Agile .NET Development – Test-driven Development using NUnit

Jason Gorman

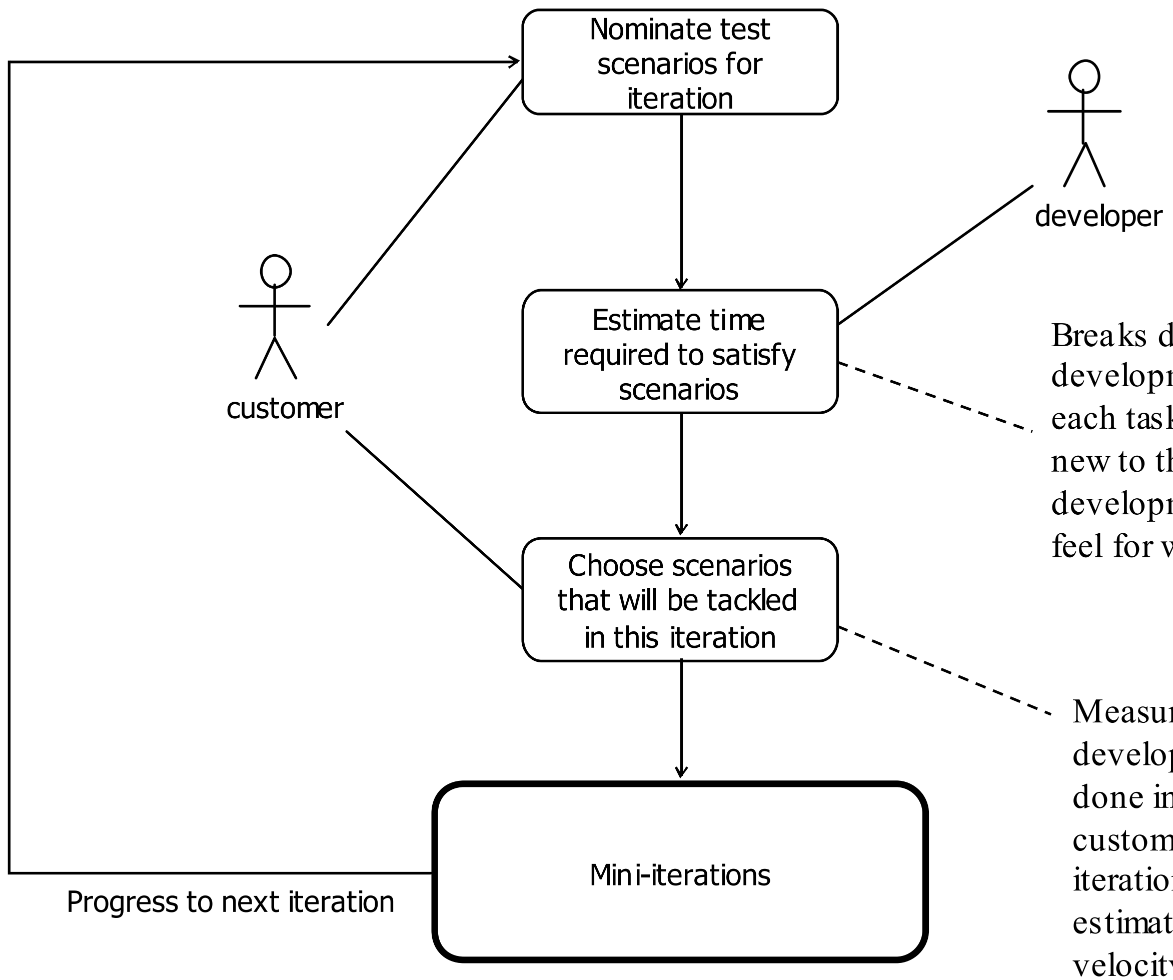
Test-driven Development

- Drive the design and construction of your code on unit test at a time
- Write a test that the system currently fails
- Quickly write the code to pass that test
- Remove any duplication from the code
- Move on to the next test

TDD Benefits

- Focuses your efforts on specific concrete test cases and helps pace development (micro-iterations)
- Provides continuous feedback about whether your code is working
  - Boosts confidence when working on complex systems
- Evolves code to do *exactly what you need* and no more
- Provides a suite of tests you can run often to be sure your code is always working
  - Code is easier to change without inadvertently breaking dependant code
  - Means better assurance when integrating your code into CVS, VSS etc

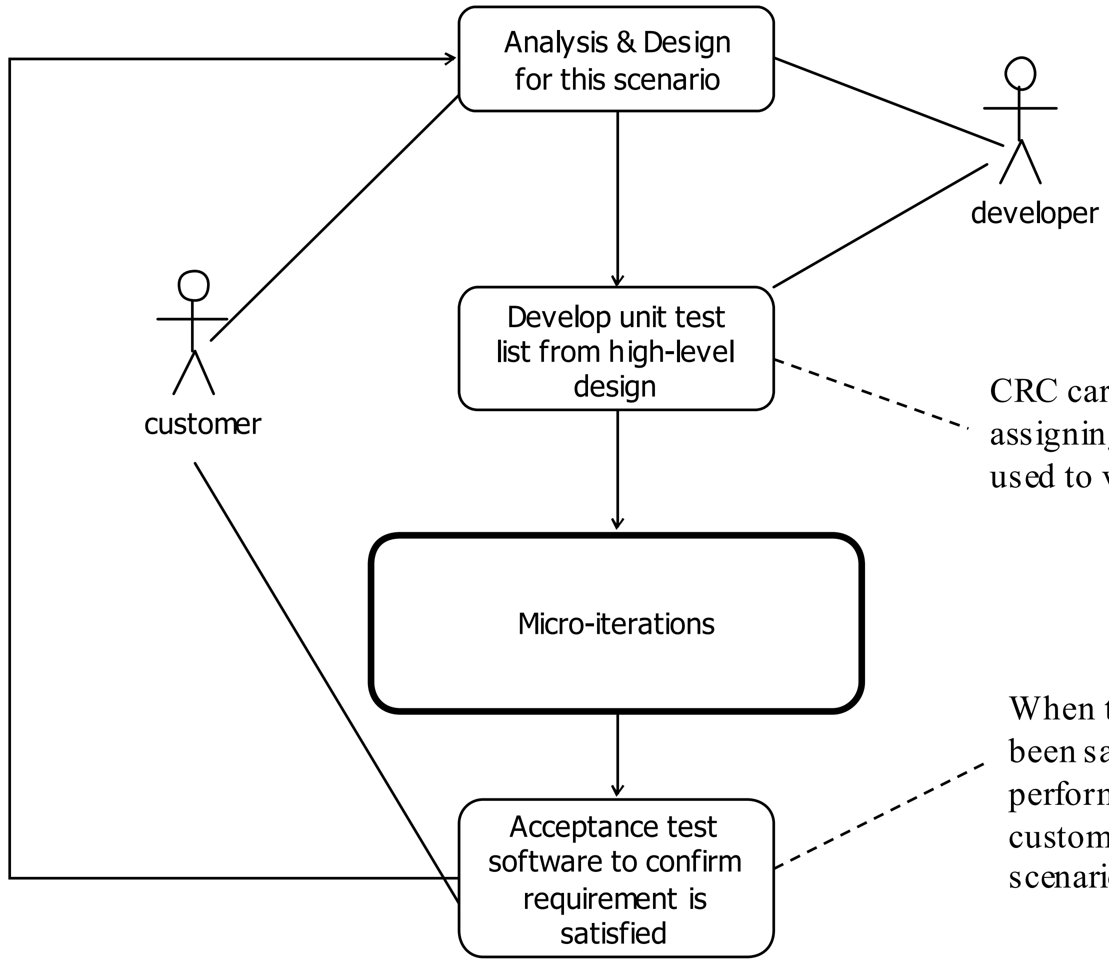
Iterations


Nominate test scenarios for iteration
developer
customer
Estimate time required to satisfy scenarios
Choose scenarios that will be tackled in this iteration
Mini-iterations
Progress to next iteration

Breaks down scenario's implementation into development tasks and estimates time required for each task. If a task cannot be estimated because it is new to the developer, do some exploratory development (sometimes called a "spike") to get a feel for what's involved.

Measure *project velocity* as the amount of development work (estimated days worth) that gets done in an average iteration, which tells the customer how much work can be scheduled for this iteration. They must choose scenarios whose estimates add up to what's achievable at the project velocity.

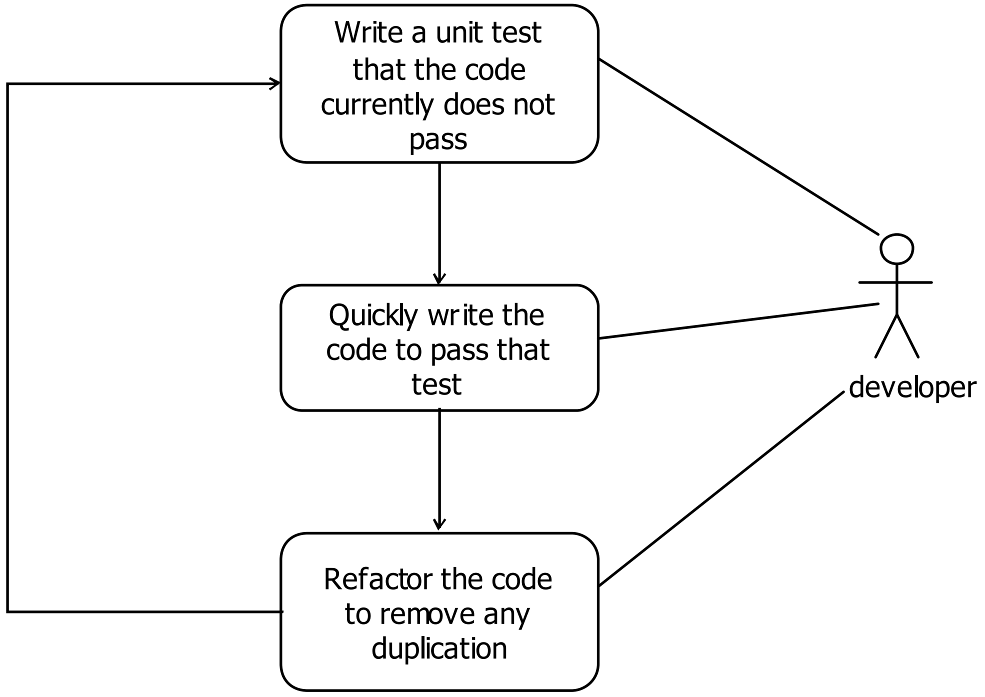
Mini-iterations


Analysis & Design for this scenario
developer
customer
Develop unit test list from high-level design
Micro-iterations
Acceptance test software to confirm requirement is satisfied

CRC cards, sequence diagrams and other ways of assigning responsibilities for test outcomes can be used to visualize unit tests required

When the developer believes that a test scenario has been satisfied, he informs the customer, who performs an acceptance test to confirm this. If the customer is satisfied, move on to the next system test scenario. If not, go back and fix the problem(s).

Micro-iterations


Write a unit test that the code currently does not pass
Quickly write the code to pass that test
Refactor the code to remove any duplication
developer

Introducing xUnit

- Original framework developed for Smalltalk by Kent Beck
- Versions for most programming languages (CppUnit for C++, JUnit for Java, Nunit for .NET etc)
- Simple concepts
  - Test fixture : a class that contains one ore more test methods
  - Test method : a method that executes a specific test
  - Test runner : an application that finds and executes test methods on test fixtures
  - Assertion : a Boolean expression that describes what must be true when some action has been executed
  - Expected Exception : the type of an Exception we expect to be thrown during execution of a test method
  - Setup : Code that is run before every test method is executed (eg, logging in as a particular user or initializing a singleton)
  - Teardown : Code that is run after every test method has finished (eg, deleting rows from a table that were inserted during the test)

parlez|uml

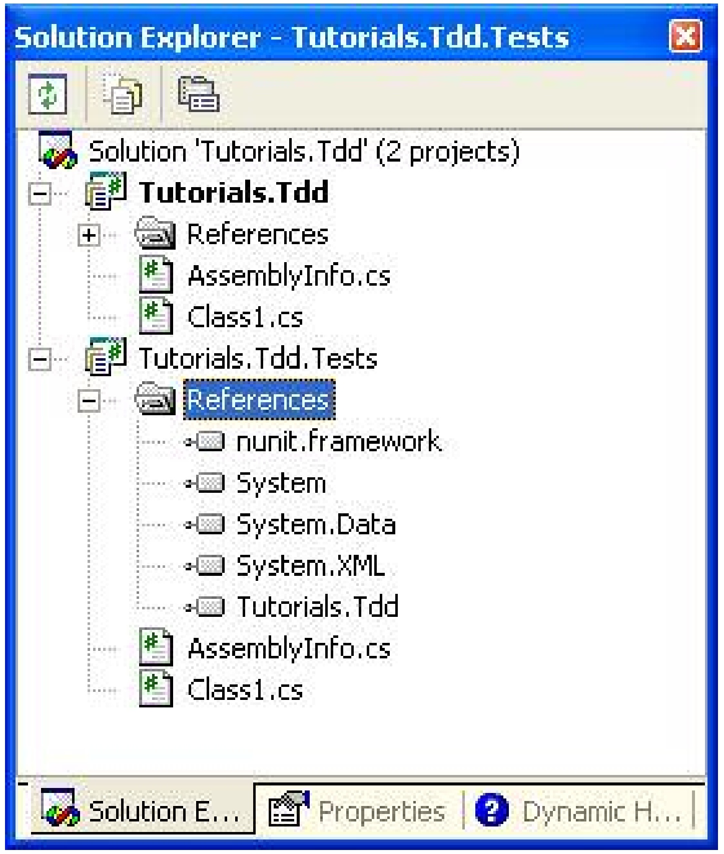
Getting Started with NUnit

1. Download and install NUnit
2. If you're using Visual Studio.NET, download and install the NUnit add-in

parlez|uml

Creating a Test Project

1. Create a library project with a name that makes it obvious which project this will be testing (eg, <project under test name>.Tests.dll)
2. In the test project, add a reference to the nunit.framework.dll library found in the \bin directory where Nunit is installed (eg, C:\Program Files\NUnit V2.0\bin\nunit.framework.dll)
3. Add a reference to the project under test



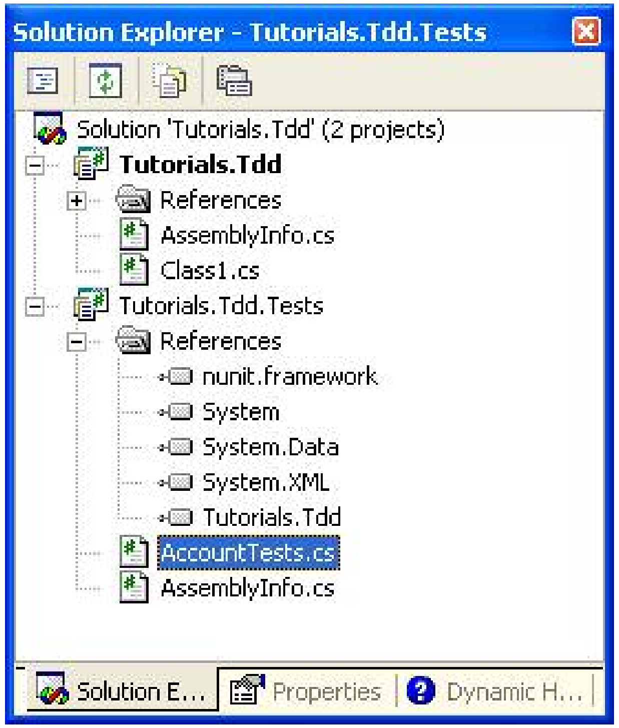
```
Solution Explorer - Tutorials.Tdd.Tests
Solution 'Tutorials.Tdd' (2 projects)
  Tutorials.Tdd
    References
    AssemblyInfo.cs
    Class1.cs
  Tutorials.Tdd.Tests
    References
      nunit.framework
      System
      System.Data
      System.XML
      Tutorials.Tdd
    AssemblyInfo.cs
    Class1.cs
```

parlez|uml

Creating a Test Fixture

1. Add a new class to the test project with a name that makes it obvious which class it will be testing (eg, <class under test name>Tests)
2. Import the NUnit.Framework namespace and add the custom attribute [TestFixture] to the class declaration

```
using System;
using NUnit.Framework;

namespace Tutorials.Tdd.Tests
{
		[TestFixture]
		public class AccountTests
		{

		}
}
```


Solution Explorer - Tutorials.Tdd.Tests

```
Solution 'Tutorials.Tdd' (2 projects)
  Tutorials.Tdd
    References
    AssemblyInfo.cs
    Class1.cs
  Tutorials.Tdd.Tests
    References
      nunit.framework
      System
      System.Data
      System.XML
      Tutorials.Tdd
    AccountTests.cs
    AssemblyInfo.cs
```

parlez|uml

Creating a Test Method

1. Declare a public method with a name that makes it obvious what the test case is (eg, <method under test><test scenario>
2. Add a [Test] attribute to the method declaration

```
[TestFixture]
public class AccountTests
{
            [Test]
            public void WithdrawWithSufficientFunds()
            {
            }
}
```

Write the Test Assertions First

1. Don't write the code needed to execute the test scenario until you've written the assertions the code needs to satisfy first
2. If there's more than one assertion, consider working one assertion at a time

```
[Test]
public void WithdrawWithSufficientFunds()
{
                    Assertion.AssertEquals(oldBalance - amount, account.Balance);
}
```

expected value

actual value

parlez|uml

Write the Test Body

1. Write the test code needed to execute the scenario

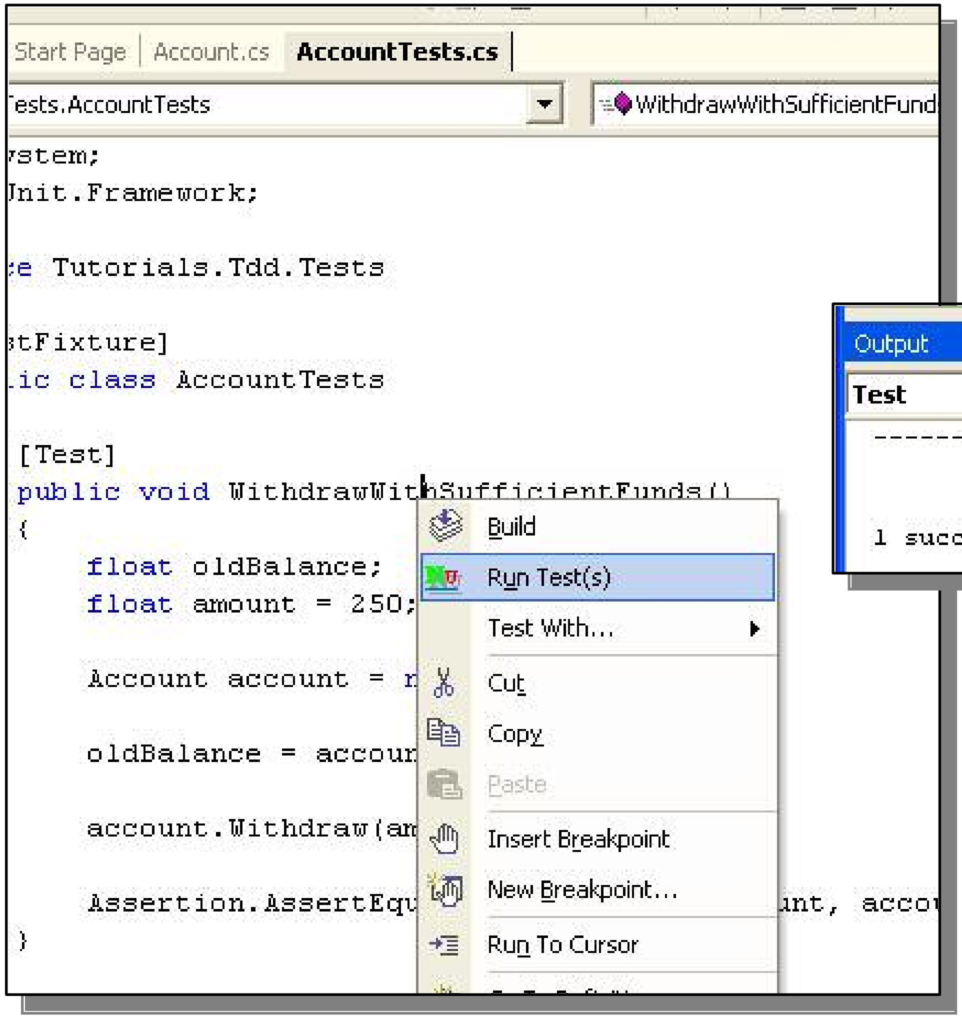
```
[Test]
public void WithdrawWithSufficientFunds()
{
            float oldBalance;
            float amount = 250;

            Account account = new Account();

            oldBalance = account.Balance;

            account.Withdraw(amount);

            Assertion.AssertEquals(oldBalance - amount, account.Balance);
}
```

Write the Code To Pass The Test

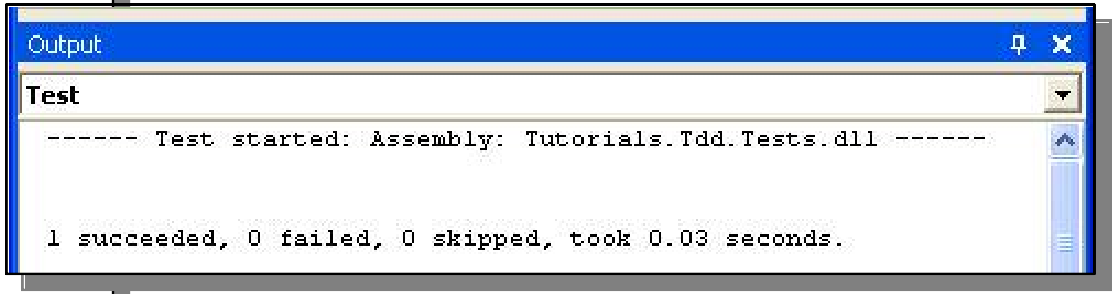
…and *only* the code needed to pass the test

```
public class Account
{
        private float balance = 0;

        public void Withdraw(float amount)
        {
                balance = balance - amount;
        }

        public float Balance
        {
                get
                {
                        return balance;
                }
        }
}
```

Running the test with the NUnit Add-in



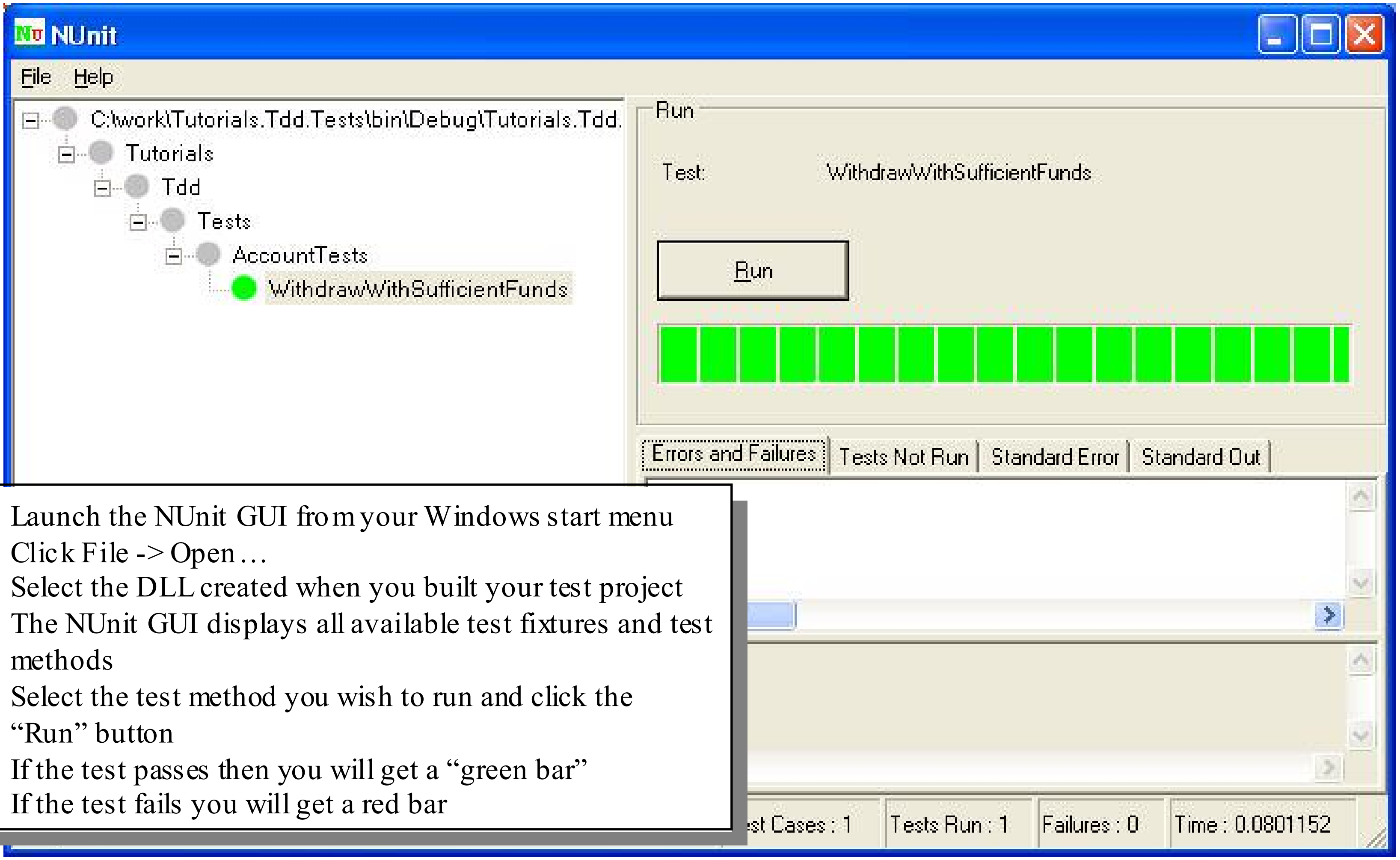
```
stem;
Jnit.Framework;

:e Tutorials.Tdd.Tests

:tFixture]
.ic class AccountTests

 [Test]
 public void WithdrawWithSufficientFunds()
 {
     float oldBalance;
     float amount = 250;

     Account account = n

     oldBalance = accour

     account.Withdraw(am

     Assertion.AssertEqu                    unt, acco
 }
```



```
------ Test started: Assembly: Tutorials.Tdd.Tests.dll ------


1 succeeded, 0 failed, 0 skipped, took 0.03 seconds.
```

1. Right click on the test method in the code window
2. Select “Run Test(s)”
3. The test results will be displayed in Output window

Running the test with the NUnit GUI



1. Launch the NUnit GUI from your Windows start menu
2. Click File -> Open…
3. Select the DLL created when you built your test project
4. The NUnit GUI displays all available test fixtures and test methods
5. Select the test method you wish to run and click the "Run" button
6. If the test passes then you will get a "green bar"
7. If the test fails you will get a red bar

Move On To The Next Test

```
[Test]
public void DepositAmountGreaterThanZero()
{
        float oldBalance;
        float amount = 250;

        Account account = new Account();

        oldBalance = account.Balance;

        account.Deposit(amount);
        Assertion.AssertEquals(oldBalance + amount, account.Balance);
}
```

parlez|uml

Get That Test To Pass Quickly

```
public class Account
{
		private float balance = 0;

		public void Withdraw(float amount)
		{
				balance = balance - amount;
		}

		public void Deposit(float amount)
		{
				balance = balance + amount;
		}

		public float Balance
		{
				get
				{
						return balance;
				}
		}
}
```

Remove Any Duplication

…from the code under test *and* from the test code (remember that the test code needs to be as easy to change as the "model" code its testing)

code under test

```
[TestFixture]
public class AccountTests
{
        private float oldBalance;
        private float amount = 250;
        private Account account;

        [SetUp]
        public void SetUp()
        {
                account = new Account();
                oldBalance = account.Balance;
        }
        [Test]
        public void WithdrawWithSufficientFunds()
        {
                account.Withdraw(amount);
                Assertion.AssertEquals(oldBalance - amount, account.Balance);
        }
        [Test]
        public void DepositAmountGreaterThanZero()
        {
                account.Deposit(amount);
                Assertion.AssertEquals(oldBalance + amount, account.Balance);
        }
}
```

parlez|uml

And So On…

TDD – The Golden Rule

Never write model code unless you have a test that requires it

TDD Best Practices

- Only work on one test at a time
  - Don't write a whole bunch of tests and then try to pass them all in one go
- Write test assertions self-explanatory so other developers can readily see what the code is supposed to be doing just by reading the tests
  - balance == oldBalance – amount rather than balance == 50
- Make the structure of your test projects follow the structure of the projects they are testing so developers can easily navigate from the tests to the code they are testing and vice-versa
- Keep the tests small
  - More than 15 minutes on the same test is probably too long. You will lose focus.
  - If a test looks complicated, think of ways to break it down. If a test involves several methods or classes that you'll have to write to pass the test then either:
    - Write tests for those first and work your way up to the bigger test (bottom-up TDD)
    - *Fake them* to pass this test and then work your way down to implement each of them one test at a time (top-down TDD)
- If you're working in a team, integrate your code changes every few unit tests (say, every hour or two)* to make sure there aren't any nasty surprises waiting for you in someone else's changes
- If you're working alone, end every day with one broken test to help you quickly refocus the next morning
- If you're working in a team, never ever leave with broken tests!
- Automate acceptance tests, too (this is a challenge, I must warn you) so that you can be confident when you're done and ready to release it to the customer, and also so that you know everything that's been accepted will stay like that…
- Take regular breaks and work reasonable hours. TDD is supposed to set a pace that can be sustained indefinitely.

TDD & Continuous Integration

- Before integrating your code changes tell all team members that you are about to. Until you have a successful build on the integration server, nobody else should consider integrating their changes
- Get any changed files that have been checked in to your team's repository (eg, VSS, CVS) since you last checked out the code and merge those with your changes
- Run *all* of the tests
- Do <u>not</u> integrate if any of the tests fail
- If all tests pass then check in your code changes
- Do a build with the latest code in the repository on the integration server and run all of the tests against that build
- If all the tests pass then you have a successful build and other developers can start to integrate
- If any tests fail then you have a broken build, which means nobody else can start integrating – your team's priority is now to find out why the test(s) fail and fix the problem ASAP