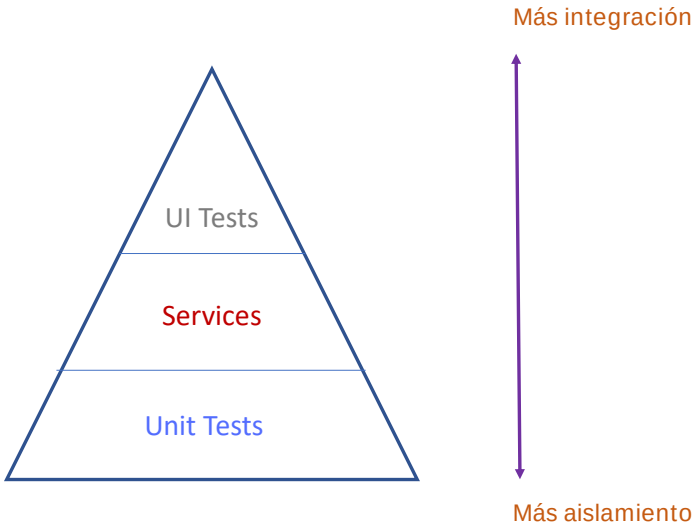
Testes unitários

Introdução

GeneXus™

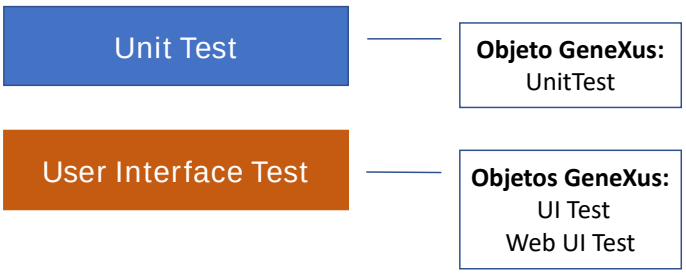
Ao longo do desenvolvimento de nossa aplicação para a Agência de viagens, mencionamos a importância de testar de forma isolada as novas funcionalidades que desenvolvemos e, em seguida, testar a aplicação inteira para garantir que o comportamento esteja correto.

Testes automáticos



À medida que a aplicação cresce, estes tipos de tarefas podem se tornar mais tediosos, e é então que GeneXus nos ajuda dando-nos funcionalidades para poder criar e executar testes automáticos, de modo a poder reduzir parte do trabalho manual de verificação.

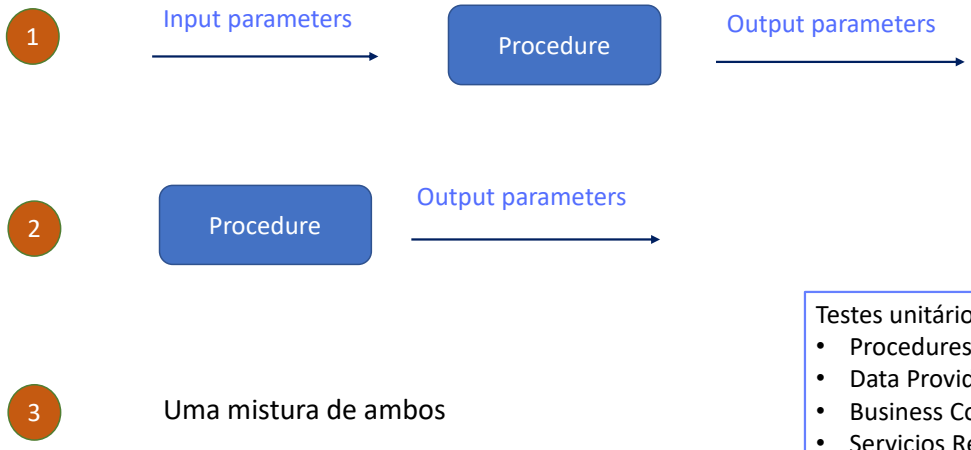
Testes automáticos



- Os **testes Unitários** nos permitem testar uma parte da aplicação de forma isolada. No GeneXus, os testes unitários sem interface se aplicam a testes em procedimentos, DataProviders e BusinessComponents. Resumindo, sobre aqueles componentes onde deveria residir a lógica de negócios de nossa aplicação, e para isso existe o objeto Unit Test.

- O **teste de Interface** –permite criar testes simulando as ações de um usuário no browser, de modo a poder testar fluxos completos da aplicação. Para isso existem os objetos UITest para interface mobile e Web UITest para interface web.

Quando é interessante testar um procedimento GeneXus? (lógica GeneXus)



Então, quando interessa testar um procedimento GeneXus?

- Quando dados alguns parâmetros de entrada, é desejado testar a saída do Procedimento comparando os resultados esperados com os resultados obtidos do processo (sejam variáveis de saída, arquivos ou registros da base de dados). Para isto são utilizadas asserções, ou seja, afirmações ou declarações incluídas no procedimento.
- Quando não há parâmetros de entrada, mas a partir de parâmetros ou configurações da base de dados, espera-se que o Procedimento retorne alguns resultados (por exemplo, variáveis ou registros da base de dados).
- E quando há uma mistura de ambos os cenários.

Isto significa que qualquer Procedimento GeneXus pode ser testado para verificar se funciona de acordo com as especificações definidas. Mas além dos Procedimentos, também podem ser definidos testes unitários associados a Data Providers, Business Components e Servicios Rest.

Benefícios:

Detecte erros no código antecipadamente.

Dê feedback imediato aos desenvolvedores.

São rápidos e compartilhados em toda a Base de Conhecimento se estiver sendo utilizado o GeneXus Server.

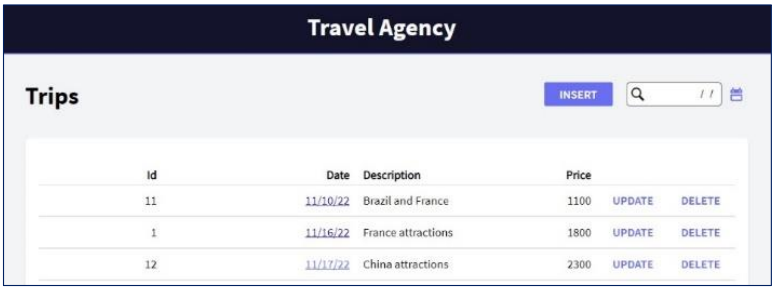
Os desenvolvedores podem executar seus testes desde o próprio IDE GeneXus sem a necessidade de outras ferramentas.

Os principais benefícios de realizar testes unitários são os seguintes:

- Permitem detectar erros no código de forma precoce.
- Dar feedback imediato aos desenvolvedores.
- São rápidos e compartilhados em toda a Base de Conhecimento se estiver sendo utilizado GeneXus Server.
- Os desenvolvedores podem executar seus testes a partir do próprio IDE de GeneXus sem necessidade de outras ferramentas

Exemplo: UpdateTripPrice

Procedimento que atualiza o preço de uma viagem de acordo com uma porcentagem recebida por parâmetro.



Bem, vamos ver então um exemplo.

A partir da launchpad, executamos o Work with Trip e vemos que temos três viagens registradas com seus custos atuais.

1100, 1800 e 2300

Exemplo: 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
```

Criamos um procedimento, chamado UpdateTripPrice, que calcula o aumento de preço de uma determinada viagem, de acordo com um percentual recebido por parâmetro. Para a Agência de viagens, o preço final de uma viagem não pode ultrapassar o valor de 2500.

Vemos então que o procedimento recebe dois parâmetros de entrada:

A variável &TripId e o percentual de aumento

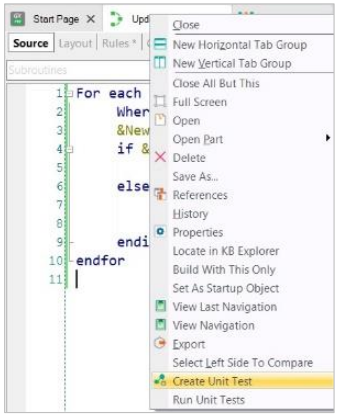
e tem um parâmetro de saída que retorna o valor obtido pelo aumento junto com um comentário que indica se foi realizada a atualização ou se foi ultrapassado o valor máximo indicado.

Então, para o TripId recebido, o procedimento calcula o valor de acordo com o percentual também recebido, e se esse valor for maior que 2500, não atualiza e retorna a mensagem correspondente.

Se o valor for menor ou igual a 2500, então atualiza o valor do atributo TripPrice e também retorna a mensagem correspondente.

Caso não exista uma viagem com o valor de TripId recebido por parâmetro, será retornada uma mensagem indicando que a viagem não está registrada.

Criação do Teste unitário associado



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 testar este procedimento vamos criar o correspondente Unit test.

Para isso, sobre a aba do procedimento, clicamos com o botão direito e escolhemos Create Unit Test.

GeneXus cria então três objetos, que podemos ver sob o novo nó Tests na janela KBExplorer:

O objeto UpdateTripPriceTestSDT,

que define a estrutura de um caso de teste concreto para o objeto que estamos testando.

Vemos que – assim como faríamos – define as duas variáveis de entrada – com o mesmo nome dos parâmetros do procedimento – e define também uma variável ExpectedUpdateResult onde poderemos definir o valor do resultado que esperamos.

Em resumo, poderemos dizer, por exemplo, que para a viagem com identificador TripId=11, com custo atual de 1100, se indicarmos um percentual de aumento de 10% esperamos um resultado de 1210 – com uma mensagem indicando que o preço foi atualizado corretamente.

Também podemos atribuir uma mensagem que queremos que seja exibida caso o resultado seja diferente do indicado.

Criação do Teste unitário associado

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 = ''
}
```

Passemos agora ao **objeto UpdateTripPriceTestData**, também criado automaticamente pelo GeneXus.

Este Data Provider está baseado no SDT que vimos anteriormente e nos permite definir um conjunto de dados. Por padrão, são criados 5 casos de teste, mas podemos modificá-lo.

Temos três viagens registradas, então vamos definir três testes com um percentual de aumento de 10%. Em cada caso indicamos o valor esperado retornado pelo procedimento.

Mas definimos também um quarto conjunto de teste para o TripId com valor igual a 8 que não existe atualmente em nossa base de dados.

Criação do Teste unitário associado

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 um resultado esperado com um resultado obtido

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

Finalmente, **o objeto UpdateTripPriceTest** é aquele que vai percorrer a coleção de casos de teste, e para cada um deles vai invocar nosso procedimento e validar se o resultado obtido é igual ao resultado esperado.

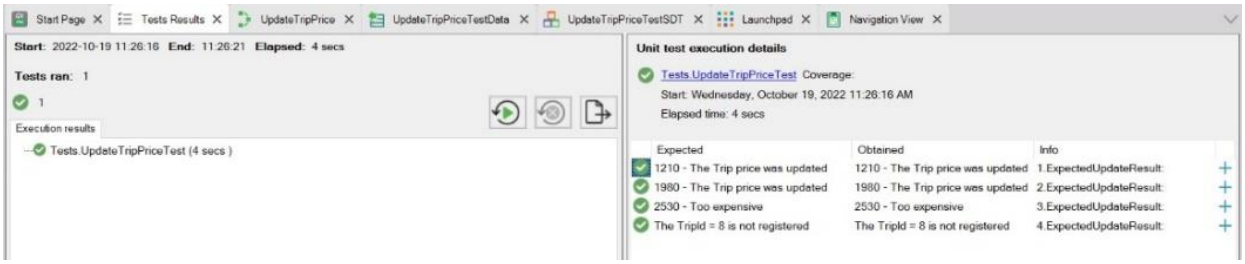
Este objeto é um procedimento GeneXus e é programado como tal, por isso vamos ver uma notação que nos é muito familiar,

Ou seja, PARA CADA caso de teste na coleção de Testes que definimos no data Provider, é feita a chamada 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 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 é dito 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, exibindo uma mensagem associada.

Aqui estamos utilizando a função AssertStringEquals para validar o resultado já que se trata de um texto, mas também existe a possibilidade de utilizar AssertBoolEquals para comparar booleanos ou AssertNumericEquals que nos permite comparar valores numéricos.

Execução do Teste unitário



Para executar, clicamos com o botão direito, Run Test

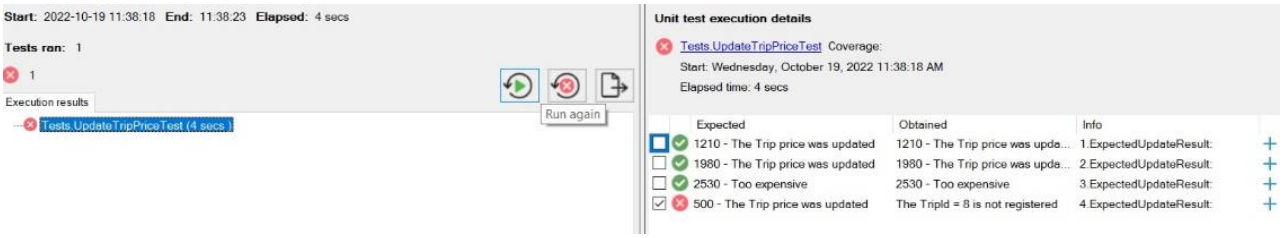
Uma vez que o teste terminar a execução, veremos a nova janela – denominada TESTS-RESULTS – onde verificamos que nosso teste foi executado (UpdateTripPriceTest) e que o resultado foi bem-sucedido, pois está marcado em verde.

Também nos dá informação sobre o tempo de execução do teste
Aqui vemos uma linha para cada Assert que definimos em nosso teste. Para cada um podemos ver o resultado esperado, o resultado obtido e também a marca verde ou vermelha dependendo se o Assert falhou ou passou.

Execução do Teste unitário

O Data Provider é modificado para gerar uma falha.

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



Vamos executar novamente o teste, mas antes vamos retornar os custos ao estado inicial e modificar o Data Provider assumindo que a viagem com identificador TripId = 8 existe em nossa base de dados e atribuímos a ela um custo determinado. A ideia é gerar uma falha neste conjunto de teste, pois nosso procedimento indicará que a viagem não está registrada.

Executamos novamente o teste pressionando este botão e vemos que um conjunto de testes falhou, pois o valor esperado e o valor obtido são diferentes:

A partir daqui, podemos ver a comparação entre os dois resultados:

E a partir daqui podemos novamente executar os conjuntos de teste que falharam. Também podemos exportar o resultado do teste para HTML

Os Testes unitários automatizam parte dos testes de software e nos ajudam a desenvolver aplicativos mais robustos.

Embora tenhamos visto um exemplo simples, como já dissemos, podemos testar todos os tipos de procedimentos, DataProviders e Business Component, Podemos realizar testes muito mais complexos – utilizando dados reais de nossa aplicação (seja em uma base de dados real ou simulada) e cobrir a validação de uma parte muito importante de nossa aplicação.

O conjunto de testes unitários que vamos construindo para cada funcionalidade automatiza parte dos testes de software e nos ajuda a desenvolver aplicações mais robustas.

GeneXus™

training.genexus.com
wiki.genexus.com