The Gallio Book

Jeff Brown
Julián Hidalgo

The Gallio Book

by Jeff Brown and Julián Hidalgo

3.0.4.0 beta

Copyright © 2008 Gallio Project

Table of Contents

Preface vi
Why This Book? vi
Who is This Book For? vi
I. Introduction 1
1. About Gallio 3
What is Gallio? 3
History 3
2. Installation 4
The Gallio Zip Distribution 4
The Gallio Installer 4
Plugins 4
Runners 5
Tools integration 5
Documentation 5
Extras 5
3. Getting Started 6
II. MbUnit v3 15
4. About MbUnit 17
What is MbUnit? 17
MbUnit v2 vs. MbUnit v3 17
Feature Matrix 17
Requirements 17
5. Unit Testing 18
6. Data Driven Testing 19
7. Web Testing 20
8. Database Testing 21
9. Recipes 22
10. Contract Verifiers 23
The Exception Contract Verifier 23
The Equality Contract Verifier 23
The Comparison Contract Verifier 23
11. Troubleshooting 24
12. Migration Guide 25
III. Test Framework Adapters 26
13. csUnit 27
14. MbUnit v2 28
15. MSTest 29
16. NUnit 30
17. xUnit.Net 31
IV. Test Runners 32
18. Icarus GUI 33
19. Sail, a Lightweight Visual Studio Test Runner 34
20. TestDriven.Net 35
21. Visual Studio Team System 36
22. ReSharper 37
23. Echo Command Line 38
24. Powershell 39
25. MSBuild 40
26. NAnt 41
V. Test Tools 42
27. Ambience 44

28. NCover 45
29. Pex 46
30. TypeMock 47
31. WatiN 48
32. CruiseControl.Net 49
33. TeamCity 50
VI. Extending Gallio and MbUnit 51
34. Understanding the Gallio Architecture 52
35. Creating Plugins 53
36. Creating Test Frameworks 54
Concepts 54
The EasyTest SampleFramework 54
EasyTest syntax 54
Low-Level Implementation 54
High-Level Implementation 54
Summary 55

Preface

Why This Book?

One of the biggest problems in many open source projects is the lack of proper documentation. This is a shame because there is not point in implementing a thousand features if no one knows about them. The Gallio development team was very aware of this issue, so documentation was given a high priority from the beginning of the project. The API reference documentation, which is automatically generated from the source code, was the first sign of this concern, but the team knew users need more: they need a guide to show them how to use the multiple features, the runners and so on under different scenarios. So the idea of writing a book was there from the beginning, in our wishlist, but we knew it was a big task so it was posponed.

Around February in 2008, the topic arised again in a conversation between Jeff Brown, Gallio's lead and main developer, and I. The project was approaching a new alpha release and a beta version was not too far away either, so there was a lot to write about. I decided I'd go ahead and write the book myself, even though I felt (and still feel) I was not the best person to do it, hoping that the rest of the team and the community would help me to fill the gaps in my English and my time.

I'm glad to see we are publishing the first draft of the book. I sincerely hope it'll help you to leverage the full power of this exciting platform and all the tools that are been built around it. Happy testing!

Who is This Book For?

- Developers.
- Testers.
- People wanting to extend Gallio.

Part I. Introduction

Table of Contents

1. About Gallio 3
    - What is Gallio? 3
    - History 3
2. Installation 4
    - The Gallio Zip Distribution 4
    - The Gallio Installer 4
        - Plugins 4
        - Runners 5
        - Tools integration 5
        - Documentation 5
        - Extras 5
3. Getting Started 6

Chapter 1. About Gallio

What is Gallio?

The Gallio Automation Platform is an open, extensible, and neutral system for .NET that provides a common object model, runtime services and tools (such as test runners) that may be leveraged by any number of test frameworks.

History

In January 2004 Marc Clifton, a frequent contributor at Codeproject, wrote a series of articles that sought to expand the unit testing discussion. Among other things, Marc proposed a formalization of various test patterns beyond basic TDD. Marc then took his ideas into code as AUT (Advanced Unit Testing), an independed project that you can find at Codeproject.

Two months later, Jonathan "Peli" de Halleux took a look at Marc's proposals and created gUnit (which was later renamed to MbUnit) while recovering from surgery in a hospital. In fact, Peli wrote most of MbUnit while still in the hospital.

MbUnit had some new ideas and concepts and it caught the attention of Jamie Cansdale who while on a trip to Brussels hooked up with Peli to work on an add-on for TD.net. TD.net started life as a NUnit project and so this made MbUnit the next framework after NUnit to be supported by TD.net, as such since the very early days of this great tool there has been MbUnit support.

In 2005 Peli made MbUnit opensource and continued working on the framework while finishing his PhD. Shortly after completing his PhD he accepted a job with Microsoft as a SDE\T on the CLR team. Unable to carry on MbUnit, he handed it over to Jamie Cansdale as short time caretaker. Peli blogged about needing someone to take on MbUnit and shortly after Andy Stopford as a long time MbUnit user read this and stepped up.

Since then MbUnit has grown as a framework and project, with two major releases and triple the downloads per release it has firmly rooted itself in main stream Microsoft .net culture as a viable unit test framework next to NUnit.

In the autumn of 2007, MbUnit v3 - a ground up rewrite of MbUnit, started. In one of those funny turn of events, v3 was to be code named "Gallileo" but due to a typo became "Gallio". The name stuck and development continued on MbUnit v3, code name: Gallio.

With MbUnit V3 developmement well under way, long time MbUnit core member Jeff Brown attended the Alt.Net conference in Austin, Texas. Following discussions with other programmers at the conference, Jeff made the case to the MbUnit team that there was value to the community at large in isolating the test runner capabilities of the system to create a neutral platform upon which MbUnit could then be hosted as one of many supported frameworks. Other open-source and commercial projects would be able to leverage the platform's services to create rich, interoperable and extensible testing solutions, thereby adding great value to the community.

After much discussion, the decision was made to separate the test runner from MbUnit and Gallio the Automation Platform was born.

Going forward the Gallio Project seeks to become visible to other open source projects so that the capabilities of the platform can bring unity and value to the many projects in the testing space.

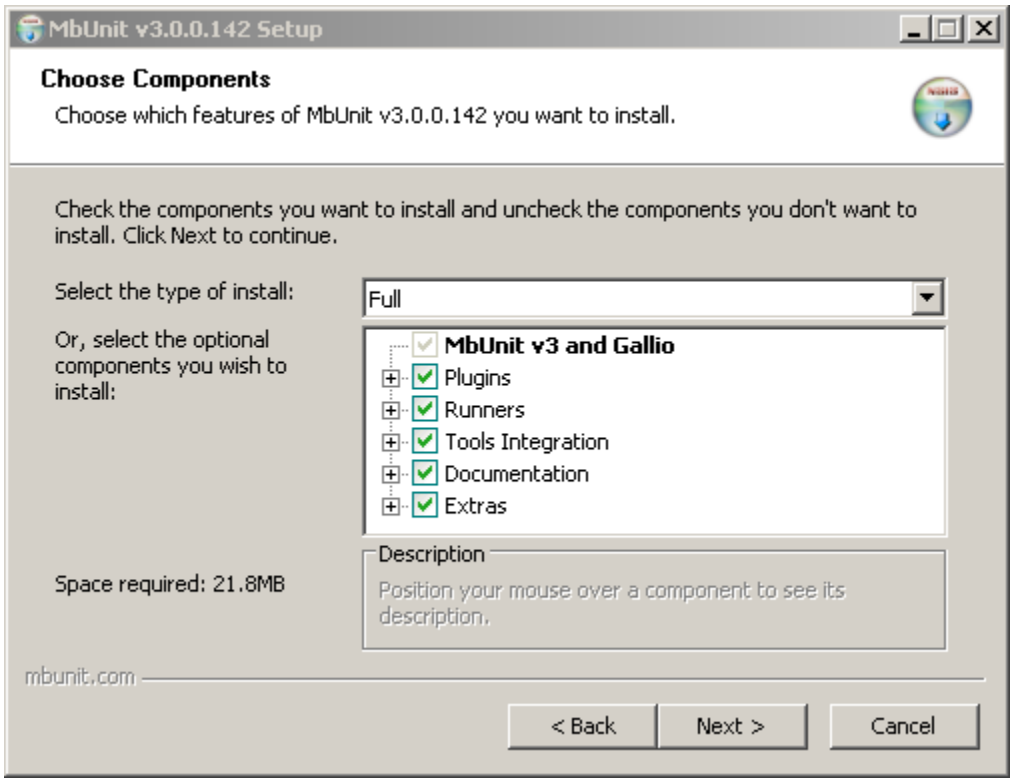
Chapter 2. Installation

Gallio comes in two flavors: an installer and a zip file. The installer is probably better for individual development. If you work on a team you might prefer the zip distribution so you can put the files in a convenient, source-controlled location.

In any case you can download the latest release from http://www.gallio.org/.

The Gallio Zip Distribution

The Gallio Installer



Plugins

Plugins add support for running tests created with frameworks other than MbUnit v3. The currently available ones are:

- MbUnit v2: Allows you to run MbUnit v2 tests with Gallio
- MSTest: Allows you to run MSTest tests with Gallio (only if you have Visual Studio 2008 / Visual Studio Team System installed)
- NUnit: Allows you to run NUnit tests with Gallio
- xUnit.NET: Allows you to run xUnit.NET tests with Gallio

Unless you need to run tests created with these frameworks, you don't need to install these plugins.

Runners

Runners are programs or plugins for other programs that provide different ways to run Gallio. The chosen runner has no effect on the availability of plugins, that is, you can for example run NUnit tests with any of the runners provided you have installed the right plugin.

- Echo: The console test runner
- Icarus: The graphical test runner
- MSBuild task: Allows you to run Gallio from an MSBuild build script
- NAnt task: Allows you to run Gallio from a NAnt build script
- PowerShell commands: Allows you to run Gallio from a PowerShell script
- ReSharper v3 plug-in: Allows you to run Gallio from ReSharper
- TestDriven.NET runner for MbUnit3: Allows you to run MbUnit v3 tests with Gallio from the TestDriven.NET Visual Studio add-in
- TestDriven.NET runner for Other Supported: Allows you to run tests from the supported frameworks (MbUnit v2, NUnit, xUnit.NET and MSTest) with Gallio from the TestDriven.NET Visual Studio add-in

You probably don't need the MSBuild and NAnt tasks unless you are implementing a build server.

Tools integration

- NCover integration: Allows you to enable code coverage with NCover by simply setting a property
- TypeMock integration: Allows you to enable TypeMock.NET by simply setting a property

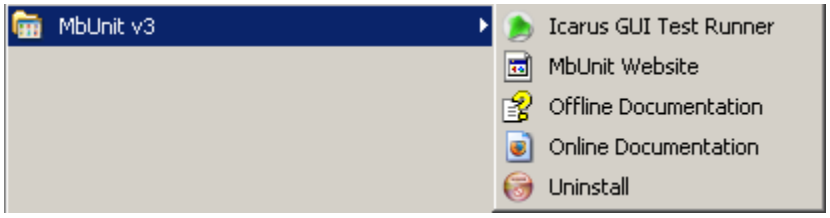
Documentation

- Standalone Help Docs: Install the CHM documentation
- Visual Studio 2005 Help Docs: Install the integrated documentation for Visual Studio 2005

Extras

- CruiseControl .NET extensions: Provides an extension to allow downloading attachment from the CCNet build report. It's only useful if you are implementing a build server.

The installer will create a folder in the Start Menu, where you will see shortcuts for the Icarus GUI runner, the MbUnit website, the offline documentation, the online documentation and the unistaller.



Chapter 3. Getting Started

In this chapter you will see step by step how to create and run a simple test project using Gallio and its default test framework, MbUnit v3. It's assumed that you already know how to create projects and add references in Visual Studio, but no prior knowledge of unit testing or test frameworks is assumed. The screenshots were taken in Visual Studio 2008, but the steps are the same for Visual Studio 2005 as well.

Let's start by creating a new class library project in Visual Studio called **SimpleLibrary**. Delete the default class (normally **Class1**) that's added by Visual Studio and add a new one called **Fibonacci**. It will be a pretty simple public class with only one method called **Calculate**, as shown here:

```
namespace SimpleLibrary
{
    public class Fibonacci
    {
        public static int Calculate(int x)
        {
            return Calculate(x - 1) + Calculate(x - 2);
        }
    }
}
```

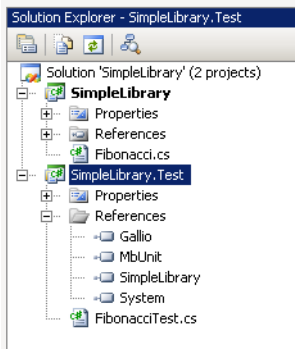
```
Public Class Fibonacci

    Public Shared Function Calculate(ByVal x As Integer) As Integer
        Return Calculate(x - 1) + Calculate(x - 2)
    End Function

End Class
```

We will write some tests to verify that this method is working properly. Many people like to write the tests first, in what is called "Test driven development", usually abreviated "TDD". If we were working in a TDD way we would have created a test, made it fail and only then we would have implemented the code required to make it pass (in this case the Calculate method's body), just to start over in what is known as the "red, green, refactor" cycle. Since the chosen development methodology doesn't affect the way we use the framework, we won't follow anyone in particular.

Tests should **never** be put in your production code, but in separate projects/assemblies. Add a new class library project to the SimpleLibrary solution called **SimpleLibrary.Test**, delete the default class and add a new one called **FibonacciTest**. In the Gallio source code the convention is to name a test project after the project it's testing plus the suffix '.Tests', and to name a unit test class after the class it's testing plus the suffix '.Test'. This is a popular convention, but as you may guess everyone has its own preferences.

Now, still in the test project, you need to add a reference to the SimpleLibrary project, and also to the Gallio.dll and MbUnit.dll assemblies that you can find in the bin folder of your Gallio install folder. In case you don't know the path, look for a shortcut to it in the Gallio program group of the Start menu. Note that the common practice is to have this assembly as well as other referenced assemblies in a custom folder in your solution, mainly for location independency and versioning issues, but we will cover this scenario in other chapters.

At this point, your solution should look like this:



With the project structure and references in place we can now write the first unit test. The code is the following:

```
using System;
using SimpleLibrary;
using MbUnit.Framework;

namespace SimpleLibrary.Test
{
    public class FibonacciTest
    {
        [Test]
        public void FibonacciOfNumberGreaterThanOne()
        {
            Assert.AreEqual(Fibonacci.Calculate(6), 8);
        }
    }
}
```

```
Imports SimpleLibrary
Imports MbUnit.Framework

Public Class FibonacciTest

    <Test()> _
    Public Sub FibonacciOfNumberGreaterThanOne()
        Assert.AreEqual(Fibonacci.Calculate(6), 8)
    End Sub

End Class
```

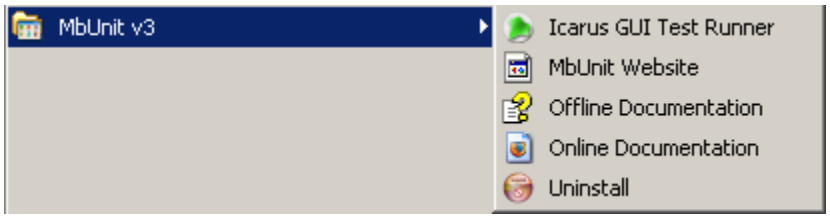
Notice the things we did in the previous code:

- We imported the **SimpleLibrary** namespace
- We imported the **MbUnit.Framework** namespace
- We made the FibonacciTest class **public** (you may need to explicitly do so in your language of choice)
- We added a new **public** method called FibonacciOfNumberGreaterThanOne, which doesn't return a value

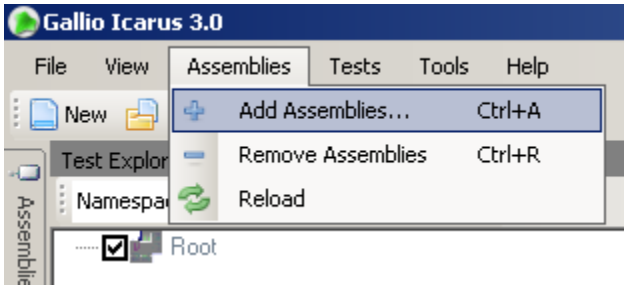
- We decorated the FibonacciOfNumberGreaterThanOne method with the **[Test]** attribute
- We added a call to the **Assert.Equal** method

You may be wondering what's the call to the Assert.AreEqual method supposed to do. Not too much in fact: it only checks that the return value of the call to Fibonacci.Calculate(6) is equal to 8. The Assert class is very important because it contains a lot of helpful methods that make writing tests easier.

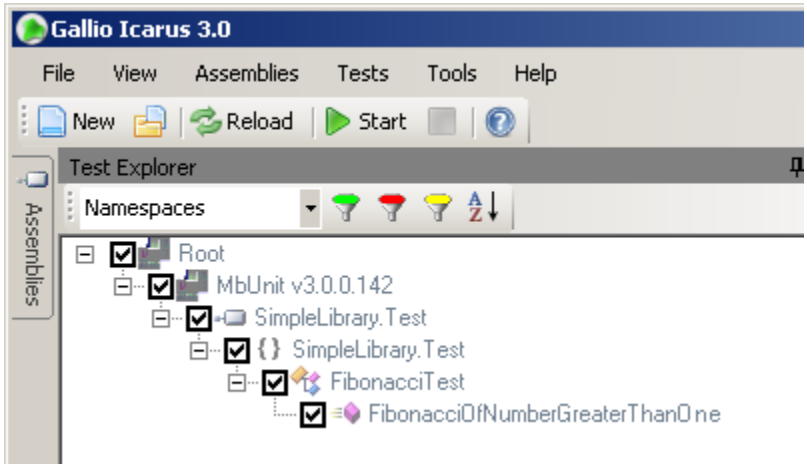
Now we have our first unit test, but how do we execute it? Gallio comes bundled with many different runners, that is, programs or plugins for other programs that allows you to execute tests. This time we will use **Icarus**, the graphical runner, because it's a standalone application (meaning that you don't need to install anything else to run it). The Gallio installer creates a shortcut for it in the programs folder of the Start menu, so it's pretty easy to launch it:



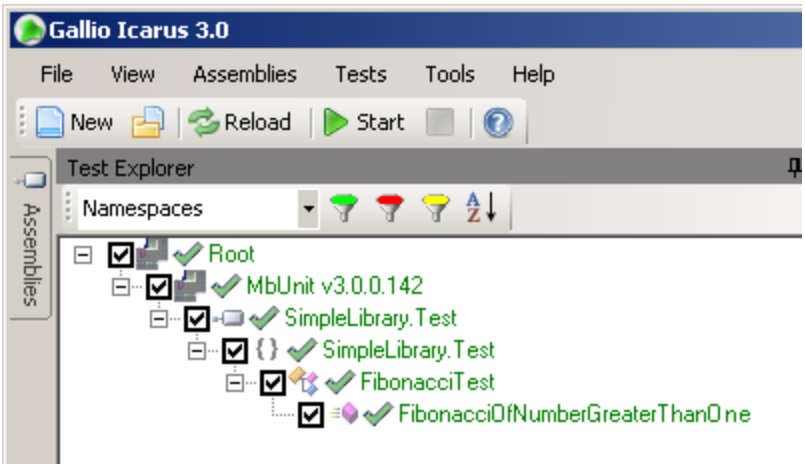
Once Icarus is running, go to **Assemblies->Add Assemblies...** in the menu:



Browse to the folder where you put the solution and look for the SimpleLibrary.Test.dll assembly (probably located in bin\Debug under the test project's folder). The test tree is populated as shown in this screenshot:



You can see the MbUnit version under the Root node, and under it the names of the assembly, the fixture and the test method. The next step is to execute the test, for which you only need to press the **Start** button. After doing so we see that it passes:



But what does it mean for a test to "pass"? It means that it was executed, all its assertions were true and no exception was thrown. This is the basic structure of a test in the so called state-based testing: you create one on more objects, call a method and assert over the state of it after doing it.

So far so good, but we need to test more scenarios to make sure that the test didn't pass because we were lucky, but because the tested method is actually working. In other words, we need to test for different values of x. Since we cannot test every possible value, the common practice is to pick a few representative cases. In our case, it's clear that we also need to test at least the following three scenarios:

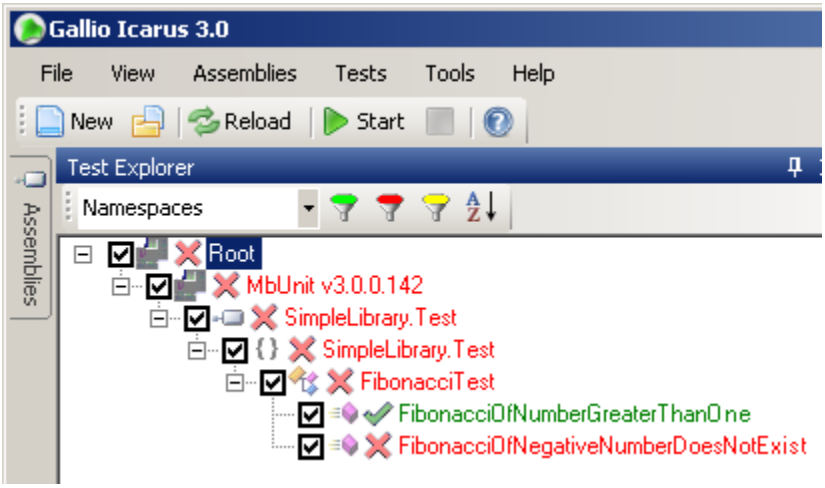
- x less than 0
- x equal to 0
- x equal to 1

We face an interesting case: what should we do if x is a negative number? The Fibonacci function is only defined for numbers greater or equal than zero, so it makes sense to throw an exception in that case. But how do we test that behavior? We said a test will only pass if no exception is thrown during its execution. One option would be to put the call to the **Fibonacci.Calculate** in a try-catch block, calling the Assert.Fail or throwing a new exception method in case the one we are waiting for is not thrown, but a better option is to use the handy **ExpectedException** attribute that will do just that for us:

```
[Test]
[ExpectedException(typeof(ArgumentException))]
public void FibonacciOfNegativeNumberDoesNotExist()
{
    Fibonacci.Calculate(-1);
}
```

```
<Test()> _
<ExpectedException(GetType(ArgumentException))> _
Public Sub FibonacciOfNegativeNumberDoesNotExist()

    Fibonacci.Calculate(-1)

End Sub
```

We run the test and we see it fails (but note that FibonacciOfNumberGreaterThanOne keeps passing):

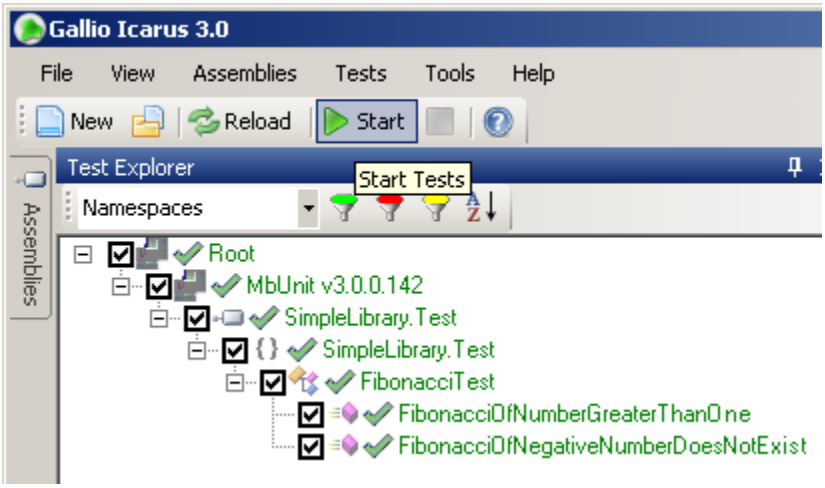


This is what we would expect, since we are not trowing any exception in the Calculate method. Let's fix that:

```
public static int Calculate(int x)
{
    if (x < 0)
        throw new ArgumentException("x must be greater than or equal to zero")

    return (x > 1) ? Calculate(x - 1) + Calculate(x - 2) : x;
}
```

```
Public Shared Function Calculate(ByVal x As Integer) As Integer

    If x < 0 Then
        Throw New ArgumentException("x must be greater than or equal to zero")
    End If

    If x < 2 Then
        Return x
    End If

    Return Calculate(x - 1) + Calculate(x - 2)

End Function
```

We execute the tests again and all of them pass this time:



All that's left is to add tests for x equal to 0 and 1. But instead of writing a test for each of them, we will use a powerful feature called row testing to reduce the amount of coding we need to do. Here's the code:

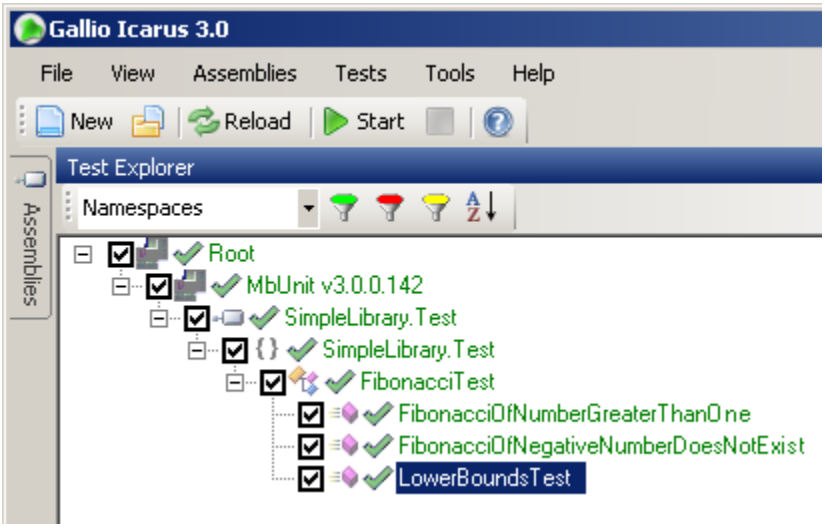
```
[Test]
[Row(0, 0)]
[Row(1, 1)]
public void LowerBoundsTest(int x, int Fibonacci)
{
    Assert.AreEqual(Fibonacci.Calculate(x), Fibonacci);
}
```

```
<Test()> _
<Row(0, 0)> _
<Row(1, 1)> _
Public Sub LowerBoundsTest(ByVal x As Integer, _
  ByVal expectedFibonacciNumber As Integer)

    Assert.AreEqual(Fibonacci.Calculate(x), expectedFibonacciNumber)

End Sub
```

There are a few differences between this test and the previous we wrote:

- We applied a **Row** attribute for each of the scenarios we want to test (two in this case)
- This test method receives **2 parameters**, just as the number of items in each Row attribute

So what does this all mean? Gallio will create test instances for each Row attribute, and will pass each value of it as a parameter to the test method (in the same order by default), converting to the right type if necessary. We run the tests in Icarus again and voilá!, they pass:

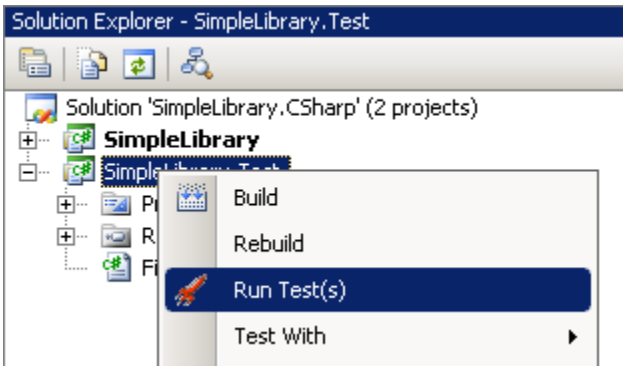


Row testing is one of the simplest cases of data-driven testing. As we will see later, Gallio has powerful data-binding capabilities, supporting a heterogeneous set of datasources and many ways to manipulate and scope them.

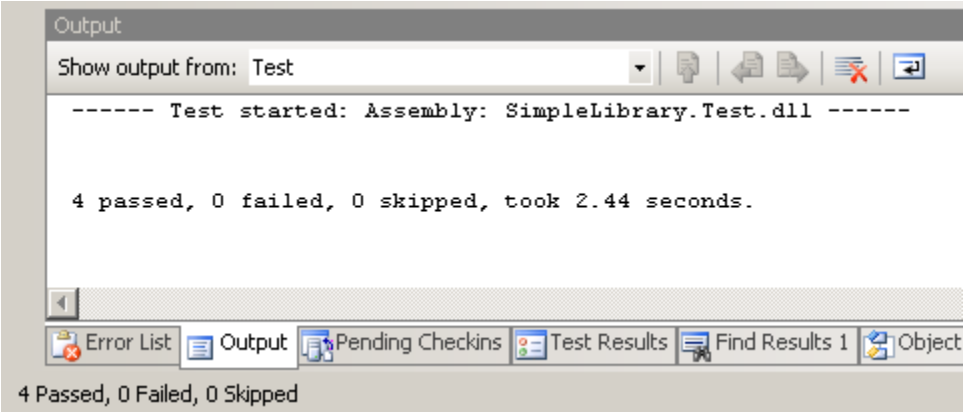
As the last thing in this chapter, we will see how to run the tests we created with the **TestDriven.NET** add-in, which is one of the most popular ways for developers to run tests in Visual Studio.

You can download TestDriven.NET from its website, http://www.testdriven.net/. It's a commercial product, but at the time of this writing there's a personal version of it.

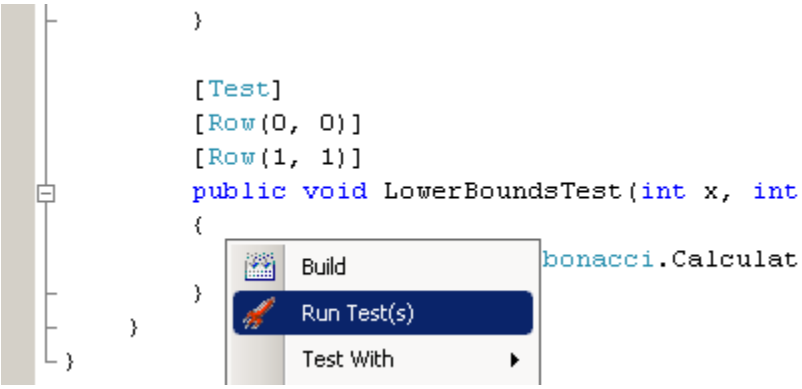
After you have downloaded and installed TestDrive.NET, open the sample solution again and right-click on the SimpleLibrary.Test project, as shown in the screenshot:



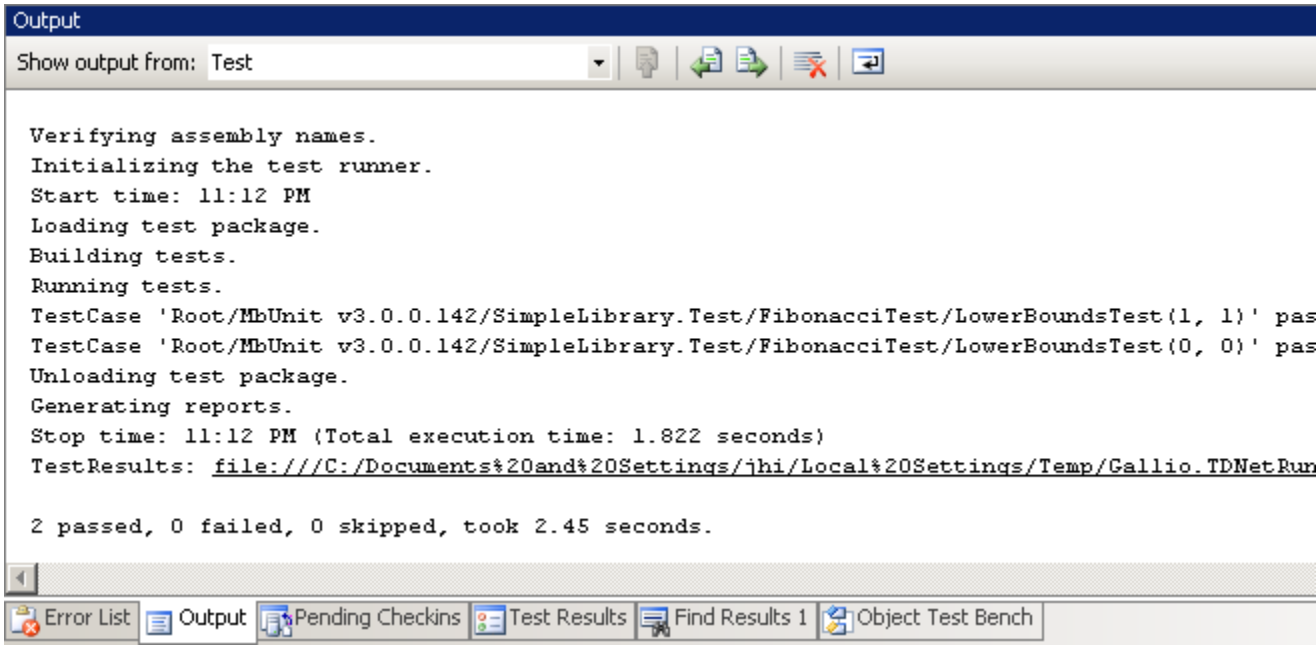
In a few seconds you see this output:



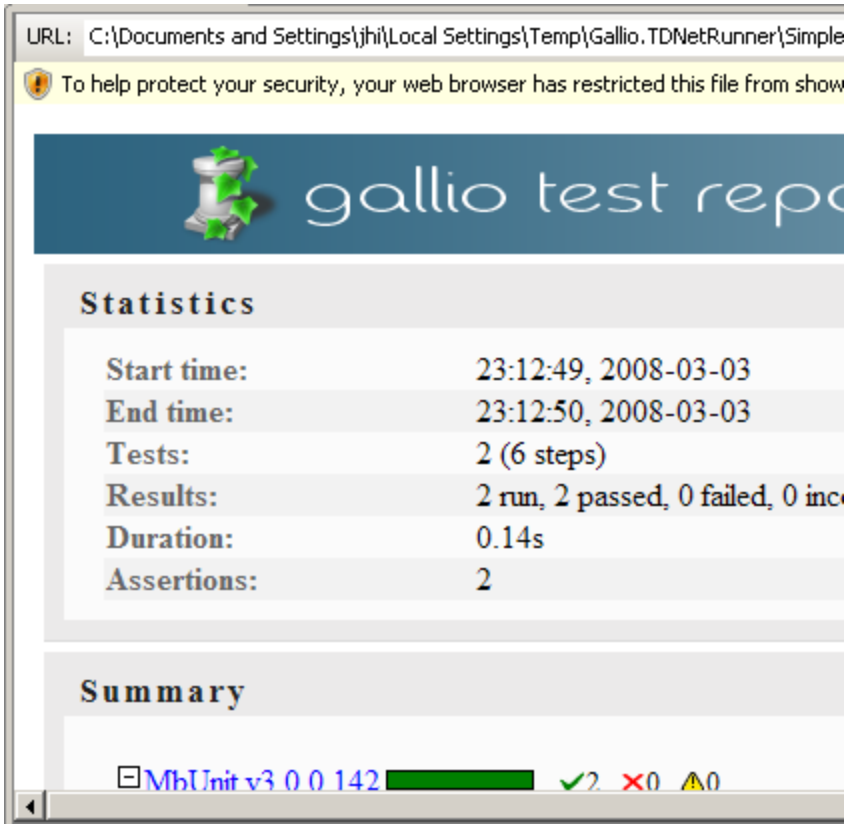
The real benefit, however, of using the TestDriven.NET add-in comes when you want to run individual tests:



In this case, the output window shows pretty useful information, like which test instances were executed, the parameters they were passed and the outcome of each one:



Also there is a link to the generated HTML report. If you Control-click it it will be opened inside Visual Studio:



This report is always generated in a folder called Gallio.TDNetRunner within your temporary files folder. Unfortunately the link is not displayed when executing tests at the assembly level, but the report is generated anyway (this is the way TestDriven.NET works, so there's nothing we can do about it other than asking its author, Jamie Cansdale, to change it).

You know now pretty much everything you need to get started. But keep reading! There is still much more to discover in the next chapters.

Part II. MbUnit v3

Table of Contents

4. About MbUnit 17
    What is MbUnit? 17
    MbUnit v2 vs. MbUnit v3 17
    Feature Matrix 17
    Requirements 17
5. Unit Testing 18
6. Data Driven Testing 19
7. Web Testing 20
8. Database Testing 21
9. Recipes 22
10. Contract Verifiers 23
    The Exception Contract Verifier 23
    The Equality Contract Verifier 23
    The Comparison Contract Verifier 23
11. Troubleshooting 24
12. Migration Guide 25

Chapter 4. About MbUnit

What is MbUnit?

MbUnit is a unit testing framework in the tradition of xUnit frameworks such as JUnit. In addition, MbUnit includes a rich suite of features designed to simplify other automation tasks that arise during integration testing.

MbUnit was originally created in 2005 by Jonathan "Peli" de Halleux. It introduced and popularized novel ideas such as "RowTests" and "CombinatorialTests".

The present incarnation of MbUnit, MbUnit v3, represents a complete rewrite and redesign of Peli's original work to improve the end-user experience, consolidate features, enhance extensibility, and enable advanced integration testing and reporting. From this synthesis, Gallio was born.

For more information about the early history of MbUnit, refer to the introductory chapter about Gallio.

MbUnit v2 vs. MbUnit v3

The subject of this section is MbUnit v3. However, we will occasionally refer to the older MbUnit v2. This section will explain a few of the differences between these versions.

MbUnit v3 is a .Net 2.0 based framework. It uses generic types and methods where possible to encourage code reuse. It also provides additional features for .Net 3.5 clients in a separate assembly. MbUnit v3 leverages the Gallio test automation platform heavily to provide integration with numerous other tools and to enable functionality such as rich reporting.

MbUnit v2 is a .Net 1.1 based framework with a few .Net 2.0 add-ons. It is stand-alone framework that includes its own suite of test runners. Gallio includes an adapter plugin so that Gallio-based tools may also be used with MbUnit v2 (when running tests in a .Net 2.0 environment). However, since MbUnit v2 was not originally designed for Gallio, it does not provide as many advanced features as MbUnit v3.

MbUnit v2 is being maintained concurrently with MbUnit v3 for the benefit of existing projects based on MbUnit v2 that have not yet migrated to MbUnit v3. For new projects, we recommend adopting MbUnit v3.

MbUnit v3 is mostly backwards compatible with MbUnit v2 except for some APIs that have been renamed or redesigned. Transitioning to MbUnit v2 is relatively straightforward.

For more information about the differences between MbUnit v2 and v3, please refer to the Migration Guide chapter.

Feature Matrix

TODO

Requirements

.Net 2.0 runtime (CLR or Mono)

TODO

Chapter 5. Unit Testing

TODO

Chapter 6. Data Driven Testing

TODO

Chapter 7. Web Testing

TODO

Chapter 8. Database Testing

TODO

Chapter 9. Recipes

TODO

Chapter 10. Contract Verifiers

<partintro>

The contract verifiers are built-in test fixtures for very common contractual objects such as custom exceptions or types implementing the generic IEquatable interface. A contract verifier is a complete and configurable test suite. It automatically enables several test methods which evaluate exhaustively the functionalities and the behaviors of your custom contractual type.

The contract verifiers are declared by adding an attribute to your test fixture. An hypothetical contract verifier for the **Foo** contract would be declared like this:

```
[TestFixture]
[FooContractVerifier(option1=value1, option2=value2)]
public class MyFooTest
{
}
```

Remark the different options passed through the constructor of the attribute. Those options are used to configure the contract verifier. They also make it more flexible by letting you enable or disable certain features at your convenience.

The current version of MbUnit v3 provides the following built-in contract verifiers:
</partintro>

The Exception Contract Verifier

The Exception Contract Verifier adds several tests to a given fixture which evaluates a custom user exception. The contract verifier is declared as an attribute with the following syntax:

The Equality Contract Verifier

TODO

The Comparison Contract Verifier

TODO

Chapter 11. Troubleshooting

TODO

Chapter 12. Migration Guide

TODO

Part III. Test Framework Adapters

Gallio provides test framework adapters for numerous test frameworks. This part describes how to use each of them.

Chapter 13. csUnit

TODO

Chapter 14. MbUnit v2

TODO

Chapter 15. MSTest

TODO

Chapter 16. NUnit

TODO

Chapter 17. xUnit.Net

TODO

Part IV. Test Runners

Gallio's test runners integrate with a variety of tools. Some test runners are standalone applications, others integrate with the IDE, and a few provide specialized tasks for release engineering purposes. This part describes how to use each of them.

Choose your favorite!

Chapter 18. Icarus GUI

TODO

Chapter 19. Sail, a Lightweight Visual Studio Test Runner

Chapter 20. TestDriven.Net

Chapter 21. Visual Studio Team System

Chapter 22. ReSharper

Chapter 23. Echo Command Line

TODO

Chapter 24. Powershell

Chapter 25. MSBuild

Chapter 26. NAnt

Part V. Test Tools

Table of Contents

27. Ambience 44
28. NCover 45
29. Pex 46
30. TypeMock 47
31. WatiN 48
32. CruiseControl.Net 49
33. TeamCity 50

Chapter 27. Ambience

TODO

Chapter 28. NCover

TODO

Chapter 29. Pex

TODO

Chapter 30. TypeMock

TODO

Chapter 31. WatiN

TODO

Chapter 32. CruiseControl.Net

TODO

Chapter 33. TeamCity

TODO

Part VI. Extending Gallio and MbUnit

Gallio and MbUnit provide many opportunities for extension. This part covers different scenarios and sample implementations.

Chapter 34. Understanding the Gallio Architecture

Chapter 35. Creating Plugins

TODO

Chapter 36. Creating Test Frameworks

Concepts

TODO

The EasyTest SampleFramework

The EasyTest sample illustrates the process of creating a new test framework using Gallio. Gallio provides the platform and test runners so the framework is only concerned with the syntax.

Our sample test framework will be called EasyTest. We present two different implementations of EasyTest dubbed Low-Level and High-Level.

EasyTest syntax

To keep the sample simple but interesting, EasyTest defines only a few different syntax elements.

TODO

Low-Level Implementation

The low-level implementation is a bare-metal implementation of the framework on Gallio. It directly implements the ITestFramework interface and performs its own reflection.

Working in this way, a framework has full control over its syntax. It can generate test structure using reflection and other means at will.

The low-level interfaces shown here are also applicable to the implementation of adapters for existing test frameworks. A test framework adapter only differs in how it generates the structure of its tests. Instead of performing reflection, it may ask the adapted test framework to generate a tree of tests which is then transformed into a Gallio test tree.

Likewise, the low-level interfaces may be used to construct tests from metadata stored in other forms such as XML-based test specifications. Or adapters may be created for test frameworks not implemented with .Net (by means of pipes, for example).

The low-level interfaces expose all of the power of the Gallio test framework API. However, if you are developing a new test framework from scratch you may leverage more Gallio infrastructure using the high-level interfaces described next.

TODO concepts

TODO implementation walkthrough

High-Level Implementation

The high-level implementation leverages an abstract test framework provided by Gallio, called the Pattern Test Framework.

The Pattern Test Framework is itself a low-level implementation of a test framework based on reflection over attributes. Each attribute implements a distinct pattern: a rule for interpreting the semantics of the attribute and collaboratively populating the test tree.

Defining new rules is as simple as subclassing existing attributes and overriding parts of their implementation to define new semantics. We will discuss this process in greater detail below.

For branding purposes, a test framework built on the Pattern Test Framework should implement IPatternTestFrameworkExtension. This ensures that the name of the derived framework can be presented to the user as appropriate.

The Pattern Test Framework is somewhat abstract but very powerful. It is designed to simplify the task of constructing test frameworks with a highly composable and general feature set. MbUnit v3 itself is based on it. However, if you require more control over the process of test exploration and test execution you may prefer using the low-level interfaces directly instead.

TODO concepts

TODO implementation walkthrough

Summary

In this chapter we have discussed two different implementation strategies for custom test frameworks on Gallio.

The low-level interfaces expose all of the power of Gallio and enable infinite customization of the test framework syntax. The high-level interfaces, in the form of the Pattern Test Framework, provide a convenient foundation for a powerful and composable test framework.

Build your own.