

GeneXus での アプリケーションのテスト

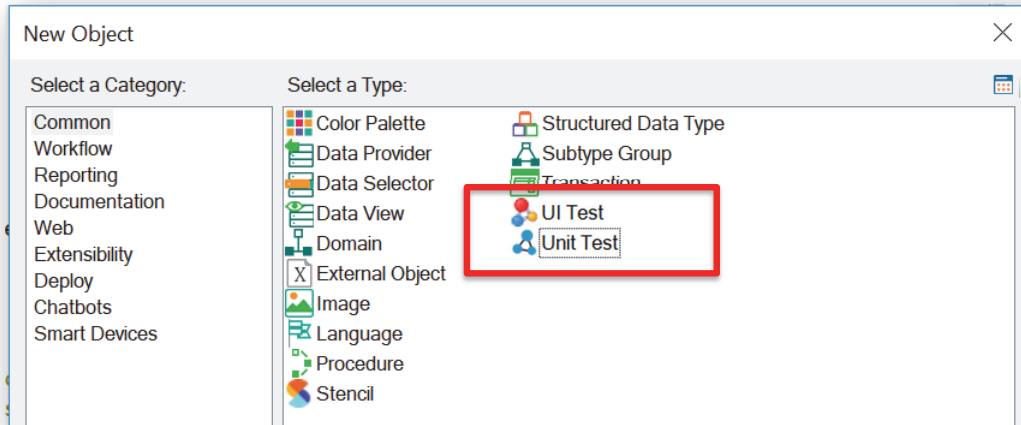
ユニットテスト概要



アプリケーションの新機能を開発するときには、開発した部分をテストし、期待どおりに機能することを確認する必要があります。さらに、アプリケーション全体でテストして、変更前に機能していたものが引き続き正常に動作することを確認する必要もあります。

アプリケーションの規模が大きくなると、何度もテストを行わなければならないため、一連のタスクがどんどん長くなります。また、毎回多くの時間が必要になるため、コストもかさみます。

UI テストとユニットテスト

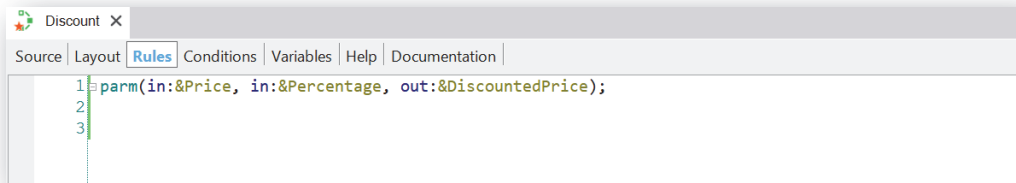


GeneXus では、ユニットテストと UI テストの両方に、自動テストを作成および実行する機能が用意されています。この機能を使用すると、手動による検証作業の負担が軽減されます。

ユニットテストでは、アプリケーションの一部を切り離してテストできます。これは、プロシージャー、データプロバイダー、およびビジネスコンポーネントのテストに利用できます。つまり、アプリケーションのビジネスロジックが存在するすべてのコンポーネントが対象となります。

UI テストでは、ブラウザーでユーザーの操作をシミュレートしてテストを作成し、アプリケーションのフロー全体をテストできます。

ユニットテスト - 例



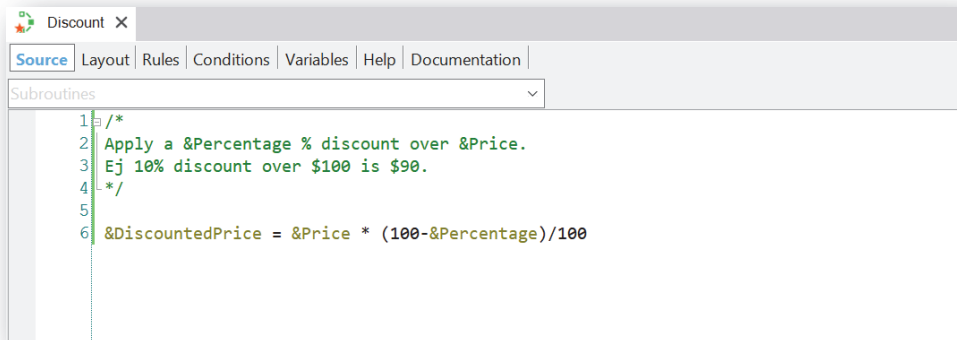
シンプルな例を使用して、ユニットテストについて説明します。

これは**割引**を計算するプロシージャーです。旅行代理店がキャンペーン料金を計算する際に使用します。

このプロシージャーには、入力パラメーターとして割引前の料金と割引率があります。また、出力パラメーターとして割引後の料金があります。

たとえば、このプロシージャーに 100 ドルと 10% の割引率を渡すと、割引後の料金である 90 ドルが返されます。

ユニットテスト - 例



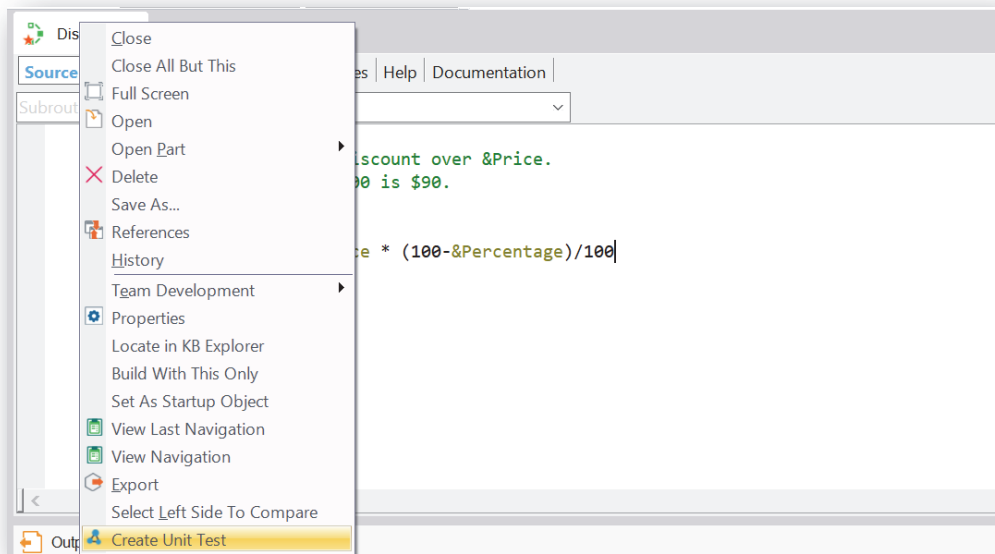
プロシージャーの実装方法を見てみましょう。

このプロシージャーが期待どおりに機能することをテストするには、どうしたらよいでしょうか。つまり、料金と割引率を入力すると、適切な割引後の料金が返されることを確認します。

料金と割引率を入力してボタンをクリックすると、プロシージャーが呼び出され、割引後の料金が表示される画面をビルドします。これにより、プロシージャーが期待どおりに動作することをインタラクティブにテストすることができます。また、異なるパラメーターを持つ `proc Discount` というプロシージャーをプログラミングします。これは、結果が期待した値と同じかどうかを検証し、結果をコンソールに出力するプロシージャーです。

しかし、画面を作成し、値を手動で入力して結果を評価するというテスト方法は、コストがかかります。また、このようなテストは、開発に使用しているコンピューターで行うことになります。動作が良好だった場合、その機能をチームのほかのメンバーと共有してから、アプリケーション本体に適切に統合されたかどうかをテストする必要もあります。

ユニットテストの背景にある考え方は、このようなタスクを自動化して繰り返し作業を排除し、タスクを簡素化することです。



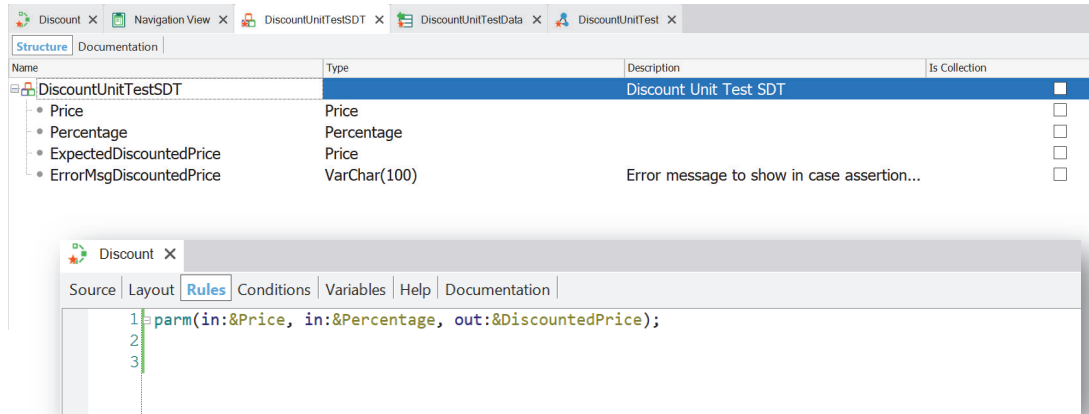
それでは、プロシーチャーのユニットテストを作成してみましょう。オブジェクトを右クリックし、[ユニットテストを作成] を選択します。

オブジェクト

- <オブジェクト名>UnitTest
- <オブジェクト名>UnitTestSDT
- <オブジェクト名>UnitTestData

ユニットテストを作成すると、3 つのオブジェクトが作成されます。詳しく見てみましょう。

<オブジェクト名>UnitTestSDT



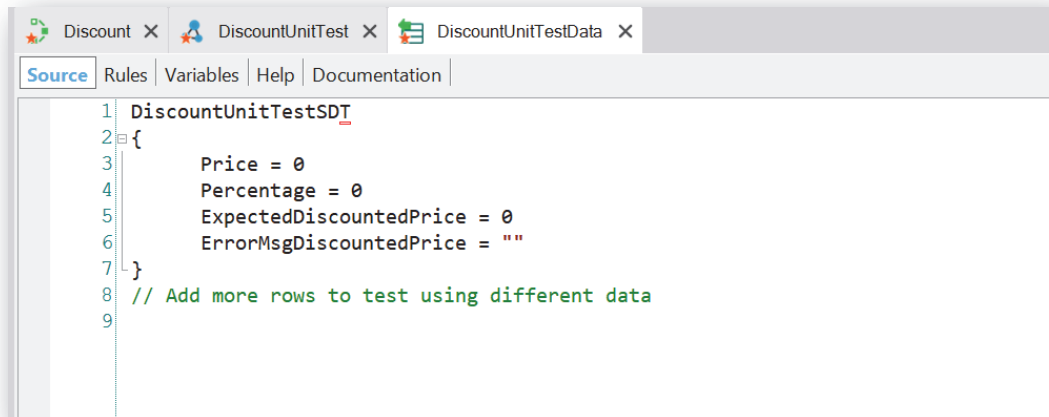
先ほど説明したように、手動でテストする場合、プロシージャーに 2 つの入力変数を渡し、出力変数の結果を表示するための、何らかの方法を備えた画面を作成する必要があります。つまり、値を入力したら、期待される結果が返されることを検証するための画面です。

スライドに表示されている DiscountUnitTestSDT は、Discount プロシージャーのユニットテストを作成したときに自動的に作成されたオブジェクトのうちの 1 つです。この SDT は、テストするオブジェクトの特定のテストケースの構造を定義しています。

プロシージャーのパラメーターと同じ名前の入力変数が 2 つ定義されており、期待する結果の値について定義する ExpectedDiscountedPrice という変数も定義されています。

料金が 100 で割引率が 10 の場合、結果は 90 になることが予測されます。

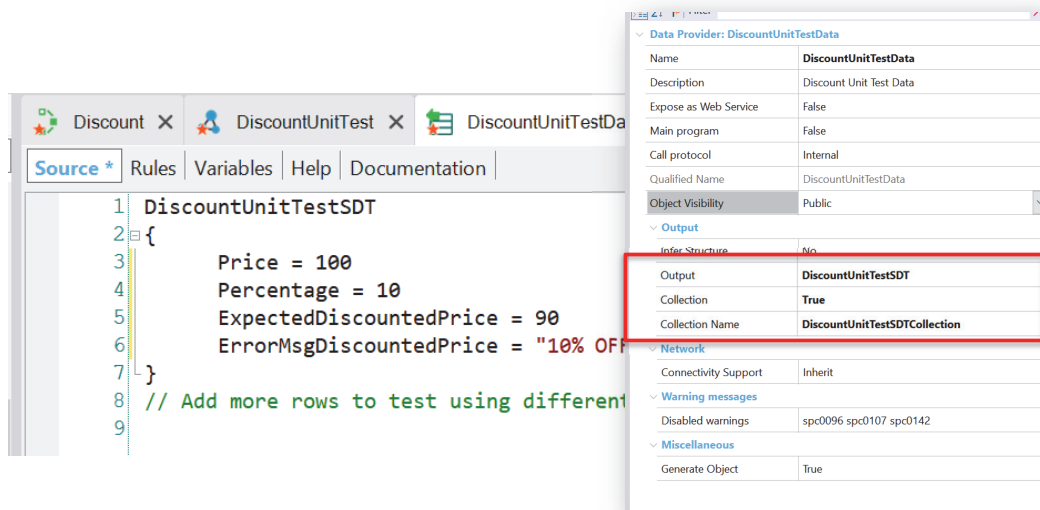
また、結果が 90 ではなかった場合に表示するメッセージを指定することもできます。



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

```
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
```

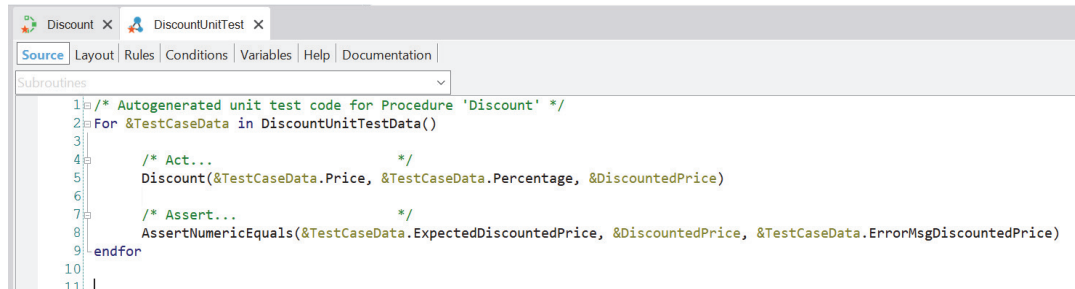
Discount プロシーチャーのテストケースの構造を確認したので、次にデータセットの定義方法を確認します。
これは、自動的に定義されたデータプロバイダーで行います。
値をインスタンス化する SDT エlementを持つ、事前定義済みのグループがあります。



[Source] に、先ほどと同じ値を入れます。

また、結果が 90 でなかった場合に表示するメッセージも指定します。

このデータプロバイダーはテストデータのコレクションを返すため、複数のテストまたは複数のデータセットを容易に定義して実行することができます。しかし、初めて行うテストのサンプルとしてはこれで十分です。



```

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

```

実行する前に、自動的に作成されたオブジェクトの 3 つ目であるユニットテスト本体を見てみましょう。

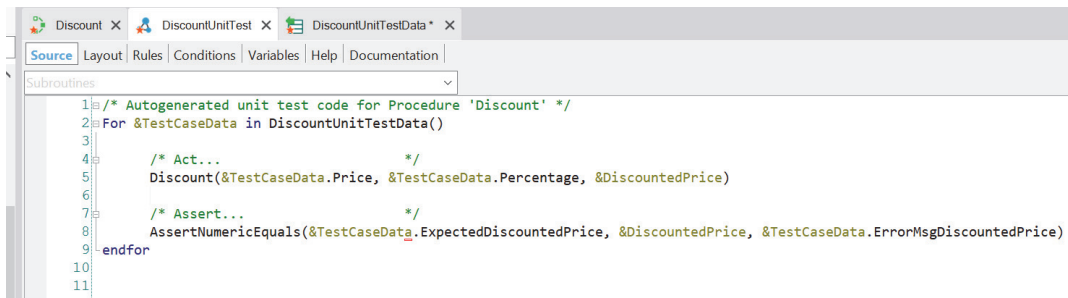
DiscountUnitTest オブジェクトがテストケースのコレクションを確認し、各テストケースに対してプロシーチャーを呼び出し、取得した結果が期待される結果と一致しているかどうかを検証します。

このオブジェクトは、そのように動作するようプログラミングされた GeneXus のプロシーチャーなので、なじみのある注釈が目に入るとと思います。

これは、データプロバイダーで定義したテストコレクションに対する For Each テストケースを表しています。テスト対象のプロシーチャーの呼び出しが作られ、テストケースで定義した 2 つの入力パラメーターと、出力値として変数が利用されています。

ユニットテストの新機能に Assert コマンドがあります。これは、テストケースに定義されている期待される結果と、実際の結果を比較するコマンドです。期待される結果と実際の結果が一致した場合、テストは成功とみなされ、[合格] と表示されます。相違がある場合、テストは失敗となり、[不合格] と表示されます。エラーがあったことを示すレポートが作成され、定義したメッセージが表示されます。

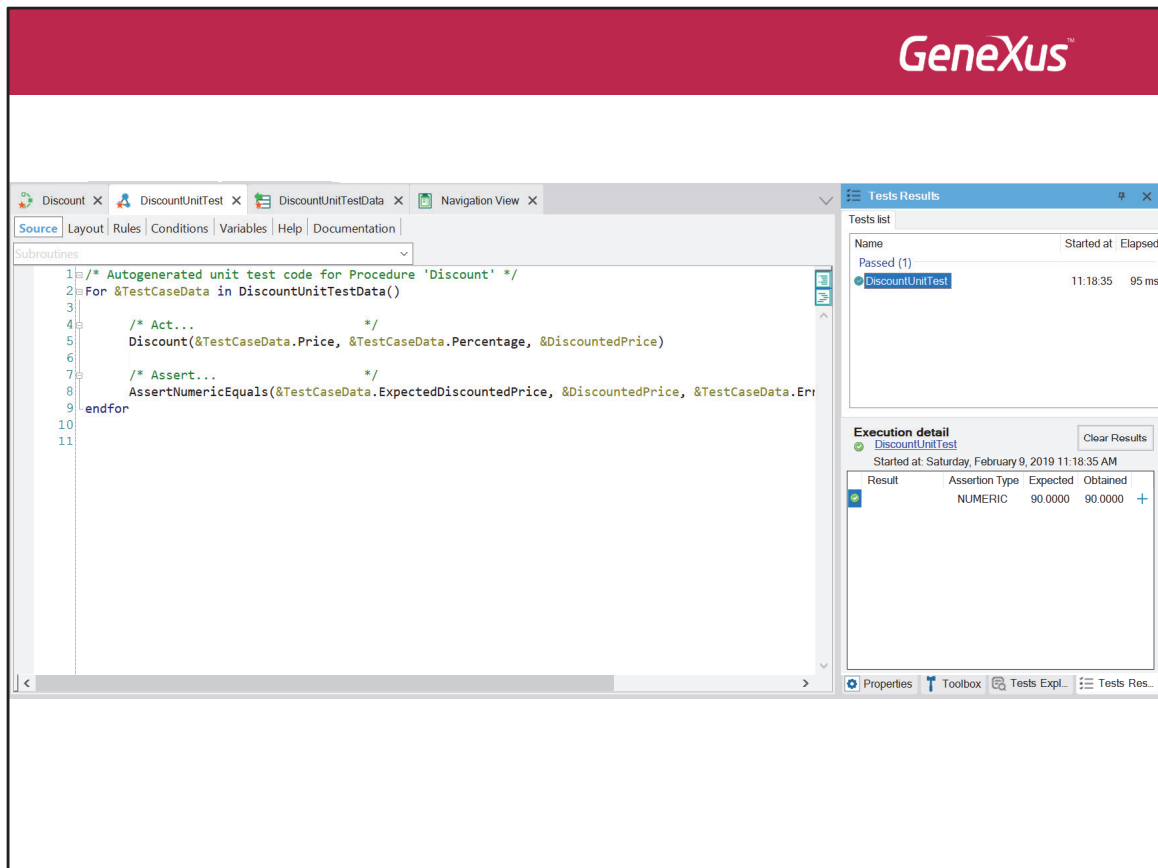
この例では、割引料金が数値であるため AssertNumericEquals 関数で結果を検証していますが、AssertBoolEquals でブール値を比較したり、AssertStringEquals でテキストを比較したりするなど、さまざまなデータタイプを比較することができます。



The screenshot shows the GeneXus IDE interface with three tabs: 'Discount', 'DiscountUnitTest', and 'DiscountUnitTestData'. The 'DiscountUnitTestData' tab is active, displaying a list of subroutines. The code is as follows:

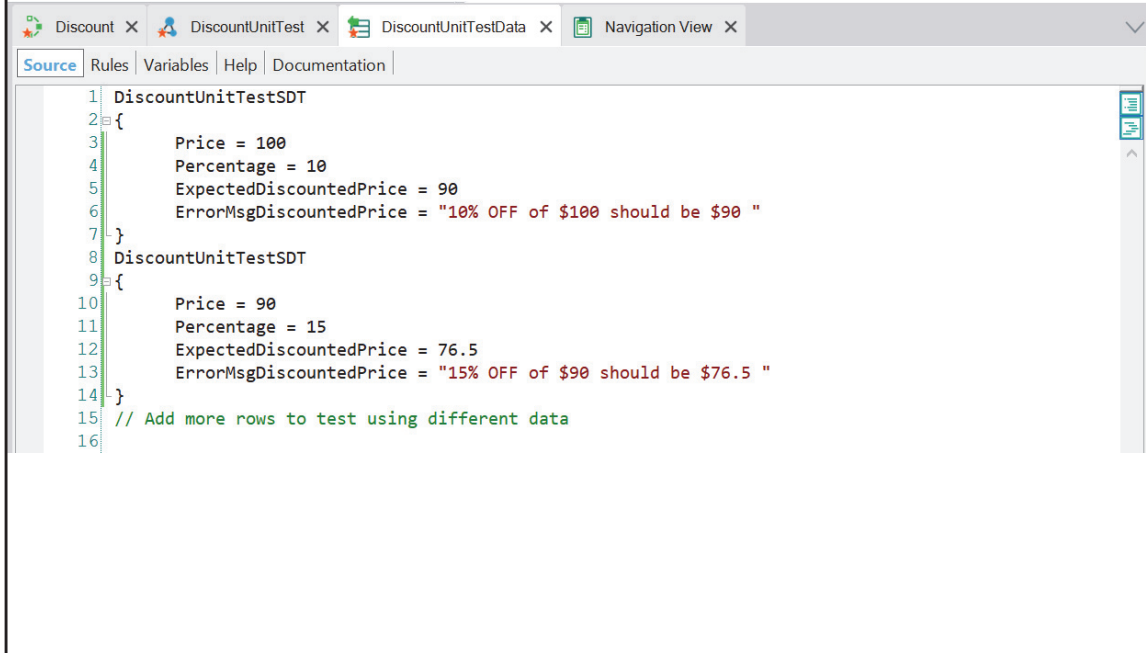
```
1 /* Autogenerated unit test code for Procedure 'Discount' */
2 For &TestCaseData in DiscountUnitTestData()
3
4     /* Act... */
5     Discount(&TestCaseData.Price, &TestCaseData.Percentage, &DiscountedPrice)
6
7     /* Assert... */
8     AssertNumericEquals(&TestCaseData.ExpectedDiscountedPrice,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

ユニットテストの作成時に自動的に作成された 3 つのオブジェクトについて確認できたので、最初のテストケースのデータを読み込み、右クリックして [このテストを実行] を選択してテストを実行します。



テストの実行が完了すると、[テスト結果] という新しいウィンドウが表示されます。ここでテスト (DiscountUnitTest) が実行されたこと、および緑色のマークがついているのでテストに成功したことが分かります。また、テストを実行した時間も表示されます。

その下の [実行詳細] ウィンドウには、テストで見つかったアサーションが表示されています。アサーションごとに、期待される結果と実際の結果、アサーションが成功した場合は緑、失敗した場合は赤のマークが表示されます。アサーションに失敗した場合、テストケースで定義したメッセージも表示されます。



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 the 'DiscountUnitTestSDT' object. The first case sets Price to 100, Percentage to 10, ExpectedDiscountedPrice to 90, and ErrorMessageDiscountedPrice to '10% OFF of \$100 should be \$90 '. The second case sets Price to 90, Percentage to 15, ExpectedDiscountedPrice to 76.5, and ErrorMessageDiscountedPrice to '15% OFF of \$90 should be \$76.5 '. A comment at the bottom suggests adding more rows 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
```

最初のテストに成功し、テストケースを容易に定義できることを理解したところで、データプロバイダーで別のケースを定義してみましょう。

テスト対象のデータの選択は重要なタスクです。チームのテスト担当者と協力して、テストに適したデータを定めてください。

今回は単純なケース (100 ドルに 10% の割引を適用) を使用して、小数値が適切に処理されるかどうかを検証するテストを追加します。そのためには、結果が小数値になる数字を選びます。

次に、もう 1 つのグループを追加します。こちらは料金が 90 で割引率が 15% の場合、期待される料金が 76.5 となります。もう一度テストを実行してみましょう。

The screenshot shows the GeneXus IDE interface. The main window displays the source code for a 'Discount' procedure. The 'Tests Results' panel on the right shows a failed test 'DiscountUnitTest' with a duration of 11:31:15 and 159 ms. The 'Execution detail' panel below it shows a table with two rows of test results. The first row is successful, and the second row shows a failure where the expected value is 90,000 and the obtained value is 76,000 for a '15% OFF of \$...' assertion.

Result	Assertion Type	Expected	Obtained
✓	NUMERIC	90.0000	90.0000
✗	NUMERIC	76.5000	76.0000

実行したテストケースごとに 1 つずつ、合計で 2 つのアサーションが表示されています。

表示を見ると、テストに成功していないことが分かります。前述したように、2 つのアサーションがあります。1 つ目は成功した最初のテストケースのものであり、2 つ目は 2 番目のテストケースのものであります。結果は 76.5 ではなく、76 と表示されています。これは、このプロシージャーでは小数値が考慮されないことを意味しています。

これは、プロシージャーにエラーがあり、割引料金の計算に使用されている変数が小数値ではなく、誤って整数値として定義されている可能性があることを示しています。

また、テストに失敗した (アサーションが成功と表示されていない) 場合に、定義したエラーメッセージが表示されることが確認できます。これは、アサーションに失敗した場合にのみ表示されます。テストに成功した場合、メッセージは表示されません。

The screenshot displays the GeneXus IDE interface. The main window shows the source code for a unit test named 'DiscountUnitTest'. 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.ErrorMessageDiscount)
9
10 Endfor
11

```

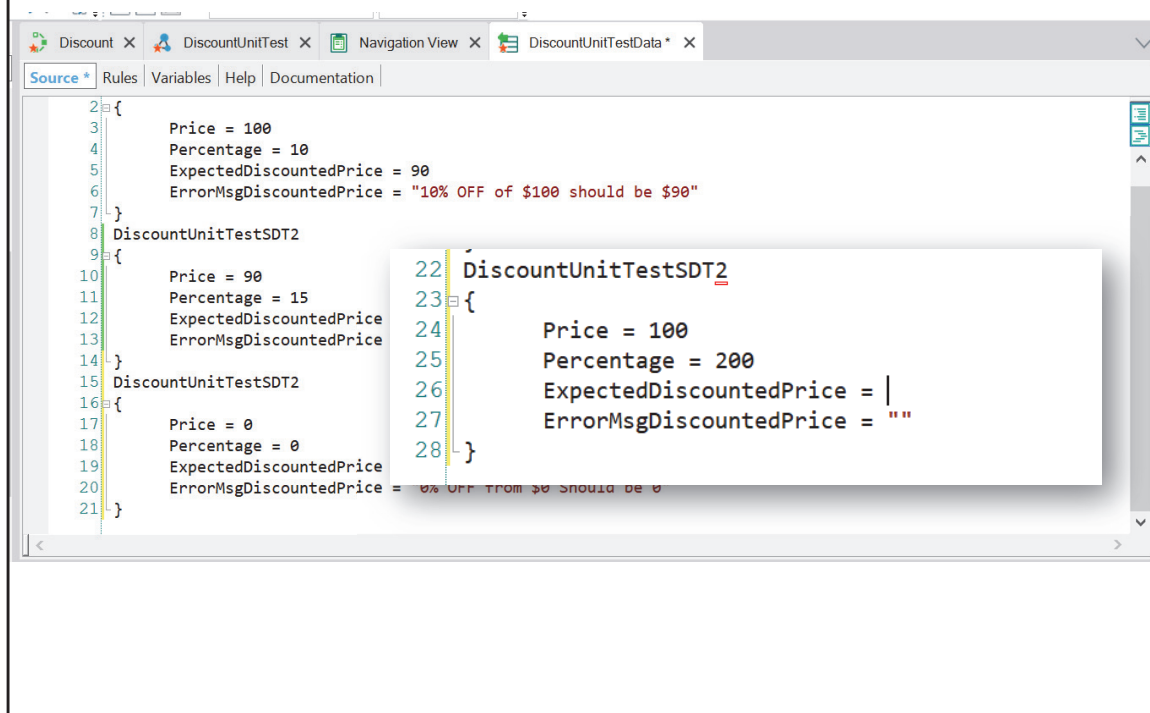
The 'Tests Results' window on the right shows a list of tests. The 'DiscountUnitTest' test is listed as 'Passed (1)' with a start time of 11:46:26 and an elapsed time of 104 ms.

The 'Execution detail' window shows the execution details for 'DiscountUnitTest'. It indicates the test started at Saturday, February 9, 2019 11:46:26 AM. Below this, a table shows the results of two assertions:

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

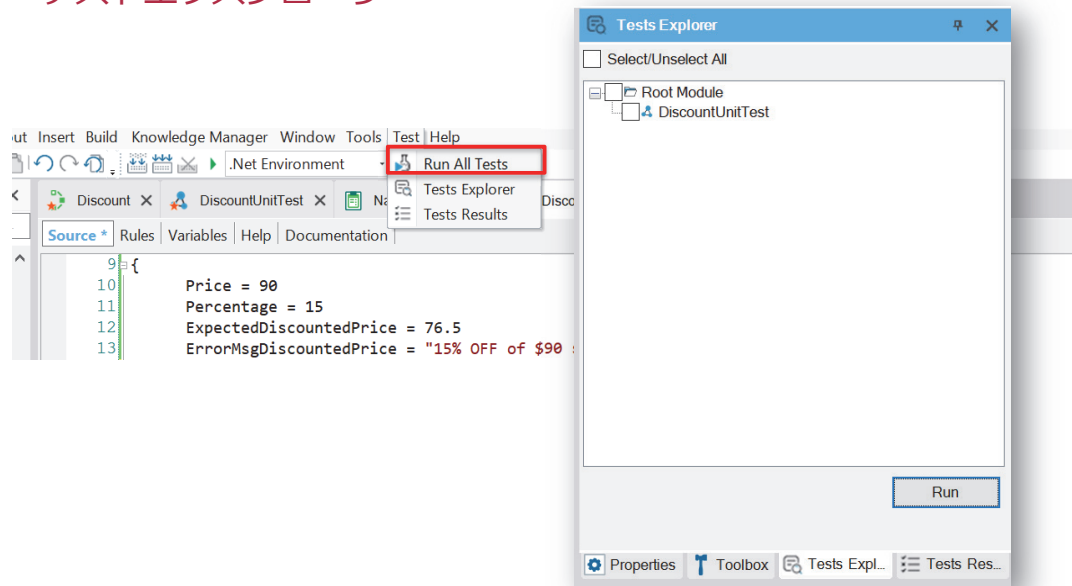
プロシーチャーの出力変数の定義に問題がある場合は、その定義を編集します。変数が定義されていない場合は、Price ドメインに基づいて適切な変数を定義します。変数を修正し、もう一度テストを実行すると、テストに成功したことが表示されます。

この例では、ユニットテストを定義する際に、ちょっとしたトリックを含めました。SDT のエレメントのデータタイプは、テストするプロシーチャーと同じデータタイプで定義されます。このトリックを含めなかった場合、両方とも値が整数になり、期待される結果も 76 だったため、アサーションの結果は成功だったはずでした。これが問題の存在のヒントになったとしても、期待される結果と実際の結果を見て、テスト結果に緑色のマークがついていれば、普通はその結果を分析して問題を見つけようとは考えないでしょう。分かりやすい例をご紹介します。このような事態は、テストに作成した後、誰かが誤って Discount プロシーチャーの出力変数を変更した場合に発生する可能性があります。



次に、境界のテストを追加します。このテスト方法の興味深いところは、テストケースについて考えやすいところです。たとえば、数学的な観点では、100 の 200% 割引は -100 になることは明らかです。しかし、これはこのコンテキストでも有効でしょうか。この Discount プロシーチャーは、負の値を返す必要があるでしょうか。または、何らかの形でエラーを指摘する必要があるでしょうか。テストケースについて考えているときに、要件の定義を改善するのに役立つ疑問を思いつくことがあります。そうして、より確実なシステムにすることができます。

テストエクスプローラー



プロシーチャーとテストを実装したら、これらをチームのほかの作業者と共有 (統合) します。ユニットテストは GeneXus のオブジェクトであるため、ナレッジベースの一部であり、ほかのオブジェクトと同じように共有する必要があることを念頭に置いておくことが重要です。

テストエクスプローラーには、ナレッジベースで定義されているすべてのテストオブジェクトが表示されます。自分で実装したものもあれば、チームのほかのメンバーによって実装されたものもあるでしょう。ワンクリックですべて実行することも、一部を選択して実行することもできます。

将来、Discount プロシーチャーを変更する必要があるときに、どこかに不具合があり、以前と同じ入力パラメーターで前回と異なる結果になった場合、テストは不合格となり、問題を早期に検出できるため、プロシーチャーの変更がより容易になります。

ユニットテスト

- 高速
- 繰り返し可能 (リグレッション)
- アプリケーションのテストの容易性が向上

ごく基本的な例を見てきましたが、同様の方法であらゆる種類のプロシージャー、データプロバイダー、ビジネスコンポーネントをテストすることができます。アプリケーションの実際のデータ (実際のデータベースまたは模擬データベース上のデータ) を使用して、より複雑なテストを実行し、アプリケーションの非常に重要な部分の検証を行うこともできます。

ユニットテストの作成と、コンソールに値を出力するテストプロシージャーの作成には、ほぼ同じ作業が含まれています。ここで興味深いのは、検証を手動で行うのではなく、テスト自体が自分で検証を実施することです。ユニットテストは短時間で迅速に行う必要があります。なぜなら、機能を開発する際のテストにだけ使用するわけではないからです。変更を実装したら、ナレッジベースで定義されているすべてのテストを再度実行し、開発した機能を実装したことでほかのテストに影響が出ないことを確認します。これは、機能ごとにビルドした一連のユニットテストによって、リグレッションテストの一部を自動化できることを意味します。

この方法で作業すると、より仕様変更に対して堅牢なアプリケーションを開発することができます。



<http://wiki.genexus.jp/hwikibypageid.aspx?38353>

ユニットテストはアプリケーションの重要な部分に対して行うことができますが、画面上でのやり取りや、アプリケーション全体のフローは対象としていません。

これらをテストするためには、UI テストでアプリケーションを実行する必要があります。つまり、ユーザーの操作をアプリケーションの画面を通じて確認します。

UI レベルのテストを自動化するために、UITest (ユーザー インターフェース テスト) オブジェクトがあります。

このようなテストは非常に複雑で、実装と実行に時間がかかります。そのため、テストのピラミッド図を見ると、アプリケーションで行われるほとんどの自動テストがユニットテストであることが分かります。ユニットテストは時間がかからず、コストも少ないからです。その次がサービスのテストです。アプリケーション内で重要なフローの UI テストのみを保存します。

UI テストについては、次の Wiki を参照してください:
<http://wiki.genexus.jp/hwikibypageid.aspx?38353>



動画 <https://www.genexus.com/community-and-support-jp/training?ja>

ドキュメント <http://wiki.genexus.jp/>

認定資格 training.genexus.com/certifications

ここでは、バージョン 16 で強化された自動テストの作成と実行について説明しました。この機能は確実なエンジニアリングプロセスの重要な要素となります。