Developer Course

Customization

T280 Testing Business Logic with the Acumatica Unit Test Framework 2022 R1

Revision: 7/25/2022

Acumatica
The Cloud ERP

Contents

Copyright

Restricted Rights

Disclaimer

Trademarks

Software Version: 2022 R1

Last Updated: 07/25/2022

Introduction

The *T280 Testing Business Logic with the Acumatica Unit Test Framework* training course teaches you how you can create unit tests for extension libraries that are used in customization projects.

This course is intended for application developers who customize Acumatica ERP. It explains how to complement the program code with unit tests that ensure the correctness of the program code even after it is changed or enhanced.

The course is based on a set of examples that demonstrate the general approach to creating unit tests for extension libraries developed for Acumatica ERP. The extension library used in this course is the one you developed in the training courses of the *T* series (which you should take before completing the current course). In this course, you will create one unit test for each graph or graph extension created for the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses.

After you complete all the lessons of the course, you will be familiar with some features of Acumatica Unit Test Framework.

We recommend that you complete the examples in the order in which they are provided in the course, because some examples use the results of previous ones.

How to Use This Course

To complete this course, you will complete the lessons of the course in the order in which they are presented and then pass the assessment test. More specifically, you will do the following:

1. Complete the *Course Prerequisites* and perform the *Initial Configuration*.
2. Complete the lessons of the training guide.
3. In Partner University, take *T280 Certification Test: Unit Testing*.

After you pass the certification test, you will receive the Partner University certificate of course completion.

What Is in a Lesson?

Each lesson consists of steps that outline the procedures you are completing and describe the related concepts you are learning.

What Are the Documentation Resources?

The complete Acumatica ERP and Acumatica Framework documentation is available on *https://help.acumatica.com/* and is included in the Acumatica ERP instance. While viewing any form used in the course, you can click the **Open Help** button in the top pane of the Acumatica ERP screen to bring up a form-specific Help menu; you can use the links on this menu to quickly access form-related information and activities and to open a reference topic with detailed descriptions of the form elements.

Licensing Information

For the educational purposes of this course, you use Acumatica ERP under the trial license, which does not require activation and provides all available features. For the production use of the Acumatica ERP functionality, an administrator has to activate the license the organization has purchased. Each particular feature may be subject to additional licensing; please consult the Acumatica ERP sales policy for details.

Course Prerequisites

To complete this course, you should be familiar with the basic concepts of Acumatica Framework and Acumatica Customization Platform. We recommend that you complete the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses before you begin this course.

Required Knowledge and Background

To complete the course successfully, you should have the following required knowledge:

- Proficiency with C#, including but not limited to the following features of the language:
  - Class structure
  - OOP (inheritance, interfaces, and polymorphism)
  - Usage and creation of attributes
  - Generics
  - Delegates, anonymous methods, and lambda expressions
- Knowledge of the following main concepts of ASP.NET and web development:
  - Application states
  - The debugging of ASP.NET applications by using Visual Studio
  - The process of attaching to IIS by using Visual Studio debugging tools
  - Client- and server-side development
  - The structure of web forms
- Experience with SQL Server, including doing the following:
  - Writing and debugging complex SQL queries (WHERE clauses, aggregates, and subqueries)
  - Understanding the database structure (primary keys, data types, and denormalization)
- The following experience with IIS:
  - The configuration and deployment of ASP.NET websites
  - The configuration and securing of IIS

Initial Configuration

You need to perform the prerequisite actions described in this part before you start to complete the course.

If you have deployed an instance for the *T220 Data Entry and Setup Forms* course and have the customization project and the source code for this course, you can use this instance and skip these steps.

Step 1: Preparing the Environment

If you have completed the *T220 Data Entry and Setup Forms* training course and are using the same environment for the current course, you can skip this step.

You should prepare the environment for the training course as follows:

1. Make sure the environment that you are going to use for the training course conforms to the *System Requirements for Acumatica ERP 2022 R1*.
2. Make sure that the Web Server (IIS) features that are listed in *Configuring Web Server (IIS) Features* are turned on.
3. Install the Acuminator extension for Visual Studio.
4. Clone or download the customization project and the source code of the extension library from the *Help-and-Training-Examples* repository in Acumatica GitHub to a folder on your computer.
5. Install Acumatica ERP. On the Main Software Configuration page of the installation program, select the **Install Acumatica ERP** and **Install Debugger Tools** check boxes.

   If you have already installed Acumatica ERP without debugger tools, you should remove Acumatica ERP and install it again with the **Install Debugger Tools** check box selected. The reinstallation of Acumatica ERP does not affect existing Acumatica ERP instances. For details, see *To Install the Acumatica ERP Tools*.

Step 2: Deploying the Needed Acumatica ERP Instance for the Training Course

If you have completed the *T220 Data Entry and Setup Forms* training course, instead of deploying a new instance, you can use the Acumatica ERP instance that you deployed and used for the training course. In this case, you do not need to perform this step.

You deploy an Acumatica ERP instance and configure it as follows:

1. Open the Acumatica ERP Configuration Wizard, and do the following:
   a. Click **Deploy New Application Instance for T-series Developer Courses**.
   b. On the **Database Configuration** page, make sure the name of the database is `PhoneRepairShop`.
   c. On the **Instance Configuration** page, do the following:
      a. In the **Local Path of the Instance** box, select a folder that is outside of the `C:\Program Files (x86)` and `C:\Program Files` folders. (We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you perform customization of the website.)

   b. In the **Training Course** box, select the training course you are taking.

   The system creates a new Acumatica ERP instance, adds a new tenant, loads the data to it, and publishes the customization project that is needed for this training course.

2. Make sure a Visual Studio solution is available in the `App_Data\Projects\PhoneRepairShop` folder of the Acumatica ERP instance folder. This is the solution of the extension library that you will modify in this course.
3. Sign in to the new tenant by using the following credentials:
   - **Username**: `admin`
   - **Password**: `setup`

   Change the password when the system prompts you to do so.
4. In the top right corner of the Acumatica ERP screen, click the username, and then click **My Profile**. The *User Profile* (SM203010) form opens. On the **General Info** tab, select *YOGIFON* in the **Default Branch** box; then click **Save** on the form toolbar.

   In subsequent sign-ins to this account, you will be signed in to this branch.
5. Optional: Add the *Customization Projects* (SM204505) and *Generic Inquiry* (SM208000) forms to your favorites. For details about how to add a form to your favorites, see *Managing Favorites: General Information*.

If for some reason you cannot complete instructions in this step, you can create an Acumatica ERP instance as described in *Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course* and manually publish the needed customization project as described in *Appendix: Publishing the Required Customization Project*.

Company Story and Mission Description

This topic describes the company story and explains the scope of this course.

Company Story

The Smart Fix company specializes in repairing cell phones of several types. The company provides the following services:

- Battery replacement: This service is provided on customer request and does not require any preliminary diagnostic checks.
- Repair of liquid damage: This service requires a preliminary diagnostic check and a prepayment.
- Screen repair: This service is provided on customer request and does not require any preliminary diagnostic checks.

To manage the list of devices serviced by the company and the list of services the company provides, two custom maintenance forms, Repair Services (RS201000) and Serviced Devices (RS202000), have been added to the company's Acumatica ERP instance. Another custom maintenance form, Services and Prices (RS203000), gives users the ability to define and maintain the price for each provided repair service. Also, the *Stock Items* (IN202500) form of Acumatica ERP has been customized to give users the ability to mark particular stock items as repair items —that is, the items (such as replacement screens and batteries) that are supplied to the customer as part of the repair services. The Repair Work Orders (RS301000) custom data entry form is used to create and manage work orders for repairs. On the Repair Work Order Preferences (RS101000) custom setup form, an administrative user specifies the company's preferences for the repair work orders.

Mission Description

To keep the business logic consistent, you need to develop unit tests for the developed forms. Adding these unit tests to the customization library will ensure that the implemented behavior will persist even if changes are made later to the customization library.

About the Acumatica Unit Test Framework

The Acumatica Unit Test Framework is the platform for testing graphs and reusable components that you create for Acumatica ERP. By using the framework, a developer can test the business logic of individual graphs.

By design, the Acumatica Unit Test Framework does not require you to sign in or access a database on a disk. You need not deploy an Acumatica ERP instance or a database instance. All actions are performed in RAM.

The Acumatica Unit Test Framework is designed to be used with the xUnit.net library.

A unit test is an automated test written by a software developer that tests some independent unit of code. Usually a unit test launches some functionality of a software component and then compares the resulting values with the expected values. The best practice for developing unit tests is to create them according to the standard *AAA* pattern: First there is an *arrange* part in which the test prepares its context and assigns some values, and then there is an *act* part and an *assert* part.

In the *arrange* part, a developer initializes the context until actions can be correctly executed. You may need to enable some application features (by using the `Common.UnitTestAccessProvider.EnableFeature<FeatureType>` generic method) or initialize the required setup DACs (by using the `Setup<PXGraph>` generic method). After that, you create an instance of a graph (by using the `PXGraph.CreateInstance<PXGraph>` generic method) and fill its caches.

Some automated tests validate the interaction of multiple independent components. Such tests are usually called *integration tests*. By using the Acumatica Unit Test Framework, you can also write more complex tests that check the interaction of multiple components within a graph.

Related Links

- *The xUnit.net Library Website*

Lesson 1: Creating a Test Project

In this lesson, you will create the project that you will use as the test project in this course. Then you will configure the project to make it a test project for use in an extension library.

Generally, for each project whose functionality is going to be tested, a developer creates a test project in the same Visual Studio solution and names it by adding *.Tests* to the name of the project being tested. The namespace used in the test project is named the same as the namespace of the project being tested with *.Tests* appended to the name. The test class is named the same as the class being tested with *Tests* appended to the name. For example, to test the `InventoryItemMaint` class, which is located in the `PhoneRepairShop_Code` namespace and contained in the `PhoneRepairShop_Code.csproj` project, you create the `PhoneRepairShop_Code.Tests.csproj` project; in this project, you create the `InventoryItemMaintTests` class and place it in the `PhoneRepairShop_Code.Tests` namespace.

Lesson Objectives

As a result of completing this lesson, you will do the following:

- Create a test project in Visual Studio
- Configure the test project for using the xUnit.net library and Acumatica Unit Test Framework

Related Links

- *Getting Started with xUnit.net Using .NET Framework with Visual Studio*

Step 1.1: Create a Test Project

You create the `PhoneRepairShop_Code.Tests.csproj` project as follows:

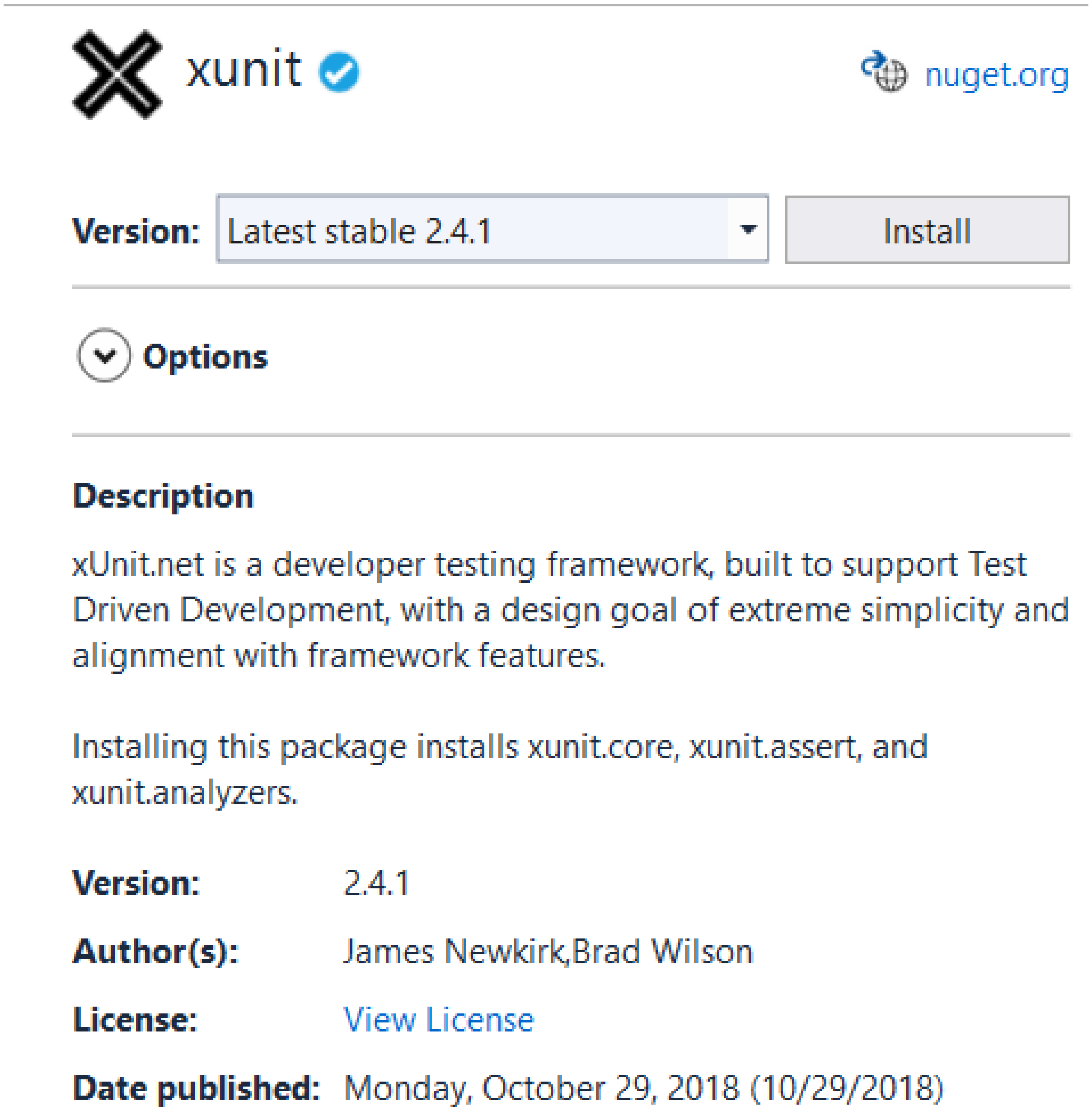
1. Open the `PhoneRepairShop_Code.sln` file located in the `\App_Data\Projects\PhoneRepairShop` subfolder of the *PhoneRepairShop* instance folder.
2. In Visual Studio, in the **Solution Explorer** panel, right-click the `PhoneRepairShop_Code` solution.
3. Select the **Add > New Project** menu item.
4. Select the *Class Library (.NET Framework)* project template.
5. Click **Next**.
6. Enter the `PhoneRepairShop_Code.Tests` project name.
7. Click **Create**.
8. Delete the `Class1.cs` item from the `PhoneRepairShop_Code.Tests.csproj` project, because you will create the required C# classes later.

Step 1.2: Configure the Test Project

You configure the `PhoneRepairShop_Code.Tests.csproj` project as follows:

1. In the **Solution Explorer** panel, right-click the `PhoneRepairShop_Code.Tests` project, and select **Manage NuGet Packages**.
2. Select the **Browse** link or make sure that it is selected.
3. In the search box, type `xunit`.

4. In the search results, select the xunit package, and click **Install**.



Version: Latest stable 2.4.1

Description

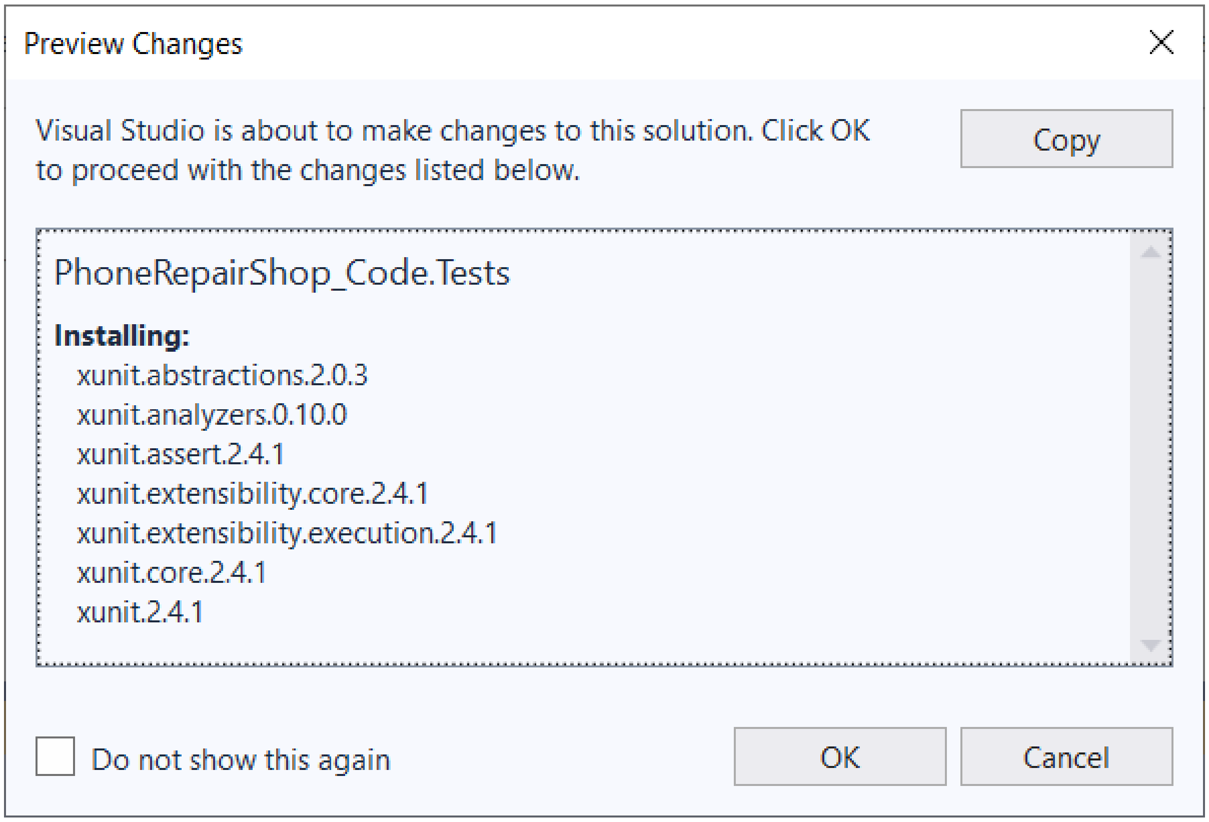
xUnit.net is a developer testing framework, built to support Test Driven Development, with a design goal of extreme simplicity and alignment with framework features.

Installing this package installs xunit.core, xunit.assert, and xunit.analyzers.

Version:	2.4.1
Author(s):	James Newkirk,Brad Wilson
License:	View License
Date published:	Monday, October 29, 2018 (10/29/2018)

Figure: xunit package information

5. In the **Preview Changes** dialog box, click **OK**. This will install the xunit package.



Visual Studio is about to make changes to this solution. Click OK to proceed with the changes listed below.

PhoneRepairShop_Code.Tests

Installing:

xunit.abstractions.2.0.3
xunit.analyzers.0.10.0
xunit.assert.2.4.1
xunit.extensibility.core.2.4.1
xunit.extensibility.execution.2.4.1
xunit.core.2.4.1
xunit.2.4.1

Figure: Preview Changes dialog box

6. Install the `xunit.runner.visualstudio` and `Microsoft.Bcl.AsyncInterfaces` packages in the same way as you installed the xunit package.
7. Optional: Copy the `PX.Tests.Unit.dll` file from the `UnitTesting` subfolder of the Acumatica ERP installation folder to any folder you choose.
8. Right-click the **References** item of the `PhoneRepairShop_Code.Tests` project, and select **Add Reference**.
9. On the **Browse** tab, click **Browse**.
10. In the `PhoneRepairShop\Bin` folder, select the `PX.Common.dll`, `PX.Data.BQL.Fluent.dll`, and `PX.Data.dll` files. Select the `PX.Tests.Unit.dll` file in the folder where you have placed it.
11. On the **Projects** tab, select the `PhoneRepairShop_Code` project.
12. Click **OK**.

You may configure your project for using another unit test library (for example, NUnit), but Acumatica Unit Test Framework does not guarantee compatibility with a unit test library other than xUnit.net.

Lesson Summary

In this lesson, you have created a project that is ready to have unit tests added to it.

Lesson 2: Testing the Graph for the Repair Services Form

In this lesson, you will create a simple unit test for a graph. The `RSSVRepairServiceMaint` graph contains logic that makes the selection of the **Walk-In Service** check box and the selection of the **Requires Preliminary Check** check box mutually exclusive. You will create a class to which you will add a test method for checking the states of the check boxes.

Lesson Objectives

As a result of completing this lesson, you will do the following:

- Create a class for unit tests
- Create a method with unit tests
- Learn how to run unit tests
- Learn how to debug unit tests

Step 2.1: Create a Test Class for the Repair Services Form

To create a test class for the Repair Services (RS201000) form, do the following:

1. In the **Solution Explorer** panel, right-click the `PhoneRepairShop_Code.Tests` project, and select **Add > New Item**.
2. Select the *Visual C# item* | *Class* template, and type `RSSVRepairServiceMaintTests.cs` as the file name.
3. Click **Add**.
4. Specify the following `using` directives in the `RSSVRepairServiceMaintTests.cs` file.

```
using Xunit;
using PX.Data;
using PX.Tests.Unit;
using PhoneRepairShop;
```

5. Make the `RSSVRepairServiceMaintTests` class public and derived from `TestBase` as follows.

```
    public class RSSVRepairServiceMaintTests : TestBase
```

Step 2.2: Create a Test Method for the Repair Services Form

To create a method for checking the states of the check boxes of the Repair Services (RS201000) form, do the following:

1. In the `RSSVRepairServiceMaintTests` class, create a public void method, and name it *PreliminaryCheckAndWalkInServiceFlags_AreOpposite*.

```
        public void PreliminaryCheckAndWalkInServiceFlags_AreOpposite()
        {
        }
```

2. Assign the *[Fact]* attribute to the method to specify that this unit test is called without parameters.

```
        [Fact]
        public void PreliminaryCheckAndWalkInServiceFlags_AreOpposite()
```

3. In the `PreliminaryCheckAndWalkInServiceFlags_AreOpposite` method, create an instance of the `RSSVRepairServiceMaint` graph as follows.

```
            var graph = PXGraph.CreateInstance<RSSVRepairServiceMaint>();
```

4. After the `RSSVRepairServiceMaint` graph is initialized, create an `RSSVRepairService` object that represents a record in the table of the Repair Services form.

```
            RSSVRepairService repairService =
                graph.Caches[typeof(RSSVRepairService)].
                Insert(new RSSVRepairService
                {
                    ServiceCD = "Service1",
                    Description = "Service 1",
                    WalkInService = true,
                    PreliminaryCheck = false
                }) as RSSVRepairService;
```

   Objects representing table records are created in the graph cache (`PXCache` objects). You can address a specific cache of a graph either by the DAC name (`graph.Caches[typeof(<DAC_name>)]`) or by the graph view that contains the DAC objects (`graph.<DACView>.Cache`).

5. After a record is created, check whether a change of the **Walk-In Service** check box state leads to the change of the **Requires Preliminary Check** check box state; also check whether a change of the **Requires Preliminary Check** check box state leads to a change of the **Walk-In Service** check box.

```
            repairService.WalkInService = false;
            graph.Caches[typeof(RSSVRepairService)].Update(repairService);
            Assert.True(repairService.PreliminaryCheck);

            repairService.WalkInService = true;
            graph.Caches[typeof(RSSVRepairService)].Update(repairService);
            Assert.False(repairService.PreliminaryCheck);

            repairService.PreliminaryCheck = false;
            graph.Caches[typeof(RSSVRepairService)].Update(repairService);
            Assert.True(repairService.WalkInService);

            repairService.PreliminaryCheck = true;
            graph.Caches[typeof(RSSVRepairService)].Update(repairService);
            Assert.False(repairService.WalkInService);
```

   This test does not use the *arrange-act-assert* pattern; instead, it contains multiple *act* parts and *assert* parts.

Assertion Methods

In the `RSSVRepairServiceMaintTests` class, you used the `Assert.True` and `Assert.False` static methods to verify field values. The following table lists additional methods of the `Xunit.Assert` class that can be useful.

Table: Assert Class Methods

Method	Description
`False(bool condition)`	Verifies that `condition` is *false*. The method also throws a `FalseException` if `condition` is not *false*.
`True(bool condition)`	Verifies that `condition` is *true*, and throws a `True-Exception` if `condition` is not *true*.
`Contains<T>(T expected, IEnumerable<T> collection)`	Verifies that a collection contains a given object. The method also throws a `ContainsException` if the collection does not contain the object.
`DoesNotContain<T>(T expected, IEnumer-able<T> collection)`	Verifies that a collection does not contain a given object. The method also throws a `DoesNotCon-tainException` if the collection contains the ob-ject.
`Empty(IEnumerable collection)`	Verifies that `collection` is empty, and throws an `EmptyException` if `collection` is not empty.
`NotEmpty(IEnumerable collection)`	Verifies that `collection` is not empty, and throws a `NotEmptyException` if `collection` is empty.
`Single(IEnumerable collection)`	Verifies that `collection` contains a single element. The method also throws a `SingleException` if `collection` does not contain exactly one element.
`Null(object obj)`	Verifies that an object reference is *null*, and throws a `NullException` if the object reference is not *null*.
`NotNull(object obj)`	Verifies that an object reference is not *null*, and throws a `NotNullException` if the object reference is *null*.
`Contains(string expectedSubstring, string actualString)`	Verifies that `actualString` contains `expect-edSubstring` by using the current culture. The method also throws a `ContainsException` if `ac-tualString` does not contain `expectedSub-string`.
`DoesNotContain(string expectedSub-string, string actualString)`	Verifies that `actualString` does not contain `ex-pectedSubstring` by using the current culture. The method also throws a `DoesNotContainExcep-tion` if `actualString` contains `expectedSub-string`.
`StartsWith(string expectedStartString, string actualString)`	Verifies that `actualString` starts with `expect-edStartString` by using the current culture. The method also throws a `StartsWithException` if `actualString` does not start with `expectedS-tartString`.

Method	Description
`EndsWith(string expectedEndString, string actualString)`	Verifies that `actualString` ends with `expectedEndString` by using the current culture, and throws an `EndsWithException` if `actualString` does not end with `expectedEndString`.
`Equal(string expected, string actual)`	Verifies that the `actual` and `expected` strings are equivalent. The method also throws an `EqualException` if the `actual` and `expected` strings are not equivalent.
`Equal<T>(T expected, T actual)`	Verifies that two objects are equal by using a default comparer, and throws an `EqualException` if the objects are not equal.
`Equal<T>(T[] expected, T[] actual)`	Verifies that two arrays of unmanaged type `T` are equal. The method also throws an `EqualException` if the arrays are not equal.

You can use some assertion library instead of the `Xunit.Assert` class (for example, `FluentAssertions`).

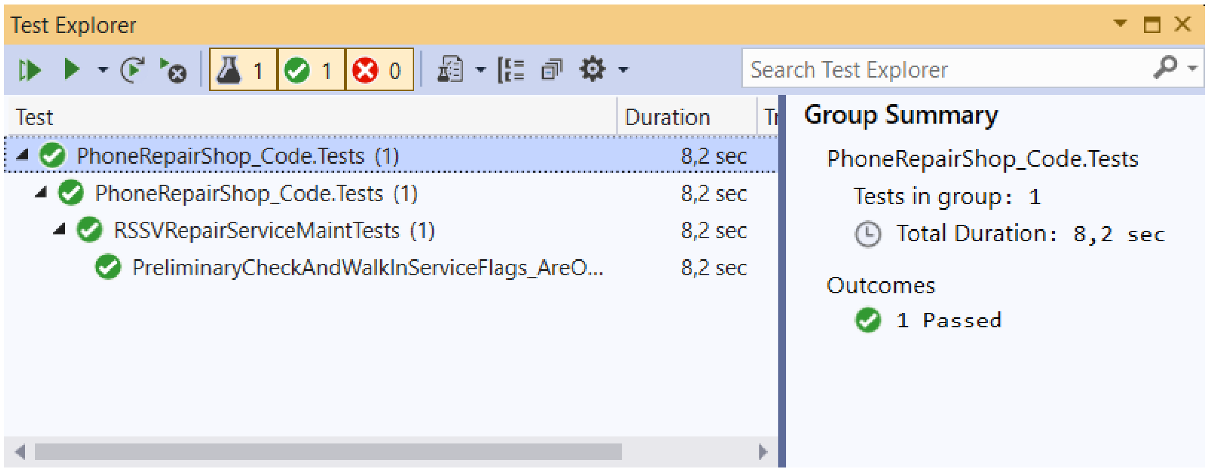
Related Links

- *PXCache<TNode> class*
- *Fluent Assertions*

Step 2.3: Run the Test Method

You should do the following to run the method created in the previous step:

1. In Visual Studio, select the **Test > Test Explorer** menu item. The **Test Explorer** dialog box opens, which currently displays no tests.
2. Click **Run All Tests In View**. The solution is built, and one test appears in the **Test Explorer** dialog box (shown in the following screenshot) and is run.



Test	Duration
PhoneRepairShop_Code.Tests (1)	8,2 sec
PhoneRepairShop_Code.Tests (1)	8,2 sec
RSSVRepairServiceMaintTests (1)	8,2 sec
PreliminaryCheckAndWalkInServiceFlags_AreO...	8,2 sec

Group Summary

PhoneRepairShop_Code.Tests

Tests in group: 1

Total Duration: 8,2 sec

Outcomes

1 Passed

Figure: Test Explorer with one test

Step 2.4: Debug the Test Method

You can debug test methods along with the extension library.

This topic contains an example of debugging the `PreliminaryCheckAndWalkInServiceFlags_AreOpposite` method of the `RSSVRepairServiceMaintTests` class and the `RSSVRepairServiceMaint` class of the extension library. You will check whether clearing the **Walk-In Service** check box causes the **Requires Preliminary Check** check box to be selected.

1. In Visual Studio, open the `RSSVRepairServiceMaintTests.cs` file.
2. Move the text cursor to the following line.

```
repairService.WalkInService = false;
```

3. Press F9 to set a breakpoint on the line.
4. Open the `RSSVRepairServiceMaint.cs` file.
5. Move the text cursor to the following line of the `_(Events.FieldUpdated<RSSVRepairService, RSSVRepairService.walkInService> e)` method.

```
row.PreliminaryCheck = true;
```

6. Press F9 to set a breakpoint on the line.
7. Select the **Test > Test Explorer** menu command to open the **Test Explorer** dialog box.
8. Right-click the `PreliminaryCheckAndWalkInServiceFlags_AreOpposite` method, and select **Debug**. After the debug process starts, execution pauses at the breakpoint in the `RSSVRepairServiceMaintTests.cs` file. You can verify that the value of `repairService.WalkInService` is *true*.
9. Step over (you can use F10 for this). The value of `repairService.WalkInService` becomes *false*.
10. Step over. This updates the `repairService` object in the cache. During the update, the `FieldUpdated` event is generated for the **Walk-In Service** check box, and the `_(Events.FieldUpdated<RSSVRepairService, RSSVRepairService.walkInService> e)` method of the `RSSVRepairServiceMaint` class is launched. Execution pauses at the breakpoint in

the `RSSVRepairServiceMaint.cs` file, which signifies that the extension library and unit test match each other.

11. Continue debugging (you can press F5 for this). The debug process continues until it completes successfully.

Lesson Summary

In this lesson, you have learned how to create test classes and test methods that have no parameters. You have also learned how to run and debug unit tests.

Lesson 3: Testing the Graph Extension for the Stock Items Form

In this lesson, you will create a unit test for a graph extension. The `InventoryItemMaint` class contains the logic for the *Stock Items* (IN202500) form that makes the availability of the **Repair Item Type** drop-down list dependent on the state of the **Repair Item** check box (that is, whether it is selected or not). You will create a class to which you will add a test method for checking the state of the **Repair Item** check box and the availability of the **Repair Item Type** drop-down list.

Lesson Objectives

As a result of completing this lesson, you will do the following:

- Create a unit test for the graph extension
- Create a parameterized test method

Step 3.1: Configure the Test Project

You change the configuration of the existing `PhoneRepairShop_Code.Tests` project as follows:

1. In the **Solution Explorer** panel, for the `PhoneRepairShop_Code.Tests` project, right-click the **References** item, and select **Add Reference**.
2. Select the **Browse** tab, and click **Browse**.
3. In the `PhoneRepairShop\Bin` folder, select the `PX.Objects.dll` file.
4. Click **OK**.

The `PX.Objects.dll` library contains the implementation of the `PX.Objects.IN.InventoryItem` class, which will be used later.

Step 3.2: Create a Test Class for the Stock Items Form

To create a class for testing the `InventoryItemMaint` graph, which extends the logic for the *Stock Items* (IN202500) form, do the following:

1. In the **Solution Explorer** panel, right-click the `PhoneRepairShop_Code.Tests` project, and select **Add > New Item**.
2. Select the *Visual C# item* | *Class* template, and type `InventoryItemMaintTests.cs` as the file name.
3. Click **Add**.
4. Specify the following `using` directives in the `InventoryItemMaintTests.cs` file.

   ```
   using Xunit;
   using PX.Data;
   using PX.Tests.Unit;
   using PX.Objects.IN;
   ```

5. Make the `InventoryItemMaintTests` class public and derived from `TestBase` as follows.

```
    public class InventoryItemMaintTests : TestBase
```

Step 3.3: Create a Test Method for the Stock Items Form

In this test method, you will make sure that selecting the **Repair Item** check box on the *Stock Items* (IN202500) form leads to the availability of the **Repair Item Type** drop-down list and that clearing the **Repair Item** check box leads to the unavailability of the **Repair Item Type** drop-down list. To do this, you will create a test method with a parameter that signifies whether the **Repair Item** is selected or cleared. Do the following:

1. In the `InventoryItemMaintTests` class, create a public void method, and name it *RepairItemTypeEnabled_WhenRepairItemSelected*.
2. To specify that this unit test is called with parameters, assign the `[Theory]` attribute to the method.

```
        [Theory]
        public void RepairItemTypeEnabled_WhenRepairItemSelected
```

3. Add the Boolean `enabled` parameter to the `RepairItemTypeEnabled_WhenRepairItemSelected` method as follows.

```
        public void RepairItemTypeEnabled_WhenRepairItemSelected
            (bool enabled)
```

4. Add two `[InlineData]` attributes, which provide two values for the `enabled` parameter.

```
        [Theory]
        [InlineData(true)]
        [InlineData(false)]
        public void RepairItemTypeEnabled_WhenRepairItemSelected
            (bool enabled)
```

   The `RepairItemTypeEnabled_WhenRepairItemSelected` method will be called twice: The first call will pass *true* as its argument, and the second will pass *false*.

   In the `[InlineData]` attribute, you specify constants as the values of the test method arguments. The number of the constants and their types must match the number and the types of the test method arguments.

   You can also specify values of the test method arguments by using the `[ClassData]` or `[MemberData]` attribute.

5. In the `RepairItemTypeEnabled_WhenRepairItemSelected` method, create an instance of the `InventoryItemMaint` graph by adding the following code.

```
            var graph = PXGraph.CreateInstance<InventoryItemMaint>();
```

6. After the `InventoryItemMaint` graph is initialized, create an `InventoryItem` object.

```
            InventoryItem item =
                (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(
                    new InventoryItem
                    {
                        InventoryCD = "Item1",
                        Descr = "Item 1"
                    });
```

7. Get the `InventoryItemExt` extension of the `InventoryItem` object.

```
            InventoryItemExt itemExt = item.GetExtension<InventoryItemExt>();
```

8. Select (or clear) the **Repair Item** check box by using the `enabled` parameter, and make sure that the **Repair Item Type** drop-down list is available (or, respectively, unavailable).

```
            itemExt.UsrRepairItem = enabled;
            graph.Caches[typeof(InventoryItem)].Update(item);
            PXFieldState fieldState =
                ((PXFieldState)graph.Caches[typeof(InventoryItem)].GetStateExt<
                InventoryItemExt.usrRepairItemType>(item));
            Assert.True(enabled == fieldState.Enabled);
```

A call of the `PXCache.Update` method for the `InventoryItem` object is sufficient for updating the object along with its extension.

As a result of the actions described in this topic, the `InventoryItemMaintTests.cs` file contains the following code.

```
using Xunit;
using PX.Data;
using PX.Tests.Unit;
using PX.Objects.IN;

namespace PhoneRepairShop_Code.Tests
{
    public class InventoryItemMaintTests : TestBase
    {
        [Theory]
        [InlineData(true)]
        [InlineData(false)]
        public void RepairItemTypeEnabled_WhenRepairItemSelected
            (bool enabled)
        {
            var graph = PXGraph.CreateInstance<InventoryItemMaint>();

            InventoryItem item =
                (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(
                    new InventoryItem
                    {
                        InventoryCD = "Item1",
                        Descr = "Item 1"
                    });

            InventoryItemExt itemExt = item.GetExtension<InventoryItemExt>();

            itemExt.UsrRepairItem = enabled;
            graph.Caches[typeof(InventoryItem)].Update(item);
            PXFieldState fieldState =
                ((PXFieldState)graph.Caches[typeof(InventoryItem)].GetStateExt<
                InventoryItemExt.usrRepairItemType>(item));
            Assert.True(enabled == fieldState.Enabled);
        }
        }
}
```

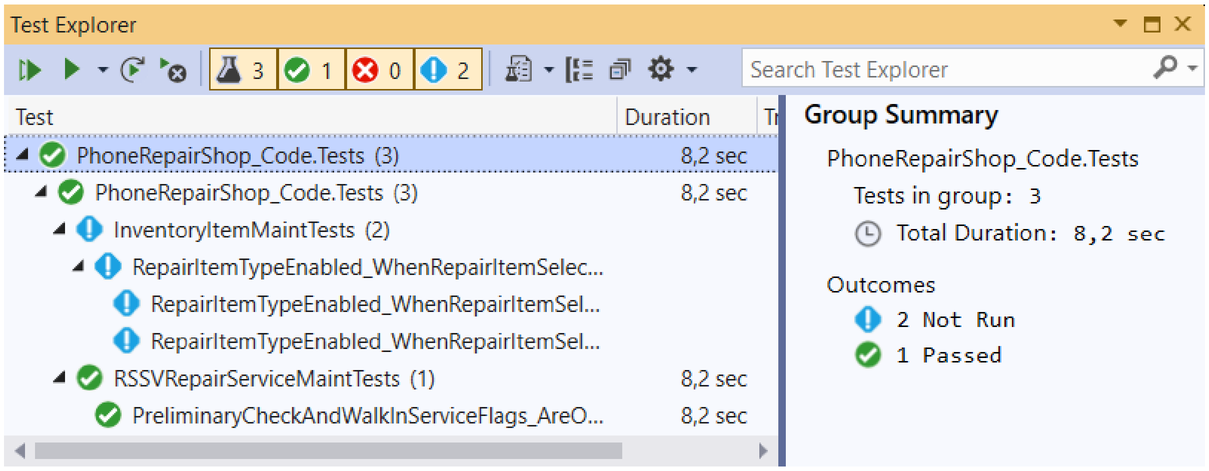
This test uses the *arrange-act-assert* pattern.

Related Links

- *PXCache.GetStateExt(Object,String)*

Step 3.4: Manage Tests

After you have added the tests, they appear in the **Test Explorer** dialog box, which is shown in the following screenshot. Note that for parameterized test methods, a test is added for each set of parameter values that you have specified for the method.



```
Test                                                      Duration
PhoneRepairShop_Code.Tests (3)                            8,2 sec
  PhoneRepairShop_Code.Tests (3)                          8,2 sec
    InventoryItemMaintTests (2)
      RepairItemTypeEnabled_WhenRepairItemSelec...
        RepairItemTypeEnabled_WhenRepairItemSel...
        RepairItemTypeEnabled_WhenRepairItemSel...
    RSSVRepairServiceMaintTests (1)                       8,2 sec
      PreliminaryCheckAndWalkInServiceFlags_AreO...       8,2 sec

Group Summary
  PhoneRepairShop_Code.Tests
    Tests in group: 3
    Total Duration: 8,2 sec
  Outcomes
    2 Not Run
    1 Passed
```

Figure: Test Explorer with added tests

You can select the desired test, test method, or group of test methods, and you can run or debug them. You can also get information about the tests' outcomes (whether they succeeded or not) and execution time.

Running a Single Test

For example, you can run the `RepairItemTypeEnabled_WhenRepairItemSelected(enabled: False)` test by right-clicking this item in the **Test Explorer** dialog box and selecting **Run** in the context menu. Note that this leads to the generation of the following error message (only part of the error message is quoted).

Autofac.Core.Registration.ComponentNotRegisteredException : The requested service 'System.Func`2[[PX.Data.PXGraph, PX.Data, Version=1.0.0.0, Culture=neutral, PublicKeyToken=3b136cac2f602b8e],[PX.Objects.CM.Extensions.IPXCurrencyService, PX.Objects, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null]]' has not been registered. To avoid this exception, either register a component to provide the service, check for service registration using IsRegistered(), or use the ResolveOptional() method to resolve an optional dependency.

See *Step 3.5: Register Services* for details about how to address this error.

Step 3.5: Register Services

Acumatica ERP uses the *dependency injection* technique to design software. With this technique, an object (a *client*) receives other objects (*services*) that it depends on; these are called *dependencies*. This software design pattern is used to decouple code components and make them easier to extend and test. For more information about dependency injection, see *https://docs.microsoft.com/en-us/dotnet/core/extensions/dependency-injection*.

Acumatica ERP uses the Autofac library to implement the dependency injection. Services can be injected into graphs, graph extensions, attributes, and custom action classes. For more information, see *Dependency Injection*.

In unit tests, mock services are used instead of real complex objects. These mock services are simplified services that mimic real ones.

To register a service, you need to override the `TestBase.RegisterServices` method and make calls to the `Autofac.ContainerBuilder.RegisterType` generic method.

In Acumatica ERP, the following mock services are often used in unit tests.

Table: Frequently Used Mock Services

Mocked Interface	Mock Service
`IFinPeriodRepository` (defined in the `PX.Objects.GL.FinPeriods` namespace)	`FinPeriodServiceMock` (defined in the `PX.Objects.Unit` namespace)
`IPXCurrencyService` (defined in the `PX.Objects.CM.Extensions` namespace)	`CurrencyServiceMock` (defined in the `PX.Objects.Unit` namespace)

Registering the Necessary Service

The error message provided in *Step 3.4: Manage Tests* tells you to register a service that is an implementation of the `PX.Objects.CM.Extensions.IPXCurrencyService` interface so that the `InventoryItemMaint` graph could be instantiated. You should register the `PX.Objects.Unit.CurrencyServiceMock` class as the required service.

To register this service, do the following:

1. In the `PhoneRepairShop_Code.Tests.csproj` project, add a reference to `Autofac.dll`. Or, install the Autofac NuGet package. Both procedures are described in *Step 1.2: Configure the Test Project*.
2. Add the following `using` directives to the `InventoryItemMaintTests.cs` file.

```
using Autofac;
using PX.Objects.CM.Extensions;
```

3. In the `InventoryItemMaintTests` class, override the `TestBase.RegisterServices` method as follows.

```
        protected override void RegisterServices(ContainerBuilder builder)
        {
            base.RegisterServices(builder);
            builder.RegisterType<PX.Objects.Unit.CurrencyServiceMock>().
                As<IPXCurrencyService>();
        }
```

Now run the `RepairItemTypeEnabled_WhenRepairItemSelected(enabled: False)` test once more. Make sure that this time the test succeeds.

Lesson Summary

In this lesson, you have created a parametrized test method and have called this method twice by adding two `[InlineData]` attributes to it. To perform the tests successfully, you have registered a mock service.

Lesson 4: Testing the Graph for the Services and Prices Form

In this lesson, you will create a more complicated unit test. You will create a unit test method for ensuring that changing the values of the **Required** and **Default** check boxes of a row on the Services and Prices (RS203000) form affects the values of these check boxes in other rows. Also, you will make sure that the prices are calculated correctly.

Lesson Objectives

As a result of completing this lesson, you will learn how to separate the initialization of objects and the assignment of values to non-key object fields.

Step 4.1: Create a Test Class for the Services and Prices Form

You create a class for testing the `RSSVRepairPriceMaint` graph, which implements the logic for the Services and Prices (RS203000) form, as follows:

1. In the **Solution Explorer** panel, right-click the `PhoneRepairShop_Code.Tests` project, and select **Add > New Item**.
2. Select the *Visual C# item* | *Class* template, and type `RSSVRepairPriceMaintTests.cs` as the file name.
3. Click **Add**.
4. Specify the following `using` directives in the `RSSVRepairPriceMaintTests.cs` file.

```
using Xunit;
using PX.Data;
using PX.Objects.IN;
using PX.Tests.Unit;
using PhoneRepairShop;
```

5. Make the `RSSVRepairPriceMaintTests` class public and derived from `TestBase` as follows.

```
    public class RSSVRepairPriceMaintTests : TestBase
```

Step 4.2: Create a Test Method for the Services and Prices Form

In this test method, you will check the statuses of check boxes in different rows on the Services and Prices (RS203000) form, as well as the values calculated based on the input values.

You will create two repair items of the *BT* (battery) type and a repair item of the *BC* (back cover) type. When the `Required` field of the second *BT* repair item is set to *false*, the `Required` field of the first *BT* repair item must be automatically set to *false*, while the `Required` field of the *BC* repair item must remain unchanged. Also, when the `IsDefault` field of the second *BT* repair item is set to *true*, the `IsDefault` field of the first *BT* repair item must be automatically set to *false*, while the `IsDefault` field of the *BC* repair item must remain unchanged.

You will also create an `RSSVLabor` object and make sure that its extended price is equal to the `DefaultPrice` value multiplied by the `Quantity` value. Also, you need to make sure that the `Price` value of the work order is equal to the sum of the `BasePrice` values of all included repair items and the sum of the `ExtPrice` values of all included `RSSVLabor` objects.

To create this method, do the following:

1. In the `RSSVRepairPriceMaintTests` class, create a public void method, and name it *TestServicesAndPricesForm*.
2. Specify the *[Fact]* attribute for the method to indicate that this unit test is called without parameters.
3. Create an instance of the `RSSVRepairPriceMaint` graph by adding the following code.

   ```
   var graph = PXGraph.CreateInstance<RSSVRepairPriceMaint>();
   ```

4. Create a device to repair (an `RSSVDevice` object) and a service for repairing the device. You will use this device and service to create an `RSSVRepairPrice` object with details about a repair service based on this device and service.

   ```
   graph.Caches[typeof(RSSVDevice)].Insert(new RSSVDevice
   {
       DeviceCD = "Device1"
   });
   graph.Caches[typeof(RSSVRepairService)].Insert(new
    RSSVRepairService
   {
       ServiceCD = "Service1"
   });
   ```

   When creating an object, you specify values for its key fields. If you do not specify a value for an object's key field during the creation of the object, you cannot change the field value later. If you try to change the value of a key field, a new object with the new value will be created instead, and the old object will keep the old value.

5. Create an `RSSVRepairPrice` object that will contain details about the repair service.

   ```
   RSSVRepairPrice repairPrice =
       (RSSVRepairPrice)graph.Caches[typeof(RSSVRepairPrice)].
       Insert(new RSSVRepairPrice());
   ```

   You should have specified the values for the key fields `DeviceID` and `ServiceID`, but you did not, because the `DeviceID` and `ServiceID` fields are assigned their values from the `Current` properties of the `RSSVDevice` and `RSSVRepairService` types.

6. Create three repair items: two of the `Battery` type, and one of the `BackCover` type.

   ```
   InventoryItem battery1 = (InventoryItem)graph.
       Caches[typeof(InventoryItem)].Insert(new
       InventoryItem
       {
           InventoryCD = "Battery1"
       });
   graph.Caches[typeof(InventoryItemCurySettings)].Insert(new
       InventoryItemCurySettings
       {
           InventoryID = battery1.InventoryID,
           CuryID = "USD"
       });
   InventoryItemExt batteryExt1 =
       battery1.GetExtension<InventoryItemExt>();
   ```

```
            batteryExt1.UsrRepairItem = true;
            batteryExt1.UsrRepairItemType = RepairItemTypeConstants.Battery;
            graph.Caches[typeof(InventoryItem)].Update(battery1);

            InventoryItem battery2 =
             (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
                InventoryItem
             {
                 InventoryCD = "Battery2"
             });
            graph.Caches[typeof(InventoryItemCurySettings)].Insert(new
                InventoryItemCurySettings
                {
                    InventoryID = battery2.InventoryID,
                    CuryID = "USD"
                });
            InventoryItemExt batteryExt2 =
                battery2.GetExtension<InventoryItemExt>();
            batteryExt2.UsrRepairItem = true;
            batteryExt2.UsrRepairItemType = RepairItemTypeConstants.Battery;
            graph.Caches[typeof(InventoryItem)].Update(battery2);

            InventoryItem backCover1 =
             (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
                InventoryItem
             {
                 InventoryCD = "BackCover1"
             });
            graph.Caches[typeof(InventoryItemCurySettings)].Insert(new
                InventoryItemCurySettings
                {
                    InventoryID = backCover1.InventoryID,
                    CuryID = "USD"
                });
            InventoryItemExt backCoverExt1 =
             backCover1.GetExtension<InventoryItemExt>();
            backCoverExt1.UsrRepairItem = true;
            backCoverExt1.UsrRepairItemType = RepairItemTypeConstants.BackCover;
            graph.Caches[typeof(InventoryItem)].Update(backCover1);
```

You add an `InventoryItemCurySettings` object to the cache for each stock item to give users the ability to specify the price of the stock items.

After you insert a new object into the cache by calling the `PXCache.Insert` method, you can change non-key fields of the object and call the `PXCache.Update` method.

7. Create a non-stock item that represents the work to be done.

```
            InventoryItem work1 = (InventoryItem)graph.
                Caches[typeof(InventoryItem)].Insert(new InventoryItem
                {
                    InventoryCD = "Work1",
                    StkItem = false
                });
```

8. Create repair items based on the stock items, and configure them.

```
            // Configure the back cover repair item
            RSSVRepairItem repairItemBackCover1 =
                (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
                    new RSSVRepairItem
                    {
                        InventoryID = backCover1.InventoryID,
                        Required = true,
                        BasePrice = 10,
                        IsDefault = true
                    });

            // Configure the first battery repair item
            RSSVRepairItem repairItemBattery1 =
                (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
                    new RSSVRepairItem
                    {
                        InventoryID = battery1.InventoryID
                    });
            repairItemBattery1.Required = true;
            repairItemBattery1.BasePrice = 20;
            repairItemBattery1.IsDefault = true;
            graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBattery1);

            // Configure the second battery repair item
            RSSVRepairItem repairItemBattery2 =
                (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
                    new RSSVRepairItem
                    { InventoryID = battery2.InventoryID });
            repairItemBattery2.Required = false;
            repairItemBattery2.BasePrice = 30;
            repairItemBattery2.IsDefault = true;
            graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBattery2);
```

This code creates three repair items: one back cover, and two batteries.

9. Add the following code to check the values of the `Required` and `IsDefault` fields.

```
            // 2nd battery is not required -> 1st battery is also not required
            Assert.False(repairItemBattery1.Required);
            // 2nd batt is used by default -> 1st batt is not used by default
            Assert.False(repairItemBattery1.IsDefault);
            // The back cover's Required and Default fields are not affected
            Assert.True(repairItemBackCover1.Required);
            Assert.True(repairItemBackCover1.IsDefault);
```

10. Add the following code to ensure that the prices are calculated correctly.

```
            RSSVLabor labor = (RSSVLabor)graph.Caches[typeof(RSSVLabor)].
                Insert(new RSSVLabor
                {
                    InventoryID = work1.InventoryID,
                    DefaultPrice = 2,
                    Quantity = 3
                });
            Assert.Equal(6, labor.ExtPrice);
            Assert.Equal(66, repairPrice.Price);
```

Related Links

- *Access to a Custom Field from a Method*

Lesson Summary

In this lesson, you have learned how to separate the creation of objects with specifying key field values and the update of objects after assigning non-key fields their values.

Lesson 5: Testing the Graph for the Repair Work Orders Form

In this lesson, you will create a unit test method for checking the following features of the `RSSVWorkOrderEntry` graph, which implements the logic for the Repair Work Orders (RS301000) form:

- Initialization of a work order with repair items and labor items of the `RSSVRepairPrice` object that corresponds to the work order
- Prohibition of negative values of labor quantities
- Labor quantities (to be compared to minimum values)
- Prohibition of repair services that require preliminary checks in low-priority work orders

Lesson Objectives

As a result of completing this lesson, you will do the following:

- Test a graph that has a setup DAC
- Learn how to pick the proper object from the cache to make checks
- Learn how to clear the cache of errors

Step 5.1: Configure the Test Project

You change the configuration of the existing `PhoneRepairShop_Code.Tests` project as follows:

1. In the **Solution Explorer** panel, for the `PhoneRepairShop_Code.Tests` project, right-click the **References** item, and select **Add Reference**.
2. Select the **Browse** tab, and click **Browse**.
3. In the `PhoneRepairShop\Bin` folder, select the `PX.Common.Std.dll` file.
4. Click **OK**.

You have added the `PX.Common.Std.dll` library to the project so that you can use graph views in the code.

Step 5.2: Create a Test Class for the Repair Work Orders Form

Create a class for testing the `RSSVWorkOrderEntry` graph for the Repair Work Orders (RS301000) form as follows:

1. In the **Solution Explorer** panel, right-click the `PhoneRepairShop_Code.Tests` project, and select **Add > New Item**.
2. Select the *Visual C# item* | *Class* template, and type `RSSVWorkOrderEntryTests.cs` as the file name.
3. Click **Add**.
4. Specify the following `using` directives in the `RSSVWorkOrderEntryTests.cs` file.

```
using PhoneRepairShop;
using PX.Data;
using PX.Tests.Unit;
using PX.Objects.IN;
using System.Linq;
using Xunit;
```

5. Make the `RSSVWorkOrderEntryTests` class public and derived from `TestBase` as follows.

```
    public class RSSVWorkOrderEntryTests : TestBase
```

Step 5.3: Create a Test Method for the Repair Work Orders Form

Create the unit test method for the Repair Work Orders (RS301000) form as follows:

1. In the `RSSVWorkOrderEntryTests` class, create a public void method, and name it *TestRepairWorkOrdersForm*.
2. Specify the *[Fact]* attribute for the method to indicate that this unit test is called without parameters.
3. Initialize the settings for the `RSSVWorkOrderEntry` graph by adding the following code.

```
            Setup<RSSVWorkOrderEntry>(new RSSVSetup());
```

   The `RSSVSetup` DAC contains settings for the `RSSVWorkOrderEntry` graph. You must call the `Setup<>` generic method to initialize an `RSSVSetup` object.

   The `Setup<Graph>(params IBqlTable[] items)` method mocks the Acumatica ERP setup preferences. This method is called before the creation of a graph instance. You must initialize and bind setup DACs for the test first; otherwise, the setup DACs will not be used by the graph. The `Setup` method should not be used to initialize non-setup DACs.

4. Create an instance of the `RSSVWorkOrderEntry` graph.

```
            var graph = PXGraph.CreateInstance<RSSVWorkOrderEntry>();
```

5. Create an `RSSVDevice` object and two `RSSVRepairService` objects (one of them is the main object, and the other is auxiliary).

```
            RSSVDevice device = (RSSVDevice)graph.Caches[typeof(RSSVDevice)].
                Insert(new RSSVDevice { DeviceCD = "Device1" });

            RSSVRepairService repairService =
                (RSSVRepairService)graph.Caches[typeof(RSSVRepairService)].
                Insert(new RSSVRepairService
                {
                    ServiceCD = "Service1"
                });
            RSSVRepairService repairService2 =
                (RSSVRepairService)graph.Caches[typeof(RSSVRepairService)].
                Insert(new RSSVRepairService
                {
                    ServiceCD = "Service2"
                });
```

6. Create an `RSSVRepairPrice` object, and initialize it with the `RSSVDevice` object and the main `RSSVRepairService` object.

```
            graph.Caches[typeof(RSSVRepairPrice)].Insert(new RSSVRepairPrice
            {
                DeviceID = device.DeviceID,
                ServiceID = repairService.ServiceID
            });
```

7. Create two stock items and one non-stock item, and make the two stock items repair items.

```
            InventoryItem battery1 =
                (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
                InventoryItem
                {
                    InventoryCD = "Battery1"
                });
            InventoryItemExt batteryExt1 =
                battery1.GetExtension<InventoryItemExt>();
            batteryExt1.UsrRepairItem = true;
            batteryExt1.UsrRepairItemType = RepairItemTypeConstants.Battery;
            graph.Caches[typeof(InventoryItem)].Update(battery1);

            InventoryItem backCover1 =
                (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
             InventoryItem
                {
                    InventoryCD = "BackCover1"
                });
            InventoryItemExt backCoverExt1 =
             backCover1.GetExtension<InventoryItemExt>();
            backCoverExt1.UsrRepairItem = true;
            backCoverExt1.UsrRepairItemType =
                RepairItemTypeConstants.BackCover;
            graph.Caches[typeof(InventoryItem)].Update(backCover1);

            InventoryItem work1 =
                (InventoryItem)graph.Caches[typeof(InventoryItem)].Insert(new
             InventoryItem
                {
                    InventoryCD = "Work1",
                    StkItem = false
                });
```

8. Create two repair items based on the two stock items.

```
            RSSVRepairItem repairItemBackCover1 =
                (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
                    new RSSVRepairItem
                    {
                        DeviceID = device.DeviceID,
                        ServiceID = repairService.ServiceID
                    });
            repairItemBackCover1.InventoryID = backCover1.InventoryID;
            repairItemBackCover1.Required = true;
            repairItemBackCover1.BasePrice = 10;
            repairItemBackCover1.IsDefault = true;
            repairItemBackCover1.RepairItemType =
                backCoverExt1.UsrRepairItemType;
            graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBackCover1);

            RSSVRepairItem repairItemBattery1 =
                (RSSVRepairItem)graph.Caches[typeof(RSSVRepairItem)].Insert(
                    new RSSVRepairItem
                {
                    DeviceID = device.DeviceID,
                    ServiceID = repairService.ServiceID
                });
            repairItemBattery1.InventoryID = battery1.InventoryID;
            repairItemBattery1.Required = true;
            repairItemBattery1.BasePrice = 20;
            repairItemBattery1.IsDefault = true;
            repairItemBattery1.RepairItemType = batteryExt1.UsrRepairItemType;
            graph.Caches[typeof(RSSVRepairItem)].Update(repairItemBattery1);
```

9. Create an `RSSVLabor` object based on the non-stock item.

```
            RSSVLabor labor = (RSSVLabor)graph.Caches[typeof(RSSVLabor)].
                Insert(new RSSVLabor
                {
                    InventoryID = work1.InventoryID,
                    DeviceID = device.DeviceID,
                    ServiceID = repairService.ServiceID
                });
            labor.DefaultPrice = 2;
            labor.Quantity = 3;
            graph.Caches[typeof(RSSVLabor)].Update(labor);
```

10. Create a work order with the same `DeviceID` and `ServiceID` values as the `RSSVRepairPrice` object created in Instruction 6.

```
            RSSVWorkOrder workOrder = (RSSVWorkOrder)graph.
                Caches[typeof(RSSVWorkOrder)].Insert(new RSSVWorkOrder());
            workOrder.DeviceID = device.DeviceID;
            workOrder.ServiceID = repairService.ServiceID;
            graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
```

11. As the first test, make sure that there are two `RSSVWorkOrderItem` objects that have been created based on the two `RSSVRepairItem` objects and an `RSSVWorkOrderLabor` object that has been created based on the `RSSVLabor` object.

```
            Assert.Equal(2, graph.RepairItems.Select().Count);
            Assert.Single(graph.Labor.Select());
```

12. As the second test, ensure that the change of the `ServiceID` field of the work order does not affect the `RepairItems` and `Labor` views.

```
            workOrder.ServiceID = repairService2.ServiceID;
            graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
            Assert.Equal(2, graph.RepairItems.Select().Count);
            Assert.Single(graph.Labor.Select());
```

13. Restore the `ServiceID` value.

```
            workOrder.ServiceID = repairService.ServiceID;
            graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
```

14. Obtain the `RSSVWorkOrderLabor` object (the value of its `Quantity` field must be *3*), and try to assign its `Quantity` field a negative value. An error message must be attached to the `Quantity` field instead.

```
            RSSVWorkOrderLabor woLabor = graph.Labor.SelectSingle();
            Assert.Equal(3, woLabor.Quantity);
            woLabor.Quantity = -1;
            graph.Caches[typeof(RSSVWorkOrderLabor)].Update(woLabor);
            PXFieldState fieldState =
                (PXFieldState)graph.Caches[typeof(RSSVWorkOrderLabor)].
                GetStateExt<RSSVWorkOrderLabor.quantity>(woLabor);
            Assert.Equal(PhoneRepairShop.Messages.QuantityCannotBeNegative,
                fieldState.Error);
            Assert.Equal(PXErrorLevel.Error, fieldState.ErrorLevel);
            graph.Labor.Cache.Clear();
```

- You can use the `PXCache.GetError`, `PXCache.GetErrors`, or `PXCache.GetStateExt` method to get the errors attached to a field.
- The `PXCache.Clear` method is called to clear the cache of the errors.

15. Assign the `Quantity` field of the `RSSVWorkOrderLabor` object a value that is less than the value of the corresponding `RSSVLabor.Quantity` field. A warning message must be attached to the `RSSVWorkOrderLabor.Quantity` field, and the field must be assigned the value of the corresponding `RSSVLabor.Quantity` field.

```
            woLabor.Quantity = 1;
            graph.Caches[typeof(RSSVWorkOrderLabor)].Update(woLabor);
            woLabor = (RSSVWorkOrderLabor)graph.
                Caches[typeof(RSSVWorkOrderLabor)].Locate(woLabor);
            fieldState = (PXFieldState)graph.
                Caches[typeof(RSSVWorkOrderLabor)].
                GetStateExt<RSSVWorkOrderLabor.quantity>(woLabor);
            Assert.Equal(PhoneRepairShop.Messages.QuantityToSmall,
                fieldState.Error);
            Assert.Equal(PXErrorLevel.Warning, fieldState.ErrorLevel);
            Assert.Equal(3, woLabor.Quantity);
```

- You can use either the `PXCache.GetWarning` method or the `PXCache.GetStateExt` method to get the warnings attached to a field.
- After actions that generate exceptions or cancel an assignment, you can use the `PXCache.Locate` method to obtain from the cache the proper `RSSVWorkOrderLabor` object that you want to check for warnings or errors.

16. Configure the second repair service `repairService2` to require a preliminary check, and assign it to the work order. Set the priority of the work order to *Low*. Make sure that an error message is attached to the `Priority` field of the work order informing the user that the priority of the work order is too low.

```
            repairService2.PreliminaryCheck = true;
            graph.Caches[typeof(RSSVRepairService)].Update(repairService2);
            workOrder.ServiceID = repairService2.ServiceID;
            workOrder.Priority = WorkOrderPriorityConstants.Low;
            graph.Caches[typeof(RSSVWorkOrder)].Update(workOrder);
            workOrder = (RSSVWorkOrder)graph.Caches[typeof(RSSVWorkOrder)].
                Locate(workOrder);
            fieldState = (PXFieldState)graph.Caches[typeof(RSSVWorkOrder)].
                GetStateExt<RSSVWorkOrder.priority>(workOrder);
            Assert.Equal(PhoneRepairShop.Messages.PriorityTooLow, fieldState.Error);
            Assert.Equal(PXErrorLevel.Error, fieldState.ErrorLevel);
```

Related Links

- *PXCache.GetStateExt(Object,String)*

Lesson Summary

In this lesson, you have used the `PXCache.Clear` method when an error occurred and the `PXCache.Locate` method to select the proper object from the cache to perform a check. Also, you have used the `Setup<>` generic method to initialize the setup DACs of a graph.

Appendix: Reference Implementation

You can find the reference implementation of the customization described in this course in the `Customization\T280` folder of the *Help-and-Training-Examples* repository in Acumatica GitHub.

Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course

If for some reason you cannot complete the instructions in *Step 2: Deploying the Needed Acumatica ERP Instance for the Training Course*, you can create an Acumatica ERP instance as described in this topic and manually publish the needed customization project as described in *Appendix: Publishing the Required Customization Project*.

You deploy an Acumatica ERP instance and configure it as follows:

1. To deploy a new application instance, open the Acumatica ERP Configuration Wizard, and do the following:
   a. On the Database Configuration page, type the name of the database: `PhoneRepairShop`.
   b. On the Tenant Setup page, set up a tenant with the *I100* data inserted by specifying the following settings:
      - **Login Tenant Name**: `MyTenant`
      - **New**: Selected
      - **Insert Data**: *I100*
      - **Parent Tenant ID**: *1*
      - **Visible**: Selected
   c. On the **Instance Configuration** page, in the **Local Path of the Instance** box, select a folder that is outside of the `C:\Program Files (x86)` or `C:\Program Files` folder. We recommend that you store the website folder outside of these folders to avoid an issue with permission to work in these folders when you perform customization of the website.

   The system creates a new Acumatica ERP instance, adds a new tenant, and loads the selected data to it.
2. Sign in to the new tenant by using the following credentials:
   - Username: `admin`
   - Password: `setup`

   Change the password when the system prompts you to do so.
3. In the top right corner of the Acumatica ERP screen, click the username and then click **My Profile**. On the **General Info** tab of the *User Profile* (SM203010) form, which the system has opened, select *YOGIFON* in the **Default Branch** box; then click **Save** on the form toolbar.

   In subsequent sign-ins to this account, you will be signed in to this branch.
4. Optional: Add the *Customization Projects* (SM204505) and *Generic Inquiry* (SM208000) forms to your favorites. For details about how to add a form to favorites, see *Managing Favorites: General Information*.

Appendix: Publishing the Required Customization Project

If for some reason you cannot complete the instructions in *Step 2: Deploying the Needed Acumatica ERP Instance for the Training Course*, you can create an Acumatica ERP instance as described in *Appendix: Deploying the Needed Acumatica ERP Instance for the Training Course* and manually publish the needed customization project as described in this topic.

You load the customization project with the results of the *T220 Data Entry and Setup Forms* training course and publish this project as follows:

1. On the *Customization Projects* (SM204505) form, create a project with the name *PhoneRepairShop*, and open it.
2. In the menu of the Customization Project Editor, click **Source Control > Open Project from Folder**.
3. In the dialog box that opens, specify the path to the `Customization\T220\PhoneRepairShop` folder, which you have downloaded from Acumatica GitHub, and click **OK**.
4. Bind the customization project to the source code of the extension library as follows:
   a. Copy the `Customization\T220\PhoneRepairShop_Code` folder to the `App_Data\Projects` folder of the website.

      By default, the system uses the `App_Data\Projects` folder of the website as the parent folder for the solution projects of extension libraries.

      If the website folder is outside of the `C:\Program Files (x86)` and `C:\Program Files` folders, we recommend that you use the `App_Data\Projects` folder for the project of the extension library.

   b. Open the solution, and build the `PhoneRepairShop_Code` project.
   c. Reload the Customization Project Editor.
   d. In the menu of the Customization Project Editor, click **Extension Library > Bind to Existing**.
   e. In the dialog box that opens, specify the path to the `App_Data\Projects\PhoneRepairShop_Code` folder, and click **OK**.
5. In the menu of the Customization Project Editor, click **Publish > Publish Current Project**.

   The **Modified Files Detected** dialog box opens before publication because you have rebuilt the extension library in the `PhoneRepairShop_Code` Visual Studio project. The `Bin\PhoneRepairShop_Code.dll` file has been modified, and you need to update it in the project before the publication.

The published customization project contains all changes to the Acumatica ERP website and database that have been performed in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses. This project also contains the customization plug-ins that fill in the tables created in the *T200 Maintenance Forms*, *T210 Customized Forms and Master-Detail Relationship*, and *T220 Data Entry and Setup Forms* training courses with the custom data entered in these training courses.

For details about the customization plug-ins, see *To Add a Customization Plug-In to a Project*. The creation of customization plug-ins is outside of the scope of this course.