Table of Contents

[Understanding Typemock Isolator 7](#_Toc408479807)

[Client-Side Integration 7](#_Toc408479808)

[Integration with MS Visual Studio 7](#_Toc408479809)

[Integration with Third-Party Coverage and Profiling Tools 8](#_Toc408479810)

[Integration with Test Runners 8](#_Toc408479811)

[Integration with Unit Test Frameworks 9](#_Toc408479812)

[Server-Side Integration 9](#_Toc408479813)

[Integration with Build Scripts 9](#_Toc408479814)

[Integration with Build Servers and Continuous Integration Servers 9](#_Toc408479815)

[Testing Methodology 11](#_Toc408479816)

[Types of Tests 11](#_Toc408479817)

[Unit Tests 11](#_Toc408479818)

[Integration Tests 11](#_Toc408479819)

[Approaches to Testing 12](#_Toc408479820)

[Test-Driven Development 12](#_Toc408479821)

[Code-Driven Development 12](#_Toc408479822)

[Installing Typemock Isolator 13](#_Toc408479823)

[Installing Typemock Isolator Client 13](#_Toc408479824)

[Installing Typemock Isolator Server 13](#_Toc408479825)

[Auto-Deploy 13](#_Toc408479826)

[Manual Installation 14](#_Toc408479827)

[Upgrading Typemock Isolator Tests 15](#_Toc408479828)

[Getting Started 17](#_Toc408479829)

[Step 1. Providing the License for the First Time 17](#_Toc408479830)

[Step 2. Getting Automatic Suggestion of Tests 17](#_Toc408479831)

[Step 3. Running Tests 18](#_Toc408479832)

[Automatic Test Suggestion 19](#_Toc408479833)

[Viewing Automatically Suggested Tests 19](#_Toc408479834)

[Getting Automatic Test Suggestions for a Specific Code 20](#_Toc408479835)

[Configuring Automatic Test Suggestion 22](#_Toc408479836)

[Mocking 23](#_Toc408479837)

[Overview 23](#_Toc408479838)

[Supported Fakeable MSCorlib Types 23](#_Toc408479839)

[Basic Structure of a Unit Test 23](#_Toc408479840)

[Mocking API Principles 25](#_Toc408479841)

[Auto-Recursive Fakes 25](#_Toc408479842)

[Argument Checking is Off by Default 25](#_Toc408479843)

[Faking Interfaces 25](#_Toc408479844)

[Faking Dependencies 25](#_Toc408479845)

[True Properties and Indexers 25](#_Toc408479846)

[Setting Up a Unit Test Project 26](#_Toc408479847)

[Step 1. Copying Typemock Isolator DLLs 26](#_Toc408479848)

[Step 2. Creating a Unit Test Project 26](#_Toc408479849)

[Step 3. Preparing to Write a Unit Test 26](#_Toc408479850)

[Step 4. Adding the Isolated Attribute to Tests 27](#_Toc408479851)

[Step 5. Defining whether to Mock Virtual and Non-Virtual Methods 28](#_Toc408479852)

[Faking an Instance 29](#_Toc408479853)

[When to Use 29](#_Toc408479854)

[Faking Objects 29](#_Toc408479855)

[Faking Future Instances 31](#_Toc408479856)

[Faking All Instances of a Type 33](#_Toc408479857)

[Faking Dependencies 36](#_Toc408479858)

[Faking Constructors 38](#_Toc408479859)

[Setting the Default Behavior of a Fake Object 47](#_Toc408479860)

[Changing Method Behavior 50](#_Toc408479861)

[Modifying the Method Behavior 50](#_Toc408479862)

[Faking Behavior for Live Objects 56](#_Toc408479863)

[Simulating a Sequenced Behavior 58](#_Toc408479864)

[Faking Method Chaining 60](#_Toc408479865)

[Faking Methods Based on Call Arguments 62](#_Toc408479866)

[Controlling Property Behavior 69](#_Toc408479867)

[Controlling PInvoke Methods 71](#_Toc408479868)

[Controlling Static Methods Behavior 73](#_Toc408479869)

[Faking Extension Methods 75](#_Toc408479870)

[Controlling Non-Public Methods 76](#_Toc408479871)

[Faking Ref and Out Parameters 81](#_Toc408479872)

[Faking Non-Public Ref and Out Parameters 82](#_Toc408479873)

[Controlling Collections and Indexers 83](#_Toc408479874)

[Faking LINQ Queries 86](#_Toc408479875)

[Calling Tested Methods 88](#_Toc408479876)

[Invoking Private Methods 88](#_Toc408479877)

[Invoke Static Constructors 91](#_Toc408479878)

[Firing Events 92](#_Toc408479879)

[Creating Instances of Types with Private Constructors 93](#_Toc408479880)

[Validating 93](#_Toc408479881)

[Verifying Method Calls 94](#_Toc408479882)

[Verifying Static Methods 96](#_Toc408479883)

[Verifying Non-Public Methods 97](#_Toc408479884)

[Verifying the Number of Times a Method Was Called 100](#_Toc408479885)

[Verifying that a Future Instance Was Created 101](#_Toc408479886)

[Verifying the Number of Created Instances 102](#_Toc408479887)

[Samples and Cookbook 103](#_Toc408479888)

[Running Tests 115](#_Toc408479889)

[Understanding Test Navigator 115](#_Toc408479890)

[Understanding Icons 116](#_Toc408479891)

[Understanding Tabs 117](#_Toc408479892)

[Summary Status of the Entire Solution 117](#_Toc408479893)

[Opening the Test Navigator Pane 117](#_Toc408479894)

[Running an Individual Test 118](#_Toc408479895)

[Running All Tests in a Group 120](#_Toc408479896)

[Running All Pending Tests 121](#_Toc408479897)

[Running Unit Tests 122](#_Toc408479898)

[Running All Tests in a Project 122](#_Toc408479899)

[Running Pending Unit Tests 122](#_Toc408479900)

[Running All Unit Tests 122](#_Toc408479901)

[Re-Running the Recently Run Test 122](#_Toc408479902)

[Setting Pending Unit Tests to Run Automatically after Build 122](#_Toc408479903)

[Setting Integration Tests to Run Automatically when Idle 123](#_Toc408479904)

[Canceling a Test Run 123](#_Toc408479905)

[Running a Test in the Sandbox 123](#_Toc408479906)

[Ignoring Tests 123](#_Toc408479907)

[Configuring SmartRunner 124](#_Toc408479908)

[Reusing the Test Runner Process 124](#_Toc408479909)

[Using a Shadow Folder for Running NUnit Tests 124](#_Toc408479910)

[Enabling and Disabling Output when Running Tests 125](#_Toc408479911)

[Specifying the Test Architecture 125](#_Toc408479912)

[Disabling SmartRunner 125](#_Toc408479913)

[Analyzing and Fixing Bugs 127](#_Toc408479914)

[Identifying Bugs 127](#_Toc408479915)

[Starting from the Failed Test 127](#_Toc408479916)

[Starting from the Method under Test 128](#_Toc408479917)

[Fake Method Highlighting 130](#_Toc408479918)

[Debugger Enhancements 131](#_Toc408479919)

[Debugging a Failed Test 132](#_Toc408479920)

[Adding a Breakpoint Manually 134](#_Toc408479921)

[Adding a Breakpoint Automatically 134](#_Toc408479922)

[Debugging the Last Debugged or Run Test 134](#_Toc408479923)

[Viewing Code Coverage 135](#_Toc408479924)

[Viewing Code Coverage for a Specific Method 135](#_Toc408479925)

[Viewing Code Coverage for a Specific Test 136](#_Toc408479926)

[Viewing Code Coverage for the Entire Solution 136](#_Toc408479927)

[Viewing Code Coverage for Recently Changed Classes 137](#_Toc408479928)

[Integrating Typemock Isolator 139](#_Toc408479929)

[Integrating with the Client 139](#_Toc408479930)

[Controlling Integration with Code Coverage Tools 139](#_Toc408479931)

[Integrating with NCover Version 4.5 and Above 139](#_Toc408479932)

[Integrating with the Server 140](#_Toc408479933)

[Integrating with Custom Build Servers 140](#_Toc408479934)

[Integrating with MSBuild 141](#_Toc408479935)

[Integrating with NAnt 144](#_Toc408479936)

[Integrating with CruiseControl.NET 148](#_Toc408479937)

[Integrating with Jenkins CI 148](#_Toc408479938)

[Integrating with TeamCity 148](#_Toc408479939)

[Integrating with MS Team Foundation Server 149](#_Toc408479940)

[Extending Typemock Isolator 159](#_Toc408479941)

[Adding Custom Attributes 159](#_Toc408479942)

[Typemock Isolator Integration API 162](#_Toc408479943)

[Sample 1: Basic Usage 162](#_Toc408479944)

[Sample 2: Linking with Code Coverage Profiler and Running the Tests 164](#_Toc408479945)

[Sample 3: Using AutoDeploy 165](#_Toc408479946)

[Upgrading Your License 167](#_Toc408479947)

[Appendix A: Useful Shortcuts 169](#_Toc408479948)

# 

# Understanding Typemock Isolator

Typemock Isolator is a tool that enables C# and Visual Basic developers to write unit tests in an easy, reliable, and fast manner. Typemock Isolator is tightly integrated with MS Visual Studio. You can write, run, and debug tests and see code coverage from within the MS Visual Studio user interface.

In addition, Typemock Isolator provides an Application Programming Interface (API). This API helps you write simple and human readable tests which are completely isolated from the production code.

Typemock Isolator consists of the following components:

| Component | Description |
| --- | --- |
| Typemock Isolator Client | Integrates with MS Visual Studio, unit test frameworks, unit test runners and code coverage tools. Typemock Isolator Client enables you to run and manage tests on a local computer. |
| Typemock Isolator Server | Integrates with build servers and continuous integration servers, such as TeamCity and Microsoft Team Foundation Server, and any build script such as MSBuild or NAnt. Typemock Isolator Server enables you to run tests as part of the server-side automated build. |
| Typemock Isolator API | Based on Arrange-Act-Assert (AAA) principles and enables you to isolate your tests from the production code. |

## Client-Side Integration

Typemock Isolator integrates with the following applications installed on your computer:

### Integration with MS Visual Studio

Typemock Isolator provides integration with MS Visual Studio starting from MS Visual Studio 2010.

|  |  |
| --- | --- |
| ! | Note  If you use a version of MS Visual Studio between 2005 and 2010, install Typemock Isolator version 7.5. |

After Typemock Isolator is installed, the Typemock menu is added to the Visual Studio menu bar. This menu enables you to access Typemock Isolator features from within the MS Visual Studio user interface.

Using Typemock Isolator, you can view:

* The status of all unit tests
* The code coverage of a specific method
* The unit tests that call a specific method and the status of each test
* The code of a specific test
* Code coverage of the entire solution
* Code coverage of classes that were recently changed
* Suggested unit tests

### Integration with Third-Party Coverage and Profiling Tools

Although Typemock Isolator provides powerful code coverage capabilities out-of-the-box, you might need to use an external code coverage tool or other profiling tools for memory and performance analysis.

Typemock Isolator integrates with the following code coverage and profiler tools:

* Visual Studio Code Coverage (Visual Studio 2005 - 2013)
* Microsoft Team Foundation Server (2010, 2012, and 2013)
* NCover. Typemock Isolator integrates with NCover up to version 4.5 out-of-the-box, and no additional configuration is required. If you use NCover version 4.5 or above, you need to manually enable the integration as described in.
* Partcover
* JetBrains dotTrace, dotCover
* ExactMagic TestMatrix
* Smartbear AQtime
* DevPartner
* Clover.NET
* CoverageEye
* NCrunch
* TestDriven.NET
* PostSharp

### Integration with Test Runners

Typemock Isolator provides its own test runner called SmartRunner. In addition, Typemock Isolator provides an out-of-the-box integration with all major unit test runners created by third-party developers:

* MS Visual Studio integrated runners:
* VS Runner
* JetBrains Runner/ ReSharper
* DevExpress Runner
* nunit-runner
* mstest runner
* ncrunch
* testdriven.net
* resharper

### Integration with Unit Test Frameworks

Typemock Isolator provides an out-of-the-box integration with all unit test frameworks. The most complete integration is provided for Nunit and MSTest frameworks.

## Server-Side Integration

When you build a project on a server in addition to a build which runs on your local computer, you need to run tests as part of the server-side automated build (an exact way you run tests depends on the build server you use).

### Integration with Build Scripts

Typemock Isolator supports integration with the following build scripts on any server:

* MSBuid. See [Integrating with MSBuild](#_D2HTopic_336).
* NAnt. See [Integrating with NAnt](#_D2HTopic_343).
* Final Builder. See [Integrating with Final Builder](#_D2HTopic_364).

### Integration with Build Servers and Continuous Integration Servers

Typemock Isolator supports integration with the following build servers and continuous integration servers:

* TeamCity. See [Integrating with TeamCity](#_D2HTopic_350).
* MS Foundation Server. See [Integrating with MS Foundation Server](#_D2HTopic_353).
* Custom build servers. See [Integrating with Custom Build Servers](#_D2HTopic_331).

# Testing Methodology

This section provides a general background on testing and explain what a typical testing process involves.

## Types of Tests

There are the following types of tests:

* Unit tests. See [Unit Tests](#_D2HTopic_32).
* Integration tests. See [Integration Tests](#_D2HTopic_33).

### Unit Tests

A unit test is a test that meets the following requirements:

* **Repeatable:** you can rerun the same test as many times as you want.
* **Consistent:** every time you run a test, you get the same result. For example, testing threads cannot be considered unit test because using threads can produce inconsistent results.
* **In Memory:** the test has no hard dependencies on anything not allocated in memory (such as file system, databases, or network).
* **Fast:** running a test should take less than half a second.
* **Checking one single concern or use case in the system:** when you check more than use case, identifying the issue becomes more difficult.

If you break any of these principles, you increase the chance that developers either do not trust the test results due to repeated false failures by the tests, or developers do not want to run the tests at all because of the significant amount of time that the tests require.

### Integration Tests

An integration test is any test that cannot be described as unit test.

An integration test might:

* Use system dependent values that change dynamically (such as DateTime.Now or Environment.MachineName)
* Create objects that cannot be controlled in a reliable manner. For example, a test can have a little control over threads or generators of random numbers.
* Make calls to external systems or have dependencies on a local machine (for example, when a test calls a Web service or uses local configuration files).
* Test multiple scenarios as part of a single test case (for example, when a test checks database integrity, system configuration, protocols, and system logic at once).

Therefore, integration tests will most likely:

* Significantly slower than unit tests
* Require significantly amount to run completely
* Make the collection and analysis of test results more problematic

## Approaches to Testing

There are the following approaches to testing:

* Test-driven development, when the process of developing a new feature starts with writing a test that checks this feature. See [Test-Driven Development](#_D2HTopic_35).
* Code-driven development, when tests are written after the code is developed. See [Code-Driven Development](#_D2HTopic_36).

### Test-Driven Development

In a test-driven development cycle, the process of adding a new feature or improving an existing feature works as follows:

1. Writing a test based on the feature requirements.
2. Running the test for the first time and verifying that it fails.
3. Writing the code for the new feature.
4. Running the test.
5. Refactoring the code.
6. Rerunning the test.

Typemock Isolator assists you in a test-driven development by providing powerful mocking capabilities and automating important tasks. For example, Typemock Isolator automatically runs the test for the first time (before the code is written), verifies that the test fails, and runs the required tests after the code is written.

In addition, Typemock Isolator provides the incremental code coverage feature. If you are following the test-driven development practices, your incremental coverage will always be 100%.

### Code-Driven Development

In a code-driven development cycle, the process of adding a new feature or improving an existing feature works as follows:

1. Writing the code for the new feature.
2. Writing a test to check the new feature.
3. Running the test.

Typemock Isolator assists you in a test-driven development by providing powerful mocking capabilities and automatically running required tests.

In addition, Typemock Isolator provides the incremental code coverage feature. If you are using code-driven development, the incremental coverage helps you verify that you do not forget to write a test for a method.

# Installing Typemock Isolator

The installation process involves installation of the client-side and server-side components.

## Installing Typemock Isolator Client

The process of installing Typemock Isolator Client is performed by a standard installation wizard. By default, Typemock Isolator Client installs the following components on your computer:

* Visual Studio integration
* Examples
* Documentation

To install Typemock Isolator Client:

1. Run the Typemock Isolator installer.
2. In the Welcome screen, click Next.
3. In the End-User License Agreement screen, click I Agree.
4. In the Choose Setup Type screen, select one of the following:

* One Click Setup, when you want to install all the components.
* Custom Install, when you want to select which components should be installed. After you selected components, click Next, and in the Ready to install screen, click Next.

1. In the Completing the Setup Wizard, click Finish.

## Installing Typemock Isolator Server

Depending on whether the build agents that run the tests have administrator privileges on the server, you might need to use one of the following installation methods:

* Auto-deploy, when you have administrator privileges. See [Auto-Deploy](#_D2HTopic_40).
* Manual installation, when you do not have administrator privileges. See [Manual Installation](#_D2HTopic_41).

|  |  |
| --- | --- |
| ! | Note  Install only the latest version on the agents and the server. |

### Auto-Deploy

|  |  |
| --- | --- |
| ! | Note  The following procedure represents the recommended way to install Typemock Isolator Server. |

Typemock Isolator provides a backward compatibility with older versions. For example, when Typemock Isolator version 8 is installed, you can run tests created with Typemock Isolator version 7 or 7.5. However, when you have Typemock Isolator version 7 installed, you cannot run tests created with Typemock Isolator version 8. This means that if you manually install Typemock Isolator, you need to verify that a newer version of Typemock Isolator is not already installed. Otherwise, when you attempt to use the old version of Typemock Isolator to run tests created with the newer version, your tests will fail.

The Auto-Deploy mode automatically checks whether a newer version is already installed. Only if there is no a newer version on the server, the version will be installed. The Auto-Deploy mode ensures that only the latest version is installed, and if there are different versions installed on the same server, they will never conflict with each other.

To install Typemock Isolator in the Auto-Deploy mode:

1. Verify that you are running the build agents as admin.
2. Copy all the files from *Typemock installation folder*/AutoDeploy to your source control repository while preserving the structure of the AutoDeploy folder.

This folder contains integration tasks for various build servers and DLLs required to run tests.

1. In your test project, add references to the required DLLs (the best practice is to keep these files in your source repository):

* TypeMock.dll
* Typemock.ArrangeActAssert.dll
* Typemock.Isolator.VisualBasic.dll

1. In your build script, run the TypemockRegister task from the source control repository with the AutoDeploy option set to true.

The following example shows how you can run the TypemockRegister task using MSBuild:

|  |
| --- |
| <Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  <PropertyGroup>   <TypeMockLocation>C:\Sources\\TypeMock\AutoDeploy</TypeMockLocation>  </PropertyGroup>   <Import Project ="$(TypeMockLocation)\TypeMock.MSBuild.Tasks"/>  <Target Name="RegisterTypeMock">  **<TypeMockRegister Company ="TypeMock" AutoDeploy="True"/>**  </Target> </Project> |

### Manual Installation

With the manual installation, when a newer version of Typemock Isolator is released, you have to upgrade Typemock Isolator manually on each build server and each agent. Otherwise, tests that you created using a previous version of Typemock Isolator, will fail

To install Typemock Isolator manually:

1. Download the server installer and install Typemock Isolator on the server as described in [Installing Typemock Isolator Client](#_D2HTopic_38).
2. Reference the Typemock Isolator’s DLLs in the source control from your test project (the best practice).
3. In the project build scripts, use the tasks and actions from the Typemock Isolator’s installation folder as described in the following sections:

* [Integrating with MSBuild](#_D2HTopic_336)
* [Integrating with NAnt](#_D2HTopic_343)
* [Integrating with Custom Build Servers](#_D2HTopic_331)

## Upgrading Typemock Isolator Tests

After you upgraded to a newer version of Typemock Isolator, you need to update references in your projects to make them point to the DLLs of the newly installed version.

To update references:

1. After the installation of the new version is completed, in MS Visual Studio, select Typemock > Update References.

The Update Typemock References window is displayed. This window shows how many projects contain references to the old DLLs.

1. Click Update.

One of the following will happen:

* If you installed Typemock Isolator to the default location (c:\Program Files (x86)\Typemock\Isolator\<version>), the reference will point to the installation folder of the latest version.
* If you installed Typemock Isolator to another location (which is the best practice), the older files will be replaced with the files of the latest version.

# Getting Started

To get started with Typemock Isolator, you need to perform the following steps:

1. Providing the license for the first time. See [Step 1. Providing the License for the First Time](#_D2HTopic_44).
2. Getting automatic suggestion of tests. See [Step 2. Getting Automatic Suggestion of Tests](#_D2HTopic_45).
3. Running a test. See [Step 3. Running Tests](#_D2HTopic_46).

## Step 1. Providing the License for the First Time

After installing Typemock Isolator and opening a solution in MS Visual Studio for the first time, you will be prompted to provide a Typemock Isolator license.

To provide the license for the first time:

1. In the Typemock window, in the Company field, enter the name of your company.
2. Do one of the following:

* If you already have a license code, enter it into the License Code field and click Set License.
* If you do not have a license code, click I need a license.

1. If you requested a license, provide your personal and contact information.
2. Do one of the following:

* If you want to immediately receive a 14-day trial, click Send me a License. The license code will be sent you by email.
* If you do not want to immediately receive a 14-day trial, click Not Now. Typemock Isolator will run with a 7-day trial license.

After your trial license expired, you can acquire a perpetual license and provide it as described in [Upgrading Your License](#_D2HTopic_374).

## Step 2. Getting Automatic Suggestion of Tests

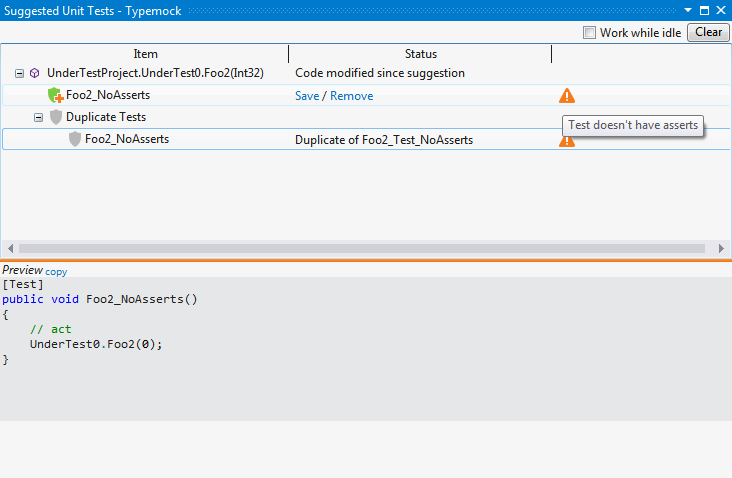
When you practice a code-driven approach, Typemock Isolator can automatically suggest tests for a method for which no tests are created yet or for a method which is not 100% covered yet.

To run automatically suggested tests in a safe isolated environment, Typemock Isolator automatically adds the [SafeNet] attribute to the class.

To get automatic suggestion of tests:

1. Right-click on the method for which you want Typemock Isolator to suggest a test.
2. From the context menu, select Suggest Unit Tests for Method.

The Typemock Suggest window is displayed.



1. On the right from the test that you want to save, click Save.

## Step 3. Running Tests

Typemock Isolator provides a real time impact analysis that runs required tests after you built the project.

For more information, see [Running Tests](#_D2HTopic_282).

# Automatic Test Suggestion

Typemock Isolator can automatically suggest tests for a method for which no tests are created yet or which is not 100% covered yet. Typemock Isolator can generate tests in following cases:

* When the computer is idle. In this case, Typemock Isolator creates tests for classes that are displayed in the Incremental Coverage window.
* When you explicitly instruct Typemock Isolator to suggest tests for the method.

To run automatically suggested tests in a safe isolated environment, Typemock Isolator automatically adds the [SafeNet] attribute to the class.

## Viewing Automatically Suggested Tests

If you chose to suggest automatic tests when the computer is idle, Typemock Isolator creates tests during the specified inactivity period. After the tests are suggested, the Typemock Suggest window is displayed. Each test is accompanied with an icon that represents the status of the test:

| Icon | Description |
| --- | --- |
|  | Low quality test because:   * There is no assertion in the method, or * The method uses a private method, or * The metod is very simple |
|  | Such a test already exists. |

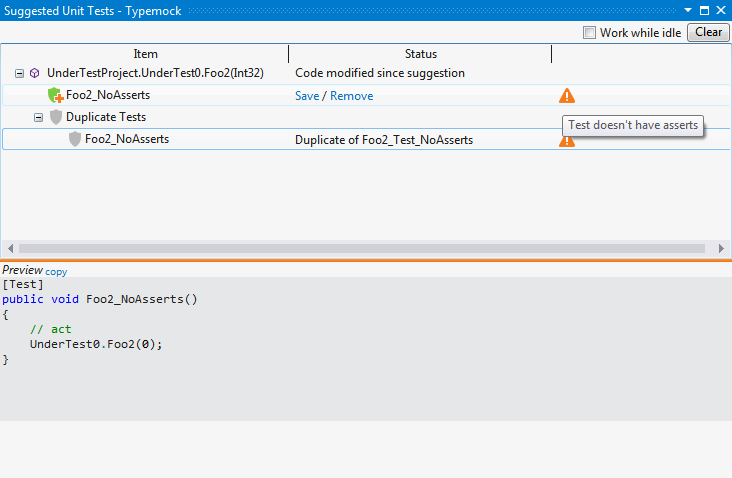
If the code of the method was changed since the last automatic suggestion, Typemock Isolator will display a special notification.

You can choose whether to accept the automatically suggested tests or discard them.

To view tests:

1. Click See the Tests.

The Suggested Unit Tests window is displayed.



1. To preview the test code, click the test name.

The code coverage indicator in the code editor highlights the code covered by the selected test

1. Do one of the following:

* When you want to save the automatically suggested test, on the right from the test name, click Save. When you save a test for the first time for the solution, the Test Suggestion Settings window is displayed. In this window, you can specify where you want to save all automatically suggested tests. The location is stored in solution\_name.isolator.config in the same folder where the solution is saved. To ensure that all automatically suggested tests are stored in the same location, share this file with the team.
* When you want to discard the automatically suggested test, on the right from the test name, click Remove.

## Getting Automatic Test Suggestions for a Specific Code

You can manually invoke automatic suggestion for any method. After tests are suggested, the Typemock Suggest window is displayed. Each test is accompanied with an icon that represents the status of the test:

| Icon | Description |
| --- | --- |
|  | Low quality test because:   * There is no assertion in the method, or * The method uses a private method, or * The metod is very simple |
|  | Such a test already exists. |

If the code of the method was changed since the last automatic suggestion, Typemock Isolator will display a special notification.

You can choose whether to accept the automatically suggested tests or discard them.

To manually invoke automatic suggestion:

1. Open the class or method for which you want Typemock Isolator to suggest a test.
2. Right-click on the class or method.
3. From the context menu, select one of the following:

* If you clicked on a class, select Suggest Unit Tests for Class.
* If you clicked on a method, select Suggest Unit Tests for Method.

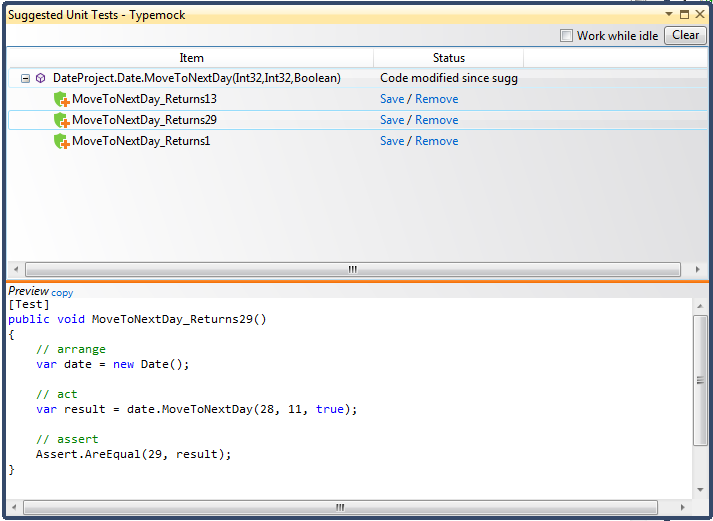
The Typemock Suggest window is displayed.

1. Click See the Tests.

The Suggested Unit Tests window is displayed.

1. To preview the test, on the right from the test, click the test name.

The code of the test is displayed underneath the list of the suggested tests.



1. Do one of the following:

* When you want to save the automatically suggested test, on the right from the test name, click Save. When you save a test for the first time for the solution, the Test Suggestion Settings window is displayed. In this window, you can specify where you want to save all automatically suggested tests. The location is stored in solution\_name.isolator.config in the same folder where the solution is saved. To ensure that all automatically suggested tests are stored in the same location, share this file with the team.
* When you want to discard the automatically suggested test, on the right from the test name, click Remove.

## Configuring Automatic Test Suggestion

[MISSING Test Suggest configuration ->naming etc] +missing option to suggest tests automatically

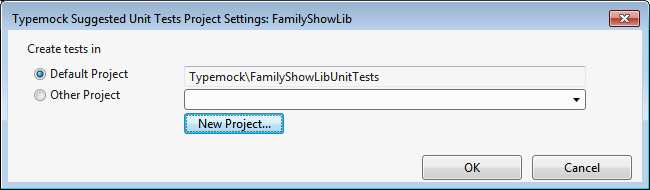
By default, when Typemock Isolator creates a test project, the test project is mapped to the project to be tested. The information about the test project is stored in a csproj file which is shared among your team members.

You can change the mapping between the test project and the project to be tested, if necessary.

To set up automatic test suggestion:

1. In Solution Explorer, right-click on the project.
2. Select Typemock Suggest Settings.

The Typemock Suggested Unit Test Project Settings window is displayed.

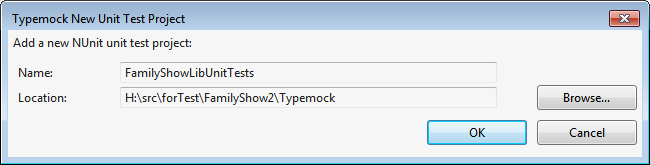


1. Select the Other Project option.
2. Do one of the following:

* If you want to map the test project to one of the existing projects, select the project from the drop-down list.
* If you want to map the test project to a new project, click New Project.

1. Click OK.

If you chose to map the test to a new project, the Typemock New Unit Test Project is displayed.



1. Specify the name and location of the new project.
2. Click OK.

# Mocking

Typemock Isolator provides the Typemock Isolator Mocking API that enables you to create unit tests for any code while protecting you from common mistakes.

The Mocking API is based on the following principles:

* The Arrange-Act-Assert (AAA) test structure is used as a convenient and readable test authoring pattern.
* Defaults are used that forces the correct usage.
* Easily discoverable API that has only one entry point
* Ability to test any code, including a legacy code
* Reduced need for unit test maintenance due to dynamic and tolerant-to-change nature of the API
* An ability to create readable tests for fast bug fixing when a failing test means a bug

## Overview

In a real-life development environment, the code might use other classes and components that cannot be tested due to various reasons. For example, some classes and components do not even exist yet because they are still under development while some of them are slow or complicated (such as classes that use a database, a logging system, or a Web service).

Typemock Isolator enables you to test your code by disconnecting and isolating any external components. This capability enables you to test the code in a localized, fast, and stable manner.

With Typemock Isolator, you do not have to change or refactor the production code which can still call an API of external components. However, these APIs will not run any actual code. Instead, they will return values that you defined for your test.

### Supported Fakeable MSCorlib Types

Typemock Isolator supports the following types:

* System
* DateTime
* Environment
* System.IO
* File
* Directory
* FileStream
* StreamReader
* System.Security.Cryptography.X509Certificates
* X509Certificate

### Basic Structure of a Unit Test

The following example shows the basic structure of a unit test:

* The test starts with the [Isolated] attribute. This attribute ensures that the fakes are cleaned up after the test.
* The test consists of the following stages:
* Arrange, in which test preconditions are defined and test objects are set up
* Act, in which the code under test is called
* Assert, in which test results are inspected and failures are reported
* A single point of entry is the Isolate statement. All mocking capabilities are available through this statement.
* DateTime.Now() is faked and returns the date every time this method is called.

|  |  |
| --- | --- |
| ! | Note  You can find all code examples in the Examples folder installed with Typemock Isolator. |

C# (BasicUnitTestExamples.cs)

|  |
| --- |
| [TestMethod,Isolated]  public void BasicUnitTestStructure()  {  // Arrange  Isolate.WhenCalled(() => DateTime.Now).WillReturn(new DateTime(2016, 2, 29));  // Act  int result = MyCode.DoSomethingSpecialOnALeapYear();  // Assert  Assert.AreEqual(100,result);  } |

VB (BasicUnitTestExamples.vb)

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub BasicTestStructure()  ' Arrange  Using TheseCalls.WillReturn(New DateTime(2016, 2, 29))  Dim dummy = DateTime.Now  End Using  ' Act  Dim result As Integer = My\_Class.DoSomethingSpecialOnALeapYear()  ' Assert  Assert.AreEqual(100, result)  End Sub |

|  |  |
| --- | --- |
| ! | Note  The Mocking API uses Lambda Expression to pass faked methods. This syntax is defined in .NET framework 3.5 and allows the API to be readable and strongly typed and support Intellisense within MS Visual Studio. |

### Mocking API Principles

The Mocking API provides a high level of isolation between tests and production code. The following sections explain the principles on which the Mocking ASPI is based.

### Auto-Recursive Fakes

By default, every fake object is auto-recursive. This means that each property and method of a fake object returns a fake object.

In addition to productivity gains of faking a whole object tree in one statement, recursive fakes reduce the decoupling level. If a test concentrates on one branch of the production code, and another branch of the object model is changed, the test stays intact.

### Argument Checking is Off by Default

When changing a method behavior, the argument checking is off by default. Regardless of the parameters at runtime, the behavior will change.

This will continue to work, even if other arguments are passed at runtime, or the method is called several times. Tests will continue to pass as long as the method is called.

For specific cases, you can explicitly check the arguments. It is recommended to keep these checks to a minimum.

### Faking Interfaces

If your code is already decoupled, and interfaces are used instead of classes as arguments, you can fake interfaces using the same APIs. You will get the same robustness because that fake interfaces will return recursive fakes.

### Faking Dependencies

Using a special method called Fake.Dependencies(), you can fake all constructor arguments in one line. In addition to a more readable and shorter test, you are immune from changing the signature of the constructor of the code under tests by adding or removing arguments of the dependencies. You can also add or remove more constructor overloads, and the test will still pass.

### True Properties and Indexers

In most cases, your production code interacts with dependencies without being aware whether these dependencies are real or faked.

Using True Properties and True Indexers not only help test code to be more readable, but also seamlessly work with the code under test. If the properties are refactored to automatic properties, tests will still pass.

## Setting Up a Unit Test Project

To set up a unit test project, perform the following steps:

1. Copy the Typemock Isolator DLLs. See [Step 1. Copying Typemock Isolator DLLs](#_D2HTopic_62).
2. Create a unit test project. See [Step 2. Creating a Unit Test Project](#_D2HTopic_63).
3. Prepare to write a test. See [Step 3. Preparing to Write a Unit Test](#_D2HTopic_64).
4. Add the Isolated attribute to tests. See [Step 4. Adding the Isolated Attribute to Tests](#_D2HTopic_65).
5. Choose the mocking method. See [Step 5. Defining whether to Mock Any Methods Choosing the Mocking Method](#_D2HTopic_69).

### Step 1. Copying Typemock Isolator DLLs

To copy Typemock Isolator DLLs:

|  |  |
| --- | --- |
| ! | Note  Although step 1 is optional, it represents the best practice and the highly recommended way to copy the Typemock Isolator DLLs. |

1. Copy all the following DLL’s from *Typemock installation folder*/AutoDeploy to your source control repository:

* TypeMock.dll
* Typemock.ArrangeActAssert.dll
* Typemock.Isolator.VisualBasic.dll

1. In your test project, change the references to the copied DLLs.

### Step 2. Creating a Unit Test Project

To create a unit test project:

1. In MS Visual Studio, create a new test project and call it UnitTest1.
2. Depending on whether you write in C# or VB.NET, do one of the following:

* For C#: Right click on Project Properties and add references to Typemock Isolator Core DLLand Typemock.ArrangeActAssert*.*
* For VB.Net: Right click on Project Properties and add references to Typemock Isolator Core DLLand Typemock Isolator VB.Net APIs*.*

1. In your test project, change the references to the Typemock Isolator DLLs that you copied to the source control repository:

* TypeMock.dll
* Typemock.ArrangeActAssert.dll
* Typemock.Isolator.VisualBasic.dll

### Step 3. Preparing to Write a Unit Test

To prepare to write a unit test:

1. Using the test project that you created, open UnitTest1.cs*.*
2. Write the following code at the beginning of the file:

C#:

|  |
| --- |
| using TypeMock.ArrangeActAssert; |

VB

|  |
| --- |
| Imports TypeMock.Isolator.VisualBasic |

### Step 4. Adding the Isolated Attribute to Tests

Because it is impossible to predict whether each test runs alone or in a group and in which order the tests will be run, the result of a test run should not depend on the order of execution. In addition, a test cannot rely on data and objects created by another test (such as behavior and fake objects left from other tests) which ran previously.

Typemock Isolator provides the [Isolated] attribute that instructs Typemock Isolator to clean up the result of the test after the test is completed.

You can use this attribute on the following levels:

* Test assembly
* Test class
* Test

Using the [Isolated] attribute at the test level overrides the behavior set on the class or assembly level. Therefore, you can set a general behavior and define a different behavior for specific cases.

|  |  |
| --- | --- |
| ! | Note  The easiest way to use the [Isolated] attribute is to define it on the assembly level. In this case, the behavior will be defined for the entire assembly. If a special behavior is required for a class or test level, you can add the [Isolated] attributed to this class or test. |

The following samples show how you can use the [Isolated] attribute on different levels.

#### Assembly Level

|  |
| --- |
| using TypeMock.ArrangeActAssert;  [assembly: Isolated()]  Class level (C#)  [TestClass, Isolated]  public class CollectionsTests  {  ...  } |

#### Class Level

|  |
| --- |
| [TestClass, Isolated]  public class CollectionsTests  {  ...  } |

#### Test Level

|  |
| --- |
| [TestMethod, Isolated]  public void SwapCollection\_WithFakeData()  {  ...  } |

### Step 5. Defining whether to Mock Virtual and Non-Virtual Methods

By default, Typemock Isolator mocks both virtual and non-virtual methods. However, in some cases (for example, when you practice a Test-Driven Development approach), you might need to instruct Typemock Isolator to mock virtual methods only.

You define whether you want Typemock Isolator to mock any methods or only virtual methods by using the Design property of the [Isolated] attribute. You can the Design property at any level where you use the [Isolated] attribute.

#### Mocking Any Methods

To mock any methods:

* Set the Design property of the [Isolated] attribute to Pragmatic.

For example:

|  |
| --- |
| [Isolated(Design = DesignMode.Pragmatic)] |

#### Mocking Virtual Methods Only

To mock virtual methods only:

* Set the Design property of the [Isolated] attribute to InterfaceOnly.

For example:

|  |
| --- |
| [Isolated(Design = DesignMode.InterfaceOnly)] |

##### Behavior of the Mocking API Methods

The following methods throw a DesignModeException when you call them with the Design property set to InterfaceOnly:

* Isolate.Fake.StaticMethods()
* Isolate.Fake.StaticConstructor()
* All Isolate.NonPublic methods
* All Isolate.Swap methods

In addition, the following methods throw a DesignModeException within conditions:

| Methods | Condition | Behavior | Throws Exception When... |
| --- | --- | --- | --- |
| Isolate.Fake.Instance<T> | *T* has some or all virtual methods | Only virtual methods can be faked and verified.  The default recursive fake behavior is applied if the returned object can also be faked. | *T* is a sealed object  All methods are non-virtual. |
| Isolate.Fake.Dependencies<T> | All argument types of T’s constructors have some virtual methods | Arguments are faked as with Fake.Instance<>. Object is created and returned. | All dependencies are sealed or have only non-virtual methods. |
| Isolate.WhenCalled… | Virtual methods | Can set any behavior | Called on non-virtual methods |
| Isolate.Verify… | Virtual methods | Can verify the behavior | Called on non-virtual methods |

## Faking an Instance

The following sections explain how you can fake objects and change the default behavior of a faked object.

### When to Use

When an object affects the unit test (for example, it is too slow or requires dependencies to be created), you should create a fake object.

### Faking Objects

Using the Mocking API, you can create fake objects of interfaces as well as concrete and sealed classes.

#### When to Use

When you want to pass a fake object to the method under test.

#### Syntax

C#

|  |
| --- |
| Isolate.Fake.Instance<Class>() |

VB

|  |
| --- |
| FakeInstance(Of Class)() |

#### Samples

C# (BasicUnitTesting.cs)

|  |
| --- |
| [TestMethod]  public void FakeAConcreteObjectExample()  {  // Arrange - Fake a Process, default is that all Members.ReturnRecursiveFakes  var fake = Isolate.Fake.Instance<Process>();    Isolate.WhenCalled(() => fake.MainModule.Site.Name).WillReturn("Typemock rocks");  // Act  var result = MyCode.IsMySiteNameTypemock(fake);  // Assert  Assert.AreEqual(true, result);  } |

VB

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub FakeObjectExample()  ' Arrange - Fake a Process, default is that all Members.ReturnRecursiveFakes  Dim fake = FakeInstance(Of Process)()  Using TheseCalls.WillReturn("Typemock rocks")  Dim dummy = fake.MainModule.Site.Name()  End Using  ' Act  Dim result = MyCode.IsMySiteNameTypemock(fake)  ' Assert  Assert.AreEqual(True, result)  End Sub |

### Faking Future Instances

In many cases, there is no clear way to pass fake objects into the code under test. In some cases, the code under test instantiates an object that you need to fake (that is a future instance). With Typemock Isolator, you can fake these instances in your unit test.

#### When to Use

When you want to fake the object that was created within the method under test.

#### Syntax

C#

|  |
| --- |
| var fakeT = Isolate.Fake.NextInstance<T>() |

VB

|  |
| --- |
| SwapNextInstance(Of T)() |

To fake a specific amount of future instances, use Isolate.Fake.NextInstance() sequentially.

Isolate.Fake.NextInstance() works with interface or abstract classes too. The next object that implements the interface or derives from the abstract class will be faked.

#### Samples

The following sample shows how to fake a future instance of Dependency.

C# (CreatingFutureFakes.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void Fake\_FutureInstance()  {  var fake = Isolate.Fake.NextInstance<Dependency><Dependency>();  var result = ClassUnderTest.AddSecurly(1, 2);  Assert.AreEqual(3, result);  }  public static int AddSecurly(int x, int y)  {  var dependency = new Dependency();  dependency.Check();  return x + y;  }  public class Dependency  {  public void Check()  {  throw new Exception("No Entry");  }  } |

VB (CreatingFutureFakes.vb)

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub Fake\_FutureInstance()  Dim fake = FakeInstance(Of Dependency)()  SwapNextInstance(Of Dependency)(fake)  Dim result = ClassUnderTest.AddSecurly(1, 2)  Assert.AreEqual(3, result)  End Sub  Public Shared Function AddSecurly(x As Integer, y As Integer) As Integer  Dim dependency = New Dependency()  dependency.Check()  Return x + y  End Function  Public Class Dependency  Public Sub Check()  Throw New Exception("No Entry")  End Sub  End Class |

### Faking All Instances of a Type

Faking all instances of the same type sets the same behavior on all past and future instances of that type. The same method used for faking all future instances also fakes instances that were previously created, even before the test ran (for example, in the test class setup method). You can use this capability to fake singletons.

#### When to Use

When you want to fake all objects of a specific type.

#### Syntax

C#

|  |
| --- |
| Isolate.Fake.AllInstances |

VB

|  |
| --- |
| SwapAllInstances(Of Dependency)(fake) |

By default, constructors are faked on future objects. To change this behavior, use the overloaded versions of Isolate.Fake.Instance() or Isolate.Fake.AllInstances().

#### Samples

##### Sample 1: Faking All Instances of the Same Type

The following sample shows how to fake all instances of the same type.

C# (CreatingFutureFakes.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void Fake\_AllFutureInstances()  {  var fake = Isolate.Fake.AllInstances<Dependency><Dependency>();  var result = ClassUnderTest.AddHeavlySecured(1, 2);  Assert.AreEqual(3, result);  } |

VB (CreatingFutureFakes.vb)

|  |
| --- |
| [TestMethod, Isolated]  <TestMethod(), Isolated()> \_  Public Sub Fake\_AllFutureInstances()  Dim fake = FakeInstance(Of Dependency)()  SwapAllInstances(Of Dependency)(fake)  Dim result = ClassUnderTest.AddHeavlySecured(1, 2)  Assert.AreEqual(3, result)  End Sub |

##### Sample 2: Faking Singletons

The following same shows how to replace the singleton object with a fake object that has a specific behavior.

|  |  |
| --- | --- |
| ! | Note  Isolate.Fake.NextInstance() fakes has precedence overIsolate.Fake.AllInstances(). |

C# (CreatingFutureFakes.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void FakeSingletonExample()  {  // Here we are setting the same behavior on all instances.  // The behavior we set on fake will apply to past instance as well  var fakeSingleton = Isolate.Fake.AllInstances<Singleton><Singleton>();  Isolate.WhenCalled(() => fakeSingleton.ReturnZero()).WillReturn(10);  // Assert that the behavior applied to all instances.  Assert.AreEqual(10, Singleton.Instance.ReturnZero());  }  public class Singleton  {  private Singleton() { }  static readonly Singleton instance = new Singleton();  public static Singleton Instance { get { return instance; } }  public int ReturnZero()  {  return 10;  }  } |

VB (CreatingFutureFakes.vb)

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub FakeSingletonExample()  Dim fakeSingleton = FakeInstance(Of Singleton)()  Using TheseCalls.WillReturn(10)  fakeSingleton.ReturnZero()  End Using  ' Here we are setting the same behavior on all instances.  ' The behavior we set on fake will apply to past instance as well  SwapAllInstances(Of Singleton)(fakeSingleton)  ' Assert that the behavior works.  Assert.AreEqual(10, Singleton.Instance.ReturnZero())  End Sub  Public Class Singleton  Private Sub New()  End Sub  Shared ReadOnly m\_instance As New Singleton()  Public Shared ReadOnly Property Instance() As Singleton  Get  Return m\_instance  End Get  End Property  Public Function ReturnZero() As Integer  Return 0  End Function  End Class |

### Faking Dependencies

With a single method, Typemock Isolator creates an object and fakes all the dependencies in its constructor. This approach saves you time and lines of code. In addition, it creates a more robust test. If the constructor changes, dependencies are added or removed, the test remains the same.

|  |  |
| --- | --- |
| ! | Note  If there are primitives, structs, or strings that are also accepted by the constructor, they are initialized to their base values. |

#### When to Use

When you want to fake all arguments of the constructor.

#### Syntax

C#

|  |
| --- |
| T real = Isolate.Fake.Dependencies<T>()  Isolate.GetFake<F>(real); |

VB

Not supported.

#### Samples

##### Sample 1: Faking All Dependencies of the Type

The following sample shows how to fake all dependencies of the type.

C# (FakingDependencies.cs)

|  |
| --- |
| [TestMethod]  public void FakeAllDependencies\_ChangeBehavior()  {  var real = Isolate.Fake.Dependencies<ClassUnderTest>();  var fake = Isolate.GetFake<Dependency>(real);  Isolate.WhenCalled(() => fake.Multiplier).WillReturn(2);  var result = real.Calculate(1, 2);  Assert.AreEqual(6, result);  }  public class ClassUnderTest  {  private int additional;  private Dependency2 d2;  private Dependency d1;  public ClassUnderTest(int additional, Dependency2 d2, Dependency d1)  {  this.additional = additional;  this.d2 = d2;  this.d1 = d1;  }  public int Calculate(int a, int b)  {  d2.Check();  return (a + b)\*d1.Multiplier+additional;  }  } |

##### Sample 2: Passing Specific Arguments to the Constructor

To pass arguments other than auto-fakes, pass them to the Fake.Dependencies() method. Typemock Isolator will try to match the type of the argument that you pass in the constructor and will swap them. This means that you do not need to enter the position of each argument, which protects you from heavy refactoring when changing the dependencies.

|  |  |
| --- | --- |
| ! | Note  You can set up a fake behavior before calling the constructor and then pass these fakes through Fake.Dependencies(). |

|  |  |
| --- | --- |
| ! | Note  To keep the interface simple, the Fake.Dependency() method works if the constructor does not accept multiple arguments of the same type (non-primitives or strings). If the constructor accepts multiple arguments of the same type, create fakes using the Fake.Instance() method and pass the arguments as an argument list. |

C# (FakingDependencies.cs)

|  |
| --- |
| [TestMethod]  public void FakeAllDependencies\_OverrideArguments()  {  var realDependency = new Dependency();  var real = Isolate.Fake.Dependencies<ClassUnderTest>(realDependency,4);  var result = real.Calculate(1, 2);  Assert.AreEqual(7, result);  } |

### Faking Constructors

A .NET code supports the following types of constructors:

* Instance: by default, Typemock Isolator fakes calls to the instance constructor. This means that any initialization done in the constructor is not performed. See [Faking Instance Constructors](#_D2HTopic_96).
* Static: the static constructor initializes class fields. The static constructor is called before the class is invoked for the first time. By default, Typemock Isolator does not fake the static constructor. See [Faking Static Constructors](#_D2HTopic_100).

You can control the behavior of both these constructors.

#### Faking Instance Constructors

By default, Typemock Isolator fakes calls to the instance constructor. This means that any initialization done in the constructor is not performed.

The default behavior for constructor is as follows:

* When using Members.CallOriginal(), the original constructor is called.
* When using all other Members enums, the constructor is not called.

##### When to Use

When you use a fake object and want to call the real constructor.

##### Syntax

Use the Isolate.Fake.Instance<T>() overload that accepts the following arguments:

| Argument | Description |
| --- | --- |
| Members behavior | Specify the default behavior for fake instance methods. |
| ConstructorWillBe | Specify whether you want to call the constructor (Called or Ignored). |
| params object [] constructorParameters | The parameters to be passed to the constructor. |

##### Samples

###### Sample 1: Faking All Methods Except the Constructor

The following sample shows how to fake all methods except the constructor and pass the constructor-specific arguments as follows:

* All methods except the constructor are faked
* The constructor will be called with the arguments 5, "Typemock"

C# (FakingConstructors.cs)

|  |
| --- |
| Isolate.Fake.Instance<Derived>(Members.ReturnRecursiveFake, ConstructorWillBe.Called); |

VB (FakingConstructors.vb)

|  |
| --- |
| FakeInstance(Of Derived)(Members. ReturnRecursiveFake, ConstructorWillBe.Called) |

###### Sample 2: Calling Original Methods and Ignoring the Constructor

The following sample shows how to use the second argument of Isolate.Fake.Instance() to call all original methods while ignoring the constructor as follows:

* All original methods are called by using Members.CallOriginal
* The constructor should be ignored by usingConstructorWillBe.Ignored

C# (FakingConstructors.cs)

|  |
| --- |
| Isolate.Fake.Instance<Derived>(Members.CallOriginal, ConstructorWillBe.Ignored); |

VB (FakingConstructors.vb)

|  |
| --- |
| FakeInstance(Of Derived)(Members.CallOriginal, ConstructorWillBe.Ignored) |

#### Ignoring Base Class Constructor

To ignore a base class constructor, use the BaseConstructor behavior of Isolate.Fake.Instance()*.*

##### When to Use

You can ignore the base class constructor when you use a fake object and want to call the real constructor while ignoring the base class constructor. For example, you can use this approach when the fake object's constructor performs initializations required for the test, but the base classes have external dependencies in their construction which should be faked.

##### Syntax

C#

|  |
| --- |
| Isolate.Fake.Instance<Derived>(Members.CallOriginal, ConstructorWillBe.Called, BaseConstructorWillBe.Ignored); |

VB

|  |
| --- |
| FakeInstance(Of Derived)(Members.CallOriginal, ConstructorWillBe.Called, BaseConstructorWillBe.Ignored) |

##### Samples

The following sample shows the TopCustomerData object. This object inherits a base class that has a dependency with a database connection.  To create a Customer, but avoid calling the base class constructor (because otherwise, it will try to connect to the external database), BaseConstructorWillBe.Ignored is used.

|  |  |
| --- | --- |
| ! | Note  A TypeMockException will be thrown if you try to fake base constructor behavior on types which are not inherited (that is inherited from System.Object) or mscorlib.dll types which are currently not supported. See [Supported Fakeable MSCorlib Types](#_D2HTopic_53). |

|  |  |
| --- | --- |
| ! | Note  To fake a base class higher in the hierarchy, send the base class type to fake to Isolate.Fake.Instance(). All constructors will be called up to that type. |

C# (FakingConstructors.cs)

|  |
| --- |
| [TestMethod]  public void CallConstructor\_FakeBaseClassConstructor()  {  // create an instance of Derived, but avoid calling the base class constructor  var dependency = Isolate.Fake.Instance<Derived>(Members.CallOriginal, ConstructorWillBe.Called, BaseConstructorWillBe.Ignored);  var result = new ClassUnderTest().GetSize(dependency);  Assert.AreEqual(100, result);  } |

VB (FakingConstructors.vb)

|  |
| --- |
| <TestMethod()> \_  Public Sub CallConstructor\_FakeBaseClassConstructor()  ' create an instance of Derived, but avoid calling the base class constructor  Dim dependency = FakeInstance(Of Derived)(Members.CallOriginal, ConstructorWillBe.Called, BaseConstructorWillBe.Ignored)  Dim result = New ClassUnderTest().GetSize(dependency)  Assert.AreEqual(100, result)  End Sub |

These are the test classes used in this sample:

C# (FakingConstructors.cs)

|  |
| --- |
| public class Base  {  public Base()  {  throw new NotImplementedException();  }  public int Size { get; set; }  }  public class Derived : Base  {  public Derived()  {  Size = 100;  }  } |

VB (FakingConstructors.vb)

|  |
| --- |
| Public Class Base  Public Sub New()  Throw New NotImplementedException()  End Sub  Public Property Size() As Integer  Get  Return m\_Size  End Get  Set(value As Integer)  m\_Size = Value  End Set  End Property  Private m\_Size As Integer  End Class  Public Class Derived  Inherits Base  Public Sub New()  Size = 100  End Sub  End Class |

#### Faking Static Constructors

By default, Typemock Isolator does not fake the static constructor. When a type is created or faked, or a static method is called, the static constructor is invoked only once by the CLR.

##### When to Use

When you want to fake the static constructor of a type.

|  |  |
| --- | --- |
| ! | Note  Typemock Isolator forces a call of the original static constructor when running a test which does not fake the static constructor. |

##### Syntax

C#

|  |
| --- |
| // Fake the static constructor  Isolate.Fake.StaticConstructor<Dependency>(); |

VB

|  |
| --- |
| ' Fake the static constructor  FakeStaticConstructor(Of Dependency)() |

|  |  |
| --- | --- |
| ! | Note  Because different runners might run tests in a different order, it is not possible to control calls of the static constructor unless the static constructor is explicitly called in every test. It is recommended to either call or fake the static constructor in every test. |

|  |  |
| --- | --- |
| ! | Note  Faking an instance might cause calling the static constructor if this is the first time the type is called. To suspend the static call, use Isolate.Fake.StaticConstructor() explicitly before calling Isloate.Fake.Instance(). |

##### Samples

C#

|  |
| --- |
| [TestMethod]  public void FakingStaticConstructor()  {  Isolate.Fake.StaticConstructor<StaticConstructorExample>();  // calling a static method on the class forces the static constructor to be called  StaticConstructorExample.Foo();  // this verifies the static constructor was faked and not called  Assert.IsFalse(StaticConstructorExample.TrueOnStaticConstructor);  } |

VB

|  |
| --- |
| <TestMethod()> \_  Public Sub FakingStaticConstructor()  FakeSharedConstructor(Of StaticConstructorExample)()  ' calling a static method on the class forces the static constructor to be called  StaticConstructorExample.Foo()  ' this verifies the static constructor was faked and not called  Assert.IsFalse(StaticConstructorExample.TrueOnStaticConstructor)  End Sub |

#### Running a Custom Logic

##### When to Use

When you want to run your logic while the constructor runs.

##### Syntax

C#

|  |
| --- |
| var handle =Isolate.Fake.NextInstance<Dependency>(Members.ReturnRecursiveFakes, context=> {  // action here }); |

##### Samples

To run custom logic in a faked method, use DoInstead after WhenCalled and provide your own delegate. DoInsteadpasses MethodCallContext with the following members:

| Member | Description |
| --- | --- |
| Instance | An instance of the calling object (null if the method is static) |
| Parameters | Array of objects of arguments passed to the method (if exist) |
| Method | A MethodBase object that contains the method metadata that can be used for choosing the required behavior |
| WillCallOriginal() | Calls the original implementation |

C#

|  |
| --- |
| [TestMethod]  public void CallConstructor\_DoCustomCode()  {  var handle = Isolate.Fake.NextInstance<Dependency>(Members.ReturnRecursiveFakes, context =>  {  if ((string) context.Parameters[1] == "typemock")  {  // contructor is faked  return;  }  context.WillCallOriginal(); // constructor will be called  });  var classUnderTest = new ClassUnderTest();  var dependency = classUnderTest.Create("typemock");  var result = classUnderTest.GetString(dependency);  Assert.AreEqual("0", result);  } |

#### Simulating Error in Constructor

##### When to Use

When you want to test a failure that occurs when constructing an object. For example, an OutOfMemoryException.

##### Syntax

C# (FakingConstructors.cs)

|  |
| --- |
| Isolate.Swap.NextInstance<Dependency>()  .ConstructorWillThrow(new OutOfMemoryException()); |

VB (FakingConstructors.vb)

|  |
| --- |
| SwapNextInstanceWithException(Of Dependency)(New OutOfMemoryException()) |

##### Samples

The following sample shows how to fake an exception that is thrown when an object is created.

C# (FakingConstructors.cs)

|  |
| --- |
| [TestMethod]  public void FutureInstance\_VerifyThrowingExceptionOnCreation()  {  // We want a memory handling exception to be thrown the next time a Dependency is instantiated  Isolate.Swap.NextInstance<Dependency>()  .ConstructorWillThrow(new OutOfMemoryException());  var classUnderTest = new ClassUnderTest();  var result = classUnderTest.Create();  Assert.AreEqual(null, result);  } |

VB (FakingConstructors.vb)

|  |
| --- |
| <TestMethod>\_  Public Sub FutureInstance\_VerifyThrowingExceptionOnCreation()  ' We want a memory handling exception to be thrown the next time a Dependency is instantiated  SwapNextInstanceWithException(Of Dependency)(New OutOfMemoryException())  Dim result = New ClassUnderTest().Create()  Assert.AreEqual(Nothing, result)  End Sub |

### Setting the Default Behavior of a Fake Object

You can set a default behavior of all methods of a class or type. Then you can specify a different behavior for a specific method.

The following default behaviors are available:

| Behavior | Behavior |
| --- | --- |
| **ReturnRecursiveFakes** | Returns:   * For reference types: fake objects * For other return types: zero or equivalent   The returned fake objects will behave in the same way.  This is the default fake object behavior. Using this option will ensure that your unit tests are stable. |
| **CallOriginal** | Calls to fake object methods will pass through to their original implementation. The object's real constructor is called during the creation of the fake object. This is useful when you need to maintain the original behavior of the test object and modify the behavior of specific methods. |
| **MustSpecifyReturnValues** | Calls to fake object methods that return a specific value. If a method that returns the specified value is called without being set up in first place, an exception is thrown, and the test will fail. Void calls are still ignored. |
| **ReturnNulls** | Calls to fake object methods will return:   * zero or equivalent: to any call of methods that return value types * null: for any methods that return a reference type.   Void calls are still ignored. |

#### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(() => fake.MainModule.Site.Name).<behavior>; |

VB

|  |
| --- |
| Using TheseCalls.<behavior>  Dim dummy = fake.MainModule.Site.Name  End Using |

#### Samples

##### Sample 1: Using Recursive Fakes

An ability to fake objects is a powerful and versatile tool. Even if the code under test calls an intricate object model which is replaced by a fake object, deep calls (chained calls several levels down the object model) will be faked automatically. Therefore, you do not have to explicitly fake return values and define behavior for each and every hierarchy level.

The following example shows how to use Members.ReturnRecursiveFakes (**default behavior)**.

|  |  |
| --- | --- |
| ! | Note  For collections in mscorlib (which implements IEnumerable and have default constructor) Members.ReturnRecursiveFake() returns an initialized empty collection. |

C# (BasicUnitTestExamples.cs)

|  |
| --- |
| [TestMethod,Isolated]  public void FakeObjectExample()  {  // Arrange  var fake = Isolate.Fake.Instance<Process>(); // default Members.ReturnRecursiveFakes  Isolate.WhenCalled(() => fake.MainModule.Site.Name).WillReturn("Typemock rocks");  // Act  var result = MyCode.IsMySiteNameTypemock(fake);  // Assert  Assert.AreEqual(true, result);  } |

VB (BasicUnitTestExamples.vb)

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub BasicTestStructure()  ' Arrange  Dim fake = FakeInstance(Of Process)() ' default Members.ReturnRecursiveFakes  Using TheseCalls.WillReturn("Typemock rocks")  Dim dummy = fake.MainModule.Site.Name  End Using  ' Act  Dim result = My\_Class.IsMySiteNameTypemock(fake)  ' Assert  Assert.AreEqual(True, result)  End Sub |

##### Sample 2: Using Stale Mocks

By default, if in test 1, you faked an instance, and in test 2 you try to use this instance, a message will be printed to the console. If you do not want this message to be printed to the console, use the [Isolated] attribute with the IgnoreStaleMocks property.

|  |
| --- |
| [Isolated(IgnoreStaleMocks = true)] |

## Changing Method Behavior

The following sections explain how you can change the behavior of faked methods.

### Modifying the Method Behavior

You can specify how you want a method to behave when you call it. For example, you can define that when the method is called, its original implementation should be invoked. Alternatively, you can force throwing an exception or returning a test collection of data.

#### When to Use

You can modify the behavior of a specific fake method when your test requires a specific return value from the method.

#### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(() => fake.Increment()).<behavior>; |

VB

|  |
| --- |
| Using TheseCalls.<behavior>()  fake.Increment()  End Using |

The following table explains possible behaviors:

| Behavior (C#) | Behavior (VB) | Description |
| --- | --- | --- |
| [CallOriginal](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IMethodBehavior_CallOriginal.htm)() | [WillCallOriginal](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillCallOriginal.htm)() | When the method is called, call the original implementation. |
| [ReturnRecursiveFakes](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IPublicNonVoidMethodHandler_1_ReturnRecursiveFake.htm)() | [WillReturnRecursiveFake](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/m../html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillReturnRecursiveFake.htm)() | Returns:   * For reference types: fake objects * For other return types: zero or equivalent   The returned fake objects will behave in the same way. |
| [WillThrow](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IMethodBehavior_WillThrow.htm)() | [WillThrow](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillThrow.htm)() | Throws a specified exception. |
| [WillReturn](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IReturnValueHandler_1_WillReturn.htm)() | [WillReturn](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillReturn.htm)() | Return a specified value. This is applicable only to those methods that return values. |
| [IgnoreCall](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IVoidActionHandler_IgnoreCall.htm)() | [WillBeIgnored](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillBeIgnored.htm)() | Returns immediately. This is applicable only to void methods. |
| [WillReturnCollectionValuesOf](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/ReplacingCollectionsAAA.html)() | Not supported | Returns a collection of test data. |
| [DoInstead](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/SettingBehaviorAAA.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#DoInstead)() | [WillBeReplacedWith](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/SettingBehaviorAAA.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#CustomBehavior)() | Calls a user code. This option is used for advanced and complex behaviors. |

#### Samples

##### Sample 1: Ignoring Call

C#

|  |
| --- |
| Isolate.WhenCalled(() => fake.Increment()).IgnoreCall(); |

VB

|  |
| --- |
| Using TheseCalls.WillBeIgnored()  fake.Increment()  End Using |

##### Sample 3: Setting Different Behavior on Overloaded Methods

To set a different behavior to overloaded methods, use WhenCalled() on each overload.

|  |  |
| --- | --- |
| ! | Note  The first overload method behavior setting is the default behavior for all overloads. |

C# (ControllingMethods.cs)

|  |
| --- |
| [TestMethod, Isolated] // Note: Use Isolated to clean up after the test  public void Overloaded\_Example()  {  var fakeDependency = Isolate.Fake.Instance<Dependency>();  // Each overloaded method will act as a separate sequence  Isolate.WhenCalled(() => fakeDependency.OverloadedMethod(1)).WillReturn(2);  Isolate.WhenCalled(() => fakeDependency.OverloadedMethod("Typemock Rocks")).WillReturn(9);  var result = new ClassUnderTest().AddOverloadedDependency(fakeDependency);  Assert.AreEqual(11, result);  }  public int AddOverloadedDependency(Dependency dependency)  {  return dependency.OverloadedMethod(12) + dependency.OverloadedMethod("typemock");  } |

VB (ControllingMethods.vb)

|  |
| --- |
| ' Note: Use Isolated to clean up after the test  <TestMethod(), Isolated()> \_  Public Sub Overloaded\_Example()  Dim fakeDependency = FakeInstance(Of Dependency)()  ' Each overloaded method will act as a separate sequence  Using TheseCalls.WillReturn(2)  fakeDependency.OverloadedMethod(1)  End Using  Using TheseCalls.WillReturn(9)  fakeDependency.OverloadedMethod("Typemock Rocks")  End Using  Dim result = New ClassUnderTest().AddOverloadedDependency(fakeDependency)  Assert.AreEqual(11, result)  End Sub  Public Function AddOverloadedDependency(dependency As Dependency) As Integer  Return dependency.OverloadedMethod(12) + dependency.OverloadedMethod("typemock")  End Function |

##### Sample 4: Running a Custom Logic

To run a custom logic in a faked method, use DoInstead after WhenCalled and provide your own delegate. DoInsteadpasses MethodCallContext with the following members:

| Member | Description |
| --- | --- |
| Instance | An instance of the calling object (null if the method is static) |
| Parameters | Array of objects of the arguments passed to the method (if exist) |
| Method | A MethodBase object that contains method metadata that can be used for choosing the required behavior |
| WillCallOriginal() | Calls the original implementation |

The following sample shows the following:

* DoInstead sends a custom delegate that will run when GetID is called.
* The actual call to GetID will call the custom delegate.

|  |  |
| --- | --- |
| ! | Note  Overloads for DoInstead also exist for void methods and non-public methods. |

C# (ControllingMethods.cs)

|  |
| --- |
| [TestMethod, Isolated] // Note: Use Isolated to clean up after the test  public void DoInstead\_Example()  {  var returnValue = 2;  var fakeDependency = Isolate.Fake.Instance<Dependency>();  // return value dynamically  Isolate.WhenCalled(() => fakeDependency.GetID()).DoInstead(  x => { return returnValue; });  returnValue = 4;  result = new ClassUnderTest().AddToDependency(1, fakeDependency);  Assert.AreEqual(5, result);  }  public int AddToDependency(int a, Dependency dependency)  {  return a + dependency.GetID();  } |

VB (ControllingMethods.vb)

|  |
| --- |
| ' Note: Use Isolated to clean up after the test  <TestMethod(), Isolated()> \_  Public Sub DoInstead\_Example()  Dim fakeDependency = FakeInstance(Of Dependency)()  ' return value dynamically  Using TheseCalls.WillBeReplacedWith(AddressOf FunctionThatReplaceGetID)  fakeDependency.GetID()  End Using  returnValue = 2  Dim result = New ClassUnderTest().AddToDependency(1, fakeDependency)  Assert.AreEqual(3, result)  returnValue = 4  result = New ClassUnderTest().AddToDependency(1, fakeDependency)  Assert.AreEqual(5, result)  End Sub  Private returnValue = 2  Private Function FunctionThatReplaceGetID(ByVal callContext As MethodCallContext) As UInt32  Return returnValue  End Function  Public Function AddToDependency(a As Integer, dependency As Dependency) As Integer  Return a + dependency.GetID()  End Function |

##### Sample 5: Running Custom Logic and Continuing with the Original Method

The following example shows how to apply a custom logic and then continue with the implementation of the original method.

C# (ControllingMethods.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void WhenCalledDoInstead\_RunOriginalAfterLogic()  {  var fakeDependency = new Dependency();  int counter=0;  Isolate.WhenCalled(() => fakeDependency.Calculate()).DoInstead(  c =>  {  // Custom logic before original call  counter++;  // Make sure the method gets called immediately after leaving  c.WillCallOriginal();  return null;  }  );  Assert.IsTrue(counter == 1);  } |

### Faking Behavior for Live Objects

A live object is a test object that was instantiated normally and is not a fake. Typemock Isolator supports controlling methods of live objects.

#### When to Use

When you want to fake a method of an existing object.

#### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(()=>dependency.<method>).<behavior>; |

VB

|  |
| --- |
| Using TheseCalls.<behavior>  <method>  End Using |

The following table explains possible behaviors:

| Behavior (C#) | Behavior (VB) | Description |
| --- | --- | --- |
| [CallOriginal](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IMethodBehavior_CallOriginal.htm)() | [WillCallOriginal](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillCallOriginal.htm)() | When the method is called, call the original implementation. |
| [ReturnRecursiveFakes](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IPublicNonVoidMethodHandler_1_ReturnRecursiveFake.htm)() | [WillReturnRecursiveFake](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/m../html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillReturnRecursiveFake.htm)() | Returns:   * For reference types: fake objects * For other return types: zero or equivalent   The returned fake objects will behave in the same way. |
| [WillThrow](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IMethodBehavior_WillThrow.htm)() | [WillThrow](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillThrow.htm)() | Throws a specified exception |
| [WillReturn](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IReturnValueHandler_1_WillReturn.htm)() | [WillReturn](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillReturn.htm)() | Return a specified value. This is applicable only to those methods that return values. |
| [IgnoreCall](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_TypeMock_ArrangeActAssert_IVoidActionHandler_IgnoreCall.htm)() | [WillBeIgnored](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/html/M_Typemock_Isolator_VisualBasic_TheseCalls_WillBeIgnored.htm)() | Returns immediately. This is applicable only to void methods. |
| [WillReturnCollectionValuesOf](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/ReplacingCollectionsAAA.html)() | Not supported | Return a collection of test data. |
| [DoInstead](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/SettingBehaviorAAA.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#DoInstead)() | [WillBeReplacedWith](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/SettingBehaviorAAA.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#CustomBehavior)() | Calls a user code. This option is used for advanced and complex behaviors. |

#### Samples

##### Sample 1: Ignoring a Method

The following sample shows how to ignore a method of a live object.

|  |  |
| --- | --- |
| ! | Note  Creating live objects is similar to creating a fake object using Members.CallOriginal(). |

|  |  |
| --- | --- |
| ! | Note  To fake non-public methods, use Isolate.NonPublic.WhenCalled(). |

C# (LiveObjects.cs)

|  |
| --- |
| var dependency = new Dependency(); // not a fake  Isolate.WhenCalled(()=>dependency.CheckSecurity(null,null)).IgnoreCall(); // Ignore CheckSecurity method |

VB

|  |
| --- |
| Dim realProduct As Product = new Product(); // this is a normal "live" object  Using TheseCalls.WillBeIgnored  realProduct.ModifyPrice(0.0)  End Using |

##### Sample 2: Faking the Internal Property Behavior

The following sample shows how to fake the behavior of a live object’s property.

C# (LiveObjects.cs)

|  |
| --- |
| Dependency dependency = new Dependency();  Isolate.NonPublic.Property.WhenGetCalled(dependency, "IsDiskFull").WillReturn(true); // faking internal property behavior on real object |

VB

|  |
| --- |
| Dim RealLogger = New RealLogger()  NonPublicWillReturn(logger, "get\_IsDiskFull",True); ' faking internal property behavior on real object |

### Simulating a Sequenced Behavior

A sequenced behavior occurs when the same method has a different behavior each time the method is called.

#### When to Use

When you want to simulate a sequenced behavior.

|  |  |
| --- | --- |
| ! | Note  After using a sequence and calling the method with the expected behavior, you can use WhenCalled on the method again. In this case, the old behavior sequence is reset, and a new one starts from the next WhenCalled. |

#### Syntax

C# (ControllingMethods.cs)

|  |
| --- |
| Isolate.WhenCalled(() => fakeDependency.GetID()).<behavior>;  Isolate.WhenCalled(() => fakeDependency.GetID()).<behavior>;  ... |

VB (ControllingMethods.vb)

|  |
| --- |
| Using TheseCalls.<behavior>  fakeDependency.GetID()  End Using  Using TheseCalls.<behavior>  fakeDependency.GetID()  End Using  ... |

The last call to Isolate.WhenCalled() defines the default behavior for the method from that moment onwards.

#### Samples

C# (ControllingMethods.cs)

|  |
| --- |
| [TestMethod, Isolated] // Note: Use Isolated to clean up after the test  public void SequencedWillReturn\_Example()  {  var fakeDependency = Isolate.Fake.Instance<Dependency>();  // Sequenced calls will return values in sequence, last value will stay the default  Isolate.WhenCalled(() => fakeDependency.GetID()).WillReturn(2);  Isolate.WhenCalled(() => fakeDependency.GetID()).WillReturn(9);  var result = new ClassUnderTest().AddToDependency3Times(1, fakeDependency);  Assert.AreEqual(21, result);  }  public int AddToDependency3Times(int a, Dependency dependency)  {  return a + dependency.GetID()+dependency.GetID()+dependency.GetID();  } |

VB (ControllingMethods.vb)

|  |
| --- |
| ' Note: Use Isolated to clean up after the test  <TestMethod(), Isolated()> \_  Public Sub SequencedWillReturn\_Example()  Dim fakeDependency = FakeInstance(Of Dependency)()  ' Sequenced calls will return values in sequence, last value will stay the default  Using TheseCalls.WillReturn(2)  fakeDependency.GetID()  End Using  Using TheseCalls.WillReturn(9)  fakeDependency.GetID()  End Using  Dim result = New ClassUnderTest().AddToDependency3Times(1, fakeDependency)  Assert.AreEqual(21, result)  End Sub  Public Function AddToDependency3Times(a As Integer, dependency As Dependency) As Integer  Return a + dependency.GetID() + dependency.GetID() + dependency.GetID()  End Function |

### Faking Method Chaining

Method chaining is a common syntax for invoking multiple method calls. Each method returns an object that allows the calls to be chained together in a single statement. A method chain causes the increase in the number of methods that come one after another in the same line that occurs as more methods are chained together even though line breaks are often added between methods.

#### When to Use

When you have a hierarchy of objects and you want to define a specific behavior for a chain of methods.

#### Syntax

C# (ControllingMethods.cs)

|  |
| --- |
| Isolate.WhenCalled(() => <fake>.<method1>.<method2>…).<behavior>; |

VB (ControllingMethods.vb)

|  |
| --- |
| Using TheseCalls.<behavior>  <fake>.<method1>.<method2>…  End Using |

#### Samples

The following sample shows how to fake a complete chain of calls at once by calling them in Isolate.WhenCalled().

|  |  |
| --- | --- |
| ! | Note  When one of the methods in the chain returns an unsupported type (including collections in mscorlib), Typemock Isolator throws an exception. |

C# (ControllingMethods.cs)

|  |
| --- |
| [TestMethod, Isolated] // Note: Use Isolated to clean up after the test  public void SetUpChain\_Example()  {  var fakeDependency = Isolate.Fake.Instance<Dependency>();  // Note The chain of calls that will be faked  Isolate.WhenCalled(() => fakeDependency.GetPatent().GetID()).WillReturn(2);  var result = new ClassUnderTest().AddToChainedDependency(1, fakeDependency);  Assert.AreEqual(3, result);  }  public int AddToChainedDependency(int a, Dependency dependency)  {  return a + dependency.GetPatent().GetID();  } |

VB (ControllingMethods.vb)

|  |
| --- |
| ' Note: Use Isolated to clean up after the test  <TestMethod(), Isolated()> \_  Public Sub SetUpChain\_Example()  Dim fakeDependency = FakeInstance(Of Dependency)()  ' Note The chain of calls that will be faked  Using TheseCalls.WillReturn(2)  fakeDependency.GetPatent().GetID()  End Using  Dim result = New ClassUnderTest().AddToChainedDependency(1, fakeDependency)  Assert.AreEqual(3, result)  End Sub  Public Function AddToChainedDependency(a As Integer, dependency As Dependency) As Integer  Return a + dependency.GetPatent().GetID()  End Function |

### Faking Methods Based on Call Arguments

To ensure that minor changes to production code will not break your test, by default, Typemock Isolator ignores all arguments passed to fake methods.

#### When to Use

When you want a fake method to return the value that varies depending on the arguments you passed. For example, you might need a fake cache to return different objects depending on the key.

#### Controlling Method Behavior Based on the Argument Value

You can control a fake method's behavior based on call arguments by requiring all arguments to match the arguments used in the WhenCalled statement.

##### Syntax

C#

|  |
| --- |
| WhenCalled(() => <method>).WithExactArguments().<behavior>; |

VB

|  |
| --- |
| Using TheseCalls.WithExactArguments.<behavior>  <method>  End Using |

##### Samples

###### Sample 1: Faking Method’s Return Value

The following sample shows how to fake the method's return value as follows:

* The return value of MethodReturnInt is set to 10 only when the arguments are "typemock" and 1.
* The return value of MethodReturnInt is set to 50 only when the arguments are "unit tests" and 2.

C# (MethodsArgument.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void FakeReturnValue\_BasedOn\_ExactMethodArgs()  {  var fake = Isolate.Fake.Instance<Dependency>();  Isolate.WhenCalled(() => fake.MethodReturnInt("typemock",1)).WithExactArguments().WillReturn(10);  Isolate.WhenCalled(() => fake.MethodReturnInt("unit tests", 2)).WithExactArguments().WillReturn(50);  var result = new ClassUnderTest().SimpleCalculation(fake);  Assert.AreEqual(60,result);  }  public int SimpleCalculation( Dependency dependency)  {  return dependancy.MethodReturnInt("typemock", 1) + dependancy.MethodReturnInt("unit tests", 2);  } |

VB (MethodsArgument.vb)

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub FakeReturnValue\_BasedOn\_ExactMethodArgs()  Dim fake = FakeInstance(Of Dependency)()  Using TheseCalls.WithExactArguments.WillReturn(10)  fake.MethodReturnInt("typemock", 1)  End Using  Using TheseCalls.WithExactArguments.WillReturn(50)  fake.MethodReturnInt("unit tests", 2)  End Using  Dim result = New ClassUnderTest().SimpleCalculation(fake)  Assert.AreEqual(60, result)  End Sub  Public Function SimpleCalculation(dependency As Dependency) As Integer  Return dependency.MethodReturnInt("typemock", 1) + dependency.MethodReturnInt("unit tests", 2)  End Function |

###### Sample 2: Faking a Void Method’s Return Value

The following sample shows how to fake the return value for void methods.

|  |  |
| --- | --- |
| ! | Note  This is the default behavior for [indexers](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/ReplacingCollectionsAAA.html), that is each index will return a different value. |

C# (MethodsArgument.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void FakeVoidMethod\_BasedOn\_ExactMethodArgs()  {  var fake = Isolate.Fake.Instance<Dependency>(Members.CallOriginal);  Isolate.WhenCalled(() => fake.VoidMethod(4)).WithExactArguments().IgnoreCall();  new ClassUnderTest().CallVoid(fake,4);  // Assert.NoExceptionWasThrown();  }  public void VoidMethod(int n)  {  throw new NotImplementedException();  } |

VB (MethodsArgument.vb)

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub FakeVoidMethod\_BasedOn\_ExactMethodArgs()  Dim fake = FakeInstance(Of Dependency)(Members.CallOriginal)  Using TheseCalls.WithExactArguments.WillBeIgnored  fake.VoidMethod(4)  End Using  Dim underTest = New ClassUnderTest()  underTest.CallVoid(fake, 4)  ' Assert.NoExceptionWasThrown();  End Sub  Public Sub VoidMethod(n As Integer)  Throw New NotImplementedException()  End Sub |

#### Controlling Method Behavior when Arguments Satisfy Custom Rules

AndArgumentsMatch accepts a predicate in which you call a custom logic to specify whether the method will be faked. To use AndArgumentsMatch*,* define the types that you want to check as your lambda parameters and pass these as arguments to the fake method.

##### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled((<type arg1>, <type arg1>) => fake.<method> (<arg1>, <arg2>))  .AndArgumentsMatch((<arg1>, <arg2>) => <check>  .<behavior>; |

VB

Not supported.

##### Samples

###### Sample 1: Using the AndArgumentsMatch Method and Checking All Arguments

The following sample shows how to use AndArgumentsMatch as follows:

* An overload of Isolate.WhenCalled() is used to accepts a lambda expression as an argument. The arguments are placeholders for the arguments to be checked.
* AndArgumentsMatch uses a delegate to perform the custom checking. The delegate receives the same arguments that Isolate.WhenCalled() receives. Within the custom logic, you can define the behavior that will be used when the actual call to the method is made.

C# (MethodsArgument.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void FakeReturnValue\_BasedOn\_CustomArgumentsChecking()  {  var fake = Isolate.Fake.Instance<Dependency>();  Isolate.WhenCalled((string s, int x) => fake.MethodReturnInt(s, x))  .AndArgumentsMatch((s, x) => s.StartsWith("Gui") && x < 300)  .WillReturn(1000);  var result = new ClassUnderTest().CallWithGuitar100(fake);  // All the arguments match our custom checker - the returned value is faked.  Assert.AreEqual(1000, result);  }  public int CallWithGuitar100(Dependency dependency)  {  return dependancy.MethodReturnInt("Guitar", 200);  } |

VB

Not supported.

###### Sample 2: Using the AndArgumentsMatch Method and Checking Part of Arguments

The following sample shows how to check only some of the method arguments by sending only the arguments to be checked to Isolate.WhenCalled()*:*

* Only the second argument of the GetQuota() method (int x) is used in Isolate.WhenCalled().
* When the actual method is called, the method is faked based on the value of only the second argument.

C# (MethodsArgument.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void FakeReturnValue\_BasedOnCustomArgumentsChecking\_CheckOneArgument()  {  var fake = Isolate.Fake.Instance<Dependency>();  Isolate.WhenCalled((int x) => fake.MethodReturnInt("", x))  .AndArgumentsMatch( x => x < 300)  .WillReturn(1000);  var result = new ClassUnderTest().CallWithGuitar100(fake);  // All the arguments match our custom checker - the returned value is faked.  Assert.AreEqual(1000, result);  } |

VB

Not supported.

#### Mixing WithExactArguments and Custom Checkers

To mix custom checkers and WithExactArguments*,* do not define the exact arguments as a lambda parameter and just pass the matching argument. Typemock Isolator checks the arguments as follows:

* If the arguments have a custom checker, the custom checker is used.
* The arguments that don't have a custom checker are matched exactly.

##### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled((<type arg1>, <type arg1>) => fake.<method> (<arg1>, <arg2>, <additional\_arguments>))  .AndArgumentsMatch((<arg1>, <arg2>) => <check>  .<behavior>; |

VB

Not supported.

##### Samples

###### Sample 1: Using Custom Checks

The following sample shows how to use the custom checkers as follows:

* A custom checker is used only on the second argument of the GetQuote() method.
* WithExactArguments is used against the second argument so only a call to GetQuote() with the string "Guitar" is faked.

|  |  |
| --- | --- |
| ! | Note  When you use AndArgumentsMatch on a chain of calls, all the arguments in the custom checker should belong to one method. |

C# (MethodsArgument.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void FakeReturnValue\_BasedOn\_MixedChecker()  {  var fake = Isolate.Fake.Instance<Dependency>();  Isolate.WhenCalled((int x) => fake.MethodReturnInt("Guitar", x))  .AndArgumentsMatch(x => x < 300)  .WithExactArguments()  .WillReturn(1000);  var result = new ClassUnderTest().CallWithGuitar100(fake);  // All the arguments match our custom checker - the returned value is faked.  Assert.AreEqual(1000, result);  } |

VB

Not supported.

###### Sample 2: Making a Call that Throws TypeMockException

The following sample shows how to make a call that throws an exception. The arguments passed to Isolate.WhenCalled() are spread across two methods. This causes Typemock Isolator to throw the exception.

C#

|  |
| --- |
| Isolate.WhenCalled((string s, int i) => fake.GetSon(s).DoSomething(i))  .AndArgumentsMatch((s, i) => s != null && i > 0)  .WillThrow(new Exception("Simulated"))); |

VB

Not supported.

### Controlling Property Behavior

Technically, properties are methods. However, when faking properties, there are several additional factors that you should consider.

There are two ways to set properties:

* Using true properties. See [Controlling Property Behavior Using True Properties](#_D2HTopic_156).
* Using Isolate.WhenCalled(). See [Controlling Property Behavior Using Isolate.WhenCalled()](#_D2HTopic_160).

#### Controlling Property Behavior Using True Properties

With a true property, setting the property value means that this value will be returned when you get the property.

Typemock Isolator simulates all fake properties as if they are auto-properties. Therefore, you can set a fake property by assigning a value to it.

#### When to Use

When your test requires a fake property to behave as an auto-property, and you have public set and get.

##### Syntax

C#

|  |
| --- |
| var fakeDependency = Isolate.Fake.Instance<Dependency>();  fakeDependency.Number=<value>; // sets the property to return the value |

VB

|  |
| --- |
| Dim fakeDependency = FakeInstance(Of <Dependency>)()  fakeDependency.Number = <value> ' sets the property to return the value |

##### Samples

###### Sample 1: Using True Properties

C# (ControllingProperties.cs)

|  |
| --- |
| [TestMethod]  [Isolated]  public void FakeAutoProperty()  {  var fakeDependency = Isolate.Fake.Instance<Dependency>();  fakeDependency.Number=5; // sets the property to return 5  Assert.AreEqual(5, fakeDependency.Number);  } |

VB (ControllingProperties.vb)

|  |
| --- |
| <TestMethod()>  Public Sub FakeAutoProperty()  Dim fakeDependency = FakeInstance(Of <Dependency>)()  fakeDependency.Number = 5 ' sets the property to return 5  Assert.AreEqual(5, fakeDependency.Number)  End Sub |

###### Sample 2: Using True Properties for Members.CallOriginal() Fakes

An ability to use true properties is not applied when Members.CallOriginal() is used because the original implementation will be executed. In this case, if you want to use true properties, use Isolate.WhenCalled().ReturnRecursiveFake() on that property as follows:

C# (ControllingProperties.cs)

|  |
| --- |
| var dependency = new Dependency(); // not fake  Isolate.WhenCalled(()=>dependency.Number).ReturnRecursiveFake(); // Number will act like an auto property  dependency.Number = 5; |

VB (ControllingProperties.vb)

|  |
| --- |
| Dim dependency = New Dependency() ' not fake  Using TheseCalls.WillReturnRecursiveFake ' Number will act like an auto property  Dim dummy = dependency.Number  End Using |

#### Controlling Property Behavior Using Isolate.WhenCalled()

Because a property expands to 2 methods, a getter and a setter, an alternative way to control a property is to use Isolate.WhenCalled().

#### When to Use

When your test requires a fake property to return a specific value.

##### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(() => <property>).WillReturn(<value>); |

VB

|  |
| --- |
| Using TheseCalls.WillReturn(<value>)  Dim dummy = fakeDependency.<property>()  End Using |

##### Samples

|  |  |
| --- | --- |
| ! | Note  To fake a set property, you need to place the statement into a block. |

C# (ControllingProperties.cs)

|  |
| --- |
| var fakeDependency = Isolate.Fake.Instance<Dependency>();  Isolate.WhenCalled(() => fakeDependency.Number).WillReturn(5); |

VB (ControllingProperties.vb)

|  |
| --- |
| Dim fakeDependency = FakeInstance(Of Dependency)()  Using TheseCalls.WillReturn(5)  Dim dummy = fakeDependency.Number()  End Using |

### Controlling PInvoke Methods

#### When to Use

When your test requires a pinvoke method to return a specific value.

#### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(() => <pInvoke-method()>).<behavior>); |

VB

|  |
| --- |
| Using TheseCalls.<Behavior>  Dim dummy = pInvoke-method()  End Using |

##### Samples

C#

|  |
| --- |
| Isolate.WhenCalled(()=> <method>).WillReturn(false);  ControlingMethods.cs  [DllImport("kernel32.dll", SetLastError = true)]  [return: MarshalAs(UnmanagedType.Bool)]  static extern bool Beep(uint dwFreq, uint dwDuration);  [TestMethod] // Note: This could be any collection from mscorlib which implements IEnumerable and has default ctor  public void PInvokeMethod\_Example()  {  Isolate.WhenCalled(()=> Beep(0,0)).WillReturn(false);  var result = Beep(100, 200);  Assert.AreEqual(false,result);  } |

VB (ControlingMethods.vb)

|  |
| --- |
| <TestMethod(), Isolated()> \_  Public Sub PInvokeMethod\_Example()  Using TheseCalls.WillReturn(False)  Beep(0, 0)  End Using  Dim result = Beep(100, 200)  Assert.AreEqual(False, result)  End Sub |

### Controlling Static Methods Behavior

Typemock Isolator supports faking and verifying the behavior of static methods as well as the behavior of instance methods.

#### Defining a Default Behavior for Static Methods

There is no need to create a fake instance in order to fake static methods.

##### When to Use

When your test requires a specific return value from the method.

##### Syntax

C#

|  |
| --- |
| Isolate.Fake.StaticMethods<Dependency>(); |

VB

|  |
| --- |
| FakeSharedMethods(Of Dependency) |

The following table explains possible behavior types:

| Behavior | Description |
| --- | --- |
| **ReturnRecursiveFakes** | Returns:   * For reference types: fake objects * For other return types: zero or equivalent   The returned fake objects will behave in the same way. |
| **CallOriginal** | Calls to fake static methods will pass through to their original implementation. |
| **MustSpecifyReturnValues** | Calls to fake object methods that return a specific value. If a method that returns the specified value is called without being set up in first place, an exception is thrown, and the test will fail. Void calls are still ignored. |
| **ReturnNulls** | Calls to fake object methods will return:   * zero or equivalent: to any call of methods that return value types * null: for any methods that return a reference type.   Void calls are still ignored. |

|  |  |
| --- | --- |
| ! | Note  Alternatively, you can use Isolate.WhenCalled(). |

##### Samples

The following sample shows how to test static methods.

|  |  |
| --- | --- |
| ! | Note  You can fake and verify chained static calls. |

C# (StaticMethods.cs)

|  |
| --- |
| [TestMethod]  public void FakeOneStaticMethod()  {  Isolate.WhenCalled(()=>Dependency.CheckSecurity(null,null)).IgnoreCall();  var result = new ClassUnderTest().Calculate(1, 2);  Assert.AreEqual(3, result);  }  public int Calculate(int a, int b)  {  Dependency.CheckSecurity("typemock", "rules");  return a + b;  } |

VB (StaticMethods.vb)

|  |
| --- |
| <TestMethod()> \_  Public Sub FakeOneStaticMethod()  Using TheseCalls.WillBeIgnored  Dependency.CheckSecurity(Nothing, Nothing)  End Using  Dim result = New ClassUnderTest().Calculate(1, 2)  Assert.AreEqual(3, result)  End Sub  Public Function Calculate(a As Integer, b As Integer) As Integer  Dependency.CheckSecurity("typemock", "rules")  Return a + b  End Function |

#### Faking All Static Methods of a Type

##### When to Use

You can fake static methods of a type when your test requires all static methods of the type to be faked.

##### Syntax

C#

|  |
| --- |
| Isolate.Fake.StaticMethods(typeof(Dependency)); |

VB

|  |
| --- |
| FakeSharedMethods(GetType(Dependency)) |

### Faking Extension Methods

Extension methods enable you to extend other classes with methods of your own. Extension methods cannot access private or protected members of the extended class. The extension class is a static class (which means all methods and fields are static).

##### When to Use

When your test requires a specific return value from an extension method.

|  |  |
| --- | --- |
| ! | Note  Faking Instance.Extension is equivalent to faking Type.Extension static method. |

##### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(() => myObject.<extension>()).<behavior>; |

VB

Not supported.

##### Samples

The following sample shows how to fake the DoubleInt() method. After faking the method, it will return 100 instead of 4.

C#

|  |
| --- |
| [TestMethod]  [Isolated]  public void TestExtensionMethods()  {  Isolate.WhenCalled(() => 1.DoubleInt()).WillReturn(100);  int n = 1;  Assert.AreEqual(100, n.DoubleInt());  } |

VB

Not supported.

### Controlling Non-Public Methods

You can control non-public methods by doing the following:

* Faking private instance methods.
* Faking private static methods
* Faking private properties and indexers
* Creating instances of types with private constructors

#### When to Use

When your test requires a specific return value from a non-public fake method.

Public methods are also supported with Isolate.NonPublic. This enables you to change the visibility of your methods without changing your tests.

#### Faking Private Methods

To fake private methods, use the Isolate.NonPublic. This API uses a string name of the method that you want to fake. Typemock Isolator will find the method through reflection and arrange the new behavior. To fake an instance method, pass the instance and the name of the method.

|  |  |
| --- | --- |
| ! | Note  To allow production code to change its method scope without failing unit tests, Isolate.NonPublic can be used for public methods too. |

##### Syntax

C#

|  |
| --- |
| Isolate.NonPublic.WhenCalled(<instance>, "<methodname>").<behavior> |

VB

|  |
| --- |
| NonPublic<behavior>(<instance>, "<methodname>", <value>) |

The following table explains possible behaviors:

| Behavior (C#) | Behavior (VB) | Description |
| --- | --- | --- |
| ReturnRecursiveFakes | Not supported | Returns:   * For reference types: fake objects * For other return types: zero or equivalent   The returned fake objects will behave in the same way. |
| WillReturn | NonPublicWillReturn | Returns a specified value. |
| WillThrow | NonPublicWillThrow | An exception will be thrown when this method will be called. |
| WillBeIgnored | NonPublicWillBeIgnored | The method will not be executed. |
| CallOriginal | NonPublicWillCallOriginal | Calls the original method. |
| AssignRefOut | Not supported | Additional information for Ref and Out parameters to be returned by the called method. |

Note that the only difference from faking public methods is the use of string names to represent the method to fake.

##### Samples

C# (PrivateMethods.cs)

|  |
| --- |
| var fakeDependency = Isolate.Fake.Instance<Dependency>(Members.CallOriginal);  Isolate.NonPublic.WhenCalled(fakeDependency, "InternalNumber").WillReturn(3); |

VB (PrivateMethods.vb)

|  |
| --- |
| Dim fakeDependency = FakeInstance(Of Dependency)(Members.CallOriginal)  NonPublicWillReturn(fakeDependency, "InternalNumber", 3) |

#### Faking Private Static Methods

To fake a private static method, pass the type and name of the method.

##### Syntax

C#

|  |
| --- |
| Isolate.NonPublic.WhenCalled<type>("<methodname>").<behavior> |

VB

|  |
| --- |
| NonPublic<behavior>( GetType (<type>), "<methodname>", <value>) |

##### **Samples**

C#

|  |
| --- |
| // Use this line to set expectation on private static method  Isolate.NonPublic.WhenCalled(typeof(Dependency), "PrivateStaticMethod").WillReturn(10);  // Or you can use this line instead to set the same expectation  Isolate.NonPublic.WhenCalled<Dependency>("PrivateStaticMethod").WillReturn(10); |

VB

|  |
| --- |
| NonPublicWillReturn(GetType(Dependency), "PrivateStaticMethod", 10) |

#### Faking Private Properties or Indexers

To fake a private property or indexer, use one of the following specialized Isolate.NonPublic APIs:

| Behavior Type | Description |
| --- | --- |
| NonPublic.Property | Fakes a private property |
| NonPublic.Indexer | Fakes a private indexer |

Then use one of the following:

| Behavior Type | Description |
| --- | --- |
| WhenGetCalled | Fakes the property getter (var x = aClass.Password) |
| WhenSetCalled | Fakes the property setter (myClass.Password = "typemock rocks";) |

##### Syntax

C#

|  |
| --- |
| Isolate.NonPublic.Property.WhenGetCalled(fakeDependency, "PrivateProp")  Isolate.NonPublic.Property.WhenSetCalled(fakeDependency, "PrivateProp") |

VB

|  |
| --- |
| NonPublicWillReturn(fake, "get\_<PrivateProperty>",3)  NonPublicWillReturn(fake, "set\_<PrivateProperty>",3) |

##### Samples

###### Sample 1: Faking Values Returned from a Private Property

To set a behavior on a private get or set property, use the same Isolate.NonPublic function (that is NonPublicWillReturn) with get\_PropertyName and set\_PropertyName as the method name.

To set a behavior on a private indexer, use the same Isolate.NonPublic function (that is NonPublicWillReturn) with get\_Item and set\_Item as the method name.

|  |  |
| --- | --- |
| ! | Note  You can set static properties by passing a type instead of an instance. |

|  |  |
| --- | --- |
| ! | Note  Indexers cannot be static. |

C#

|  |
| --- |
| var fakeDependency = Isolate.Fake.Instance<Dependency>(Members.CallOriginal);  Isolate.NonPublic.Property.WhenGetCalled(fakeDependency, "PrivateProp").WillReturn(3); |

VB

|  |
| --- |
| Dim fake = FakeInstance(Of Dependency)(Members.CallOriginal)  NonPublicWillReturn(fake, "get\_PrivateProperty",3) 'expectation for a get private property |

###### Sample 2: Faking a Private Property when the Private Method Has Generic Arguments

The following sample shows how to fake a private property when the private method has generic arguments.

C#

|  |
| --- |
| Isolate.NonPublic.WhenCalled(fakeDependency, "PrivateProp").WithGenericArguments  (typeof(MyClass)).WillReturn(3);  [TestMethod]  public void PrivateGenericMethod\_Return()  {  Isolate.NonPublic.WhenCalled<Dependency>("PrivateCallGuardGeneric").WithGenericArguments(typeof(int)).WillReturn(3);  var result = Dependency.CallsGuardGeneric<int>();  Assert.AreEqual(3,result);  }  private static T PrivateCallGuardGeneric<T>()  {  return default(T);  } |

### Faking Ref and Out Parameters

To return Refand Out parameters, set the value of these arguments before faking the call to the method.

#### When to Use

You can fake Ref and Out parameters when your test requires a specific return value from them.

#### Syntax

C#

|  |
| --- |
| Var outValue = <value>;  Isolate.WhenCalled(() => fake.SomeMethod(ref outValue)).<behavior> |

VB

Not supported.

#### Samples

The following sample shows how to fake the first and second parameter of SomeMethod() as follows:

* Required return values are assigned to the variables.
* The variables are passed to the fake method in Isolate.WhenCalled().

C# (RefOutArguments.cs)

|  |
| --- |
| [TestMethod]  public void RefArgument\_Example()  {  string outStr = "typemock";  List<int> outList = new List<int>() { 100 };  var fake = new Dependency();  Isolate.WhenCalled(() => fake.SomeMethod(ref outStr, out outList)).IgnoreCall();  var result = new ClassUnderTest().GetString(fake);  Assert.AreEqual("typemock1", result);  }  public void SomeMethod(ref string name, out List<int> list)  {  throw new NotImplementedException();  } |

### Faking Non-Public Ref and Out Parameters

To return Refand Out parameters of non-public methods, set the value of these arguments before faking the call to the method.

#### When to Use

When your test requires a specific return value from Ref and Out parameters.

#### Syntax

C#

|  |
| --- |
| var fakeParam = <result>;  Isolate.NonPublic.WhenCalled<Dependency>("method").AssignRefOut(fakeParam).<bahavior>; |

#### Samples

C# (PrivateMethods.cs)

|  |
| --- |
| [TestMethod]  public void VerifyPrivateStaticMethodWithRef\_Return()  {  int fakeParam = 3;  Isolate.NonPublic.WhenCalled<Dependency>("PrivateMethodOutParam").AssignRefOut(fakeParam).IgnoreCall();  var result = Dependency.CallPrivateMethodOutParam();  Isolate.Verify.NonPublic.WasCalled(typeof(Dependency), "PrivateMethodOutParam");  Assert.AreEqual(3,result);  } |

### Controlling Collections and Indexers

To return fake collections which are enumerable, use Isolate.WhenCalled.WillReturnCollectionValuesOf()**.**

When Isolate.WhenCalled.WillReturnCollectionValuesOf()is used, Typemock Isolator will **implicitly** replace calls of GetEnumerator to the original collection with calls to GetEnumerator of the test collection.

#### When to Use

You can control collections and indexers when your test requires that a fake method to return an enumerable collection.

#### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(() => <your\_collection>).WillReturnCollectionValuesOf(<items>); |

VB

Not supported.

#### Samples

|  |  |
| --- | --- |
| ! | Note  Typemock Isolator does not replace collections that are defined in mscorlib.dll. |

C# (Collections.cs)

|  |
| --- |
| [TestMethod]  [Isolated]  public void SwapCollection\_WithFakeData()  {  var dependency = new Dependency();  Isolate.WhenCalled(() => dependency.GetList()).WillReturnCollectionValuesOf(new int[] { 1, 2, 3 });  var result = new ClassUnderTest().Sum(dependency);  Assert.AreEqual(6, result);  }  public int Sum(Dependency dependency)  {  int total = 0;  foreach (var i in dependency.GetList())  {  total += i;  }  return total;  } |

#### Faking Indexers

When you fake indexers, only the specified index values will behave according to the specified behavior. Other indices will behave according to the fake object default behavior.

##### When to Use

You can fake indexers when your test requires to fake an indexer method.

##### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(()=> <fake>[<key>]).<behavior>; |

VB

Not supported.

##### Samples

The following sample shows how to fake the behavior index 2 of the indexer. All other indexes call the original implementation.

C#

|  |
| --- |
| public class ClassWithIndexer  {  public string this[int index]  {  get { return "Cat"; }  }  }  [TestMethod]  public void TrueIndexersTest()  {  var fake = Isolate.Fake.Instance<ClassWithIndexer>(Members.CallOriginal);  // We fake only the return value of index 2  Isolate.WhenCalled(()=> fake[2]).WillReturn("Dog");  // Here we call non fake indexer  Assert.AreEqual("Cat", fake[0]);  // Calling our fake indexer  Assert.AreEqual("Dog", fake[2]);  } |

##### Default Values of Undefined Indexes

Creating fake collections just to access a specific index can be a tedious task. Typemock Isolator automatically creates a complete collection as soon as Isolate.WhenCalled() is called on a specific integer index.

To implicitly create a fake collection, call the last index of the collection that you want to fake in Isolate.WhenCalled(). For example, Isolate.WhenCalled(() => myCollection[19]) will create 20 item collections.

###### Samples

|  |  |
| --- | --- |
| ! | Note  This feature works only on collections that are not defined in the mscorlib.dll assembly. In addition, this feature works only when using indexers with an integer argument. Typemock Isolator cannot identify the collection size if the indexer argument is not a numeric type. |

C# (Collections.cs)

|  |
| --- |
| [TestMethod]  [Isolated]  public void ImplictCollectionCreation\_ByFakingLastItem()  {  var dependency = new Dependency();  // A fake collection of size of 6 is created  Isolate.WhenCalled(() => dependency.GetList()[5]).WillReturn(3);  var result = new ClassUnderTest().Count(dependency);  Assert.AreEqual(6, result);  Assert.AreEqual(3, dependency.GetList()[5]);  } |

### Faking LINQ Queries

You can fake the result of a LINQ query.

##### When to Use

When your test requires to fake a SQL query or query result.

#### Syntax

C#

|  |
| --- |
| Isolate.WhenCalled(()=><linq\_query>.WillReturnCollectionValuesOf(fakedList); |

VB

Not supported.

#### Samples

##### Sample 1: Faking the Result of a LINQ Query

The following sample shows how to fake the result of a LINQ query.

C#

|  |
| --- |
| public void ProcessCustomerList()  {  var nycCustomers =  from customer in customerList.USCustomers  where customer.City == "NYC"  select customer;  // Process list  } |

The following sample shows how to fake the entire query and return your own customer list instead.

C# (ControllingMethods.cs)

|  |
| --- |
| Isolate.WhenCalled(()=>  from customer in customerList.USCustomers  where customer.City == "NYC"  select customer)  .WillReturn(fakedList); |

##### Sample 2: Returning a Queryable Fake Result

If the customer list is retrieved from the database, you might want to return a faked set from the USCustomers property. The following sample shows how to fake the USCustomers property and return it as IQueryable.

C# (ControllingMethods.cs)

|  |
| --- |
| Isolate.WhenCalled(() => customerList.USCustomers).WillReturn(fakedList as IQueryable); |

## Calling Tested Methods

### Invoking Private Methods

To invoke private methods, use Isolate.Invoke.Method().

#### When to Use

You can invoke private methods when your test requires a private method to be called.

#### Syntax

C#

|  |
| --- |
| Isolate.Invoke.Method(<instance>, "<method\_name>", <list\_of\_arguments>); |

VB

Not supported.

#### Samples

##### Sample 1: Invoking an Instance Private Method

The following sample shows how to invoke an instance private method. Arguments which are passed to the method from Isolate.Invoke.Method() are 2 and 5.

C# (InvokingMethods.cs)

|  |
| --- |
| [TestMethod]  public void InvokePrivateMethod()  {  var underTest = new ClassUnderTest();  var result = Isolate.Invoke.Method(underTest, "Sum", 2, 5);  Assert.AreEqual(7, result);  } |

##### Sample 2: Calling a Base Method

You can use Isolate.Invoke.Method() to call non-public methods that are defined in a base class of the object. To call an overridden method in the base class, use Isolate.Invoke.MethodFromBase().

The following sample shows how to a base method (this works for both hidden and virtual methods).

C# (InvokingMethods.cs)

|  |
| --- |
| Isolate.Invoke.MethodFromBase<what-base-type>(<instance>,"method-name", list-of-args);  [TestMethod]  public void InvokeBaseMethod()  {  var underTest = new ClassUnderTest();  var result = Isolate.Invoke.MethodFromBase<ClassUnderTest>(underTest,"Subtract", 2, 5);  Assert.AreEqual(7, result);  } |

##### Sample 3: Invoking a Static Method

C# (InvokingMethods.cs)

|  |
| --- |
| [TestMethod, Isolated]  public void InvokePrivateStaticMethod()  {  var result = Isolate.Invoke.Method<ClassUnderTest>("Multiply", 2, 5);  Assert.AreEqual(10, result);  } |

##### Sample 3: Passing Arguments

When a method has arguments that need to be passed as Ref or Out parameters, you can use the Args class to create the arguments. The following sample shows how to pass three arguments as follows:

* One argument is passed by value
* One argument is passed as a Ref parameter
* One argument is passed as an Out parameter

To read the returned value the IBox<int> property is used.

C# (InvokingMethods.cs)

|  |
| --- |
| [TestMethod]  public void InvokeMethod\_WithIntRefAndOutArgument\_RefAndOutValuesAreAssigned()  {  var t = new ClassWithPrivateMethods();  IBox<int> val1 = Args.Ref(10);  IBox<int> val2 = Args.Out<int>();  var result = (int)Isolate.Invoke.Method(t, "MethodWithIntRefAndOutArgument", 10, val1, val2);  Assert.AreEqual(result, 4);  Assert.AreEqual(val1.Value, 1);  Assert.AreEqual(val2.Value, 2);  } |

##### Sample 4: Handling Overloaded Methods Passing Null Values as Parameters

Typemock Isolator automatically calls the correct method based on the arguments you passed. However, when nulls are sent, you cannot collect that information from the passed arguments. To be able to collect this information, use Args.Null().

The following sample shows that in order to pass values for Nullable types as parameters, the type is specified using Args.Null().

C# (InvokingMethods.cs)

|  |
| --- |
| [TestMethod]  public void InvokeMethod\_CreateNullArgumentForOverloadedMethod\_NullableType()  {  var nullParamToPrivateMethod = new NullParamToPrivateMethod();  var result = (int)Isolate.Invoke.Method(nullParamToPrivateMethod, "CallMe", Args.Null<int?>());  Assert.IsTrue(result == 1);  } |

##### Sample 3: Passing Ref Arguments

To pass Ref or Out parameters, use Args.Out or Args.Ref as follows:

1. Create a Ref or Out object
2. Pass it to the method
3. Get the value.

C# (InvokingMethods.cs)

|  |
| --- |
| [TestMethod]  public void InvokePrivateMethodWithRef()  {  var byRef = Args.Ref(5);  Isolate.Invoke.Method<ClassUnderTest>("MultiplyByRef", byRef, 2);  var result = byRef.Value;  Assert.AreEqual(10, result);  } |

### Invoke Static Constructors

Because a static constructor for a type is executed only once, when you fake a static constructor, you need to invoke it in a test that requires a normal execution. While Typemock Isolator does this automatically, you can force a static constructor call.

#### When to Use

When you want to make sure that the type is reinitialized (for example, when testing a singleton and making sure the instance is null).

#### Syntax

C#

|  |
| --- |
| Isolate.Invoke.StaticConstructor(typeof(<type>)); |

#### Samples

C# (StaticMethods.cs)

|  |
| --- |
| [TestMethod]  public void CallingStaticConstructorTest()  {  StaticConstructorExample.TrueOnStaticConstructor = false;  // force static constructor to be called  Isolate.Invoke.StaticConstructor(typeof(StaticConstructorExample));  Assert.IsTrue(StaticConstructorExample.TrueOnStaticConstructor);  } |

### Firing Events

To fire static or instance events, use Isolate.Invoke.Event()*.*

#### When to Use

When your test requires an event to be fired.

#### Syntax

C#

|  |
| --- |
| Isolate.Invoke.Event(() => <event> += <optional list of arguments>); |

|  |  |
| --- | --- |
| ! | Note  When no arguments are sent, a default of (this,EventArgs.Empty) is sent. |

VB

Not supported.

#### Samples

The following sample shows how to fire an event. The syntax () => underTest.RunEvent+= nullmeans that the LogEntryCreated event should be fired.

C# (InvokingMethods.cs)

|  |
| --- |
| [TestMethod]  public void FireEvent\_RunEvent()  {  var underTest = new ClassUnderTest();  var counter = new Counter(underTest);  // Note how adding a dummy event is the way to fire it  Isolate.Invoke.Event(() => underTest.RunEvent+= null, 0);  Assert.AreEqual(1, counter.Times);  } |

### Creating Instances of Types with Private Constructors

#### When to Use

When you need to instantiate a type that does not have a public constructor. Typemock Isolator creates the object from the specified type using the provided argument list and matches the correct constructor.

#### Syntax

|  |
| --- |
| var obj = Isolate.NonPublic.CreateInstance<type>(<list of arguments>); |

#### Sample

C# (PrivateMethods.cs)

|  |
| --- |
| [TestMethod]  public void CreateInstanceWithAPrivateConstructor()  {  ClassWithNoPublicConstructor obj = Isolate.NonPublic.CreateInstance<ClassWithNoPublicConstructor>(1, 2);  Assert.AreEqual(obj.x, 1);  Assert.AreEqual(obj.y, 2);  } |

## Validating

When mocking, you might need to verify that a specific call was actually made, and the correct parameters were passed into that call.

### Verifying Method Calls

To verify that a method was called, use Isolate.Verify.<Verification\_Statement>.

#### When to Use

When you want to check a case when a specific method is called.

#### Syntax

C#

|  |
| --- |
| Isolate.Verify<Verification\_Kind>(() => fakeDependancy.<method>); |

VB

|  |
| --- |
| Using AssertCalls. <Verification\_Kind>  fakeDependancy.<method>  End Using |

The following table explains the possible verification statements:

| Verification Statement (C#) | Verification Statement (VB) | Description |
| --- | --- | --- |
| WasCalledWithAnyArguments() | HappenedWithAnyArguments() | Passes the verification if during the test execution, the call was made at least once. |
| WasCalledWithExactArguments() | HappenedWithExactArguments() | Passes the verification, the call was made at least once with matching arguments. |
| WasCalledWithArguments() | Not supported | Passes the verification if the call was made at least once with arguments that match the predicate |
| GetTimesCalled() | Not supported | Returns the number of times that the method was called, which later can be asserted on. |
| WasNotCalled() | NeverHappened() | Passes the verification if the call was never made. |

|  |  |
| --- | --- |
| ! | Note  Unlike Isolate.WhenCalled() which can refer to all overloaded methods, Isolate.Verify() always checks a specific method, according to the method signature. |

|  |  |
| --- | --- |
| ! | Note  You can perform verification on a chain of methods. When one of the methods in the chain returns an unsupported type (including collections in mscorlib), Typemock Isolator throws an exception. |

#### Samples

##### Sample 1: Verifying whether the Method Was Called with Any Arguments

C# (AssertingCallsWhereMade.cs)

|  |
| --- |
| Isolate.Verify.WasCalledWithAnyArguments(() => fakeDependancy.CheckSecurity(null, null)); |

VB

|  |
| --- |
| Using AssertCalls.HappenedWithAnyArguments  fake.Method()  End Using |

##### Sample 2: Verifying that the Method Was Not Called

The following sample shows how to verify that the method was never called.

C# (AssertingCallsWhereMade.cs)

|  |
| --- |
| Isolate.Verify.WasNotCalled(() => fakeDependancy.CallGuard().CheckSecurity(null,null)); |

VB

|  |
| --- |
| Using AssertCalls.NeverHappened  fake.GetSon().DoSomething()  End Using |

##### Sample 3: Verifying that the Method Was Called With Specific Arguments

The following sample shows how to verify more complex argument verification with a custom logic, using Isolate.Verify.WasCalledWithArguments() and Matching().

C# (AssertingCallsWhereMade.cs)

|  |
| --- |
| Isolate.Verify.WasCalledWithArguments(() => fakeDependancy.CheckSecurity(null, null)).Matching(  a => (a[0] as string).StartsWith("type") &&  (a[1] as string).StartsWith("rule")); |

### Verifying Static Methods

To verify that a static method was called, use the same API as fake objects. You can verify calls only on the types that were activated with Isolate.WhenCalled().

#### When to Use

When you want to check a case when a specific static method is called.

##### Syntax

C# (StaticMethods.cs)

|  |
| --- |
| Isolate.Verify<Verification\_Kind>(() => <static\_method>); |

VB

|  |
| --- |
| Using AssertCalls. <Verification\_Kind>  <static\_method>  End Using |

##### Samples

The following sample shows how to verify whether a static method was called.

C# (StaticMethods.cs)

|  |
| --- |
| [TestMethod]  public void VerifyStaticMethodWasCalled()  {  Isolate.Fake.StaticMethods<Dependency>(Members.ReturnRecursiveFakes); // Must Fake At Least One Static Method  var result = new ClassUnderTest().Calculate(1, 2);  Isolate.Verify.WasCalledWithAnyArguments(() => Dependency.CheckSecurity(null, null));  } |

VB (StaticMethods.vb)

|  |
| --- |
| <TestMethod()> \_  Public Sub VerifyStaticMethodWasCalled()  FakeSharedMethods(Of Dependency)(Members.ReturnRecursiveFakes)  Dim result = New ClassUnderTest().Calculate(1, 2)  Using AssertCalls.HappenedWithAnyArguments  Dependency.CheckSecurity(Nothing, Nothing)  End Using  End Sub |

### Verifying Non-Public Methods

#### When to Use

When you want to check a case when a specific private method is called.

To verify that a non-public method was called, use Isolate.Verify.NonPublic() with one of the following verification statements:

| Statement | Description |
| --- | --- |
| WasCalled | Passes the verification if during the test execution, the call was made at least once. |
| WasNotCalled | Passes the verification if the call was never made. |

To verify properties and indexers, add the following before the verification statement:

| Behavior Type | Description |
| --- | --- |
| Property | Specialized API for properties. |
| Indexer | Specialized API for indexers. |

##### Syntax

| Method Type | API |
| --- | --- |
| Instance | instance Isolate.Verify.NonPublic.WasCalled(<**instance**>, "PrivateMethodname"); |
| Static | Isolate.Verify.NonPublic.<WasCalled>(**typeof(Dependency**), "PrivateMethodname");. |
| Static Method Generic | Isolate.Verify.NonPublic.WasCalled(typeof(<type>), "<method>", <list of generic types>); |

C#

|  |
| --- |
| static Isolate.Verify.NonPublic.<WasCalled>(typeof(Dependency), "PrivateMethodname");  instance Isolate.Verify.NonPublic.WasCalled(<instance>, "PrivateMethodname");  property Isolate.Verify.NonPublic.Property.WasCalledGet(realDependency, "PrivateProp");  Isolate.Verify.NonPublic.Indexer.WasCalledGet(realDependency); |

VB

Not supported.

Public methods are also supported with the Isolate.NonPublic API. This allows you to change the visibility of your methods without having to change the tests.

##### Samples

###### Sample 1: Verifying the Non-Public Method Was Called

The following sample shows how to verify that the non-public method was called.

C# (PrivateMethods.cs)

|  |
| --- |
| [TestMethod]  public void VerifyPrivateStaticMethod\_WasCalledWithAnyArg()  {  Isolate.NonPublic.WhenCalled<Dependency>("CallGuard").IgnoreCall();  var classUnderTest = new ClassUnderTest();  var result = classUnderTest.Calculate(1, 2);  Isolate.Verify.NonPublic.WasCalled(typeof (Dependency), "CallGuard");  } |

VB

Not supported.

###### Sample 2: Verifying the Property and Indexer Was Called

The following sample shows how to verify the property and indexer.

C#

|  |
| --- |
| var fake = Isolate.Fake.Instance<Dependency>(Members.CallOriginal);  Isolate.NonPublic.Property.WhenGetCalled(fake, "PrivateProperty").WillReturn(10);  Isolate.NonPublic.Indexer.WhenSetCalled(fake).IgnoreCall();  // Act Here  Isolate.Verify.NonPublic.Property.WasCalledGet(fake, "PrivateProperty"); //verification for private get property  Isolate.Verify.NonPublic.Indexer.WasCalledSet(fake); //verification for private set indexer |

###### Sample 3: Verifying that the Private Method Was Called With Specific Arguments

Typemock Isolator does not match arguments of the verified called methods. To verify that a private method was called with a specific arguments, use Isolate.Verify.NonPublic.WasCalled.WithArguments() and pass the matching arguments.

C#

|  |
| --- |
| Isolate.Verify.NonPublic.WasCalled(fake, "PrivateMethod").WithArguments(<arguments>) |

C# (PrivateMethods.cs)

|  |
| --- |
| [TestMethod]  public void VerifyPrivateMethod\_WasCalledWithExactArg\_Example()  {  Isolate.NonPublic.WhenCalled<Dependency>("CallGuard").IgnoreCall();  var result = new ClassUnderTest().Calculate(1, 2);  Isolate.Verify.NonPublic.WasCalled(typeof(Dependency), "CallGuard").WithArguments("typemock", "rocks");  } |

###### Sample 4: Verifying that a Private Static Generic Method Was Called

C#

|  |
| --- |
| [TestMethod]  public void VerifyPrivateGenericMethod\_WasCalled()  {  Isolate.NonPublic.WhenCalled<Dependency>("PrivateCallGuardGeneric").WithGenericArguments(typeof(int)).WillReturn(3);  var result = Dependency.CallsGuardGeneric<int>();  Isolate.Verify.NonPublic.WasCalled(typeof (Dependency), "PrivateCallGuardGeneric", typeof (int));  } |

### Verifying the Number of Times a Method Was Called

You might want to count how many times a method was called. While using Isolate.Verify.<Verification\_Statement> passes for at least one call, it might be useful to know the exact number of calls made.

To get the number of calls to a method, use Isolate.Verify.GetTimesCalled().

#### When to Use

When the number of times a specific method is called is important for the test.

|  |  |
| --- | --- |
| ! | Note  Do not overuse this feature because it might make your test brittle. |

#### Syntax

C#

|  |
| --- |
| int count = Isolate.Verify.GetTimesCalled(() => <method>); |

VB

Not supported.

#### Samples

The following sample shows how to get the number of times that fakeBag.AddItem() was called.

|  |  |
| --- | --- |
| ! | Note  Use Verify.GetTimesCalled() to count calls on any public instance or static methods. |

|  |  |
| --- | --- |
| ! | Note  Verify.GetTimesCalled() returns total call count, regardless of arguments. |

C#

|  |
| --- |
| [TestMethod]  public void GetTimesCalledExample\_FillBag()  {  var fakeBag = Isolate.Fake.Instance<Bag>();  var thief = new Shoplifter();  thief.FillBag(fakeBag);  int count = Isolate.Verify.GetTimesCalled(() => fakeBag.AddItem(""));  Assert.AreEqual(2, count);  }  public class Shoplifter  {  public void FillBag(Bag bag)  {  bag.AddItem("Stolen Watch");  bag.AddItem("Stolen Necklace");  }  } |

### Verifying that a Future Instance Was Created

You can verify whether a specific instance was created.

#### When to Use

When it is important that an instance was created.

#### Syntax

C# (AssertCallsWhereMade.cs)

|  |
| --- |
| Isolate.Fake.NextInstances<type>();  Isolate.Verify.WasCalledWithAnyArguments (() => new Dependency()); |

#### Samples

The following sample shows how to verify that a future instance was created.

C# (AssertCallsWhereMade.cs)

|  |
| --- |
| [TestMethod]  public void Verify\_AFutureInsance\_WasCreated()  {  var fakeDependencyHandle = Isolate.Fake.NextInstance<Dependency>();  var dependancy1 = DependancyFactory.Create();  Isolate.Verify.WasCalledWithAnyArguments(() => new Dependency());  } |

### Verifying the Number of Created Instances

You can count how many times an instance was created.

#### When to Use

When the number of instances created is important for the test.

#### Syntax

C# (AssertCallsWhereMade.cs)

|  |
| --- |
| Isolate.Fake.AllInstances<type>();  …  Isolate.Verify.GetTimesCalled(() => new Dependency()); |

#### Samples

The following sample shows how to verify how many instances was created.

C# (AssertCallsWhereMade.cs)

|  |
| --- |
| [TestMethod]  public void Verify\_FutureInsance\_WasCreated3Times()  {  var fakeDependency = Isolate.Fake.AllInstances<Dependency>();  var dependancy1 = DependancyFactory.Create();  var dependancy2 = DependancyFactory.Create();  var dependancy3 = DependancyFactory.Create();  var count = Isolate.Verify.GetTimesCalled(() => new Dependency());  Assert.AreEqual(3, count);  } |

## Samples and Cookbook

This section provides samples that show how to use the Typemock Isolator Mocking API.

#### Sample 1: Ignoring a Method Call

This sample shows:

* How to ignore a method call
* How to verify that the method was actually called during the test

##### APIs Used

| Method | Description |
| --- | --- |
| Isolate.WhenCalled() | The method inside the Isolate.WhenCalled() will be ignored and will return a fake value. |
| Isolate.Verify.WasCalledWithExactArguments() | Verify that the method inside the Isolate.WhenCalled() was called during the test with the exact arguments. |

##### Scenario

1. Set all calls to MessageBox.Show() to be ignored and return a fake value.
2. Call MessageBox.Show(). When running the test the MessageBox does not appear.
3. Check that the static function Show() was called during the test with the parameter "This is a message".

##### Code

C#

|  |
| --- |
| [TestMethod]  public void SimpleTestUsingMessageBox()  {  // Arrange  Isolate.WhenCalled(()=>MessageBox.Show(null)).WillReturn(DialogResult.OK);  // Act  MessageBox.Show("This is a message"); <br>  // Assert  Isolate.Verify.WasCalledWithExactArguments(()=>MessageBox.Show("This is a message"));  } |

VB

|  |
| --- |
| Public Sub SimpleTestUsingMessageBox()  ' Arrange<br>  Using TheseCalls.WillBeIgnored  MsgBox(String.Empty)  End Using  ' Act  MsgBox("This is a message")  ' Assert  Using AssertCalls.HappenedWithExactArguments  MsgBox("This is a message")  End Using  End Sub |

#### Sample 2: Testing Interaction between Two Classes

This sample shows a test of the interaction between two classes:

* SomeClass, which has a shared sub called MyMethod.
* UserOfSomeClass, which has a sub called DoSomething.

This test checks that every exception thrown from SomeClass is shown to the user using MessageBox.

##### APIs Used

| API | Description |
| --- | --- |
| Isolate.WhenCalled(..).WillThrow (C#) TheseCalls.WillThrow (VB) | The method inside the Isolate.WhenCalled() will throw a specified exception when the method is called. |
| Isolate.WhenCalled(..).WillReturn (C#) TheseCalls.WillBeIgnored (VB) | The method inside the Isolate.WhenCalled() will be ignored and return a fake value. |
| Isolate.Verify.WasCalledWithExactArguments (C# and VB) | Verify that the method inside the Isolate.WhenCalled() was called during the test with the exact arguments. |

##### Scenario

1. Set a behavior: when MyMethod() is called, throw a specified exception.
2. Define that all calls to MessageBox.Show() are ignored and return a fake value.
3. Create a new instance of UserOfSomeClass and call DoSomething().
4. Check that MessageBox.Show() was called during the test with the string "Exception caught: foo".

##### Code

C#

|  |
| --- |
| public class SomeClass  {  public static void MyMethod()  {  // do work  }  }  public class UserOfSomeClass()  {  public void DoSomething()  {  try  {  SomeClass.MyMethod();  }  catch (Exception exc)  {  MessageBox.Show("Exception caught: " + exc.Message);  }  }  } |

VB

|  |
| --- |
| Public Class SomeClass  Public Shared Sub MyMethod()  ' do work  End Sub  End Class  Public Class UserOfSomeClass  Public Sub DoSomething()  Try  SomeClass.MyMethod()  Catch exc As Exception  MsgBox("Exception caught: " + exc.Message)  End Try  End Sub  End Class |

C#

|  |
| --- |
| [TestMethod]  public void ComplexTest()  {  // Arrange  Isolate.WhenCalled(()=>SomeClass.MyMethod()).WillThrow(new Exception("foo"));  Isolate.WhenCalled(()=>MessageBox.Show(String.Empty)).WillReturn(DialogResult.Cancel);  // Act  UserOfSomeClass user = new UserOfSomeClass();  user.DoSomething();  // Assert  Isolate.Verify.WasCalledWithExactArguments(()=>MessageBox.Show("Exception caught: foo"));  } |

VB

|  |
| --- |
| <TestMethod()>  Public Sub ComplexTest()  ' Arrange  Using TheseCalls.WillThrow(New Exception("foo"))  SomeClass.MyMethod()  End Using  Using TheseCalls.WillBeIgnored  MsgBox(String.Empty)  End Using  ' Act  Dim user As New UserOfSomeClass()  user.DoSomething()  ' Assert  Using AssertCalls.HappenedWithExactArguments  MsgBox("Exception caught: foo")  End Using  End Sub |

#### Mocking a SharePoint Site

This section provides the following samples:

* Faking items retrieval from a SharePoint site
* Explicitly faking an SPItem collection

##### Sample 1: Faking Items Retrieval from a SharePoint Site

###### Original Method

The following sample includes a function that retrieves all of the items from a list called Tasks. The list has the Urgent priority.

|  |
| --- |
| public List<string> GetUrgentTasks()  {  var site = new SPSite("http://sharepoint.typemock.com");  var taskList = site.OpenWeb().Lists[TASKS\_LIST\_NAME];  var urgentTasks = new List<string>();  foreach (SPListItem item in taskList.Items)  {  if (item[PriorityFieldName] != null &&  item[PriorityFieldName].ToString() == Priority.Urgent.ToString())  {  urgentTasks.Add(item.Name);  }  }    return urgentTasks;  } |

###### Typemock Isolator Features Used

* Recursive Fakes
* Swap Future Instance
* Implicit Collection Creation

###### Scenario

1. Fake the SPSite class. Because you need all of the object-model classes instantiated from SPSite to be faked as well, use the Recursive Fakes feature.
2. Because SPSite is instantiated inside the test code, use Swap.NextInstance to define that the next instance of SPSite will be replaced with the faked one.
3. Populate the fake SPList object with items. Using the implicit collection creation, you can define a collection of 10 items and populate them with only two items.
4. Call GetUrgentTasks() and check the result using Assertions.

###### Code

|  |
| --- |
| [TestMethod]  [Isolated]  public void GetUrgentTasks\_QuerySiteForAllTasksAndReturnTheNamesOfTheUrgentTasks\_TwoUrgentTasksFound()  {  // Fake recursively the next SPSite and all its returned classes are faked  var fakeSite = Isolate.Fake.NextInstance<SPSite>();  // Retrieve a fake SPList to set behavior  var fakeTaskList = fakeSite.OpenWeb().Lists[SharePointLogic.TASKS\_LIST\_NAME];  // Isolator will automatically create an internal fake list with 10 items in it (based on the highest indexer)  // Two of those items (1 and 9) have "Urgent" Priority ,while the rest are not important for the test  string urgentPriorityString = SharePointLogic.Priority.Urgent.ToString();  var priorityFieldName = SharePointLogic.PriorityFieldName;  Isolate.WhenCalled(() => fakeTaskList.Items[1][priorityFieldName]).WillReturn(urgentPriorityString);  Isolate.WhenCalled(() => fakeTaskList.Items[9][priorityFieldName]).WillReturn(urgentPriorityString);  Isolate.WhenCalled(() => fakeTaskList.Items[1].Name).WillReturn("Do the laundry");  Isolate.WhenCalled(() => fakeTaskList.Items[9].Name).WillReturn("Wash the dishes");    // Call The Method under test  var urgentTasks = classUnderTest.GetUrgentTasks();  Assert.AreEqual(2, urgentTasks.Count);  Assert.AreEqual("Do the laundry", urgentTasks[0]);  Assert.AreEqual("Wash the dishes", urgentTasks[1]);  } |

##### Sample 2: Explicitly Faking an SPItem Collection

###### Original Method

The following sample shows a method that returns all of the items in a specific list:

|  |
| --- |
| public List<SPItems> GetUrgentTasks()  {  var site = new SPSite("http://sharepoint.typemock.com");  var taskList = site.OpenWeb().Lists[TASKS\_LIST\_NAME];  var urgentTasks = new List<SPItem>();    foreach (SPItem item in taskList.Items)  {  urgentTasks.Add(item);  }  return urgentTasks;  } |

###### Typemock Isolator Features Used

* Recursive Fakes
* Swap Future Instance
* Replacing Collections

###### Scenario

1. Fake the SPSite class. Because you need all of the object-model classes instantiated from SPSite to be faked as well, use the Recursive Fakes feature.
2. Create a fakeItem object.
3. Explicitly declare a collection of three fakeItem objects that will be returned when SPList.Items() is called.
4. Call the method and check the returned values.

###### Code

|  |
| --- |
| [TestMethod]  [Isolated]  public void GetAllTasks\_ThreeItemsInTasksList\_ThreeTaskItemsFound()  {  // Fake recursively the next SPSite and all its nested classes are faked  var fakeSite = Isolate.Fake.NextInstance<SPSite>();  // Retrieve a fake SPList to set behavior  var fakeTaskList = fakeSite.OpenWeb().Lists[SharePointLogic var>.TASKS\_LIST\_NAME];  // Create a fake SPListItem  var fakeItem = Isolate.Fake.Instance<SPListItem>();  // List collection shell return three fake items// List collection shall return an array containing three fake items  IsolateWillReturnCollectionValuesOf(new[] {fakeItem, fakeItem, fakeItem});  // Call the method under test  var urgentTasks = classUnderTest.GetAllTasks();  Assert.AreEqual(3, urgentTasks.Count);  } |

#### Typemock Isolator Mocking API Cookbook

The following sections summarize the Mocking API methods.

##### Faking an Instance

| Method | Description |
| --- | --- |
| SealedClass fake = Isolate.Fake.Instance<SealedClass>(); (use Members.ReturnRecursiveFakes or Members.CallOriginal) | Creates fake instances (can be used of interfaces, abstract, concrete and sealed types). |
| SealedClass fake = Isolate.Swap.NextInstance<SealedClass>(). WithRecursiveFake(); | Fakes a future instance. |
| Isolate.Fake.AllInstances | Fakes all Instances. |
| UnderTest real = Isolate.Fake.Dependencies<UnderTest>([args])  var fake = Isolate.GetFake<F>(real); | Fakes all dependencies of a type. |
| Isolate.Fake.Instance<Derived>(Members.ReturnRecursiveFake,  ConstructorWillBe.Called); | Fakes all methods except the constructor. |
| Isolate.Fake.Instance<Derived>(Members.CallOriginal,  ConstructorWillBe.Called, BaseConstructorWillBe.Ignored); | Calls the original method and ignores the constructor. |
| Isolate.Fake.Instance<Derived>(Members.CallOriginal,  ConstructorWillBe.Called, BaseConstructorWillBe.Ignored); | Ignores the base class constructor. |
| Isolate.Fake.StaticConstructor<Dependency>(); | Fakes a static constructor. |
| var handle  =Isolate.Fake.NextInstance<Dependency>(Members.ReturnRecursiveFakes,  context=>  {  // action here  }); | Runs a custom logic. |
| Isolate.Swap.NextInstance<Dependency>()  .ConstructorWillThrow(new OutOfMemoryException()); | Fakes an error in a constructor. |

##### Changing the Method Behavior

| Method | Description |
| --- | --- |
| Isolate.WhenCalled(() => fake.Increment()).<behavior>; | Modifies the behavior of a method. |
| Isolate.WhenCalled(()=>dependency.<method>).<behavior>; | Fakes the behavior for a live object. |
| Isolate.WhenCalled(() => fakeDependency.GetID()).<behavior>;  Isolate.WhenCalled(() => fakeDependency.GetID()).<behavior>;  ... | Fakes a sequenced behavior. |
| Isolate.WhenCalled(() => <fake>.<method1>.<method2>…).<behavior>; | Fakes a chain of methods. |
| WhenCalled(() => <method>).WithExactArguments().<behavior>; | Fakes a method based on call arguments. |
| Isolate.WhenCalled((<type arg1>, <type arg1>) => fake.<method>  (<arg1>, <arg2>))  .AndArgumentsMatch((<arg1>, <arg2>) => <check>  .<behavior>; | Uses a custom checker on arguments. |
| Isolate.WhenCalled((<type arg1>, <type arg1>) => fake.<method>  (<arg1>, <arg2>, <additional\_arguments>))  .AndArgumentsMatch((<arg1>, <arg2>) => <check>  .<behavior>; | Mixes WithExactArguments and custom checkers. |
| var fakeDependency = Isolate.Fake.Instance<Dependency>();  fakeDependency.Number=<value>; // sets the property to return the value | Controls the behavior of a property using true properties. |
| Isolate.WhenCalled(() => <property>).WillReturn(<value>); | Controls the behavior of a property using Isolate.WhenCalled(). |
| Isolate.WhenCalled(() => <pInvoke-method()>).<behavior>); | Controls PInvoke methods. |
| Isolate.Fake.StaticMethods<Dependency>(); | Defines the default behavior for a static method. |
| Isolate.Fake.StaticMethods(typeof(Dependency)); | Fakes all static methods of a type. |
| Isolate.WhenCalled(() => myObject.<extention>()).<behavior>; | Fakes an extension method. |
| Isolate.NonPublic.WhenCalled(<instance>, "<methodname>").<behavior> | Fakes a private method. |
| Isolate.NonPublic.WhenCalled<type>("<methodname>").<behavior> | Fakes a private static method. |
| Isolate.NonPublic.Property.WhenGetCalled(fakeDependency, "PrivateProp")  Isolate.NonPublic.Property.WhenSetCalled(fakeDependency, "PrivateProp") | Fakes a private property or indexer. |
| Var outValue = <value>;  Isolate.WhenCalled(() => fake.SomeMethod(ref outValue)).<behavior> | Fakes a Ref and Out parameter. |
| var fakeParam = <result>;  Isolate.NonPublic.WhenCalled<Dependency>("method").AssignRefOut(fakeParam).<bahavior>; | Fakes a non-public Ref and Out parameter. |
| Isolate.WhenCalled(() =>  <your\_collection>).WillReturnCollectionValuesOf(<items>); | Controls collections and indexers. |
| Isolate.WhenCalled(()=> <fake>[<key>]).<behavior>; | Fakes an indexer. |
| Isolate.WhenCalled(()=><linq\_query>.WillReturnCollectionValuesOf(fakedList); | Fakes a LINQ query. |

##### Calling Tested Methods

| Method | Description |
| --- | --- |
| Isolate.Invoke.Method(<instance>, "<method\_name>", <list\_of\_arguments>); | Invokes a private method. |
| Isolate.Invoke.StaticConstructor(typeof(<type>)); | Invokes a static constructor. |
| Isolate.Invoke.Event(() => <event> += <optional list of arguments>); | Fires an event. |
| var obj = Isolate.NonPublic.CreateInstance<type>(<list of arguments>); | Creates an instance of a type with a private constructor. |

##### Validating

| Method | Description |
| --- | --- |
| Isolate.Verify<Verification\_Kind>(() => fakeDependancy.<method>); | Verifies a method call. |
| Isolate.Verify<Verification\_Kind>(() => <static\_method>); | Verifies a static method. |
| static Isolate.Verify.NonPublic.<WasCalled>(typeof(Dependency),  "PrivateMethodname");  instance Isolate.Verify.NonPublic.WasCalled(<instance>,  "PrivateMethodname");  property Isolate.Verify.NonPublic.Property.WasCalledGet(realDependency,  "PrivateProp");  Isolate.Verify.NonPublic.Indexer.WasCalledGet(realDependency); | Verifies a non-public method. |
| int count = Isolate.Verify.GetTimesCalled(() => <method>); | Verifies the number of times a method was called. |
| Isolate.Fake.NextInstances<type>();  Isolate.Verify.WasCalledWithAnyArguments (() => new Dependency()); | Verifies that a future instance was created. |
| Isolate.Fake.AllInstances<type>();  …  Isolate.Verify.GetTimesCalled(() => new Dependency()); | Verifies the number of created instances. |

# Running Tests

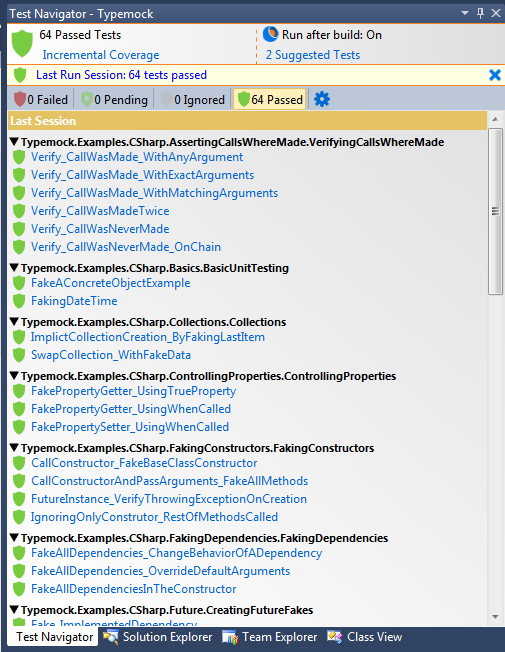
When you load a solution for the first time, you will see all your existing tests pending. You need to run these tests manually (see [Running All Pending Tests](#_D2HTopic_291)).

A Typemock Isolator’s component that runs test is called SmartRunner. After SmartRunner ran the solution’s tests for the first time, only those tests are run that were affected since the last build. This includes all modified tests and those tests that need to run because the code they test changed.

After tests are completed, you can view the result of each test along with their coverage directly in the code. Failed tests can be debugged and run again directly from the test itself or from the code.

## Understanding Test Navigator

The Test Navigator window is integrated into the MS Visual Studio user interface. The window displays the solution status. The following figure shows the Test Navigator window.



### Understanding Icons

Test Navigator provides various icons that represent types and statuses of the test. The following sections explain these icons.

#### Test Type

| Icon | Description |
| --- | --- |
|  | Unit test |
|  | Integration test |
|  | Non-active test. The test was ignored because you added the [DontRun] or [Ignore] attribute. |

#### Test Status

| Icon | Description |
| --- | --- |
|  | The test successfully passed. |
|  | The test is pending because the code that the test checks changed. |
|  | The test failed. |
|  | A new test. This test was written by a developer or automatically suggested by Typemock Isolator and was not run yet. |

### Understanding Tabs

To help you easier navigate through the tests, Test Navigator displays the tests and their status on the following tabs:

* Failed
* Pending
* Ignored
* Passed

On each tab, you can group tests by the full class name or project, namespace, or class name. If tests are grouped, you can run all tests within the group in one click. In addition, you can run all tests on the selected tab.

### Summary Status of the Entire Solution

Test Navigator summarizes the testing status of the entire solution since the last build. The status is represented by the icon displayed above the tests and can be one of the following:

| Icon | Description |
| --- | --- |
|  | All tests passed successfully. |
|  | Some tests are pending after the code changed. |
|  | Some tests failed. |
|  | The solution build failed. If you click the icon, the errors will be displayed. |

## Opening the Test Navigator Pane

To open the Test Navigator pane:

* Select Typemock > Windows > Navigator.

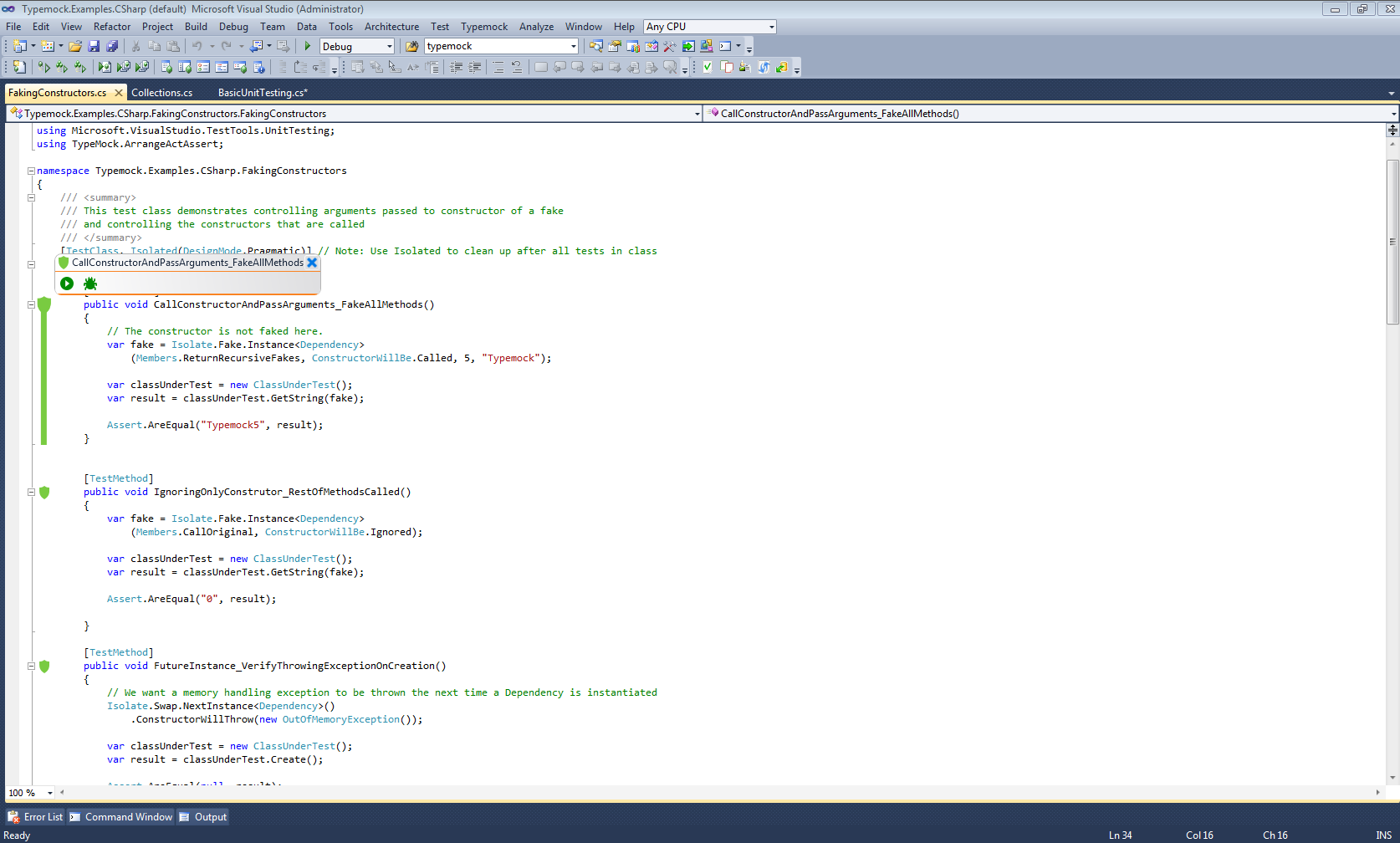
## Running an Individual Test

You can run any test at any time without having to wait for next build or when your computer is idle. You can run a test either from the code of the test or from the code of the method under test.

To run an individual test from the code of the test:

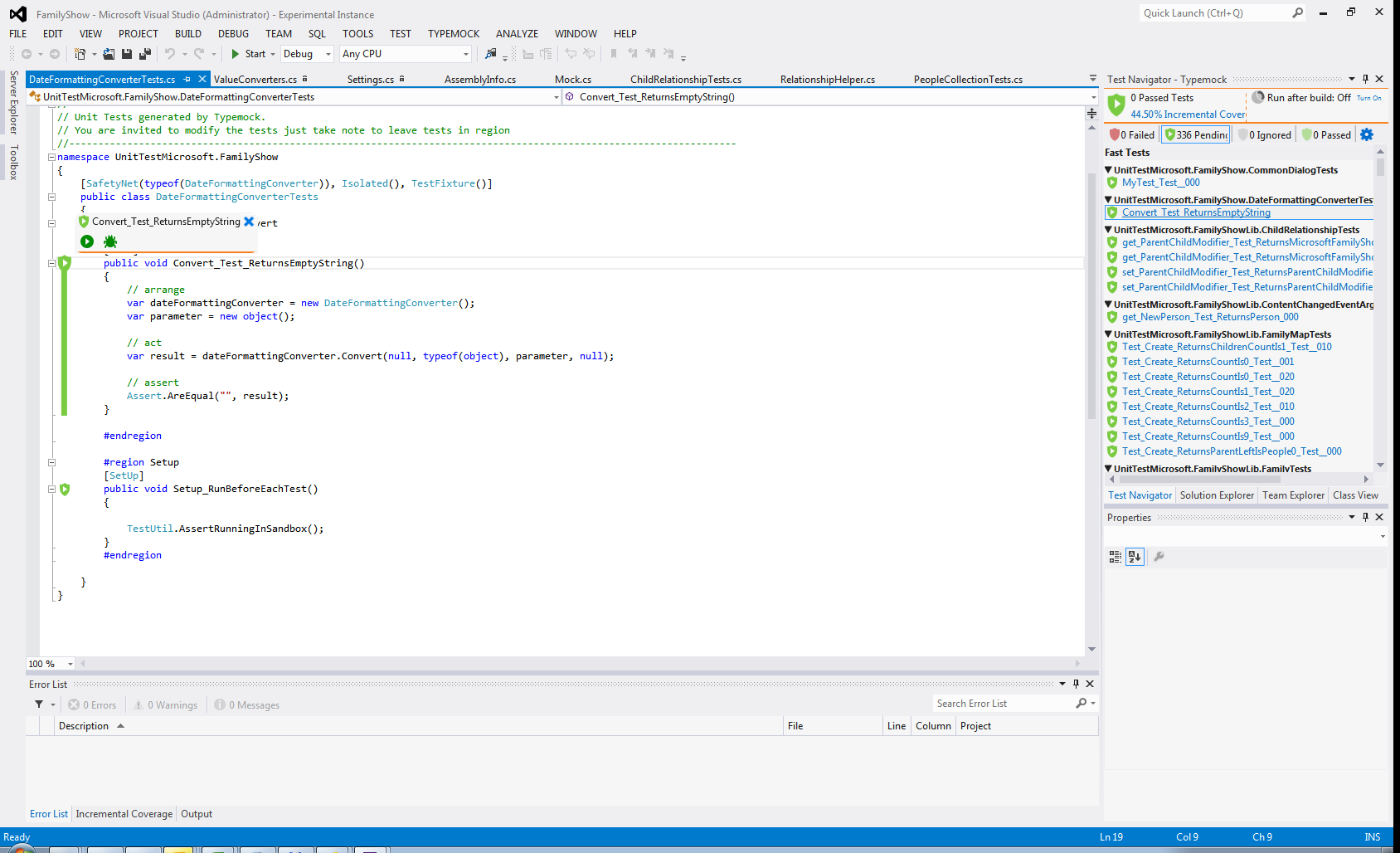
1. In the Test Navigator window, click the test that you want to run.

The code of the test is opened in the code window.



1. On the left from the method, click the shield icon .

The Test window is displayed above the shield icon.

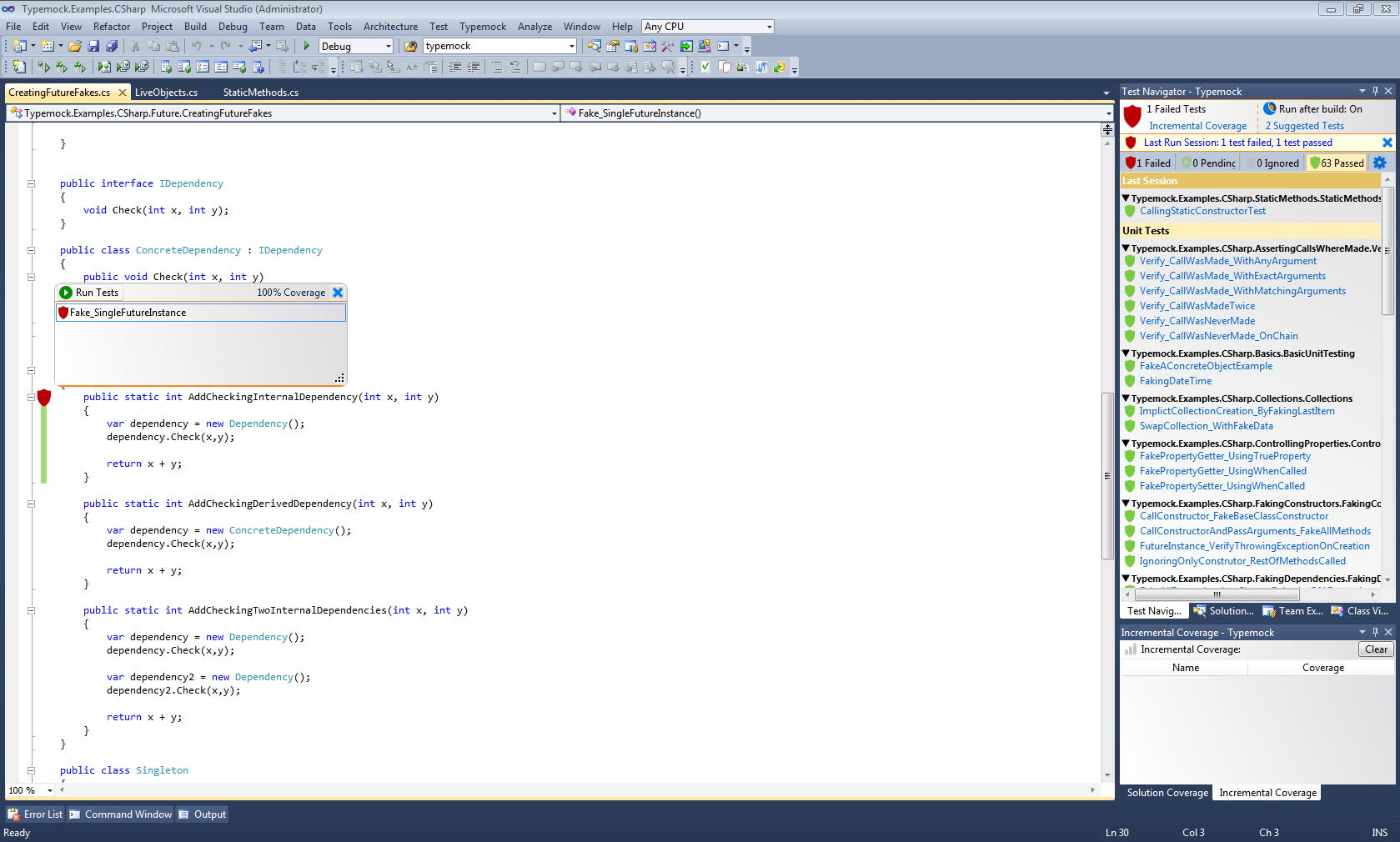


1. In the Test window, click Run .

To run an individual test from the code of the method under test:

1. Navigate to the method.
2. On the left from the method, click the shield icon .

All tests that run on the method appear above the shield. If you clicked the shield icon of a collapsed group of methods, then all tests that run on the collapsed methods are displayed.



1. Do one of the following:

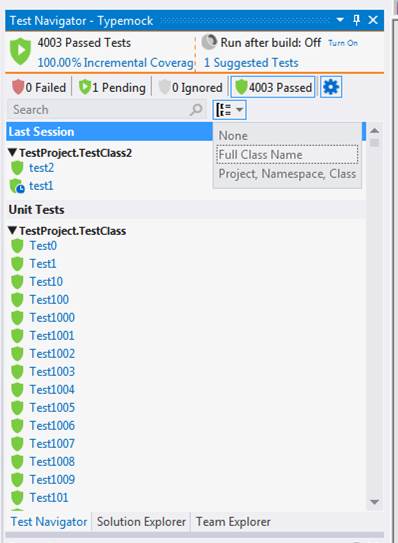
* To run all the tests, click Run Tests ****.
* To run an individual test, select the test, and in the Test window displayed on the right, click Run .

## Running All Tests in a Group

If you selected to group tests by the full class name or project, namespace, or class name, you can run all tests within the group in one click. In addition, you can run all tests on the selected tab in Test Navigator.

To run all tests in a group:

1. In Test Navigator, click the tab on which you want to run tests.
2. From the filter displayed on the right from the Search field, select how you want to group the tests on the tab.



1. Do one of the following:

* If you want to run all tests on the tab, right-click on the tab and select the required action.
* If you want to run tests within a specific group on the tab, right-click on the group and select Run Tests of Group.

## Running All Pending Tests

You can run both unit tests and integration tests at once. This feature is useful when you do not want to run each test individually or wait till next build.

To run all pending tests:

* Select Typemock > Run > All Pending Tests.

Typemock Isolator will compile and run pending unit tests and integration tests. The result will be displayed in the Test Navigator window.

## Running Unit Tests

You can perform the following actions for unit tests:

* Running all tests in a project. See [Running All Tests in a Project](#_D2HTopic_294).
* Running pending unit tests. See [Running Pending Unit Tests](#_D2HTopic_295).
* Running all unit tests. See [Running All Unit Tests](#_D2HTopic_296).
* Re-running the recently run test. See [Re-Running the Recently Run Test](#_D2HTopic_297).
* Setting pending unit tests to run automatically after build. See [Setting Pending Unit Tests to Run Automatically after Build](#_D2HTopic_298).

### Running All Tests in a Project

You can run all tests in a project at once.

To run all tests in a project:

1. In Solution Explorer, right click on the project for which you want to run tests.
2. Select Run Tests in Project.

### Running Pending Unit Tests

You can instruct Typemock Isolator to run only unit pending tests and skip integration tests. This feature is useful when you do not want to run each test individually or wait till next build.

To run unit pending tests:

* Select Typemock > Run > Unit Pending Tests.

Typemock Isolator will compile and run all pending tests since the last build. The result will be displayed in the Test Navigator window.

### Running All Unit Tests

You can run all unit tests in the solution without having to wait when your computer is idle.

To run all unit tests:

* Select Typemock > Run > All Unit Tests.

Typemock Isolator will compile and run all unit tests. The result will be displayed in the Test Navigator window.

### Re-Running the Recently Run Test

You can re-run the test that was run last time.

To re-run the test:

* Select Typemock > Last Test > Run.

### Setting Pending Unit Tests to Run Automatically after Build

In addition to the ability to run unit tests manually whenever you need it, you can set unit pending tests to run automatically every time you build the project.

To set unit pending tests to run after build:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. Select the Run unit pending tests after build check box.
3. Click OK.

Typemock Isolator will now automatically run all unit pending tests after every build.

## Setting Integration Tests to Run Automatically when Idle

In addition to the ability to run tests manually whenever you need it, you can set integration tests to run automatically when your computer is idle. Any operation done on the computer stop the runs. The Test Navigator window and corresponding shields will display the tests whose run is completed.

To set integration tests to run when in idle:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. Select the Run unit pending tests in background when idle check box.
3. Click OK.

Typemock Isolator will now automatically run all integration tests when the computer is idle.

## Canceling a Test Run

You can cancel a test run at any time. The status of those tests that are completed by the time you canceled is displayed in the Test Navigator window. The tests that were running when you canceled the run are stopped. The status of the tests that did not begin running is set to pending.

To cancel a test run:

* Select Typemock > Cancel Run.

## Running a Test in the Sandbox

To ensure that your test does not crash your data or production environment, you can run it in the sandbox. A test running in the sandbox is isolated and cannot harm your system.

Typemock Isolator runs all suggested tests in the sandbox. However, integration tests cannot run in the sandbox.

To run a test in the sandbox:

* To the test class that you want to run in the sandbox, add the [SafetyNet] attribute.

For example:

|  |
| --- |
| [SafetyNet(typeof(UnderTest),TestMethod,Isolated] |

## Ignoring Tests

You can instruct Typemock Isolator to ignore a test by adding one of the following attributes to the unit test method:

| Attribute | Description |
| --- | --- |
| [DontRun] | Typemock Isolator skips the test. However, the test framework runs the test. |
| [Ignore] | Both Typemock Isolator and the test framework skip the test. |

|  |
| --- |
| [AttributeUsage(AttributeTargets.Method)]  public class DontRunAttribute : System.Attribute  {}  [DontRun]  [TestMethod]  public void SimpleTestUsingMessageBox()  {  // Arrange  Isolate.WhenCalled(()=>MessageBox.Show(null)).WillReturn(DialogResult.OK);  // Act  MessageBox.Show("This is a message"); <br>  // Assert  Isolate.Verify.WasCalledWithExactArguments(()=>MessageBox.Show("This is a message"));  } |

## Configuring SmartRunner

You can configure how SmartRunner works by:

* Reusing the test runner process. See [Reusing the Test Runner Process](#_D2HTopic_304).
* Using a shadow folder for running NUnit tests. See [Using a Shadow Folder for Running NUnit Tests](#_D2HTopic_305).
* Enabling or disabling output when running tests. See [Enabling and Disabling Output when Running Tests](#_D2HTopic_306).
* Specifying the test architecture. See [Specifying the Test Architecture](#_D2HTopic_307).
* Disabling SmartRunner. See [Disabling SmartRunner](#_D2HTopic_304).

### Reusing the Test Runner Process

To reuse the test runner process:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. In the Advanced area, select the Reuse Test Runner process check box.
3. Click OK.

### Using a Shadow Folder for Running NUnit Tests

When a shadow folder is used, Typemock Isolator copies the build DLLs to a shadow folder and runs from this folder.

To use a shadow folder for running NUnit tests:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. In the Advanced area, select the Use Shadow Copy for Nunit Tests check box.
3. Click OK.

### Enabling and Disabling Output when Running Tests

To enable or disable output when running tests:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. In the Advanced area, do one of the following:

* If you want to enable output, select the Show Output when build starts check box.
* If you want to disable output, clear the Show Output when build starts check box.

1. Click OK.

### Specifying the Test Architecture

You can specify whether you want to force Typemock Isolator to run tests on an x86-bit or x64-bit architecture.

To specify the test architecture:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. In the Advanced area, in the Test Architecture drop-down list, select one of the following:

* If you want SmartRunner to choose a x64-bit architecture unless your tests explicitly reference an x86-bit DLL, select Auto.
* If you want to force Typemock Isolator to run tests on an x86-bit architecture, select X86.
* If you want to force Typemock Isolator to run tests on an x64-bit architecture, select X64.

1. Click OK.

### Disabling SmartRunner

By default, Typemock Isolator uses SmartRunner to run tests. If you want to use third-party runners, you can disable SmartRunner. Typemock Isolator provides an out-of-the-box integration with a wide variety of runners (for a complete list of supported test runners, see [Integration with Test Runners](#_D2HTopic_25)).

To disable SmartRunner:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. Clear the Enable SmartRunner check box.
3. In the list, select Typemock > Mocking Integration.
4. Verify that the Enable Visual Studio mocking integration check box is selected.
5. Click OK.

# Analyzing and Fixing Bugs

The following topics explain how you can use Typemock Isolator to analyze and fix bugs.

## Identifying Bugs

Using Test Navigator, Method window, and Test Preview window you can identify and resolve the issue that caused the test to fail. Depending on whether you want to approach the debugging process from the failed test or from the method under test, the debugging procedure will be different.

### Starting from the Failed Test

To identify bugs:

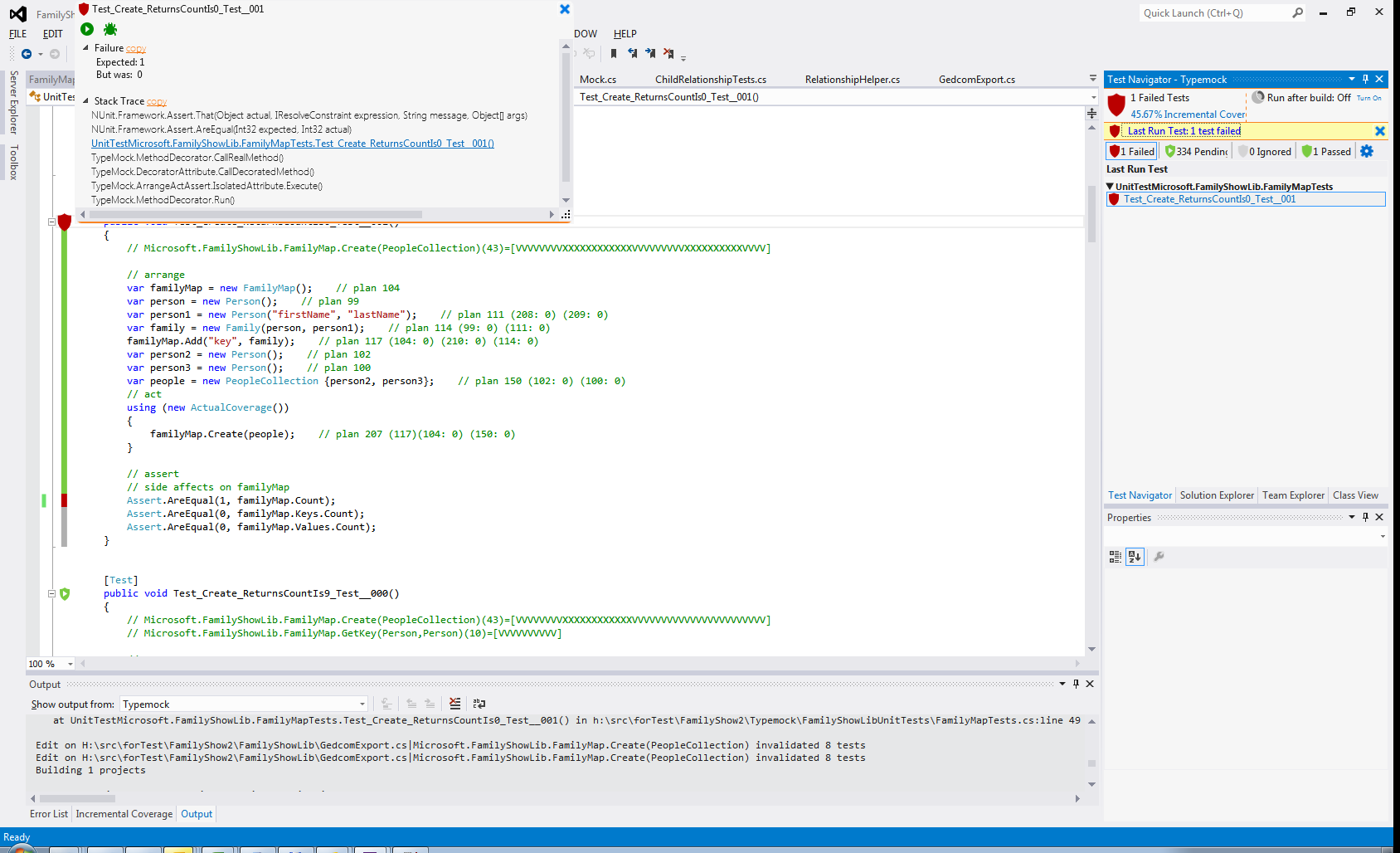
1. In Test Navigator, click the Failed tab.

The tab shows a list of failed tests. The list is grouped based on the last run of the tests.

1. Click the test that you want to further investigate.

The code of the test is opened in the code window. The code coverage indicator is red where the test fails. In addition, the Test Preview window is displayed. This window displays:

* Failure message
* Stack trace
* Output of the test
* Test code



1. In the Stack Trace area, click the method that failed.

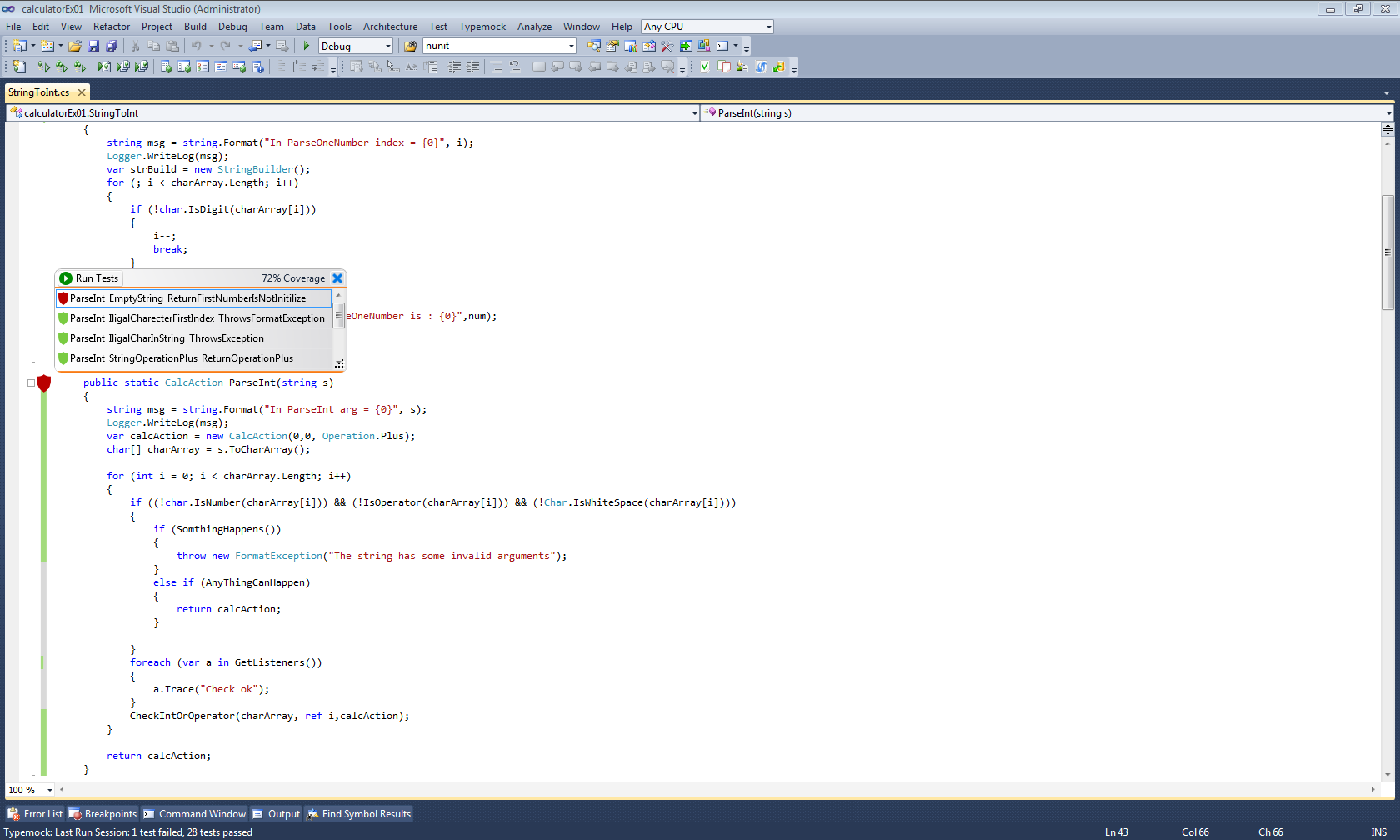
The code of the method is opened. In addition, the shield and the Test Preview window are displayed. To snap the Test Preview window so you can always see both the test and the code you need to debug, detach the Test window from the Method window by dragging the Test Preview window and dropping it somewhere on the screen.

### Starting from the Method under Test

To identify bugs:

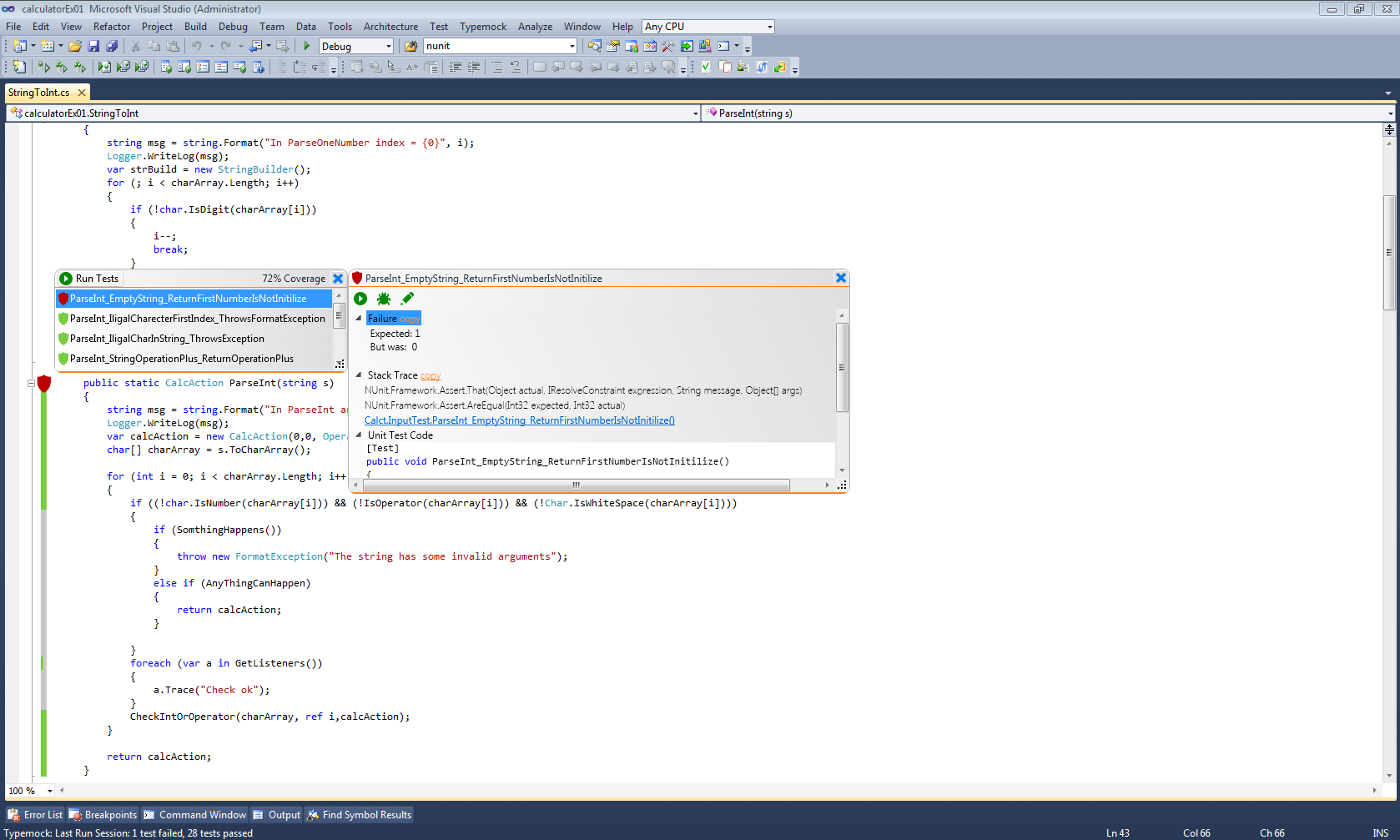
1. In the method on which the failed test runs, click the shield icon .

The Method window is displayed. The window shows a list of all tests that call the method.



1. Click the failed test.

The Test Preview window is displayed. In this window, you can view the test code, console output, and stack trace. When the Method window opens, you can also see the combined coverage of all tests. This gives you a complete view of the problem. You can see which scenarios behave incorrectly. In most cases, this is enough to resolve the problem without actual debugging.

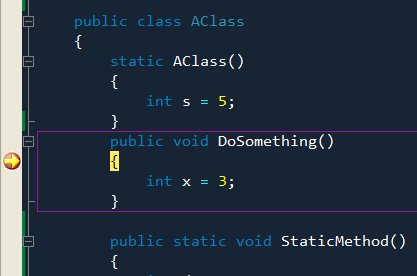


## Fake Method Highlighting

|  |  |
| --- | --- |
| ! | Note  This feature is available for C# only. |

When debugging, it is easy to get confused when you enter the mocked method. Although the code is there and includes a breakpoint, when you step into the function, it is the mocked code that actually runs rather than the actual code.

To help you identify the mocked method when you reach the method and set a breakpoint at the beginning, Typemock Isolator highlights the method as shown on the following figure:



When you step into the method, it will be highlighted only if the next call is mocked. In the following code, when the DoSomething() method is called for the first time, it will be mocked and highlighted. When DoSomething() is called next time, it is displayed as a regular code because no expectation was set.

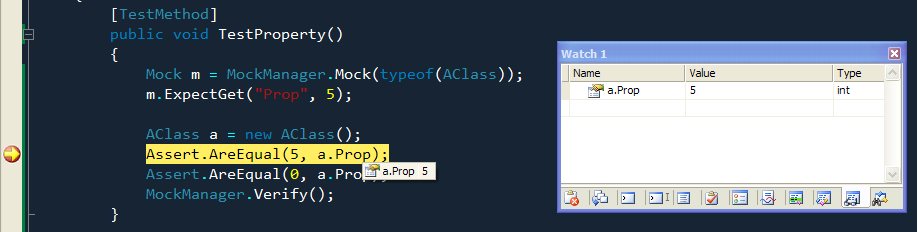
|  |
| --- |
| public void TestDoSomething() {    Mock mock = MockManager.Mock(typeof(AClass));    mock.ExpectCall("DoSomething");        AClass instanceA = new AClass();    AClass instanceB = new AClass();     // This call will be mocked    instanceA.DoSomething();    // This call will not be mocked    instanceB.DoSomething();     MockManager.Verify(); } |

## Debugger Enhancements

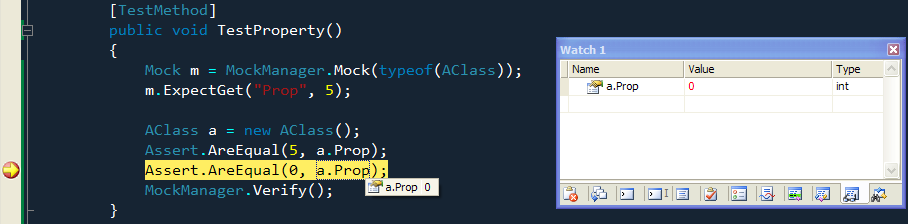
Using either the watch window or tooltip to evaluate the function return value is very helpful during debugging. Typemock Isolator enhances these tools.

If a method is faked, the watch window or tooltip for evaluation present the expected return value. If the method or instance is not faked, the watch or tooltip evaluate the actual code.

In the following sample, there is an expectation set on Prop. When you set a breakpoint before the first assertion which expects the mocked return value, both watch window and tooltip show the expected mock value 5.



The following figure shows the next assertion. In this case, the real value (in this case, it is 0) should be returned. The watch window and tooltip show this value.



## Debugging a Failed Test

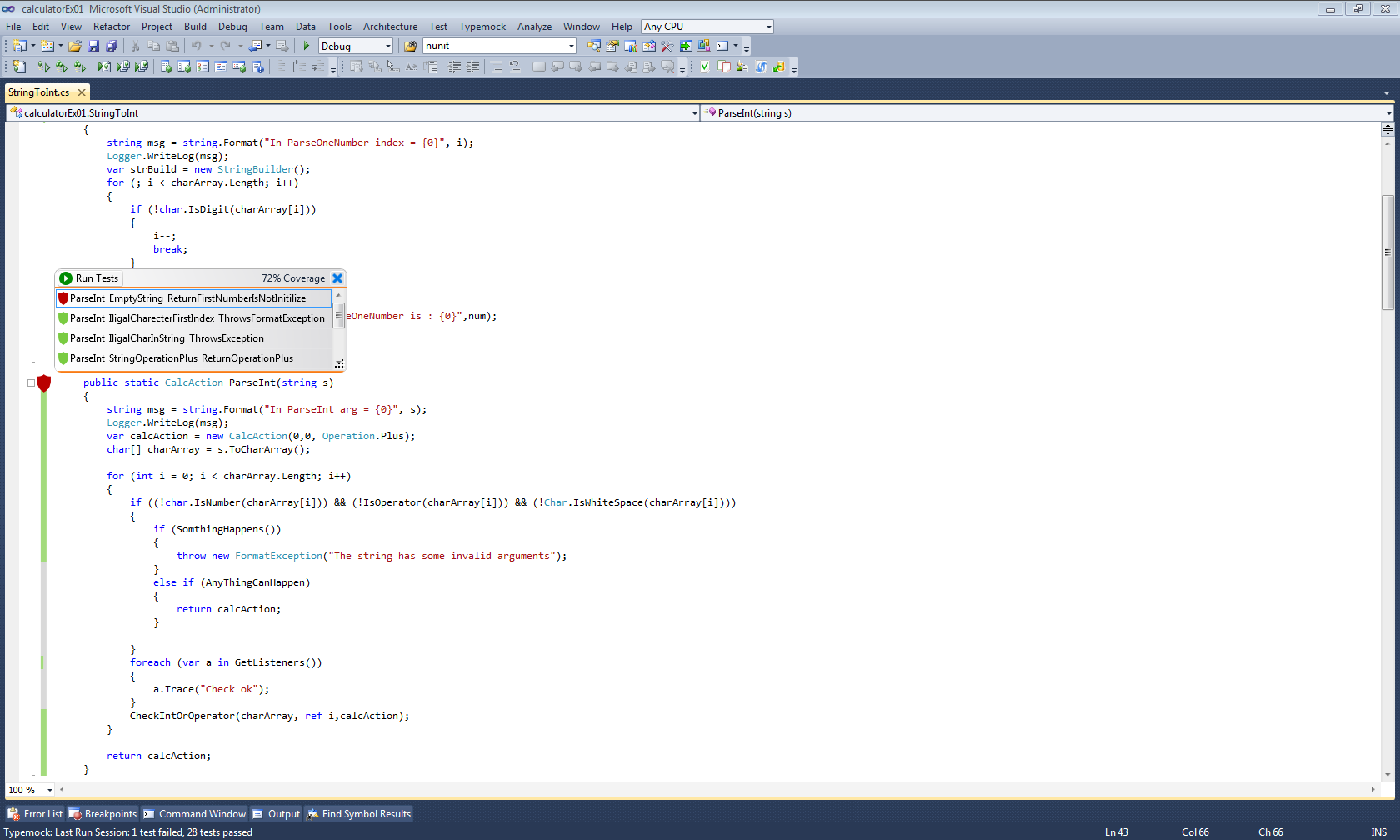
You can debug a specific failed unit test. Typemock Isolator adds a breakpoint in the beginning of the test. When running from the Method window, Typemock Isolator adds a breakpoint in the beginning of the method. You can run directly to the beginning of the method to debug it. The breakpoints are removed at the end of the debug session.

If you decide to ignore a specific failed test, and fix other test, the failed test remains in the Failed list in the Test Navigator until the test passes successfully.

To debug a test:

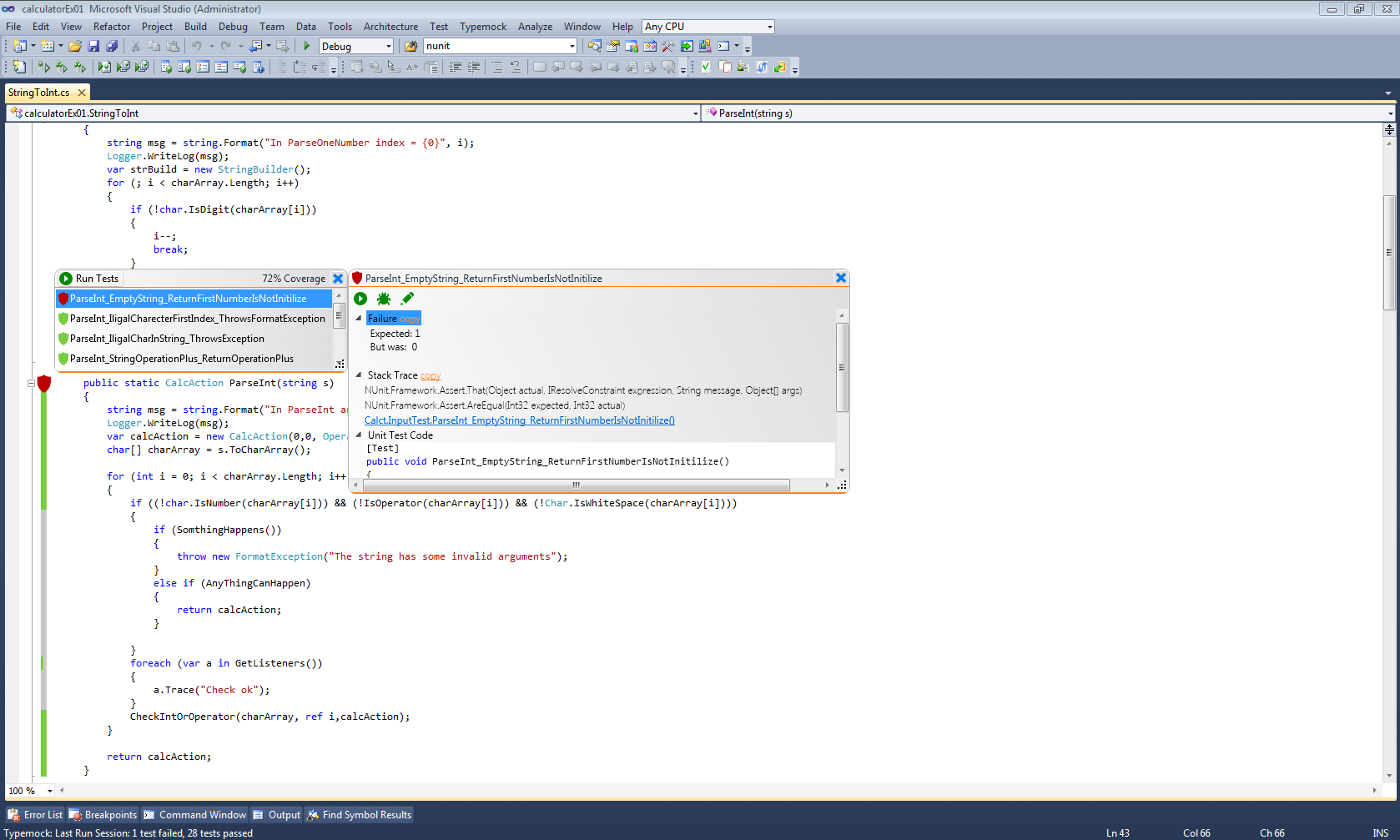
1. Navigate to the method whose call was failed.
2. On the left from the method, click the shield icon .

The Method window is displayed.



1. Select the failed test.

The Test Preview window is displayed.



1. In the Test Preview window, click Debug .

## Adding a Breakpoint Manually

You can add a breakpoint to the method under the failed test.

To add a breakpoint:

1. Within the method whose call was failed, navigate to the place where you want to add a breakpoint.
2. Right click on the method.
3. Select Debug and Break Here.

A breakpoint will be added to the current position. If there are many failed tests, the last failed test that you debugged will be run.

## Adding a Breakpoint Automatically

You can instruct Typemock Isolator to automatically add a breakpoint to the beginning of the method or test that you want to debug.

To add a breakpoint automatically:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > SmartRunner.
2. Select the Add BreakPoint when Debugging Tests check box.
3. Click OK.

## Debugging the Last Debugged or Run Test

You can debug the last debugged or last run test. The breakpoint will be added as in the last debugging session.

To debug the last debugged or run test:

* Select Typemock > Last Test > Debug.

# Viewing Code Coverage

You can view code coverage on the following levels:

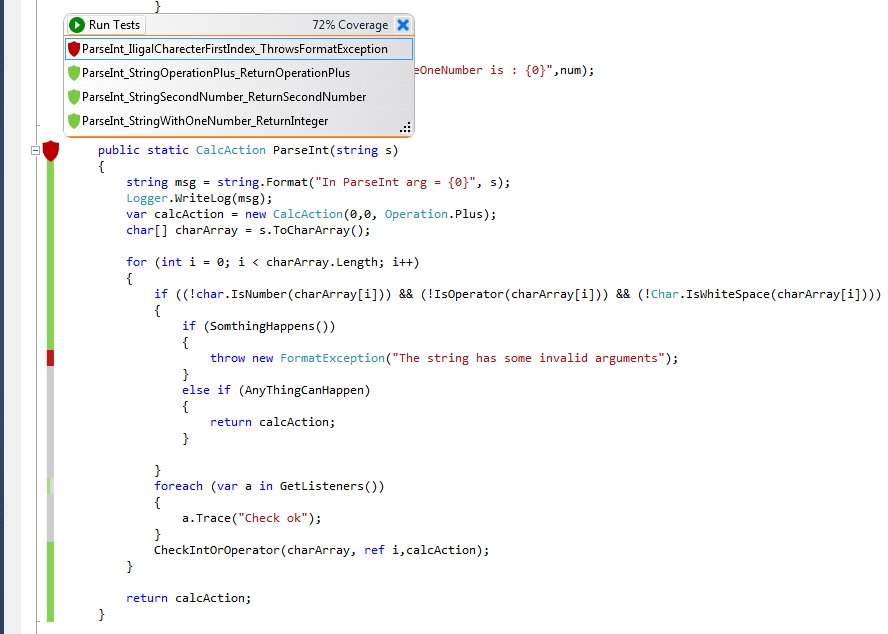
* For a specific method. See [Viewing Code Coverage for a Specific Method](#_D2HTopic_320).
* For the entire solution. See [Viewing Code Coverage for the Entire Solution](#_D2HTopic_321).
* For the recently changed classes. See [Viewing Code Coverage for Recently Changed Classes](#_D2HTopic_323).

## Viewing Code Coverage for a Specific Method

Code coverage for a method is shown by the coverage indicator that is displayed as a vertical strip on the left from the code. The coverage indicator can be one of the following:

* Green, which means the code is covered.
* Gray, which means the code is not covered.
* Red, which means the code failed.

For example, the following figure demonstrates a partially covered method whose test failed.



To view code coverage for a specific method:

1. Open the method shield and review the vertical strip on the left from the method.
2. To view the tests that cover the method, click the shield icon.

The Method window is displayed with a list of tests.

## Viewing Code Coverage for a Specific Test

A part of the method code covered by a specific test is shown by the coverage indicator that is displayed as a vertical strip on the left from the code. The coverage indicator can be one of the following:

* Green, which means the code is covered.
* Gray, which means the code is not covered.

To view code coverage of a specific test:

1. In the method, click the shield icon.

The Method window is displayed.

1. Click the test whose code coverage you want to view.

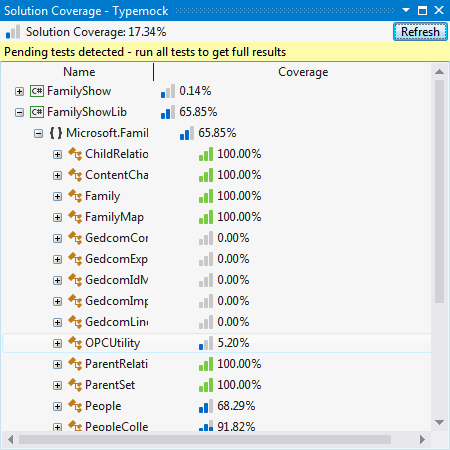
The vertical strip will show the part of the method code covered by the selected test. In addition, in the Test Preview window, the test itself is displayed.

## Viewing Code Coverage for the Entire Solution

To view code coverage for the entire solution:

1. Select Typemock > Windows > Solution Wide Code Coverage.

The Solution Coverage window is displayed. If any tests fail or are pending, a warning is displayed.



1. To refresh the code coverage, click Refresh.

## Viewing Code Coverage for Recently Changed Classes

To let you view in real time how the coverage of your code changes, Typemock Isolator provides the incremental coverage feature. The incremental coverage dynamically shows you coverage for the classes in which you made last 100 changes. Every time you make a change, the incremental coverage is automatically updated. Using the incremental code coverage, you can always see the impact of your latest modifications on the codebase.

You can always clear the list of the last 100 changes.

If you are following the test-driven development practices, your incremental coverage will always be 100%. If you are using code-driven development, the incremental coverage helps you verify that you do not forget to write a test.

To view incremental code coverage:

* Select Typemock > Windows > Incremental Code Coverage.

The Incremental Code coverage pane is displayed on the right from the code window.



To clear the list of last 100 changes:

* In the Incremental Code Coverage window, click Clear.

# Integrating Typemock Isolator

## Integrating with the Client

Typemock Isolator integrates with most of the tools, such as code coverage tools and unit testing frameworks out-of-the-box without any need for an additional configuration. However, you might need to manually select the code coverage tools with which you want to link Typemock Isolator. In addition, NCover 4.5 and above requires manual linking.

### Controlling Integration with Code Coverage Tools

By default, Typemock Isolator integrates with all code coverage tools out-of-the-box without any need for an additional configuration. However, you might need to explicitly define the specific tools to which you want to link Typemock Isolator.

To manually select the tools to which you want to link Typemock Isolator:

1. Run the Typemock Isolator configuration tool.
2. Click the Mocking Integration tab.
3. Clear the Enable auto linking check box.
4. Do one of the following:

* If you want to view all code coverage tools to which Typemock Isolator can be linked, clear the Show only available profilers check box.
* If you want to view only those code coverage tools that are installed on your computer, verify that the Show only available profilers check box is selected.

1. In the drop-down list, select the code coverage tools that you want to link to Typemock Isolator.
2. Click Link with Typemock Isolator.

### Integrating with NCover Version 4.5 and Above

If you use NCover version 4.5 and above, you need to link NCover with the following:

* Typemock Isolator. See [Linking NCover Version 4.5 and Above with Typemock Isolator](#_D2HTopic_328).
* Build server. See [Integrating NCover Version 4.5 and Above with Build Servers](#_D2HTopic_329).

#### Linking NCover Version 4.5 and Above with Typemock Isolator

To link NCover version 4.5 and above with Typemock Isolator:

1. Run the Typemock Isolator configuration tool.
2. Click the Mocking Integration tab.
3. In the drop-down list, select NCover.
4. Click Link with Typemock Isolator.

#### Integrating NCover Version 4.5 and Above with Build Servers

If you use a build server, you need to link NCover with your build server.

To link NCover with a build server:

* Do one of the following:
* If you use a custom build server, set the link argument to NCover. See [Integrating with Custom Build Servers](#_D2HTopic_331).
* If you use MS Build or NAnt, set the link parameter to NCover. See [Integrating with MSBuild](#_D2HTopic_336) or [Integrating with NAnt](#_D2HTopic_343).

## Integrating with the Server

### Integrating with Custom Build Servers

If you have a custom build server, you can use the TMockRunner tool provided as part of Typemock Isolator.

TMockRunner launches external programs, such as an external code coverage tool, a test runner, a memory or CPU profiler. For example, using TMockRunner, you can run mstest.exe and specify the name of the DLL to be tested.

To run TMockRunner:

* Run TMockRunner.exe as follows:

|  |
| --- |
| TMockRunner.exe [-version] [-thisFolder] [-register <company> <license>] [-target <version>] [-logpath <path>] [–link <profiler>] program [arguments...] |

| Parameter | Description |
| --- | --- |
| **-version** | Displays the version of **Typemock Isolator** to the console. This parameter cannot be used with other arguments. |
| **-thisFolder** | Runs with the version of Typemock binaries from the current folder. This is the normal operation on server deployment.  Use this parameter only in conjunction with the program parameter. |
| **-register** | Registers Typemock Isolator with your license. |
| **-target** | Changes the version of.NET.   Possible values:   * 2.0 * 3.0 * 3.5 * 4.0 * 4.5 |
| **-logpath** | Specifies the path where log files are stored. By default, logs are stored in the Typemock Isolator installation folder. |
| **-link <profiler>** | Integrates Typemock Isolator with a specified profiler. |
| **program** | Specifies the program to be run with mocking. |
| arguments | Specifies a list of arguments to pass to the program. |
| supportPreV8Settings | Uses legacy registry (Machine Level) |
| upgradePreV8Settings | Changes the registry to the user space. |

#### Samples

##### Sample 1: Running MSTest

The following example shows how to run MS-Test for test.dll with Typemock Isolator using the binaries from the installed version.

|  |
| --- |
| **TM**ockRunner mstest.exe test.dll |

##### Sample 2: Running Nunit Using .NET 4.0

The following example shows how to run NUnit using .NET 4.0 for test.dll with Typemock Isolator enabled from the copied AutoDeploy folder.

|  |
| --- |
| **TM**ockRunner **-thisFolder -target 4.0** nunit-console.exe test.dll |

##### Sample 3: Running Code Coverage Tool with Visual Studio 2013 and 2015

The following example shows how to run coverage with MS Visual Studio 2012 test runner for test.dll with Typemock Isolator enabled from the copied AutoDeploy folder.

|  |
| --- |
| **TM**ockRunner -thisFolder vstest.console.exe test.dll /EnableCodeCoverage /settings:CodeCoverage.runsettings |

### Integrating with MSBuild

To integrate Typemock Isolator with MSBuild, you need to add Typemock Isolator’s tasks to the MSBuild project file and define parameters of these tasks, if necessary.

Typemock Isolator provides the following tasks:

* [TypeMockRegister](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/MSBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#register): registers the Typemock Isolator license
* [TypeMockStart](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/MSBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#start): starts using Typemock Isolator
* [TypeMockStop](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/MSBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#stop): stops using Typemock Isolator

|  |  |
| --- | --- |
| ! | Note  To run nUnit with Typemock Isolator, nUnit must be run in a new executable with the <Exec> task (the <Nunit> task will not work). |

It is recommended to run the executable to support different NUnit versions than those versions that are distributed with MSBuild.

|  |  |
| --- | --- |
| ! | Note  Plug-and-Play Profiling works with MSBuild tasks too. Run the coverage or profiling tool in the contained task. |

#### Registering the Typemock Isolator License

The TypeMockRegister task registers the Typemock Isolator license.

To register the Typemock Isolator license:

* Add the TypeMockRegister task to the project file with the following optional parameters:

| Parameter | Description | Type | Mandatory/Optional |
| --- | --- | --- | --- |
| **Company** | The company name as defined in your license | String | Mandatory |
| **License** | The activation key as defined in your license | String | Mandatory |
| **AutoDeploy** | Specifies whether to run Typemock Isolator tests without installation on the server | **Boolean** | **Optional** |

##### Samples

|  |
| --- |
| <Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  <PropertyGroup>   <TypeMockLocation>C:\Sources\TypeMock</TypeMockLocation>  </PropertyGroup>   <Import Project ="$(TypeMockLocation)\**TypeMock.MSBuild.Tasks**"/>  <Target Name="RegisterTypeMock">  **<TypeMockRegister Company ="TypeMock" License="YourLicense" AutoDeploy**=**”True”/>**  </Target>  </Project> |

#### Starting Typemock Isolator

The TypeMockStart task runs Typemock Isolator on the server.

To run TypeMockStart:

* Add the TypeMockStart task to the project file with the following optional parameters:

| Parameter | Description | Type | Mandatory/Optional |
| --- | --- | --- | --- |
| **Target** | .NET version.  Possible values:   * 2.0 * 3.0 * 3.5 * 3.0 * 4.0 * 4.5 | String | Optional |
| **LogPath** | Path to log files (by default, it is the current user temporary directory) | String | Optional |
| **Link** | Integrates Typemock Isolator with a specified profiler | String | Optional |
| **DisableAutoLink** | Disables automatic integration with profilers | Boolean | Optional |

##### Samples

###### Sample 1: Running NUnit Tests with Typemock Isolator

Because the best practice recommends to copy the Typemock Isolator DLLs to the source control repository, the following example assumes that the DLLs are stored there (see the TypemockLocation parameter). Your build can retrieve the files from the source control repository.

The original location of the DLLs is C:\Program Files (x86)\TypeMock\Isolator\<version>\TypeMock.MSBuild.dll.

|  |
| --- |
| <Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  <PropertyGroup>  <TypeMockLocation>C:\Sources\TypeMock\ </TypeMockLocation>  <NUnit>"C:\Program Files\NUnit\bin\nunit-console.exe"</NUnit>  </PropertyGroup>    <Import Project ="$(TypeMockLocation)\**TypeMock.MSBuild.Tasks**"/>  <Target Name="TestWithTypeMock">  <**TypeMockStart**/>  <Exec ContinueOnError="true" Command="$(NUnit) Test.dll"/>  <**TypeMockStop**/>  </Target>  </Project> |

###### Sample 2: Running NUnit Tests with Typemock Isolator and NCoverage in .NET 4.0

|  |
| --- |
| <Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">  <PropertyGroup>TypeMockLocation  <TypeMockLocation>C:\ Sources\TypeMock </TypeMockLocation>  <NUnit>"C:\Program Files\NUnit\bin\nunit-console.exe"</NUnit>  <NCover>"C:\Program Files\NCover\NCover.Console"</NCover>  </PropertyGroup>   <Import Project ="$(TypeMockLocation)\**TypeMock.MSBuild.Tasks**"/>  <Target Name="TestWithTypeMock">  <**TypeMockStart** **Target**="4.0"/>  <Exec ContinueOnError="true" Command=" $(NCover) //a Tested $(NUnit) Test.dll"/>  <**TypeMockStop**/>  </Target>  </Project> |

#### Stopping Typemock Isolator

The TypeMockStop task stops Typemock Isolator on the server.

You must call TypeMockStop after TypeMockStart even if the test failed and set the ContinueOnError parameter to "true".

To run TypeMockStop:

* Add the TypeMockStop task to the project file after TypeMockStart**.**

##### Samples

The following example shows you how to stop Typemock Isolator.

|  |
| --- |
| <Target Name="TestWithTypeMock">  <TypeMockStart/>  <Exec ContinueOnError="true" Command="$(NUnit) Test.dll"/>  <TypeMockStop/>  </Target> |

### Integrating with NAnt

To integrate Typemock Isolator with NAnt, you need to add Typemock Isolator’s tasks to the NAnt project file and define parameters of these tasks, if necessary.

Typemock Isolator provides the following tasks:

* [typemockregister](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/NAntBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#register): registers the Typemock Isolator license
* [typemockstart](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/NAntBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#start): starts using Typemock Isolator
* [typemockstop](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/NAntBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#stop): stops using Typemock Isolator

|  |  |
| --- | --- |
| ! | Note  To run nUnit with Typemock Isolator, nUnit must be run in a new executable with the <Exec> task (the <Nunit2> task will not work). |

It is recommended to run the executable to support different NUnit versions than those distributed with NAnt.

|  |  |
| --- | --- |
| ! | Note  Plug-and-Play Profiling works with NAnt tasks too. Run the tool as part of the contained task parameters. |

#### Registering Typemock Isolator License

The typemockregister task registers the Typemock Isolator license.

To register the Typemock Isolator license:

* Add the typemockregister task to the project file with the following optional parameters:

| Parameter | Description | Type | Mandatory/Optional |
| --- | --- | --- | --- |
| **Company** | The Company Name received in your license | String | Mandatory |
| **License** | The activation key received in your license | String | Mandatory |
| **AutoDeploy** | Specifies whether to run Typemock Isolator tests without an installation on the server | **Boolean** | **Optional** |

##### Samples

The following example shows how to register a Typemock Isolator license.

|  |
| --- |
| <project name="Typemock Isolator Examples" default="test" basedir=".">  <property name="typemock.dir" value="C:\Program Files\TypeMock\TypeMock" />    <target name="register">  <!-- Dynamically load TypeMock task. -->  <loadtasks assembly="${typemock.dir}\**TypeMock.NAntBuild.dll**" />  <**typemockregister company ="TypeMock" license="TypeMockLicense" AutoDeploy**=**”True”**/>  </target>  </project> |

#### Starting Typemock Isolator

The typemockstart task runs Typemock Isolator on the server.

To run typemockstart:

* Add the typemockstart task to the project file with the following optional parameters:

| Parameter | Description | Type | Mandatory/Optional |
| --- | --- | --- | --- |
| **version** | Typemock Isolator version | String | Optional |
| **target** | .NET version.  Possible values:   * 2.0 * 3.0 * 3.5 * 3.0 * 4.0 * 4.5 | String | Optional |
| **logPath** | Path to log files (by default, it is the Typemock Isolator installation directory) | String | Optional |
| **Link** | Integrates Typemock Isolator with a specified profiler | String | Optional |
| **DisableAutoLink** | Disables automatic integration with profilers | Boolean | Optional |

##### Samples

###### Sample 1: Running NUnit Tests and Typemock Isolator

Because the best practice recommends to copy the Typemock Isolator DLLs to the source control repository, the following example assumes that the DLLs are stored there (see the TypemockLocation parameter). Your build can retrieve the files from the source control repository.

The original location of the DLLs is C:\Program Files (x86)\TypeMock\Isolator\<version>\TypeMock.NAnt.dll.

|  |
| --- |
| <project name="Typemock Isolator Examples" default="test" basedir=".">  <property name="nunit" value="C:\Program Files\NUnit\bin\nunit-console.exe" />  <property name="typemock.dir" value="C:\Sources\TypeMock" />    <!--Example: running NUnit tests with Typemock Isolator-->  <target name="test">  <!-- Dynamically load Typemock NAnat tasks -->  <loadtasks assembly="${typemock.dir}\**TypeMock.NAntBuild.dll**" />  <!-- Start Typemock Isolator -->   **<typemockstart/>**  <!-- Stop Typemock Isolator -->   **<typemockstop/>**    </target> </project> |

###### Sample 2: Running NUnit Tests with Typemock Isolator and NCover 2.x Coverage in .NET 4.0

This is done using the <ncover> NAnt task provided with NCoverExplorer, which comes with the NCover 4.x installation.

|  |  |
| --- | --- |
| ! | Note  Pre .NCover 4. Typemock Isolator requires registering its own profiler when linking with the NCover profiler. If you do not set **registerProfiler="false"**, NCover will try to register its profiler when run and unregister Typemock Isolator, causing unexpected test failure. |

|  |
| --- |
| <project name="Typemock Isolator Examples" default="test" basedir=".">  <property name="nunit" value="C:\Program Files\NUnit\bin\nunit-console.exe" />  <property name="ncover.dir" value="C:\Program Files\NCover\NCover.Console.exe" />  <property name="typemock.dir" value="C:\Sources\TypeMock" />    <!-- Example: running NUnit tests with Typemock Isolator and NCover 2.x coverage -->  <target name="test" description="Execute NUnit tests with Typemock Isolator, and generate coverage information with NCover 2.x">  <!-- Dynamically load TypeMock Isolator NAnt tasks -->  <loadtasks assembly="${typemock.dir}\TypeMock.NAntBuild.dll" />    <!-- Load NCover NAnt tasks -->  <loadtasks assembly="${ncover.dir}\Build Task Plugins\NCoverExplorer.NAntTasks.dll" />    <!-- Start TypeMock Isolator; -->  **<typemockstart target="2.0"/>    <!-- Run NCover with NUnit; Do NOT register the NCover profiler - this conflicts with the Typemock/NCover link -->  <ncover program="${ncover.dir}\NCover.Console.exe"  commandLineExe="${nunit}"  commandLineArgs="Tests.dll"  registerProfiler="false"  />**  <!-- Stop Typemock Isolator -->   **<typemockstop/>**  </target> </project> |

You can use the same syntax for usage with NCover 1.5.8 by downloading the [NCoverExplorer Extras package](http://www.kiwidude.com/dotnet/DownloadPage.html).

###### Sample 3: Running Ncover 4 Tests with Typemock Isolator and NCover 1.5.8 Coverage

The following example shows how to run NUnit tests with Typemock Isolator and NCover 1.5.8 coverage by using the generic NAnt <exec> task.

|  |
| --- |
| <project name="Typemock Isolator Examples" default="test" basedir=".">  <property name="nunit" value="C:\Program Files\NUnit\bin\nunit-console.exe" />  <property name="ncover.dir" value="C:\Program Files\NCover\NCover.Console.exe" />  <property name="typemock.dir" value="C:\Sources\TypeMock" />    <!-- Example: running NUnit tests with Typemock Isolator and NCover 1.5.8 coverage -->  <target name="test" description="Executes NUnit tests with Typemock Isolator, and generates coverage information using NCover 1.5.8" >  <!-- Dynamically load TypeMock Isolator NAnt tasks -->  <loadtasks assembly="${typemock.dir}\TypeMock.NAntBuild.dll" />    <!-- Start TypeMock Isolator; -->  **<typemockstart target="2.0" />    <!-- Run NCover with NUnit using the NAnt exec task -->  <exec failonerror="false" program="${ncover.dir}\NCover.Console.exe">  <arg value="${nunit}" />  <arg value="Tests.dll" />  </exec>**  <!-- Stop Typemock Isolator -->   **<typemockstop/>**  </target> </project> |

#### Stopping Typemock Isolator

The typemockstop task stops Typemock Isolator on the server. You must call typemockstop after typemockstart even if the test failed.

To run typemockstop:

* Add the typemockstop task to the project file after typemockstart**.**

##### Samples

The following example shows how to stop Typemock Isolator.

|  |
| --- |
| <typemockstop/> |

### Integrating with CruiseControl.NET

To integrate Typemock Isolator with CruiseControl .NET, you need to configure CruiseControl .NET to run with TMockRunner.

Because the best practice recommends to copy the Typemock Isolator DLLs to the source control repository, the following example assumes that the DLLs are stored there (see the executable parameter). Your build can retrieve the files from the source control repository.

The original location of the DLLs is C:\Program Files (x86)\TypeMock\Isolator\<version>\TMockRunner.dll.

To configure CruiseControl .NET:

* In ccnet.config stored in the CruiseControl.NET root folder, add the following code to run the tests:

|  |
| --- |
| <exec>  <executable>C:\Sources\TypeMock\TMockRunner.exe</executable>  <baseDirectory><your directoty></baseDirectory>  <buildArgs>"C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\MSTest.exe" /testcontainer:Tests.dll</buildArgs>  </exec> |

### Integrating with Jenkins CI

You can integrate Typemock Isolator with Jenkins CI using MSBuild tasks or a Windows batch command.

To integrate Typemock Isolator with Jenkins CI using MSBuild tasks:

1. In Jenkins CI, under Build, Add Build Step.
2. Select Choose Build a Visual Studio project or solution using MSBuild.
3. In the MSBuild Build File field, specify the MSBuild build file that contains Typemock Isolator tasks.
4. Click Save.

To integrate Typemock Isolator with Jenkins CI using Windows batch command:

1. In Jenkins CI, under Build, Add Build Step.
2. Select Execute windows Batch command.
3. In the Command field, add the following command to run TMOCKRUNNER:

|  |
| --- |
| <location>TMockRunner <location>mstest.exe /testcontainer:"<location>test.dll" |

1. Click Save.

### Integrating with TeamCity

You can integrate Typemock Isolator with TeamCity using MSBuild tasks or a Windows batch command.

To integrate Typemock Isolator using MSBuild tasks:

1. In TeamCity, click Add New Build Step.
2. In the Runner type drop-down list, select MSBuild.
3. Specify the MSBuild file that contains Typemock Isolator tasks.
4. Fill in the rest of the fields as required.
5. Click Save.

To integrate Typemock Isolator using a Windows batch command:

1. In TeamCity, click Add New Build Step.
2. In the Runner type drop-down list, select Command Line.
3. In the Custom script field, add the following command to run TMOCKRUNNER:

|  |
| --- |
| **<location>TM**ockRunner <location>mstest.exe /testcontainer:"<location>test.dll" |

1. Fill in the rest of the fields as required.
2. Click Save.

### Integrating with MS Team Foundation Server

Typemock Isolator integrates with the following versions of MS Foundation Server:

* MS Team Foundation Server 2010. See [Integrating with MS Team Foundation Server 2010](#_D2HTopic_354).
* MS Team Foundation Server 2012. See [Integrating with MS Team Foundation Server 2012](#_D2HTopic_357).
* MS Team Foundation Server 2013. See [Integrating with MS Team Foundation Server 2013](#_D2HTopic_360).

#### Integrating with MS Team Foundation Server 2010

The following steps show how to use the custom MS Team Foundation Server activity and the template provided with Typemock Isolator for use of different scenarios.

The initial setup described in the following sections is forward compatible. With future versions, only the AutoDeploy folder should be updated in the source control. Deploying the activity and the template is not required with newer versions of Typemock Isolator.

The following sections explain the recommended way of working with Typemock Isolator in MS Team Foundation Server. It requires checking the AutoDeploy folder to source control. It allows for side-by-side use of different versions of Typemock Isolator in the same server.

##### Using the Auto-Deploy Mode

The Auto-Deploy mode enables you to run tests with Isolator in environments where Typemock Isolator is not installed. The Auto-Deploy mode creates a temporary local framework on the agent which enables it to run the tests. Afterwards it restores the environment.

It is recommended to use the Auto-Deploy mode because:

* There is no need to update agents when you upgrade Isolator's version.
* The agents can work side-by-side on a multiple solutions using different versions of Typemock Isolator.

However, if for some reason you decide not to use the Auto-Deploy mode, the DLLs form the AAA folder should be placed in your source control repository.

##### Integrating with MS Team Foundation Server 2010

To integrate with MS Team Foundation Server 2010:

1. From the Typemock Isolator installation folder, copy the AutoDeploy folder to the projects root directory in your source control tree.
2. In MS Team Foundation Server, click the Excluded tab.
3. Select all the items.
4. Click Include.

|  |  |
| --- | --- |
| ! | Note  MS Team Foundation Server might prompt you to map your workspace. Map your server folder to your local project directory. |

1. Check-in the AutoDeploy folder into the MS Team Foundation Server.
2. In Source Control Explorer, create a new folder in the repository under your team's project, and name it CustomActivities.
3. Copy TypemockTFS2010.dll from the AutoDeploy folder to the CustomActivities folder.
4. Copy DefaultTemplateWithTypemock.xaml from AutoDeploy folder to the BuildProcessTemplates folder.
5. Check-in the changes.
6. In Team Explorer, right-click on Builds.
7. Select Manage Build Controllers.

The Manage Build Controllers window is displayed.

1. Click Properties.

The Manage Build Controllers Properties window is displayed.

1. On the right from the Version control path to custom assemblies field, click ....

The Browse window is displayed.

1. Navigate to the CustomActivities folder in your repository.
2. Click OK.
3. Restart MS Visual Studio.
4. In Team Explorer, right click on your build definition.
5. Select Edit Build Definition.

The Build Definition window is displayed.

1. Select Process.
2. In the Build Process Template area, click Show.

The New Build Process Template window is displayed.

1. Select the Select an existing XAML file option.
2. Click Browse.

The Browse window id displayed.

1. Select DefaultTemplateWithTypemock.xaml.
2. Click OK.
3. In the Build Definition window, select Workspace.
4. Copy the path displayed in the Source Control Folder column.
5. Click Process.
6. Scroll down to the bottom of the Build process parameters list to the Typemock properties
7. In the Build process parameters list, paste the source control folder path next to AutoDeploy server directory and add the AutoDeploy directory.
8. Fill in the license details.

#### Integrating with MS Team Foundation Server 2012

The following steps represent the recommended way of working with Typemock Isolator in MS Team Foundation Server. It requires checking the AutoDeploy folder to your source control repository and referencing the Typemock Isolator DLLs from there. It allows for side-by-side use of different versions of Isolator in the same server.

Typemock Isolator requires that the service running the build will have admin privileges on the build machines and be defined in the Project Collection Build Service Accounts. Set the service to run under Build Service Properties, in the Run the Service As window.

Plug-and-Play Profiling works with MS Team Foundation Server activities too. You need to enable code coverage in MS Team Foundation Server.

When installing Typemock Isolator, the AutoDeploy folder is copied to the installation folder. The AutoDeploy folder contains all the files required for running tests without installing Typemock Isolator on the build machine.

The integration with MS Team Foundation Server involves the following:

1. Setting up the integration in MS Team Foundation Server.
2. Specifying the default Typemock Isolator build template or customizing the existing template.

##### Setting Up the Integration with MS Team Foundation Server

To integrate with MS Team Foundation Server 2012:

1. In MS Visual Studio, in Source Control Explorer, browse to the BuildProcessTemplates folder.
2. From the AutoDeploy folder in the Typemock installation folder, copy DefaultTemplate.11.1.WithTypemock.xaml into the BuildProcessTemplates folder.
3. In the BuildProcessTemplates folder, create a new folder called TypemockActivities.
4. From the AutoDeploy folder, copy the following files to the TypemockActivitiesfolder:

* Configuration.dll
* TypeMock.CLI.Common.dll
* TypeMock.Integration.dll
* Typemock.Interceptors.dll
* TypeMock.TFS2012.dll

|  |  |
| --- | --- |
| ! | Note  MS Team Foundation Server might prompt you to map your workspace. Map your server folder to your local project directory. |

1. Check-in the changes.
2. In Team Explorer, click Builds.
3. Under Actions, select Manage Build Controllers.

The Manage Build Controllers window is displayed.

1. Click Properties.

The Build Controller Properties window is displayed.

1. Next to the Version control path to custom assemblies field, click ....

The Browse window is displayed.

1. Navigate to the TypemockActivities folder in your repository.
2. Click OK.
3. From the Version control path to custom assemblies field, copy the path.
4. In Team Explorer, right click your build definition.
5. Select Edit Build Definition.

The Build Definition window is displayed.

1. Select Process.
2. In the Build Process Template area, click Show Details.
3. In the Build Process File field, click New.

The New Build Process Template window is displayed.

1. Select the Select an existing XAML file option.
2. Click Browse.

The Browse window is displayed.

1. Select DefaultTemplate.11.1.WithTypemock.xaml.
2. Click OK.

##### Selecting the Default Typemock Isolator Template

|  |  |
| --- | --- |
| ! | Note  Follow this procedure only if you want to use the default Typemock Isolator template. If you want to customize an existing template, follow the steps described in [Selecting the Default Typemock Isolator Template](#_D2HTopic_359). |

Typemock Isolator integrates with the build process using one of the following ways:

* Per team project collection, or
* Per team project or branch

###### Integrating Typemock Isolator per Team Project Collection

This strategy integrates Typemock Isolator for all builds in a team project collection. Use this strategy when all projects in that collection will use the same version of Typemock Isolator.

With this strategy, all of the Typemock Isolator files are loaded and deployed on the build controller level.

To integrate Typemock Isolator per team project collection:

1. In Team Explorer, right click on your build definition.
2. Select Edit Build Definition.

The Edit Build definition window is displayed.

1. Select Process.
2. In the Build process template area, select DefaultTemplate.11.1.WithTypemock.xaml.
3. Scroll down to the bottom of the Build process parameters list to the Typemock properties.
4. Verify that the Binaries Directory field does not contain any value.
5. Provide the license details.
6. From the AutoDeploy folder, copy all the files to the TypemockActivities folder.

###### Integrating Typemock Isolator per Team Project or Branch

This strategy integrates Typemock with a MS Team Foundation Server team project or a branch. Use this strategy when more than one Typemock Isolator version exists in the collection.

With this strategy, only the base activity files are loaded on the build controller level, and the version-specific Typemock Isolator files loaded from Binaries Directory during the build.

To integrate Typemock Isolator per team project or branch:

1. In Team Explorer, right click on your build definition.
2. Select Edit Build Definition.

The Edit Build definition window is displayed.

1. Select Process.
2. In the Build process template area, select DefaultTemplate.11.1.WithTypemock.xaml.
3. In Source Control Explorer, navigate to the location of the build definition.
4. From the Source location field, copy the location.
5. In the Edit Build Definition window, in Process, scroll down to the bottom of the Build process parameters list to the Typemock properties.
6. Verify that the Binaries Directory field does not contain any value.
7. Paste the location of the build.
8. Provide in the license details.

#### Integrating with MS Team Foundation Server 2013

The following steps represent the recommended way of working with Typemock Isolator in MS Team Foundation Server. It requires checking the AutoDeploy folder to your source control repository and referencing the Typemock Isolator DLLs from there. It allows for side-by-side use of different versions of Isolator in the same server.

Typemock Isolator requires that the service running the build will have admin privileges on the build machines and be defined in the Project Collection Build Service Accounts. Set the service to run under Build Service Properties, in the Run the Service As window.

Plug-and-Play Profiling works with MS Team Foundation Server activities too. You need to enable code coverage in MS Team Foundation Server.

When installing Typemock Isolator, the AutoDeploy folder is copied to the installation folder. The AutoDeploy folder contains all the files required for running tests without installing Typemock Isolator on the build machine.

The integration with MS Team Foundation Server involves the following:

1. Setting up the integration in MS Team Foundation Server.
2. Specifying the Default Typemock Isolator build template.

##### Setting Up the Integration with MS Team Foundation Server

To integrate with MS Team Foundation Server 2013:

1. In Source Control Explorer, under your project, create a new folder named TypemockActivities**.**
2. From the AutoDeploy folder stored in the Typemock Isolator installation folder, copy DefaultTemplate.12.0.WithTypemock.xaml to the Typemock Activities folder.
3. From the AutoDeploy folder, copy the following files to the TypemockActivitiesfolder:

* Configuration.dll
* TypeMock.CLI.Common.dll
* TypeMock.Integration.dll
* Typemock.Interceptors.dll
* TypeMock.TFS2013.dll

|  |  |
| --- | --- |
| ! | Note  MS Team Foundation Server might prompt you to map your workspace. Map your server folder to your local project directory. |

1. Check-in the changes.
2. In Team Explorer, click Builds.
3. Under Actions, select Manage Build Controllers.

The Manage Build Controllers window is displayed.

1. Click Properties.
2. Next to the Version control path to custom assemblies field, click ....

The Browse window is displayed.

1. Browse to the TypemockActivities folder in your repository.
2. Click OK.
3. In Team Explorer, right click on your build definition.
4. Select Edit Build Definition.

The Edit Build definition window is displayed.

1. Select Process.
2. In the Build process template area, click Show Details.
3. In the Build process file field, click New.

The Browse window is displayed with the team projects shown as default.

1. In the Version control path field, click Browse.

The Browse window is displayed.

1. Navigate to the TypmockActivities folder.
2. Select DefaultTemplate.12.0.WithTypemock.xaml
3. Click OK.

##### Selecting the Default Typemock Isolator Template

|  |  |
| --- | --- |
| ! | Note  Follow this procedure only if you want to use the default Typemock Isolator template. If you want to customize an existing template, follow the steps described in [Selecting the Default Typemock Isolator Template](#_D2HTopic_362).. |

Typemock Isolator integrates with the build process using one of the following ways:

* Per team project collection
* Per team project or branch

###### Integrating Typemock Isolator per Team Project Collection

This strategy integrates Typemock Isolator for all builds in a team project collection. Use this strategy when all projects in the collection will use the same version of Typemock Isolator.

With this strategy, all of the Typemock Isolator files are loaded and deployed on the build controller level.

To integrate Typemock Isolator per team project collection:

1. In Team Explorer, right click on your build definition.
2. Select Edit Build Definition.

The Edit Build definition window is displayed.

1. Select Process.
2. In the Build process template area, select DefaultTemplate.12.1.WithTypemock.xaml.
3. Scroll down to the bottom of the Build process parameters list to the Typemock properties.
4. Verify that the Binaries Directory field does not contain any value.
5. Paste the location of the build.
6. Provide the license details.
7. From the AutoDeploy folder, copy all the files to the TypemockActivities folder.

###### Integrating Typemock Isolator per Team Project or Branch

This strategy integrates Typemock with a MS Team Foundation team project or a branch. Choose this strategy when more than one Typemock version exists in the collection.

With this strategy, only the base activity files are loaded on the build controller level, and the version-specific Isolator files loaded from Binaries Directory during the build.

To integrate Typemock Isolator per team project or branch:

1. In Team Explorer, right click on your build definition.
2. Select Edit Build Definition.

The Edit Build definition window is displayed.

1. Select Process.
2. In the Build process template area, select DefaultTemplate.12.1.WithTypemock.xaml.
3. In Source Control Explorer, navigate to the location of the build.
4. From the Source location field, copy the location.
5. Scroll down to the bottom of the Build process parameters list to the Typemock properties.
6. Verify that the Binaries Directory field does not contain any value.
7. Paste the location of the build.
8. Provide in the license details.

#### Customizing an Existing Build Template

If you do not want to use the build template that comes with Typemock Isolator, you can modify the existing template to support running tests.

To customize an existing build template:

1. Open your custom template in a text editor.
2. Add the following namespace to the <Activity> element as follows:

For MS Team Foundation Server 2012:

|  |
| --- |
| xmlns:tm="clr-namespace:TypeMock.TFS2012;assembly=TypeMock.TFS2012" |

For MS Team Foundation Server 2013:

|  |
| --- |
| xmlns:tm="clr-namespace:TypeMock.TFS2013;assembly=TypeMock.TFS2013" |

1. Add the following properties to the <x:Members> element:

|  |
| --- |
| <x:Property Name="Company" Type="InArgument(x:String)" />  <x:Property Name="License" Type="InArgument(x:String)" />  <x:Property Name="AutoDeployDir" Type="InArgument(x:String)" /> |

1. Add the following elements to the <mtbw:ProcessParameterMetadataCollection> element:

|  |
| --- |
| <mtbw:ProcessParameterMetadata Category="#400 Typemock" Description="This path defines the server path to the Typemock binaries path, for example: '$/MyTeamProject/TypemockAutoDeploy'.&#xA;If the location of the custom activities (defined in the build controller) is the same as the binaries directory then this field can be left empty." DisplayName="Binaries Directory" ParameterName="AutoDeployDir" />  <mtbw:ProcessParameterMetadata Category="#400 Typemock" DisplayName="Company" ParameterName="Company" Required="True" />  <mtbw:ProcessParameterMetadata Category="#400 Typemock" DisplayName="Server License" ParameterName="License" Required="True" /> |

1. Save the custom template.
2. Check-in the changes.

#### Adding Typemock Isolator Tasks

To integrate Typemock Isolator with MS Team Foundation Server using a custom template, you need to add Typemock Isolator’s tasks to the template file and define parameters of these tasks, if necessary.

Typemock Isolator provides the following tasks:

* [TypeMockStart](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/MSBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#start): starts using Typemock Isolator
* [TypeMockStop](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/MSBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#stop): stops using Typemock Isolator
* [TypeMockRegister](http://docs.typemock.com/Isolator/Content.aspx/Ref.chm/Documentation/MSBuild.html?__versionId=E4A58701A5208E0AD80251BD7C3D27E5#register): registers the Typemock Isolator license

##### Starting Typemock Isolator

The TypeMockStart task runs Typemock Isolator on the server.

To run TypeMockStart:

* Add the TypemockStart task to your custom template to the beginning of the <Sequence DisplayName="Run Tests"> sequence with the following parameters:

| Parameter | Description | Type | Mandatory/Optional |
| --- | --- | --- | --- |
| **Link** | Integrates Typemock Isolator with a specified profiler. | String | Optional |
| LogLevel | Specifies the log verbosity.  Possible values:   * 0: no logging * 3: logging is on | Integer | Optional |
| LogPath | Path to log files (by default, it is the current user temporary directory) | String | Optional |
| ProfilerLaunchedFirst | Specifies whether the profiler linked to Typemock Isolator should run before Typemock Isolator.  Possible options:   * true * false   This parameter is not relevant when the auto-deploy mode is used. | Boolean | Optional |
| DisableAutoDeploy | Disables the Auto-Deploy mode of installation. |  |  |
| Version | Gets the Typemock Isolator version. | Output string | Optional |
| Target | .NET version.  Possible values:   * 2.0 * 3.0 * 3.5 * 3.0 * 4,0 * 4.5 | String | Optional |

The following example shows how you can start Typemock Isolator:

|  |
| --- |
| <tm:TypeMockStart EvaluationFolder="{x:Null}" Link="{x:Null}" LogLevel="{x:Null}" LogPath="{x:Null}" ProfilerLaunchedFirst="{x:Null}" DisableAutoDeploy="{x:Null}" Target="{x:Null}" Version="{x:Null}" /> |

##### Stopping Typemock Isolator

The TypeMockStop task stops Typemock Isolator on the server. You must call TypeMockStop after TypeMockStart even if the test failed.

To run TypeMockStop:

* Add the TypemockStop task to your custom template after TypeMockStart with the following parameter**:**

| Parameter | Description | Type | Mandatory/Optional |
| --- | --- | --- | --- |
| Undeploy | Uninstalls Typemock Isolator.  Possible values:   * true * false | Boolean | Optional |

The following example shows you how to stop Typemock Isolator.

|  |
| --- |
| <tm:TypeMockStop Undeploy ="{x:Null}" /> |

##### Registering the Typemock Isolator License

The TypeMockRegister task registers the Typemock Isolator license.

To register the Typemock Isolator license:

* Add the TypeMockRegister task to your custom template with the following parameters:

| Parameter | Description | Type | Mandatory/Optional |
| --- | --- | --- | --- |
| **AutoDeploy** | Specifies whether to run Typemock Isolator tests without installation on the server. | **Boolean** | **Optional** |
| **Company** | The company name as defined in your license | String | Mandatory |
| **License** | The activation key as defined in your license. | String | Mandatory |

The following example shows how to register Typemock Isolator:

|  |
| --- |
| <tm:TypeMockRegister AutoDeployDir="[AutoDeployDir]" Company="[Company]" License="[License]" /> |

# Extending Typemock Isolator

You can extend the existing Typemock Isolator capabilities by:

* Adding your own custom attributes. See [Adding Custom Attributes](#_D2HTopic_369).
* Integrating your own test runners and CI frameworks with Typemock Isolator. See [Typemock Isolator Integration API](#_D2HTopic_370).

## Adding Custom Attributes

With Typemock Isolator you can extend any testing framework by writing custom attributes that will be executed while the tests run. This will give you full control over the flow of the test and an ability to add functionality to the test framework.

Examples of custom decorators are automatic rollback, timeout, and multiple runs. Typemock Isolator comes with prebuilt decorators for best practice usage.

To write a custom attribute, only execute methods need to be implemented. In this example, the trace message is written to the console before and after the method is run. CallDecoratedMethod is used to activate the original method.

|  |  |
| --- | --- |
| ! | Note  Custom attributes cannot be used in ASP.net tests or in generic methods or types. |

|  |  |
| --- | --- |
| ! | Note  The following properties are also defined in the DecoratorAttribute class:   * OriginalMethod: returns a MethodBase object representing the decorated method. * OriginalContext: holds the instance in which the decorated method was called. * OriginalParams: holds the parameters passed to the decorated method |

The following example shows how to write a custom attribute. The attribute will display a trace of the test method's entry and exit.

C#

|  |
| --- |
| /// <summary>  /// An example of how to write new attributes.  /// Decorating a method with this attribute will cause a message to be  /// written to the console at the start and end of the method execution.  /// </summary>  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = false, Inherited = true)]  public sealed class TraceableAttribute : DecoratorAttribute  {  /// <summary>  /// In this execute, we added logic for tracing to a text file before and after activation of the  /// decorated method.  /// </summary>  /// <returns>  /// Results from activating the original invoker on the test (unless overridden) .  /// </returns>  public override object Execute()  {  // Trace start of method  Console.WriteLine("Entering Method: " + OriginalContext.GetType() + " -- " + OriginalMethod.ToString());  try  {  // Execute the original method  return base.CallDecoratedMethod();  }  finally  {  // Trace end of method  Console.WriteLine("Exiting Method: " + OriginalContext.GetType() + " -- " + OriginalMethod.ToString());  }    }    /// <summary>  /// This will cause the attribute to be applied to all methods and not just test  /// </summary>  /// <param name="methodBase">Method being executed</param>  /// <returns>True to decorate the code</returns>  /// <remarks>Default action is to execute only methods that have a [Test] or [TestMethod] attribute</remarks>  protected override bool DecorateMethodWhenAttributeIsClass(MethodBase methodBase)  {  return true;  }  } |

VB

|  |
| --- |
| ' An example of how to write new attributes.  ' Decorating a method with this attribute will cause a message to be  ' written to the console at the start and end of the method execution.  Public Class TraceableAttribute  Inherits DecoratorAttribute  ' In this execute, we added logic for tracing to a text file before and after activation of the decorated method.  ' Returns the results from activating the original invoker on the test (unless overridden).  Public Overrides Function Execute() As Object  Console.WriteLine(String.Format("Entering Method: {0} -- {1}", OriginalContext.GetType(), OriginalMethod.ToString()))  Try  Return CallDecoratedMethod()  Finally  Console.WriteLine(String.Format("Exiting Method: {0} -- {1}", OriginalContext.GetType(), OriginalMethod.ToString()))  End Try  End Function  ' This will cause the attribute to be applied to all methods and not just to test methods.  ' By default, attributes decorating a class will only be applied to test methods.  Protected Overrides Function DecorateMethodWhenAttributeIsClass(ByVal methodBase As MethodBase) As Boolean  Return True  End Function  End Class |

## Typemock Isolator Integration API

The Typemock Isolator Integration API is intended for developers of unit tests runners and CI frameworks who want to integrate their code with Typemock Isolator.

To use the Typemock Isolator Integration API, you need to link your application to TypeMock.Integration.dll.

The Integration API provides two classes:

| Class | Description |
| --- | --- |
| TypeMock.Integration.Service | Provides methods for checking the Typemock Isolator license, the installation path, and Auto-Deploy mode. |
| TypeMock.Integration.TypeMockProcess | Provides methods for running processes with Typemock Isolator and linking Typemock Isolator with code coverage tools. |

### Sample 1: Basic Usage

The following sample shows how to create a class for running unit tests with Typemock Isolator. The UnitTestRunner property is used to set the framework that will run the test. The UnitTestRunnerParams property is used to set the runner command line arguments. In this case, only the assembly with the tests is set.

|  |
| --- |
| class TestsRunner  {  private string unitTestRunner;  private string unitTestRunnerParams;  /// <summary>  /// get or sets the unit test runner (nunit, mbunit etc ...)  /// </summary>  public string UnitTestRunner  {  get { return unitTestRunner; }  set { unitTestRunner = value; }  }  /// <summary>  /// get or set the unit test runner parameters  /// </summary>  public string UnitTestRunnerParams  {  get { return unitTestRunnerParams; }  set { unitTestRunnerParams = value; }  }  public void RunTest()  {  //set the process to run and its command line arguments  ProcessStartInfo info = new ProcessStartInfo(UnitTestRunner, UnitTestRunnerParams);  //create process with TypeMock enabled  TypeMockProcess typeMockProcess = new TypeMockProcess(info);  //run the process  typeMockProcess.Start();  }  }  class Program  {  static void Main(string[] args)  {  //calling the code  TestsRunner runner = new TestsRunner();  runner.UnitTestRunner = @"C:\Program Files\NUnit-Net-2.0 2.2.8\bin\nunit-console.exe";  runner.UnitTestRunnerParams = @"..\..\..\..\csharp\bin\Debug\Examples.csharp.dll";  runner.RunTest();  }  } |

### Sample 2: Linking with Code Coverage Profiler and Running the Tests

The following sample shows how to run a test is run with NCover:

1. The license is checked.
2. The GetCoverageTools() method prints all available profilers on the machine. When the GetCoverageTools() method has a false argument, the method will return a list of all the profilers with which Typemock Isolator can be linked.
3. The code unlinks any previous profiler that was linked with Typemock Isolator.
4. The static TypeMockProcess.Start() method runs the tests.

|  |
| --- |
| public void RunTest(string profilerName)  {  if(! Service.HasLicenseToLink())  {  Console.WriteLine("In order to link Typemock Isolator you need a valid licence");  return;  }    Console.WriteLine("Availble profilers on this machine:");  foreach(string profiler in Service.GetCoverageTools(true))  {  Console.WriteLine(profiler);  }  //First unlink Typemock Isoaltor with any previous profiler  TypeMockProcess.UnlinkWithCovarage();    //Set the process to run and its command line arguments  ProcessStartInfo info = new ProcessStartInfo(UnitTestRunner, UnitTestRunnerParams);    //Run the process with TypeMockEnabled and link with profiler  using(TypeMockProcess.Start(info, profilerName))  {  }  } |

|  |  |
| --- | --- |
| ! | Note  When you are debugging code that uses TypeMockProcess, remember to disable Typemock Isolator from the MS Visual Studio's Tools menu. If Typemock Isolator is enabled, any process that you create will be enabled as well. This will not represent a real-life situation when the application is running outside of MS Visual Studio. |

### Sample 3: Using AutoDeploy

To use the Auto-Deploy mode, use the TypeMock.Integration.Service.InstallFrom() method. You can check if the machine has a valid license using the TypeMock.Integration.Service.HasLicenseForAutodeploy() method.

|  |
| --- |
| public void DoAutoDeploy(string path)  {  if(! Service.HasLicenseForAutodeploy())  {  Console.WriteLine("You don't have license for Auto Deploy Typemock Isolator");  return;  }  Service.InstallFrom(path);  } |

# Upgrading Your License

After your trial license expired, you can provide a new license code.

To provide a new license code:

1. Select Typemock > Options.

The Options window is displayed.

1. In the options list, select Typemock > License.
2. In the License Code field, enter the new license code.
3. Click Set License.

# Appendix A: Useful Shortcuts

| Action | Shortcut |
| --- | --- |
| Open Test Navigator | Alt + t + n |
| Run test | Alt + t + t |
| Open Method Shield window | Alt + t + s |
| Rerun last test | Alt + t + l |
| Run unit pending tests | Alt + t + p |
| Cancel test run | Alt + t + Esc |
| Debug | Alt + t + d |

This is a body text.

## Heading 2

## Heading 2

## Heading 2

# Chapter 2

## Heading 2

### Heading 3

#### Heading 4

#### Heading 4

## Heading 2

### Heading 3

### Heading 3

#### Heading 4