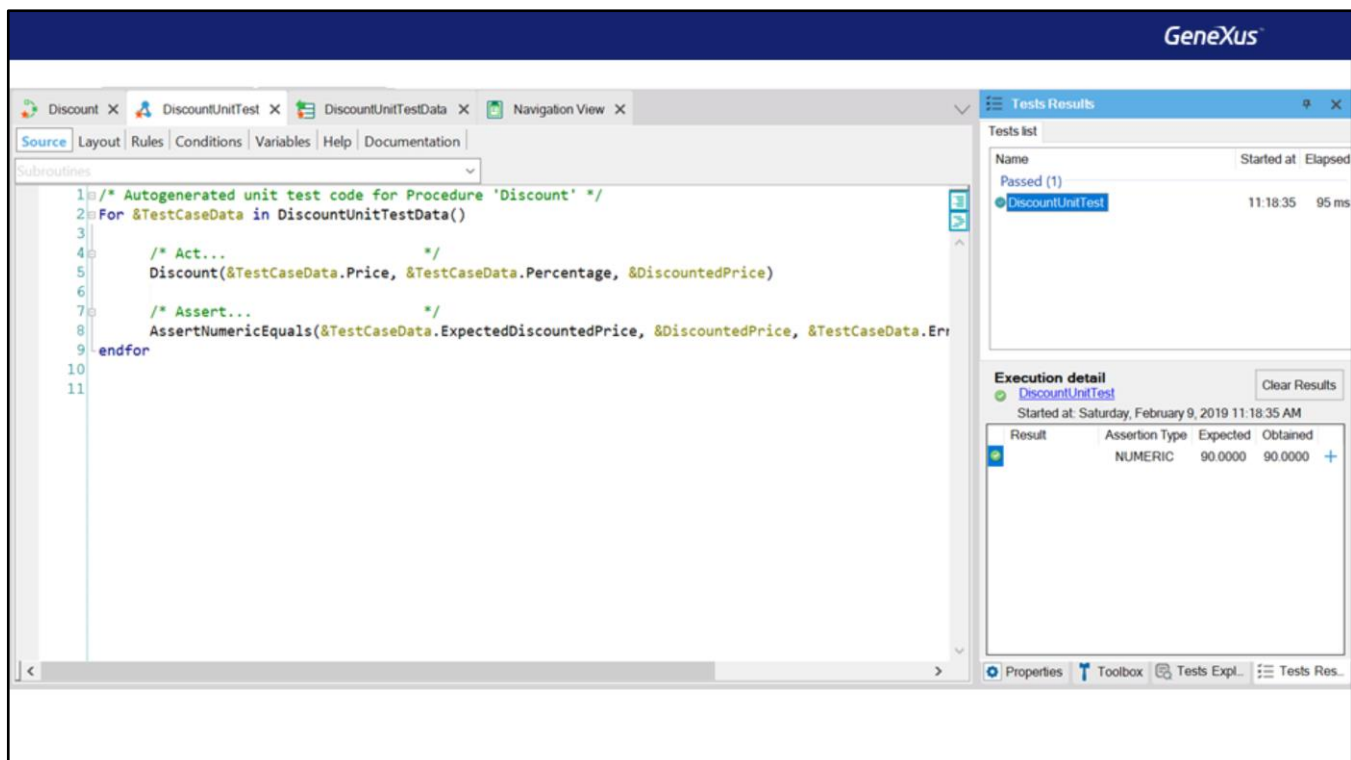
O que é novo no teste de unitário é o comando ASSERT. Que basicamente compara um resultado esperado – definido como parte do caso de teste – com o resultado obtido. Se o resultado esperado e o resultado obtido forem iguais, o teste é bem-sucedido e se diz que PASSA ou é um PASS, e se houver alguma diferença, o teste falha ou é um FAIL e é relatado que houve um erro mostrando uma mensagem associada.

Aqui estamos utilizando a função AssertNumericEquals para validar o resultado, já que o preço com desconto é um valor numérico, mas há também a possibilidade de utilizar AssertBoolEquals para comparar valores booleanos ou AssertStringEquals que nos permite comparar texto e, portanto, qualquer tipo de dados mais complexo



```
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
```

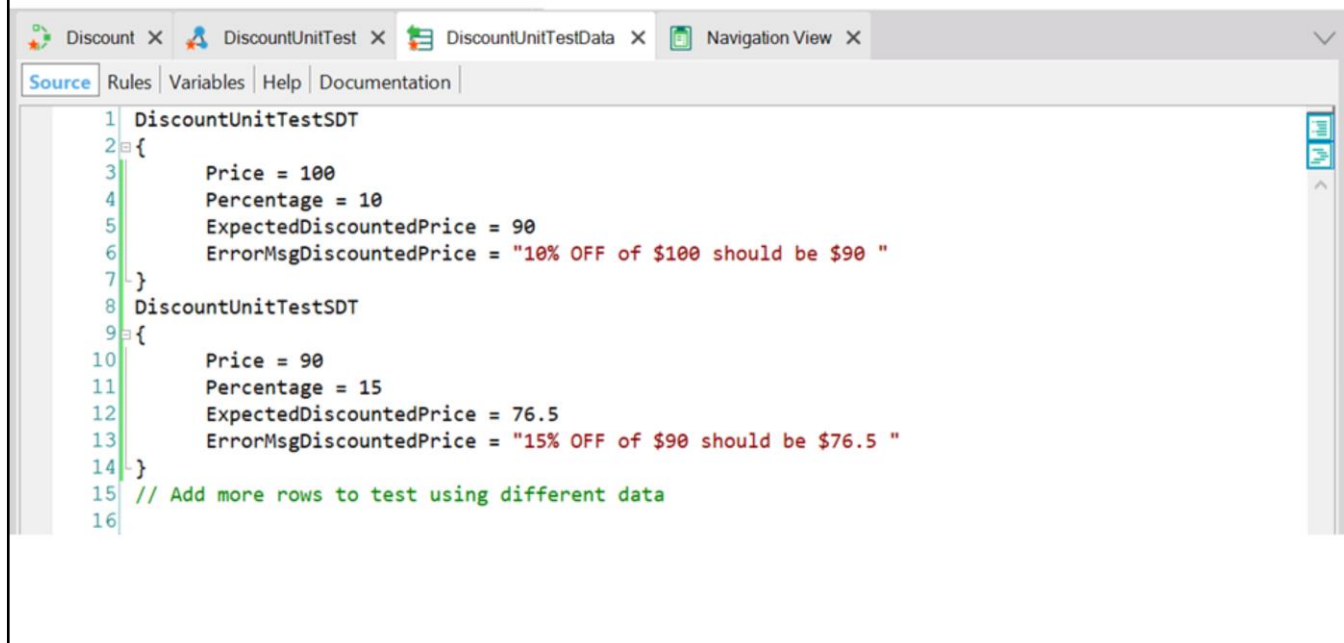
Agora que vimos os três objetos que foram criados automaticamente ao criar nosso teste unitário, que carregamos os dados para o nosso primeiro caso de teste, vamos executar o teste clicando com o botão direito e escolhendo a opção [Run This Test]



Uma vez que o teste conclui a execução, veremos a nova janela - chamada TEST-RESULTS - onde podemos ver que nosso teste (DiscountUnitTest) foi executado e que o resultado foi bem-sucedido, pois está marcado em verde.

Também nos dá informações sobre o tempo de execução do teste

Abaixo - no setor Execution Detail - veremos uma linha para cada Assert que haja em nosso teste. Para cada um podemos ver o resultado esperado, o resultado obtido e também a marca verde ou vermelha de acordo se o Assert falhou ou passou. Caso o Assert falhe, veremos a mensagem que definimos em nosso caso de teste.



The screenshot shows the GeneXus IDE interface with four tabs: Discount, DiscountUnitTest, DiscountUnitTestData, and Navigation View. The 'DiscountUnitTest' tab is active, displaying a test script in the 'Source' view. The script defines two test cases for a discount calculation. The first case (lines 2-7) sets Price to 100, Percentage to 10, ExpectedDiscountedPrice to 90, and ErrorMessageDiscountedPrice to '10% OFF of \$100 should be \$90 '. The second case (lines 9-14) sets Price to 90, Percentage to 15, ExpectedDiscountedPrice to 76.5, and ErrorMessageDiscountedPrice to '15% OFF of \$90 should be \$76.5 '. A comment on line 15 suggests adding more rows for testing with different data.

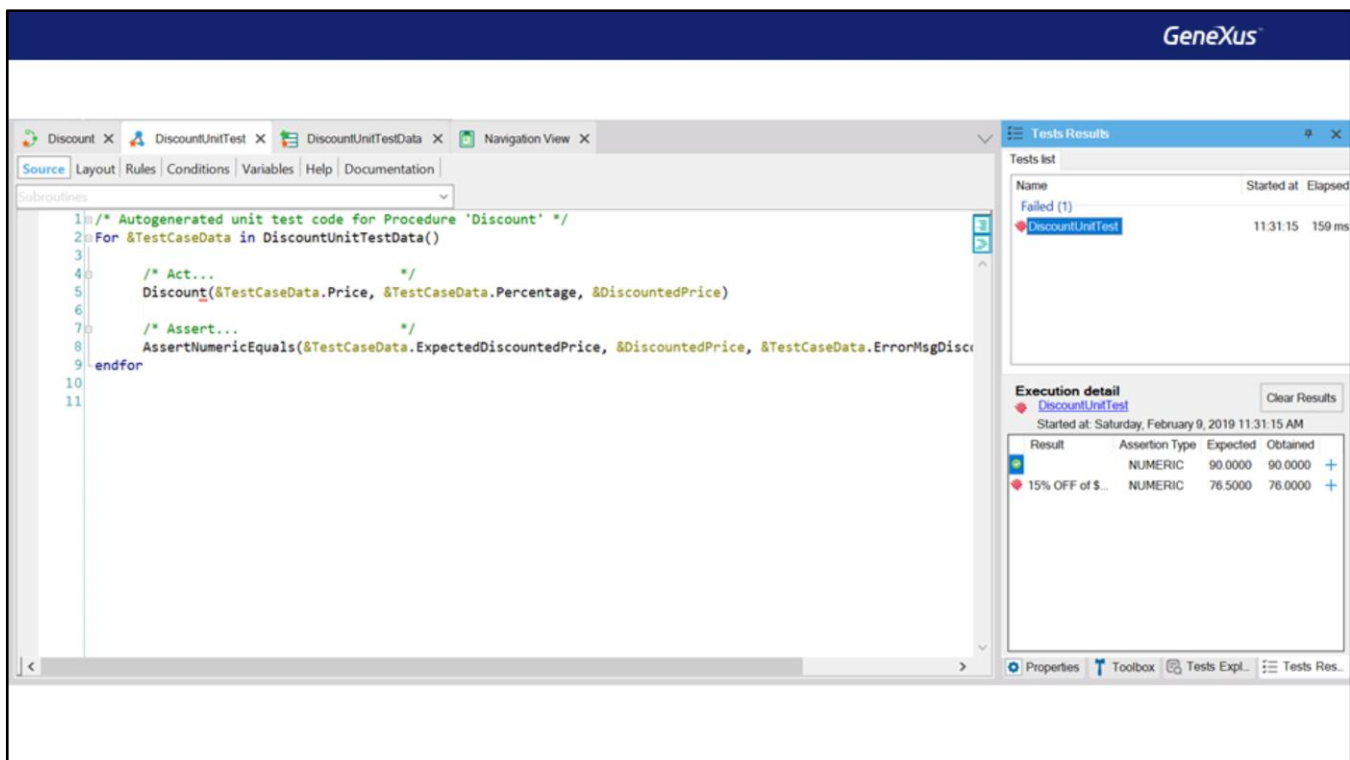
```
1 DiscountUnitTestSDT
2 {
3     Price = 100
4     Percentage = 10
5     ExpectedDiscountedPrice = 90
6     ErrorMessageDiscountedPrice = "10% OFF of $100 should be $90 "
7 }
8 DiscountUnitTestSDT
9 {
10    Price = 90
11    Percentage = 15
12    ExpectedDiscountedPrice = 76.5
13    ErrorMessageDiscountedPrice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

Agora que fizemos nosso primeiro teste bem-sucedido e vimos que é simples definir um caso de teste, definiremos outros casos em nosso DataProvider

A seleção dos dados a serem testados é uma tarefa importante, e uma boa oportunidade para colaborar com o tester da equipe, a fim de definir os testes que nos dão uma melhor cobertura.

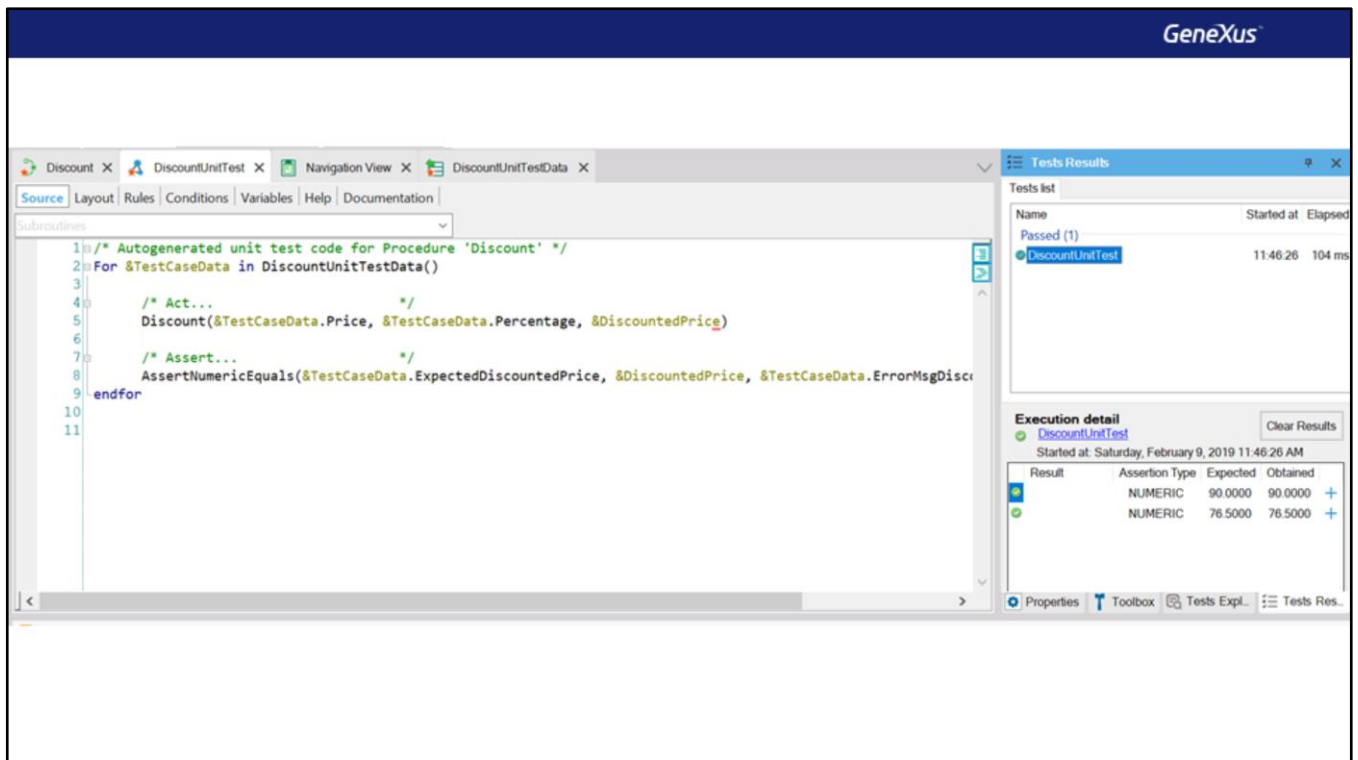
Neste caso, escolhemos um caso qualquer e simples (aplicar um desconto de 10% a um preço de \$100) e agora vamos adicionar um teste que valide que os decimais são corretamente tratados, por isso escolhemos números que nos dão um resultado com decimais.

Então, adicionamos outro grupo e agora dizemos que, dado um preço de 90 e uma porcentagem de desconto de 15, o preço esperado é de 76,5. E novamente executamos nosso teste



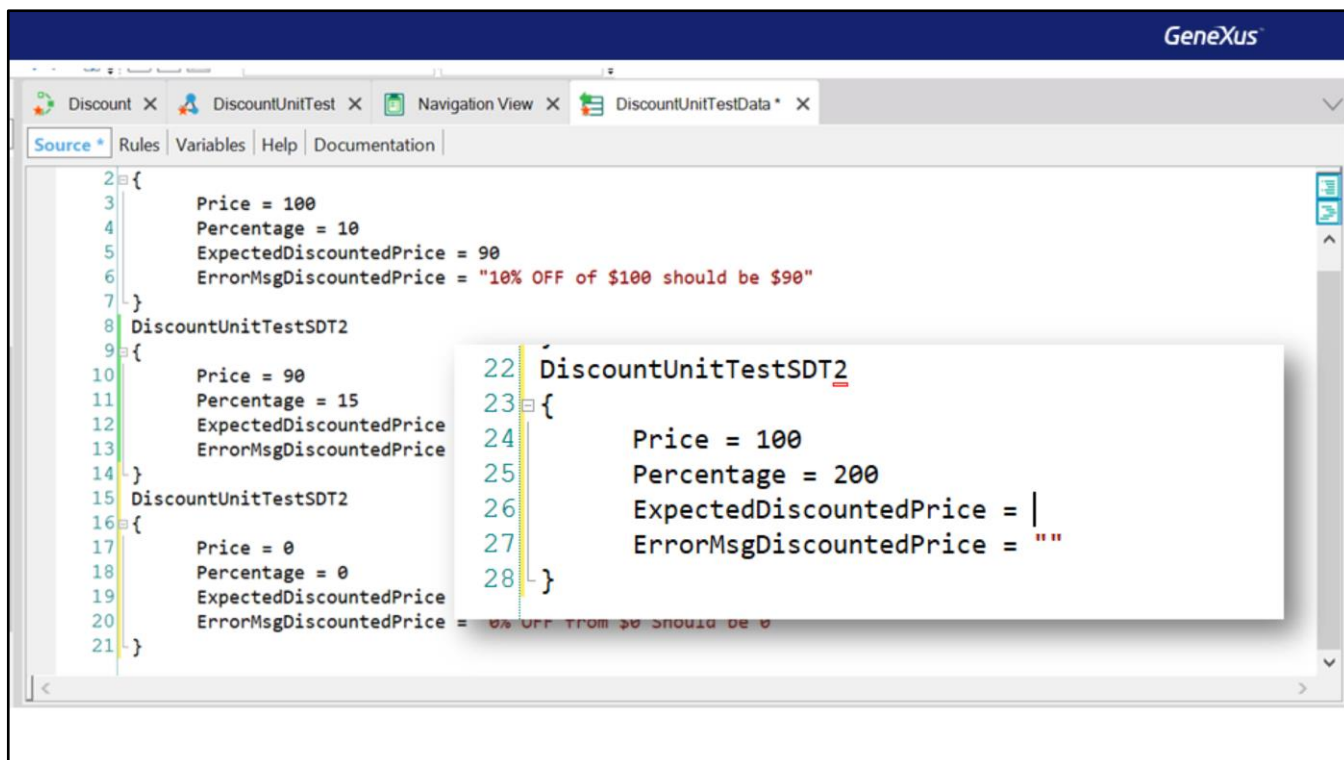
Veremos que agora vamos ter 2 Assert, um para cada caso de teste que estamos executando. Agora vemos que para nossa surpresa que o teste não foi bem-sucedido, como dissemos que temos 2 Assert, o primeiro é referente ao primeiro caso de teste que obteve sucesso e o segundo corresponde ao 2º caso de teste em que o resultado obtido foi 76 em vez de 76,5, ou seja, nosso procedimento não está considerando decimais.

Isto nos mostra que há um erro no procedimento e que a variável onde o preço com desconto é calculado provavelmente está definida incorretamente como um número inteiro em vez de ter decimais. Aqui também vemos que, no caso de o teste falhar, ou seja, que o Assert não dê um resultado bem-sucedido, vemos a mensagem que definimos no caso de erro. Só vemos isso em caso em que o Assert falhe, se o teste for bem-sucedido, a mensagem não é mostrada.



Sabendo que temos um problema na definição da variável de saída do nosso procedimento, vamos editá-lo, e vemos que a variável na verdade não foi definida. Agora, a definimos corretamente com base no domínio Price. Depois de corrigir a variável, executamos o teste novamente e veremos que agora ele é bem-sucedido.

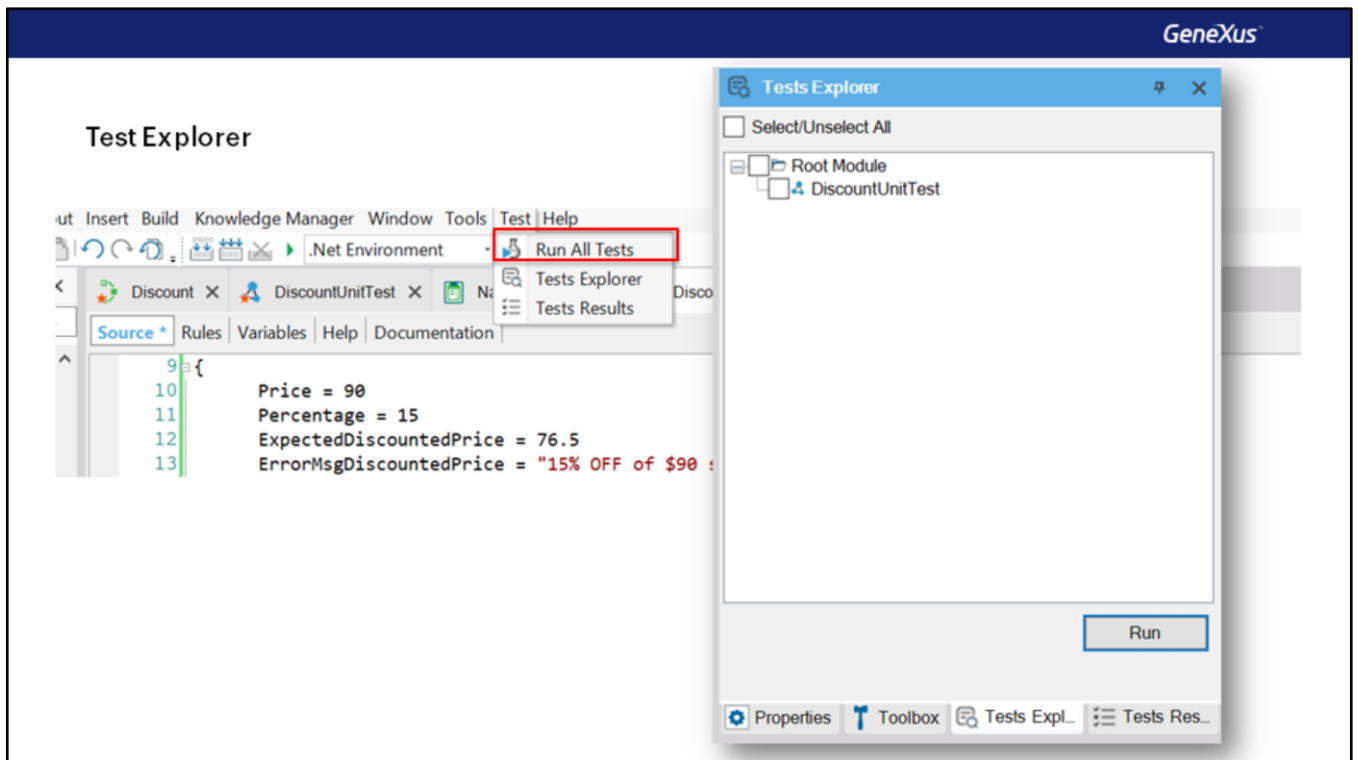
Para este exemplo eu trapaceei um pouco, porque quando definimos o teste unitário, os tipos de dados dos elementos do SDT são definidos com o mesmo tipo de dados que os do procedimento a ser testado, portanto se não tivesse trapaceado, o resultado do Assert teria sido um PASS já que ambos os valores teriam sido inteiros e o resultado esperado também teria sido 76. Embora isso nos tenha dado uma pista de que houve um problema, quando se olha para o resultado esperado e o resultado obtido, não é a ideia de que se descubra problemas analisando os resultados de um teste em verde. Mas eu queria mostrar um exemplo simples e, na verdade, este tipo de coisa poderia acontecer na vida real se alguém tivesse alterado equivocadamente a variável de saída do procedimento Discount DEPOIS que o teste já tivesse sido criado, como foi o caso aqui.



Podemos continuar aumentando nossos testes adicionando casos de borda... Algo que é bom nesta forma de testar é que nos facilita pensar em casos... por exemplo... está claro que matematicamente falando, 200% de desconto sobre 100 seria -100, agora, bem... é um caso válido em nosso contexto? Nosso procedimento Discount deveria retornar um valor negativo ou de alguma maneira indicar um erro?

Ao pensar em casos de teste, surgem questões que nos ajudam a melhorar a definição de requisitos e a tornar nosso sistema mais robusto.





Quando terminamos de implementar nosso procedimento e nossos testes e compartilhamos – ou integramos – nosso trabalho com o restante da equipe, é importante notar que os testes unitários, como são objetos GeneXus, são parte da base de conhecimento e os compartilharemos da mesma forma que compartilhamos outros objetos.

No Test Explorer podemos ver todos os objetos de testes definidos em nossa KB, talvez alguns implementados por nós ou outros por nossos colegas e podemos executar todos ou uma seleção deles a partir daqui com apenas um clique.

Se amanhã, nós ou outro desenvolvedor, tivermos que modificar o procedimento de Discount – irá fazê-lo mais tranquilo porque se danifica algo – ou seja, se o resultado obtido para os mesmos parâmetros de entrada for diferente do anterior, o teste falhará e o problema será detectado cedo.

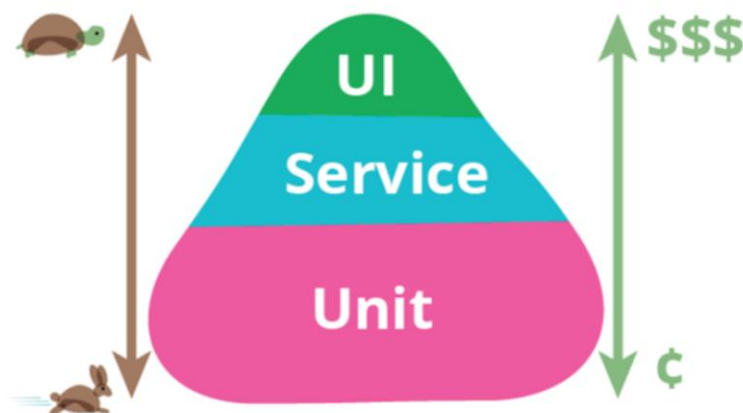
## Unit Tests

- Rápidos
- Podem ser Repetidos (Regressão)
- Melhoram a Capacidade de Teste da aplicação

Embora tenhamos visto exemplos muito básicos, podemos testar todos os tipos de procedimentos, DataProviders e BC desta maneira. Podemos realizar testes muito mais complexos – utilizando dados reais de nossa aplicação (seja em uma base de dados real ou simulada) e cobrindo a validação de uma parte muito importante de nossa aplicação.

Talvez, criar o teste unitário nos dê o mesmo trabalho que criar um procedimento de teste que imprima valores no console, mas tem a vantagem de que a verificação seja feita pelo próprio teste e não por nós. Os testes unitários devem ser rápidos para executar – porque vamos querer usá-los não apenas para testar enquanto desenvolvemos a funcionalidade, mas também, vamos querer repetir todos os testes que estejam definidos na KB facilmente, depois de fazer mudanças, para ver que não danificamos nenhum outro teste ao implementar nossa funcionalidade. Ou seja, o conjunto de testes unitários que vamos construindo para cada funcionalidade, automatiza parte dos testes de regressão.

Também vimos que, ao trabalhar desta maneira, nos ajuda a desenvolver uma aplicação mais robusta.



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

Embora, com os testes unitários, cobriremos uma parte importante da aplicação, não estamos cobrindo nenhuma das interações que ocorrem na tela ou os fluxos completos da aplicação.

Para isto, não temos escolha a não ser executar a aplicação utilizando a interface da mesma, isto é, navegando pelas telas da aplicação como um usuário faria.

Para automatizar os testes no nível da Interface é que temos o objeto UITest ou User-Interface-Test

Estes testes são mais complexos e levam mais tempo para serem implementados e executados, por isso, a pirâmide de testes nos lembra que a maioria dos testes automatizados de nossa aplicação deve ser unitária – já que são mais rápidos e menos caros – depois no nível de serviço e reservar os testes de Interface apenas para aqueles fluxos que são críticos em nossa aplicação.

Para saber mais sobre o Test de UI, acesse  
<https://wiki.genexus.com/commwiki/servlet/wiki?38353,UI+Test+Automation>,



Videos

[training.genexus.com](https://training.genexus.com)

Documentation

[wiki.genexus.com](https://wiki.genexus.com)

Certifications

[training.genexus.com/certifications](https://training.genexus.com/certifications)

Vimos então as facilidades da versão 16 para criar e executar testes automáticos, o que é um elemento chave em um bom processo de engenharia.