Tutorial 2

In the last tutorial we developed a very simple 3D application that drew animated objects on the screen. The application was written in C#, and made use of OpenGL via the Tao Framework, and the tutorial was both an introduction to using the Tao Framework and Visual Studio 2005 Express, and a tutorial in Object Oriented design and refactoring.

In this tutorial we will start to write a very simple physics engine; to allow an object to move under force, but with no rotations (yet), so that following this tutorial we will get our sphere to fall realistically under gravity.

We will go about developing the physics engine using Test-Driven Development (TDD). TDD is one of the practices used in agile software development processes such as eXtreme Programming (XP), and can produce better-designed code with less bugs.

The aim of this tutorial is to investigate using TDD to develop a working physics library, to be consumed by the simple 3D application in the next tutorial. It's quite difficult to give a picture of doing TDD in a small example, so this real-world example should show the thought processes and rhythm of doing TDD, with no big upfront design.

This tutorial stands alone from the previous tutorial, and doesn't depend on any of the code from tutorial 1, but I will refer to some points from that tutorial as we're going along.

Test-Driven Development and Unit Testing

The idea of Test-Driven Development is to write unit tests for all code in the system, but in such a way that the feedback from writing the tests and the code at each point allows the code to be designed as it is written; rather than have a big upfront design, each test is implemented and then any duplication in the code is refactored so that the code is always being designed to match the requirements and the knowledge of the system.

Having a set of unit tests around the code means that there is no *fear* of the code base; major refactorings to improve the design or performance should be a fairly safe activity, allowing the code can be continually improved. Code designed to accommodate unit testing (not necessarily test-driven) usually ends up better designed, as it ends up being loosely coupled; an example of how the code is better designed is that we couldn't write calls to pop up Message Boxes in the middle of our code as it would halt the tests, but would call on an interface to allow us to test that the code was making appropriate calls on the interface (via a mock object implementing that interface). This decoupling makes the code easier to maintain, as it has the side-effect of allowing the messages to be easily logged, or suppressed that wouldn't be possible it the code wasn't written to be tested.

Of course, the tests are only any use if they have a high coverage, and it takes some practice to find the correct level of testing. Unit testing is sometimes called Grey Box Testing, because you have knowledge of the internal workings of the class, and want to

make sure that you're testing that the various code paths in the system, as in White Box testing, but you're coding using the class's public interface, as in Black Box testing.

NUnit can be used to test interactions across more than one class. Many unit testing purists seem to say that each class should be tested independently, using mock objects to test for interactions with other objects. This is fine, and should be done, but they seem concerned that people want to use unit testing tools to test for actual interactions between constructed classes. I think that this is fine too; we don't ship individual separate objects to the customer to glue together, and it's OK to use NUnit to test at a higher level, similar to testing using a scripting interface.

That's the basics of unit testing. Test Driven Development makes use of unit testing, but the difference is, is that the actual design of the system develops through the process of testing.

The TDD process is typically described as:

Red bar – Green bar – Refactor

But of course there is more to it than that, as this doesn't include the step about thinking what tests should be written in which order. This is well described in this blog post: http://www.jamesshore.com/Blog/Microsoft-Gets-TDD-Completely-Wrong.html

The most difficult part is to get a feel for what test to write at a given time – you should know approximately what you want the system to do (or how else would you go about coding it?). Agile processes say that you would probably want to code the most important or most risky part of the project first, and the TDD process is to start with the simplest test cases you can think of and moving on from there. The basic idea of TDD is to keep the code simple, so that it can be easily modified and refactored to support the next piece of functionality; and the unit tests make continual refactoring feasible.

The advantages of TDD are the *rhythm* that you get when developing, and all your concentration is either focused on the next test, or refactoring the current test. It works well with pair programming, as you don't get distracted and keep a fast pace throughout the day. Enough background; there are many more resources available that explain the theory better than this, but this tutorial should give some insight into the actual procedure.

The functionality we require is to model a rigid body in 3D moving under a force. We'll start off with the simplest case, being only one dimension, stating that a body at rest stays at rest, and will build up from there.

Writing Tests in NUnit

Let's get started by writing tests to run in NUnit. NUnit is a .NET port of a collection of tools that include JUnit (for Java). To write tests we write test fixture classes that contain test methods that are ran by NUnit. NUnit works by using reflection to look for attributes that decorate classes and methods that describe them as being `TestFixtures` or `Tests`.

This tutorial was written against NUnit 2.2.8 and NUnit 2.4 beta 1. Download and install NUnit – you can run the .msi to install NUnit to your Program Files folder, or you can download the zip and use it locally.

Now, open up the Taumuon.Jabuka solution developed in the last tutorial and add a new C# Class Library, named Taumuon.Jabuka.Physics.Test. If you're following this tutorial standalone, you can just add a project to a new solution called Taumuon.Jabuka. Set the properties on this class to Set Warnings As Errors, and Build XML Documentation, as shown in the last tutorial.

Add a reference to the NUnit.Framework.dll assembly from wherever you installed it to. Rename Class1.cs to TestSphereNewtonLaws.cs, and this will also change the name of the class in the file.

We'll go ahead and decorate our class with a `TestFixtureAttribute`, so that NUnit knows that this class contains tests, and that it should instantiate an instance of this class. To do this, we'll also add a `using` statement for the `NUnit.Framework` namespace:

```
using System;

using NUnit.Framework;

namespace Taumuon.Jabuka.Physics.Test
{
    /// <summary>
    /// Tests that a sphere moves according to Newton's
    /// laws of motion.
    /// </summary>
    [TestFixture]
    public class TestSphereNewtonLaws
    {
    }
}
```

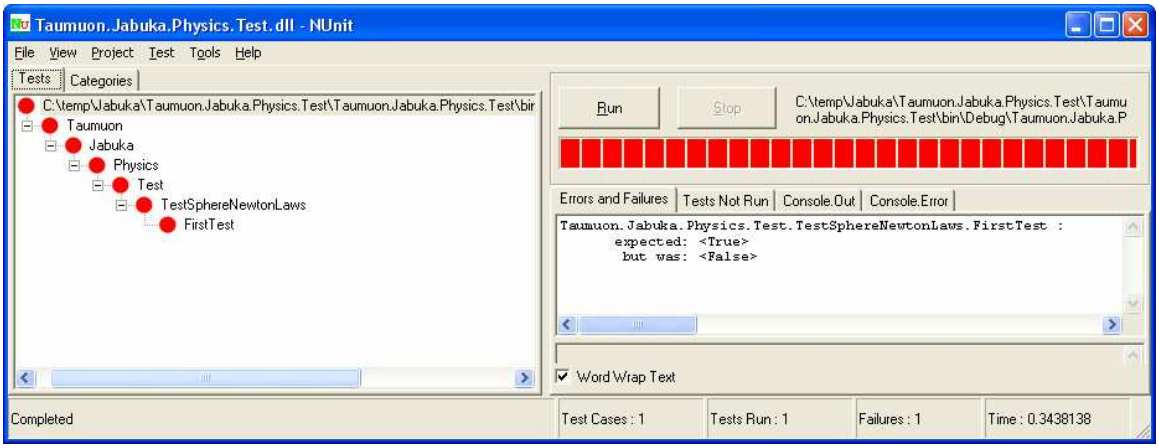
Build the project. Now we want to run NUnit so that we can run our tests. Run the NUnit GUI from wherever it was installed to. In the GUI, select File | Open… and navigate to where our Taumuon.Jabuka.Physics.Test.dll assembly was just built. You'll see that the tree view is broken down to the parts of our namespace separated by full stops. There's obviously nothing interesting to see as we haven't got any tests.

You can keep NUnit open while developing the assembly – NUnit detects when the assembly has been rebuilt and loads the latest version. It can get tedious to reopen it if you're too hasty with the close icon though, so in the next section we'll see how to get NUnit to launch from clicking the Run button from within Visual Studio Express.

Add a method called `FirstTest()` and decorate it with the `[Test]` attribute. This indicates that this is a method that NUnit will call during the running of the tests:

```
/// <summary>
/// First Test.
/// </summary>
[Test]
public void FirstTest()
{
    Assert.IsTrue(false);
}
```

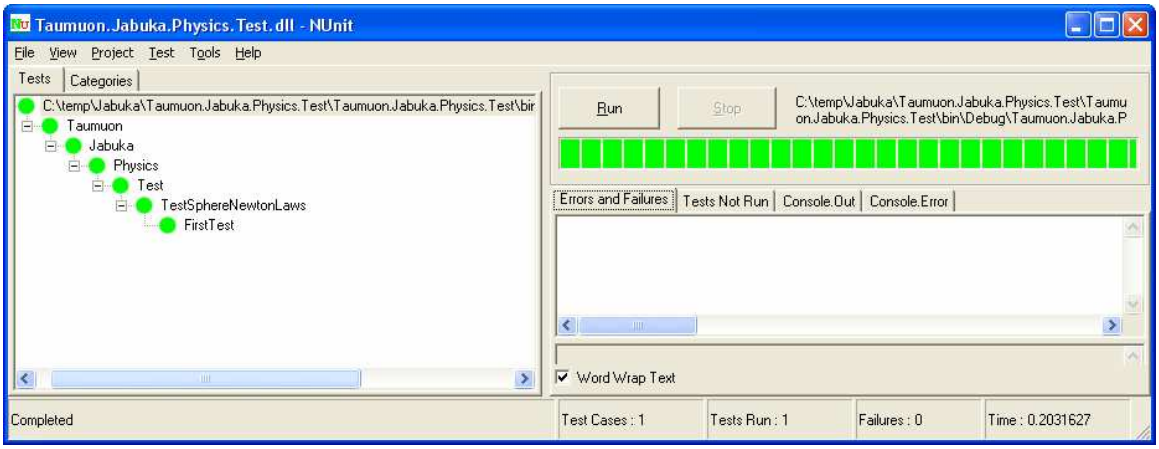
The `Assert` class has lots of static methods that can be used to check the condition of the program. There are various methods, with various overloads, to check the program state. Obviously in this case, it's just checking that true is false, so we get a red bar.



Now change the contents of the method to the following:

```
Assert.IsTrue(true);
```

If you run NUnit it will now show that we have a successfully running test, albeit a totally useless one.



Launching NUnit from Visual Studio Express

Now we'll see how we can launch NUnit from Visual Studio 2005 Express. This is important for debugging, as the Express version of Visual Studio 2005 doesn't let you

attach the debugger to a separate process so this is the only way to debug your tests (well, assuming that you're not running your tests by launching NUnit from its source files built with Express).

These instructions were found on a post at
http://thespoke.net/forums/923583/ShowPost.aspx

Close the solution, and open up the project file for our project, Taumuon.Jabuka.Physics.Test.csproj in a text editor. In the project group containing the following element:

```
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' ==
'Debug|AnyCPU' ">
```

Insert the following:

```
    <StartAction>Program</StartAction>
    <StartProgram>PathToExecutable\nunit-gui.exe</StartProgram>
    <StartArguments>Taumuon.Jabuka.Physics.Test.dll</StartArguments>
```

and insert the same into the `PropertyGroup` for the `Release` configuration.

Now reopen Visual Studio, set the Taumuon.Jabuka.Physics.Test project as the active project and run the solution.

Note that if you're using the full and not the express version of Visual Studio 2005, you can do this by opening up the project's properties, and in the Debug tab select Start External Program: and navigate to the NUnit executable, and set Taumuon.Jabuka.Physics.Test.dll as the Command Line Arguments:

An Object at Rest Stays at Rest

Newton's first law of motion is given on http://en.wikipedia.org as:
"*Objects in motion tend to stay in motion, and objects at rest tend to stay at rest unless an outside force acts upon them.*"

And it goes on to say that this comprises two parts; that an object at rest stays at rest, and that an object travels at a constant velocity until a force acts on it.

We'll be checking first that the location and velocity remain unchanged for a sphere with zero initial location and velocity, and the next test following that will be to check that the location changes but that the velocity remains constant for a sphere with a non-zero initial velocity.

The location and velocity of our sphere are measured relative to some world frame of reference. All measurements will be assumed to be in S.I. units, we can change this later on if needs be (test-driven development means that we don't have to second guess our requirements too easily, but we can rely on having a comprehensive test suite to safely make changes).

Let's go ahead and write a test. Thinking of what tests to write in which order is probably the hardest part of TDD, but it's not really that hard if you know, at least vaguely, what your software should be doing. We'll be testing that a sphere moves under forces according to Newton's laws of motion, and the first part of the first law is the simplest case.

The route that we'll take is to write tests, and functionality for Newton's First and Second laws, in one dimension only. Then we'll expand on this into three dimensions.

Let's go and property implement our first test. Add a method called `TestRemainsAtRest()`, decorated with the `[Test]` attribute. In our test code we'll construct a new `Sphere` object, with the initial location and initial velocity as constructor parameters:

```
Sphere sphere = new Sphere(0.0, 0.0);
```

Wait a second you might be thinking – intellisense doesn't show us anything for the `Sphere` class, but that's the `Sphere` class doesn't even *exist* yet. If you're following along from tutorial 1, don't get confused with the `Sphere` class in the MainApp – this sphere is (or rather will be) a totally different one. Note that from now on when I refer to the `Sphere` class, I'm referring to the one that we're going to create, rather than the one in the MainApp, unless I say otherwise.

This is test-driven development. We're coding our test to demonstrate the intent of the code, before writing the actual implementation.

As an aside, note that this is our first design decision; to have a separate `Sphere` in the physics project. This should hopefully be obvious, we don't want our Physics project to be dependent on the ThreeD project, the dependency should be the other way around – we may want to use our Physics code in a DirectX engine, or to produce input files for POV-ray (http://www.povray.org). Also, the physics and 3D spheres have different responsibilities. It will probably turn out that we won't have a Physics sphere in the end, but just some form of general physics object containing an integrator, as the physics code only depends on the mass and the motion of inertia. Of course, the collision detection does depend on the shape, but we won't get ahead of ourselves. Going through the process of designing one thing at a time via the tests will get us to the solution, and we don't have to second guess ourselves now.

Now, remember from the previous tutorial that the behaviour we desire is for the `Sphere` to have a method `Update()`, that updates the `Sphere`'s physical parameters by a given timestep. We'll do that here with a fairly large timestep of 10 seconds:

```
sphere.Update(10.0);
```

And finally, we'll check that after ten seconds the `Sphere`'s location and velocity remain unchanged. We use the `Assert.AreEqual()` method to do this. In this case, if the actual

and expected values passed into `AreEqual()` are not equal, then an `AssertionException` is thrown by the test, and caught internally in NUnit from where it called our test method.

```
        /// <summary>
        /// Tests that a sphere initially at rest remains at rest.
        /// </summary>
        [Test]
        public void TestRemainsAtRest()
        {
            Sphere sphere = new Sphere(0.0, 0.0);
            sphere.Update(10.0);
            Assert.AreEqual(0.0, sphere.Location, "Location not"
                            + " as expected");
            Assert.AreEqual(0.0, sphere.Velocity, "Velocity not"
                            + " as expected");
        }
```

Unfortunately, we can't demonstrate that just yet, as the project won't even compile! We'll go ahead and create the `Sphere` class now.

Adding the Sphere

We could add the `Sphere` class directly into the `Test` project, but it's better to separate the tests out into a separate assembly. An application that consumes the physics engine shouldn't have a dependency on NUnit.

That being said, create a new Class Library Project (Add | New Project) to the solution called Taumuon.Jabuka.Physics. Delete class1.cs from the project.

Add a new class to the Physics project, called `Sphere`. To get the code to compile, add two public properties to obtain the location and velocity, with constructor parameters to set these variables:

```
using System;

namespace Taumuon.Jabuka.Physics
{
    /// <summary>
    /// A physical sphere.
    /// </summary>
    public class Sphere
    {
        #region Object Lifetime

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="location">Initial location.</param>
        /// <param name="velocity">Initial velocity.</param>
        public Sphere(double location, double velocity)
        {
        }
```

```
        #endregion Object Lifetime

        #region Public Accessors

        /// <summary>
        /// Retrieves the current location.
        /// </summary>
        public double Location
        {
            get
            {
                return -1.0;
            }
        }

        /// <summary>
        /// Retrieves the current velocity.
        /// </summary>
        public double Velocity
        {
            get
            {
                return -1.0;
            }
        }

        #endregion Public Accessors
    }
}
```

Add a project reference to the Physics project from the Test project, and add a `using` statement at the top of the `Test` class for the `Taumuon.Jabuka.Physics` namespace.

The project still won't compile, so let's go ahead and add our `Update` method to the `Sphere`:

```
        #region Public Methods

        /// <summary>
        /// Allows the sphere to update its physical state.
        /// </summary>
        /// <param name="deltaTime">Time increment, in seconds.</param>
        public void Update(double deltaTime)
        {
        }

        #endregion Public Methods
```

Build the project, run the test and you'll get a red bar. You may wonder why I didn't go ahead and add the member variables, set these in the constructor and retrieve these from the properties. Well, they're not necessary for this test.

Also, you may be wondering why I'm returning a value of -1.0 for the location and velocity. Well, I chose a value other than zero to get the test to fail. The idea of the *Red Bar* stage is to implement the test to fail, but not just by putting in an `Assert.Fail()`, but in a way such that the test will pass once the non-test code is implemented correctly, *without changing the test*. The idea of the red bar is to make sure that we're testing the right thing, to avoid occasions where we may, for example, have forgotten to write any assertions, or maybe test the value of the wrong variable inadvertently leading to a passing test (a false positive).

Now that we've proved that our test is working correctly, let's change the `Sphere` code to pass the test. Change the `Location` and `Velocity` properties to return zero, and test passes.

```
/// <summary>
/// Retrieves the current location.
/// </summary>
public double Location
{
    get
    {
        return 0;
    }
}

/// <summary>
/// Retrieves the current velocity.
/// </summary>
public double Velocity
{
    get
    {
        return 0;
    }
}
```

Obviously, currently the sphere will *always* be at rest, but we don't implement the logic that is obviously soon to be in the class, we keep things simple and approach this one test at a time.

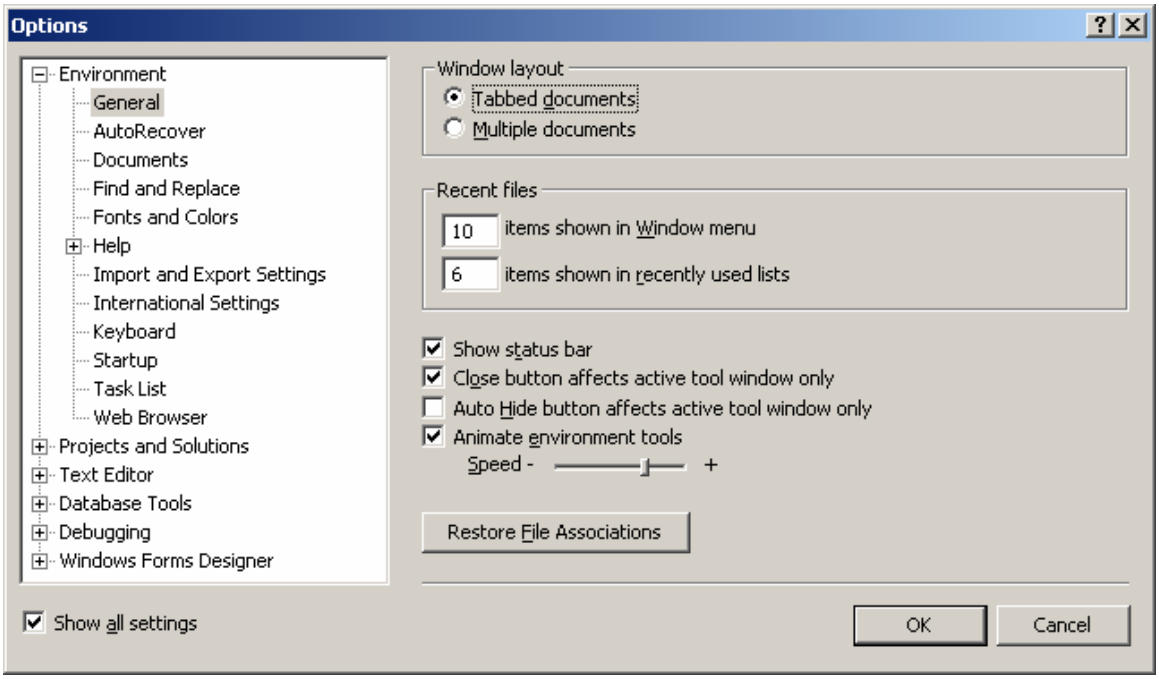
Note that the `Sphere` class could be called `Particle` throughout this tutorial, as we're not constructing it with a size, and there are no rotations (all forces act through the centre of gravity). However, we will want to add rotations and collision detection in at some later point.

Setting the Project Settings

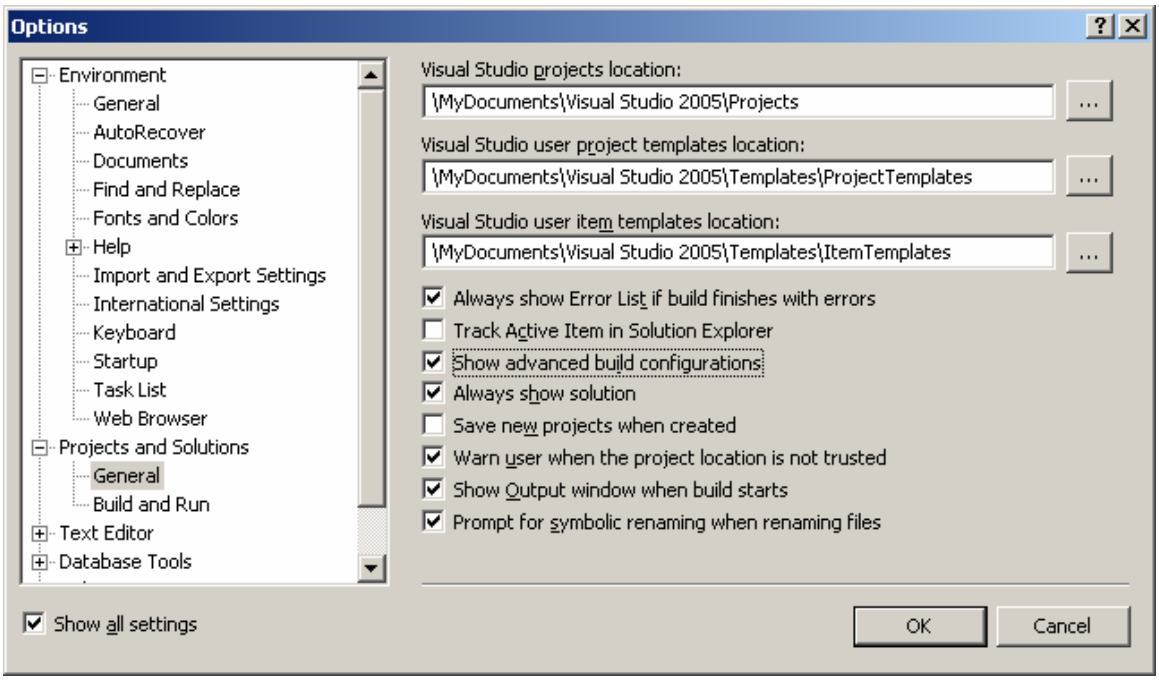
We quickly created our Taumuon.Jabuka.Physics project to get the test to pass, and now we'll set some properties for the project, but first we want to make some changes so that we can set different properties in Visual Studio 2005 Express, depending on whether we're building in debug or release build configurations (note that this step is unnecessary in the full version of Visual Studio 2005).

Showing all project configurations

Go to Tools | Options and select the Show all settings checkbox in the lower left of the dialog.



Now, check the Projects and Solutions | General | Show Advanced Build Configurations checkbox

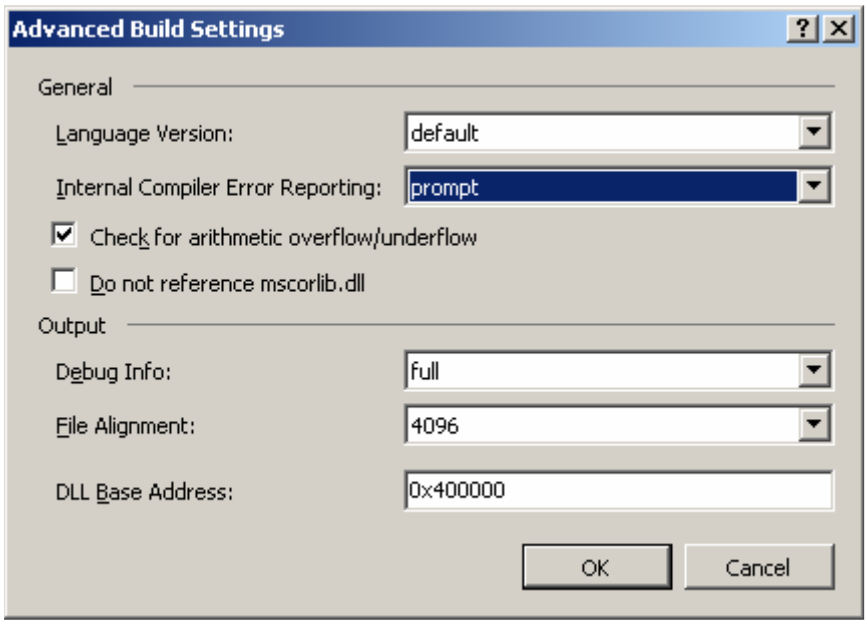


The project properties

As in Tutorial 1, we want to set the project's Build properties to Treat Warnings as Errors, and to Build XML Documentation. This time though, on the Build tab of the Project Properties make sure that the Configuration: drop down is set to All Configurations.

Now, we're going to make a couple of configuration specific changes. Change the Configuration: drop down to Debug.

Click the Advance button on the Build settings tab. On the dialog that appears, check the Check for arithmetic overflow/underflow checkbox. This throws an exception if a value overflows the data type that it's in. We only set this for Debug builds for performance reasons – if we have a comprehensive test harness that we are confident in, that we run for both debug and release builds, then we expect that this setting will catch all occurrences of arithmetic under/overflows in debug mode.



Running our unit tests against debug/release builds is a good idea, as it catches instances where we may have, for example, performed floating point comparisons that have been changed due to optimisation (in release floating point calculations may be performed on the processor's higher precision registers, read http://blogs.msdn.com/davidnotario/archive/2005/08/08/449092.aspx for a discussion on this).

Let's go ahead and add another test.

An Object's Velocity Remains Constant for No Applied Force

Let's add a second test to our test class. This method will create a Sphere located at zero position, but with a non-zero initial velocity. It's not subject to any external forces, so we expect our velocity to remain constant.

This time, however, we expect our location to be updated according to the velocity.

s = vt.

Add the following method, compile the project and check that the new test is recognised in NUnit. Run it and as expected we get a red bar:

```
/// <summary>
/// Tests that a sphere with a non-zero initial velocity
/// moves with a constant velocity.
/// </summary>
[Test]
public void TestNoForceVelocityRemainsConstant()
{
    Sphere sphere = new Sphere(0.0, 0.5);
    sphere.Update(10.0);
    Assert.AreEqual(5.0, sphere.Location, "Location not"
        + " as expected");
    Assert.AreEqual(0.5, sphere.Velocity, "Velocity not"
                    + " as expected");
}
```

Now, obviously, we can't hardcode the `Location` property to return 5.0, as that will break the first test. We'll have to actually insert some logic to update the position.

In the `Sphere` class, add member variables to hold our location and velocity, and set these in the constructor:

```
#region Member Variables

private double location = 0.0;
private double velocity = 0.0;

#endregion Member Variables

#region Object Lifetime

/// <summary>
/// Constructor
/// </summary>
/// <param name="location">Initial location.</param>
/// <param name="velocity">Initial velocity.</param>
public Sphere(double location, double velocity)
{
    this.location = location;
    this.velocity = velocity;
}
```

```
        #endregion Object Lifetime
```

Change the `Location` and `Velocity` properties to return the value of the member variables rather than hard-coded values.

```
        /// <summary>
        /// Retrieves the current location.
        /// </summary>
        public double Location
        {
            get
            {
                return this.location;
            }
        }

        /// <summary>
        /// Retrieves the current velocity.
        /// </summary>
        public double Velocity
        {
            get
            {
                return this.velocity;
            }
        }
```

Rebuild and retest. We haven't yet got our new test working, but it gives us confidence that we haven't broken our first test.

It's worth noting at this point, that the second test is failing because the location isn't as expected. If the location and velocity assertions were in separate tests then the velocity test would pass – the velocity remains at the initial value.

This may be an indication that we're testing too many things in this test; there are some people in the unit testing community who advocate having one `Assert()` call per test. However, I think that it's OK to have one logical assertion per test - many NUnit assertions can be used to test one atomic operation. Or even more, as long as if a failure happens it's easy to quickly track down the source of the operation. If you are interested in the debate though, there's a discussion of this here:

http://www.testdriven.com/modules/newbb/viewtopic.php?topic_id=363&forum=6

Having more than one assertion per test means that you should do the redbar/greenbar/refactor for every assertion present in the test. What we will do here, is to change the check for the velocity to be above the location, as the velocity calculation is simpler, and as the location depends on the velocity, the location will be wrong if the velocity isn't calculated correctly.

Now, implement the `Update()` method as:

```
/// <summary>
/// Allows the sphere to update its physical state.
/// </summary>
/// <param name="deltaTime">Time increment, in seconds.</param>
public void Update(double deltaTime)
{
    this.location += this.velocity * deltaTime;
}
```

And both tests pass. This only works as the acceleration is constant. We'll get onto non-constant integration when we discuss integration shortly. Note that we're comparing floating point values; checking that two doubles are equal will only work if the two values are exactly the same (which you have to be careful of, due to e.g. rounding errors, or optimisation changing the order of calculations or storing one of the values in a larger 80 bit register).

Notice that both tests create a `Sphere`, call `Update()` on it, and check the location and velocity, so we'll refactor out the common code.

Let's add a helper method to the class `TestSphereNewtonLaws`. This will call `Update()` on the sphere and check that the resulting location and velocity are as expected:

```
#region Helper Methods

private void UpdateSphereAndCheckState(Sphere sphere,
    double expectedVelocity, double expectedLocation)
{
    const double deltaTime = 10.0;
    sphere.Update(deltaTime);
    Assert.AreEqual(expectedVelocity, sphere.Velocity,
        "Velocity not as expected");
    Assert.AreEqual(expectedLocation, sphere.Location,
        "Location not as expected");
}

#endregion Helper Methods
```

Note that even though the time step is still hard-coded into the method, it's stored in a local constant. Now, our test methods are amended to call this helper:

```
/// <summary>
/// Tests that a sphere initially at rest remains at rest.
/// </summary>
[Test]
public void TestRemainsAtRest()
{
    Sphere sphere = new Sphere(0.0, 0.0);
    UpdateSphereAndCheckState(sphere, 0.0, 0.0);
}
```

```
        /// <summary>
        /// Tests that a sphere with a non-zero initial velocity
        /// moves with a constant velocity.
        /// </summary>
        [Test]
        public void TestNoForceVelocityRemainsConstant()
        {
            Sphere sphere = new Sphere(0.0, 0.5);
            UpdateSphereAndCheckState(sphere, 0.5, 5.0);
        }
```

Check that we get a green bar. After making changes to the test framework, it is sometimes worth checking that the tests still catch any bugs, by forcing the tested code to fail. In this case, we might want to get our `Location` and `Velocity` properties to return -1 to check that the tests fail as expected.

*Force = mass * acceleration*

Now, we'll test that an object under the influence of a force experiences an acceleration proportional to that force:

F=ma
Or
a = F/m

What we want to find are the object's location, and velocity, at a given time. We do this by finding the acceleration at each time step, and then by integrating to find the velocity and location.

We don't integrate the analytical equation. Instead, the velocity is integrated by breaking up the graph of acceleration into small time intervals, and adding the rectangles approximating the area together. The explicit Euler method is (from Numerical Methods for Physics, Second Edition, Alejandro L. Garcia, Prentice Hall, ISBN 0-13-906744-2):

$\mathbf{v}_{n+1} = \mathbf{v}_n + \tau\mathbf{a}_n$
$\mathbf{s}_{n+1} = \mathbf{s}_n + \tau\mathbf{v}_n$

where τ is the time step, $\mathbf{v}_n$ is the velocity at time t_n, and $\mathbf{s}$ is the displacement.

In pseudocode, the sequence of steps are:

- Calculate the acceleration
- Update the displacement from the current velocity
- Update the velocity from the acceleration

A superior form is given, named the Euler-Cromer method, that first calculates the velocity, then updates the displacement based on the newly calculated velocity:

$\mathbf{v}_{n+1} = \mathbf{v}_n + \tau\mathbf{a}_n$

$\mathbf{s}_{n+1} = \mathbf{s}_n + \tau\mathbf{v}_{n+1}$

This is the form that we're using. In code the steps are:

- Calculate the acceleration
- Update the velocity from the acceleration
- Update the displacement from the updated velocity

Search on http://www.gamedev.net for some discussions of the advantages of this method. Note that it's also named semi-implicit Euler, Newton-Störmer-Verlet or Symplectic Euler (e.g. this link http://www.gamedev.net/community/forums/topic.asp?topic_id=382993)

We've only got one object here, but note that it calculates its own state. We could combine the states into a larger vector to investigate getting better performance from parallelising with vector/SIMD processing, but that would make it less OO – we'll get to look at performance once we have a full app working that we can benchmark.

We'll now take a look at some actual figures, so that we can write a test. Now, velocity is integral of acceleration. For constant acceleration the integral of acceleration over time is simply:

$v(t) = v(0) + at$

Location is integral of velocity. For constant acceleration

$s(t) = s(0) + v(0)t + 0.5at^2$

So for initial conditions $s(0) = 0$, $v(0) = 0$.

$v(10) = 10 * a.$
$s(10) = 0.5 * a * 100 = 50 * a.$

Now, we want to apply an acceleration to our body, we do this by applying a force. We'll assume that the sphere has a default mass of 1kg, so that applying a force of 1N gives an acceleration of 1m/s. After ten seconds, our sphere should be at location 50m, and have a velocity of 10m/s.

Let's go and write the test for this. We won't be using the helper method that we just wrote yet; we'll write this test in isolation and then refactor out common code:

```
/// <summary>
/// Tests that a sphere subject to a constant applied
/// force updates its location and velocity.
/// </summary>
[Test]
public void TestForceVelocityAndLocationChange()
{
```

```
            Sphere sphere = new Sphere(0.0, 0.0);
            sphere.ApplyForce(1.0);
            sphere.Update(10.0);
            Assert.AreEqual(50.0, sphere.Location, "Location not"
                + " as expected");
            Assert.AreEqual(10.0, sphere.Velocity, "Velocity not"
                            + " as expected");
        }
```

Now, let's add the `ApplyForce()` method to our `Sphere` class to get the project to compile.

```
        /// <summary>
        /// Applies a force to the sphere.
        /// </summary>
        /// <param name="force">Force, in Newtons.</param>
        public void ApplyForce(double force)
        {
        }
```

Now, run the tests just to make sure that this test fails and that we haven't broken the previous tests. It's going to take some work to get this one to pass.

The `ApplyForce()` method needs to set some state that will be used in the `Update()` method. Let's store the force in a private member variable. Add a private member variable of type double to the Sphere class, `force`, and assign to it the value passed into `ApplyForce()`:

```
        #region Member Variables

        private double location = 0.0;
        private double velocity = 0.0;
        private double force = 0.0;

        #endregion Member Variables

        public void ApplyForce(double force)
        {
            this.force = force;
        }
```

Now, the semi-implicit Euler method says that we update the velocity, then location, as follows:

velocity += acceleration * dt
location += velocity *dt

So, change the `Update()` method to be:

```
        public void Update(double deltaTime)
        {
            const double mass = 1.0;
```

```
        double acceleration = this.force / mass;
        this.velocity += acceleration * deltaTime;
        this.location += this.velocity * deltaTime;
    }
```

Run the test, and see that the failing test is now failing due to the location being wrong (is at 100m rather than 50m) – we've got the velocity behaving correctly. So why is the test failing? Well, the integration approximates the area under the graph by a series of rectangles. The acceleration is constant so there is no error in finding the velocity by calculating the area under the graph for any time step. The velocity is non-constant so there will be an error in finding the area under the velocity-time graph to find the location, but the error will be smaller for smaller time steps (up to a point, where round-off error, the error due to the precision with which we can store values in our data types, becomes important, but we shouldn't be reaching that point in a game engine).

Adding in Multiple Time Steps

Let's change the test to repeatedly call `Update()` with multiple time steps. First off, we'll put the loop in, but with just 1 time step to check that we get the same result as previously.

```
    /// <summary>
    /// Tests that a sphere subject to a constant applied
    /// force updates its location and velocity.
    /// </summary>
    [Test]
    public void TestForceVelocityAndLocationChange()
    {
        Sphere sphere = new Sphere(0.0, 0.0);
        sphere.ApplyForce(1.0);
        const int numSteps = 1;
        for (int i = 1; i <= numSteps; ++i)
        {
            sphere.Update(10.0 / (double)numSteps);
        }
        Assert.AreEqual(10.0, sphere.Velocity, "Velocity not"
            + " as expected");
        Assert.AreEqual(50.0, sphere.Location, "Location not"
            + " as expected");
    }
```

Now, change the value of `numSteps` to be 2 and the final location is at 75m, much closer to the 50m we're expecting. Obviously 2 steps isn't a very good approximation, so let's change the number of steps to be 10. We're now at 55m, within 10% of the result which is better. But let's go and change the number of steps to be 100. Now the velocity test has failed, due to the small numeric differences compounded 100 times. We're within 14 decimal places of the correct result. This is pretty good, so we'll say that if we're within 10 decimal places then the test is good enough.

Let's change the `Assert()` methods to use the overload with the tolerance parameter. Note that this is an absolute tolerance. We may, in the course of refactoring the tests,

decide to not use the overload on the `AreEqual()` method, and add a method ourselves to check that values are within a certain percentage of each other, rather than within an absolute tolerance.

```
const double tolerance = 1e-10;
Assert.AreEqual(10.0, sphere.Velocity, tolerance,
    "Velocity not as expected");
Assert.AreEqual(50.0, sphere.Location, tolerance,
    "Location not as expected");
```

The velocity test now passes, and we can see that the location fails due to a 1% difference over the calculated value, with 100 steps for 10 seconds. This means that calculating 10 steps in 1 second would give us 99% accuracy. That's not too bad; for a real-time 3D game we're aiming to be running at at least 30 frames per second so we know that we'll be accurate within 1%. Stability is more of a concern, but we'll take a look at that much later.

Let's see what happens if we use 1000 steps. The result is accurate to within 0.1%. This is the value that we'll test against, so change the velocity assertion to check for a value of `50.0499999999`.

```
/// <summary>
/// Tests that a sphere subject to a constant applied
/// force updates its location and velocity.
/// </summary>
[Test]
public void TestForceVelocityAndLocationChange()
{
    Sphere sphere = new Sphere(0.0, 0.0);
    sphere.ApplyForce(1.0);
    const int numSteps = 1000;
    for (int i = 1; i <= numSteps; ++i)
    {
        sphere.Update(10.0 / (double)numSteps);
    }
    const double tolerance = 1e-10;
    Assert.AreEqual(10.0, sphere.Velocity, tolerance,
        "Velocity not as expected");
    Assert.AreEqual(50.0499999999, sphere.Location, tolerance,
        "Location not as expected");
}
```

Investigating the Results

We've just seen above that changing the step size gives smaller error. The Euler method is a first order method, so the error is of the second order. This is the local error – the error in the approximation of a single time step to the analytical value.

Time step of 1.

```
0       0       0
1       1       1
```

2	2	3
3	3	6
4	4	10
5	5	15
6	6	21
7	7	28
8	8	36
9	9	45
10	10	55

Time step of 0.1

0	0	0
0.1	0.1	0.01
0.2	0.2	0.03
0.3	0.3	0.06
0.4	0.4	0.1
…	…	…
9.7	9.7	47.53
9.8	9.8	48.51
9.9	9.9	49.5
10	10	50.5

For a time step of 1 second, for the first time step the value of the location should be 1*1/2 = 0.5, but the value is 1, giving an error of 0.5.

For a time step of 0.1 seconds, the first time step should be 0.1 * 0.1 / 2 = 0.005, but we get a value of 0.01, giving an error of 0.005. So it can be seen that dividing the time step by 10 gives an error 100 times less.

This is the value for the local error. The global error is the accumulated error from adding all of the sections together. After 10 seconds, the error is 5 for a time step of 1, and 0.5 for a time step of 0.1. So dividing the time step by 10 also divides the error by 10, the global error is of the same order as the time step.

This intuitively makes sense, the local error for a time step is proportional to the square of the time step, but with a smaller time step we have to make more time steps. So a time step a 10^{th} of the size gives a local error a hundredth of the size, but we have to take 10 times as many steps. As pointed out in Numerical Methods for Physics, the actual global error isn't that easy to calculate, as it depends on whether the local errors add or cancel each other out.

Refactoring the Tests

Let's refactor this method, and then beef up the number of tests that we have. Looking at this test, it follows the same pattern as `UpdateSphereAndCheckState()` – it creates a sphere with initial state, calls `Update()` on it, and then checks that the velocity and location are as expected. The differences are that it also applies a force, and calls update more than once. `UpdateSphereAndCheckState()` can be seen to be a subset of this

behaviour, so we'll change `UpdateSphereAndCheckState()` to add a force parameter, and a number of iterations:

```
private void UpdateSphereAndCheckState(Sphere sphere,
    double expectedVelocity, double expectedLocation,
    double appliedForce, int numberOfIterations)
{
    const double time = 10.0;
    sphere.ApplyForce(appliedForce);
    for (int i = 1; i <= numberOfIterations; ++i)
    {
        sphere.Update(time / numberOfIterations);
    }
    const double tolerance = 1e-10;
    Assert.AreEqual(expectedVelocity, sphere.Velocity,
        tolerance, "Velocity not as expected");
    Assert.AreEqual(expectedLocation, sphere.Location,
        tolerance, "Location not as expected");
}
```

Writing this test straight away shows the semantics of the `ApplyForce()` method – we expect this force to be applied for the life of the object, it's not applied for just one `Update()`. This may or not be what we want in the future, we may need to add a method to reset the forces on the object, but writing the tests makes us think about these sort of API design decisions whereas if we were just writing our component we may not notice things like this until much later.

Now change `TestRemainsAtRest` and `TestNoForceVelocityRemainsConstant` to call this method with an applied force of zero, and with 1 iteration. Change `TestForceVelocityAndLocationChange` to call this method:

```
[Test]
public void TestRemainsAtRest()
{
    Sphere sphere = new Sphere(0.0, 0.0);
    UpdateSphereAndCheckState(sphere, 0.0, 0.0, 0.0, 1);
}

[Test]
public void TestNoForceVelocityRemainsConstant()
{
    Sphere sphere = new Sphere(0.0, 0.5);
    UpdateSphereAndCheckState(sphere, 0.5, 5.0, 0.0, 1);
}

[Test]
public void TestForceVelocityAndLocationChange()
{
    Sphere sphere = new Sphere(0.0, 0.0);
    UpdateSphereAndCheckState(sphere, 10.0, 50.0499999999,
        1.0, 1000);
}
```

Testing the Mass

Now, we've got to the stage where the object moves in one dimension under force, but with a hardcoded mass of 1kg (as we're assuming SI units throughout). Let's add a test with double the mass, to see that it behaves correctly. We'll first write the test with the behaviour we want:

```
/// <summary>
/// Tests that a sphere subject to a constant applied
/// force updates its location and velocity, with a mass
/// of two kilograms.
/// </summary>
[Test]
public void TestForceMassTwoKilograms()
{
    Sphere sphere = new Sphere(0.0, 0.0, 2.0);
    UpdateSphereAndCheckState(sphere, 5.0, 25.0249999999,
        1.0, 1000);
}
```

We obtained the mass from:

$s(t) = s(0) + v(0)t + 0.5at^2$

and the acceleration, a, is inversely proportional to the mass; doubling the mass halves the acceleration and so halves the final displacement at a given time t.

You can see that the mass is set in a new constructor parameter. The question is whether it makes sense to have both overloads, whether there should be an overload to set the location and velocity without setting the mass. It's best to keep things simple – we don't know that that's a common scenario and one of the aims of Agile development is to keep things as simple as possible – what's the point of having code to test an overloaded constructor that no-one uses?

Removing the original constructor will cause quite a few breakages, so let's temporarily add an overloaded `Sphere` constructor to get the tests to compile:

```
/// <summary>
/// Constructor
/// </summary>
/// <param name="location">Initial location.</param>
/// <param name="velocity">Initial velocity.</param>
/// <param name="mass">The sphere's mass.</param>
public Sphere(double location, double velocity, double mass)
{
    this.location = location;
    this.velocity = velocity;
}
```

Obviously, the test won't pass as we're not using the mass in the calculations. Let's fix the sphere so that the mass is taken into account. Add a member variable to hold the

mass, and assign it in the constructor. Now, all we have to do is to remove the local mass variable from the `Update()` method.

```
#region Member Variables
...
private double mass = 1.0;

#endregion Member Variables
```

Now that all the tests pass, let's do some tidying up. We'll remove the original constructor overload and fix the other tests. Now this may be making you break out in a cold sweat, and if it doesn't then it should! Our existing tests are quite brittle at the moment, it's not too bad in this case, we just have 3 constructor calls to fix, but imagine if we had 50 tests, or 100! We'll add a factory method to create a `Sphere` into `TestSphereNewtonLaws` and will call that from the three existing tests:

```
/// <summary>
/// Creates a sphere with a default mass of 1kg.
/// </summary>
/// <param name="location">The initial location</param>
/// <param name="velocity">The initial velocity</param>
/// <returns> A new sphere</returns>
private Sphere CreateSphere(double location, double velocity)
{
    return new Sphere(location, velocity);
}
```

The original test methods will now construct a sphere as follows:

```
public void TestRemainsAtRest()
{
    Sphere sphere = CreateSphere(0.0, 0.0);
    UpdateSphereAndCheckState(sphere, 0.0, 0.0, 0.0, 1);
}
```

Now, remove the `Sphere` constructor that takes only two parameters. Obviously the test won't compile. Luckily, we've only got one place to make the change. Add a new parameter into the `CreateSphere()` method, to construct the sphere with a mass of 1.0 (this is the same implicit behaviour as before, we've just made the mass used in each of the tests set explicitly). Having the `CreateSphere` method has the added benefit that we can derive from our test fixture and override only this method to test that a class deriving from `Sphere` passes the tests.

Let's do a bit more tidying up – the last test that we added is still constructing a sphere. Add an overload `CreateSphere()`, taking the mass as an additional parameter, and call the `Sphere` constructor:

```
/// <summary>
/// Creates a sphere.
/// </summary>
/// <param name="location">The initial location</param>
/// <param name="velocity">The initial velocity</param>
```

```
/// <param name="mass">The sphere's mass</param>
/// <returns>A new sphere</returns>
private Sphere CreateSphere(double location,
    double velocity, double mass)
{
    return new Sphere(location, velocity, mass);
}
```

Call this factory method from the last added test. There's one further change we can make to make the tests less brittle. If we at some point add another constructor argument, then we can isolate the changes to just one method by getting the `CreateSphere()` overload that has two arguments, to call the one that takes three arguments:

```
private Sphere CreateSphere(double location, double velocity)
{
    return CreateSphere(location, velocity, 1.0);
}
```

Now we've got the mass included in the calculations, and have let the user specify the sphere's mass via the constructor. This gives the possibility of a divide by zero, as nothing prevents somebody from creating a sphere of zero mass. We would rather catch this when they're creating a sphere, rather than getting a value of infinity some time later when `Update()` is called. We'll get the constructor to throw an `ArgumentException` if the mass is zero. Let's add a test for this:

```
/// <summary>
/// Tests that a sphere with zero mass can't be created.
/// </summary>
[Test]
[ExpectedException(typeof(ArgumentException),
    "mass cannot be zero")]
public void TestCreateASphereWithZeroMass()
{
    Sphere sphere = CreateSphere(0.0, 0.0, 0.0);
}
```

Notice that we use the `ExpectedException` attribute to check we get the correct, and expected, exception, and that it is of the correct type and contains the correct message.

Let's get this test to pass, add the following into the start of the `Sphere` constructor:

```
if (mass == 0.0)
{
    throw new ArgumentException("mass cannot be zero");
}
```

Moving to the Third Dimension

So now we've got a sphere moving under force, but only in one dimension, which isn't that exciting for a 3D games or physics engine, so let's go and change this! Test-driven development and agile development doesn't rely on big upfront design, but lots of little whiteboard design sessions, so that designing happens as tests introduce knowledge about the system. And this is one of those little design sessions.

It's probably obvious though, that the location and velocity of the sphere will turn out to be a `Vector` class, with properties for the three dimensions. We'll introduce the vector into the tests, making sure that the code is never too far from compiling, and that the current tests are never broken for too long.

The first thing we'll do is to keep all the tests the same – there will be an assumption that all the values they're setting is still in one, say the x, dimension. However, we'll change the `CreateSphere()` helper method overload that actually creates a sphere to pass in a vector class:

```
        private Sphere CreateSphere(double location,
            double velocity, double mass)
        {
            Vector locationVector = new Vector(location, 0.0, 0.0);
            Vector velocityVector = new Vector(velocity, 0.0, 0.0);
            return new Sphere(locationVector, velocityVector, mass);
        }
```

Now, we want the tests to compile again as quickly as possible, so add a `Vector` class into the Physics project, taking 3 doubles:

```
    /// <summary>
    /// Represents a Vector in 3 dimensions.
    /// Can represent a point, or a vector from the origin.
    /// </summary>
    public class Vector
    {
        #region Object Lifetime

        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="x">Initial x location</param>
        /// <param name="y">Initial y location</param>
        /// <param name="z">Initial z location</param>
        public Vector(double x, double y, double z)
        {
        }

        #endregion Object Lifetime
    }
}
```

Now, let's change the sphere constructor to accept instances of the `Vector` class.

```
        public Sphere(Vector location, Vector velocity, double mass)
        {
            if (mass == 0.0)
            {
                throw new ArgumentException("mass cannot be zero");
            }
            this.mass = mass;
```

```
        }
```

The only test that fails is `TestNoForceVelocityRemainsConstant`, as this test relies on the sphere being constructed with non-zero velocity, it sets the initial velocity to 0.5. The fact that we've got a failing test is good, as we've got coverage for errors in the `Sphere` constructor, but it's only catching the error with the velocity.

Note that now we're passing in vectors, we want to check for the values being null, and have a test for that, but we won't add that quite yet, as we don't want to be doing too many things at once; we want to get to a green bar as quickly as possible without doing any tidying or adding any functionality. Making changes without seeing our green bar is *dangerous* as we're not sure whether we're introducing any bugs. It's handy to keep a list next to of tasks to-do as you go along.

Let's quickly introduce a test for a non-zero initial location. We'll base it on `TestForceMassTwoKilograms()` but change the initial location.

```
        /// <summary>
        /// Tests that a sphere subject to a constant applied
        /// force updates its location and velocity, with a mass
        /// of two kilograms, and non-zero initial velocity.
        /// </summary>
        [Test]
        public void TestMassTwoKilogramsNonZeroLocation()
        {
            Sphere sphere = CreateSphere(-5.0, 0.0, 2.0);
            UpdateSphereAndCheckState(sphere, 5.0, 20.0249999999,
                1.0, 1000);
        }
```

This test should pass when we fix the constructor.

We need to add a property of type double, `X`, to the `Vector`, and a double member holding this property which we set in the constructor. Don't add the properties or member variables for the `Y` and `Z` dimensions yet, as there not yet any clients making use of them.

```
    public class Vector
    {
        #region Private Member Variables
        private double x = 0.0;
        #endregion Private Member Variables

        #region Object Lifetime
        /// <summary>
        /// Constructor
        /// </summary>
        /// <param name="x">Initial x location</param>
        /// <param name="y">Initial y location</param>
        /// <param name="z">Initial z location</param>
        public Vector(double x, double y, double z)
        {
            this.x = x;
```

```
        }
        #endregion Object Lifetime

        #region Accessors
        /// <summary>
        /// Returns the vector's X dimension value.
        /// </summary>
        public double X
        {
            get
            {
                return x;
            }
        }
        #endregion Accessors
    }
```

And now we change the code in the `Sphere`'s constructor to make use of these values:

```
            ... // Argument validation

            this.location = location.X;
            this.velocity = velocity.X;
            this.mass = mass;
```

The tests are now creating a `Sphere`, passing in the location and velocity as `Vectors`.

Tidy up

Now that the tests pass again, let's do a bit of tidying up before we add any more tests. First, change the `CreateSphere` methods to take `Vectors` rather than doubles. Only set the `x` values in the constructors. Remember that when we changed our `Sphere` constructor to take vectors rather than double values, that we wanted to check for null values? Well, we'll add the tests for these now. Here's the test for the location, the velocity test is obviously similar:

```
        /// <summary>
        /// Tests that a sphere with zero location can't be created.
        /// </summary>
        [Test]
        [ExpectedException(typeof(ArgumentNullException),
@"Value cannot be null.
Parameter name: location")]
        public void TestCreateASphereWithNullLocation()
        {
            Sphere sphere = CreateSphere(null,
                new Vector(0.0, 0.0, 0.0), 1.0);
        }
```

The `ExpectedException` message here is broken over two lines, to reflect the fact that the message in the exception thrown contains an `Environment.NewLine`. Run the tests to prove that we get our expected red bar, and then insert the following into the top of the `Sphere` constructor:

```
            if (location == null)
            {
                throw new ArgumentNullException(
                    "location");
            }
```

Run the tests to check that we get the expected green bar on the test checking for null location, and then implement a similar test that checks for the velocity instance being passed in being null.

Debug Assertions versus Exceptions

In the last tutorial I promised that I would say in a later tutorial what the best strategy is for validating the arguments passed into a method; whether it's better to have Asserts or to throw exceptions.You should throw exceptions on any public method, where any external user of your code passes in arguments. This is to give direct feedback to the user, in a situation similar to where a user constructs an object with a null reference, but without the check in the constructor. A `NullReferenceException` would only be thrown when calling a method on this object, maybe much later, and obviously the cause of this would be much less obvious than having an `ArgumentNullException` thrown on construction.

Assertions are used to catch internal coding errors, they shouldn't be fired in normal operation, their purpose is to catch bugs in the codebase, and they are commonly put in places where it is anticipated that they will catch newly-introduced bugs, as well as bugs that may have not been exercised in testing. You should use assertions liberally for private and internal methods.

You can use assertions to validate the arguments for a method for a private method (mainly for performance reasons, for most methods it's a good idea to throw exceptions anyway), as even though you know everywhere in your code base the method is called from calls it with correctly-formed parameters, it catches bugs where you may make changes to the arguments passed in (they could be obtained indirectly from another method that changes).

Another useful place for Asserts is in the default case in a switch statement, so if you're switching on e.g. an enum entry, or a combobox entry it can catch places where you forgot to add handling for the new entry. An assertion thrown in the code is a bug that needs fixing. This is especially apparent with unit testing, as an assertion dialog holds up the tests and should be taken to be a test failure.

An exception to the rule of having exceptions thrown in the public interface may be if you have a performance-critical method that is called often, and you'd leave it to the caller to ensure that the arguments were correct. This is obviously not the general case.

Making use of vectors

Let's change the interface of the `Sphere` class to be as desired, by showing what we want it to be in the test, before coding it, in the `UpdateSphereAndCheckState()` test helper: We want the `Sphere`'s location and velocity to be expressed as vectors, and for `ApplyForce` to take a `Vector`:

```
        private void UpdateSphereAndCheckState(Sphere sphere,
            double expectedVelocity, double expectedLocation,
            double appliedForce, int numberOfIterations)
        {
            ...
            sphere.ApplyForce(new Vector(appliedForce, 0.0, 0.0));
            ...
```

```
            Assert.AreEqual(expectedVelocity, sphere.Velocity.X,
                tolerance, "Velocity not as expected");
            Assert.AreEqual(expectedLocation, sphere.Location.X,
                tolerance, "Location not as expected");
        }
```

Obviously, we've broken the build again.

Change the `Sphere` and `Vector` properties and associated member variables on the `Sphere` to be of type `Vector` rather than double. Set the vectors to be null in their declarations, e.g:

```
        private Vector velocity = null;
```

Now, the `Sphere` constructor can directly assign the member variables from the parameters passed in.

```
            this.location = location;
            this.velocity = velocity;
```

The `Sphere`'s `ApplyForce()` method is changed to take a `Vector`:

```
        public void ApplyForce(Vector force)
        {
            this.force = force.X;
        }
```

Change the `Sphere`'s `Update()` method to get and set the x dimension of the vector, remember that we're only changing the interface of our `Sphere` class to be 3D, and currently only the x dimension. This will be changed to be more generic once the tests are in place.

```
        public void Update(double deltaTime)
        {
            double acceleration = this.force / mass;
            this.velocity.X += acceleration * deltaTime;
            this.location.X += this.velocity.X * deltaTime;
        }
```

To get this to compile add a setter to the `Vector`'s `X` property:

```
        public double X
        {
            get { return x; }
            set { x = value; }
        }
```

Now, the tests all compile again, and they all pass. Internally the `Sphere` is still only using one dimension, it's just pulling all the information that it needs off the `Vector`'s `X` dimension, but the progress we've made is to get the tests to make use of 3 dimensions in the interface that it's calling, even if it's using just one of them. Next, we'll go and introduce another dimension.

No Longer One-Dimensional

Let's introduce another dimension into the tests. We'll write a quick test that's a duplication of the other tests to get the functionality in place, then refactor the code and the tests.

Add a new test. You'll see that the code for `CreateSphere()` and `UpdateSphereAndCheckState()` is duplicated here – that's because those helper methods are tied to the x dimension, and we don't want to too drastically break the tests. We'll use this method to factor out the tests making everything more generic.

```
[Test]
public void TestMassTwoKilogramsNonZeroLocationYDimension()
{
    Vector location = new Vector(0.0, -5.0, 0.0);
    Vector velocity = new Vector(0.0, 0.0, 0.0);
    Sphere sphere = new Sphere(location, velocity, 2.0);

    double expectedVelocity = 5.0;
    double expectedLocation = 20.0249999999;
    double appliedForce = 1.0;
    int numberOfIterations = 1000;

    const double time = 10.0;
    sphere.ApplyForce(new Vector(0.0, appliedForce, 0.0));
    for (int i = 1; i <= numberOfIterations; ++i)
    {
        sphere.Update(time / numberOfIterations);
    }
    const double tolerance = 1e-10;
    Assert.AreEqual(expectedVelocity, sphere.Velocity.Y,
        tolerance, "Velocity not as expected");
    Assert.AreEqual(expectedLocation, sphere.Location.Y,
        tolerance, "Location not as expected");
}
```

Add a dummy `Y` property to the `Vector` class to get this to compile:

```
/// <summary>
/// Returns the vector's X dimension value.
/// </summary>
public double Y
{
    get
    {
        return double.NaN;
    }
}
```

You may be wondering why we return a value of `double.NaN` rather than throwing a `NotImplementedException`. This is because we want to see that the red bar fails, but we want to have confidence that our test is actually testing for the correct thing. For example, we may write something stupid in our test like

```
// SomeMethod accidentally returns sphere.Location
```

```
Vector myVector = SomeMethod(sphere);
Assert.AreEqual(myVector, sphere.Y);
```

This test will fail if the `Y` property threw an exception, but pass when this exception is removed giving the wrong impression that we had successfully been through the red bar-green bar stage. The red bar stage didn't indicate that we're testing the wrong thing, whereas if the `Y` property returned `double.NaN`, we'd see an unexpected green bar.

Let's get the tests to pass. First, add a member variable, `y`, to the `Vector` class, and implement the getter and setter as for the `Y` property, and set it in the `Vector`'s constructor.

Now we've got to the point where we're having to change the `Sphere`'s internal workings to be 3D to get all of the tests to pass. Firstly, our force is currently non-directional. Change the `Sphere`'s force member to be a `Vector`. So now `ApplyForce()` can directly copy the force over.

The `Update()` method will be the largest change, and will give us direction on the next piece of development.

We want the acceleration to be a `Vector`, with each of its components to be the force component divided by the mass.

```
public void Update(double deltaTime)
{
    Vector acceleration = new Vector(this.force.X / mass,
        this.force.Y / mass, 0.0 );
```

Now, change the velocity calculation to make use of the acceleration vector:

```
    this.velocity.X += acceleration.X * deltaTime;
    this.location.X += this.velocity.X * deltaTime;
```

Run the tests, to confirm that we haven't broken any of our existing tests. We haven't, so let's go and get our new test to pass. Insert the following two lines:

```
    this.velocity.Y += acceleration.Y * deltaTime;
    this.location.Y += this.velocity.Y * deltaTime;
```

Our tests pass, but the previous code may have made you feel a bit queezy, with the duplication of the calculations for the x and y dimensions. Why didn't we go ahead and implement an overloaded addition operator straight away? With test-driven development the pace of development goes

Red bar
Green bar
Refactor

We've first had to go through the red bar-green bar stage. We didn't want to add new functionality (the overloaded operator) until we'd proved that the functionality that we were in the middle of adding (the y dimension) was working. Now we've proved that the implementation is correct in the quickest possible route, we can go ahead and clean the code up. If we'd introduced the addition operator in the middle of the other stuff and figured out that the tests didn't pass, we'd have much more code to look through to find the cause of the failure; the code for the `Update()` method and the addition operator code itself.

Overloaded operators

Now that the `Update()` method works as expected though, let's remove the duplication of logic for the addition of the vector components. We'll go ahead and express the intent and functionality of the overloaded addition operator in a new set of tests, for the `Vector` class. This may seem a bit like too much effort – the operator will be tested via its use in the `Update()` method, but the `Vector` should have its own tests. Firstly, we may like to use the `Vector` class elsewhere, and don't want to rely on running the Physics engine tests to test it, and secondly, it helps with debugging any errors that we may introduce in the `Update()` method if we can see that the `Vector` tests pass and so can eliminate that area from investigating the cause of the bug.

Add a new class, `TestVector`, to the Test project. Make it public, add a suitable comment, add a `using` statement for the `NUnit.Framework` and `Taumuon.Jabuka.Physics` namespaces, and mark the class with a `TestFixture` attribute.

Add a new public method, marked with the `Test` attribute, `TestVectorAddition()`. Insert the following implementation for the method:

```
public void TestVectorAddition()
{
    Vector vector1 = new Vector(1.0, 2.0, 3.0);
    Vector vector2 = new Vector(-2.0, 1.0, 1.0);
    Vector vector3 = vector1 + vector2;
    // Check that the original vectors are unchanged
    //  following addition
    Assert.AreEqual(vector1.X, 1.0);
    Assert.AreEqual(vector1.Y, 2.0);
    Assert.AreEqual(vector1.Z, 3.0);
    Assert.AreEqual(vector2.X, -2.0);
    Assert.AreEqual(vector2.Y, 1.0);
    Assert.AreEqual(vector2.Z, 1.0);
    // Check the result of addition
    Assert.AreEqual(vector3.X, -1.0);
    Assert.AreEqual(vector3.Y, 3.0);
    Assert.AreEqual(vector3.Z, 4.0);
}
```

Now, this method isn't very elegant – the checking of each of the vector dimensions is messy and hints at the necessity of having a check for vector equality. However, this test

method is good enough for now, the first thing to do is to get the tests to compile again (the same rhythm that you're hopefully getting used to by now).

Add the `Z` property onto the `Vector` class, but have the getter return `double.NaN` and the setter throw a `NotImplementedException` for now.

Add the overloaded operator as follows:

```
/// <summary>
/// Overloaded addition operator
/// </summary>
/// <param name="lhs">The left-hand operand</param>
/// <param name="rhs">The right-hand operand</param>
/// <returns>
/// Result of addition, leaving the operands unchanged.
/// </returns>
public static Vector operator +(Vector lhs, Vector rhs)
{
    return null;
}
```

The build succeeds, so run the tests to check that our new test is present, and fails, and also that the other tests still work.

To get the test to pass, add a member `z` to the `Vector` class, assign it in the constructor, and implement the `Z` property as the others.

Change the addition operator implementation to the following:

```
return new Vector(lhs.X + rhs.X, lhs.Y + rhs.Y,
                                        lhs.Z + rhs.Z);
```

Now we get a green bar. Note that we really should add an overloaded subtraction operator, so we don't surprise users of the class, but for now we'll just add the functionality we need to tidy up the `Update()` method.

Looking in the `Update()` method, we can see that we'll want to multiply our vector by a scalar variable, representing the time difference. So let's add a test for this to our `TestVector` class:

```
[Test]
public void TestVectorMultiplyByScalar()
{
    Vector vector1 = new Vector(1.0, 2.0, 3.0);
    Vector vector2 = vector1 * 2.0;
    // Check that the original vector is unchanged
    //  following multiplication by a scalar
    Assert.AreEqual(vector1.X, 1.0);
    Assert.AreEqual(vector1.Y, 2.0);
    Assert.AreEqual(vector1.Z, 3.0);
    // Check the result of multiplication by a scalar
    Assert.AreEqual(vector2.X, 2.0);
```

```
            Assert.AreEqual(vector2.Y, 4.0);
            Assert.AreEqual(vector2.Z, 6.0);
        }
```

To get the project to build, add the following:

```
        /// <summary>
        /// Overloaded multiplication operator, for a vector and
        /// a scalar.
        /// </summary>
        /// <param name="lhs">The left-hand operand</param>
        /// <param name="rhs">The right-hand operand</param>
        /// <returns>
        /// Result of multiplication, leaving operands unchanged.
        /// </returns>
        public static Vector operator *(Vector lhs, double rhs)
        {
            return null;
        }
```

We could insert the implementation straight away, but remember that we want to see the red bar before to satisfy ourselves that we're actually testing something.

Now, implement the method as follows:

```
            return new Vector(lhs.X * rhs, lhs.Y * rhs, lhs.Z * rhs);
```

Back to the `Update()` method

We'll leave the refactoring of the `Vector` tests for a while – we'll be revisiting that class soon enough. Note that we should add functionality and tests for multiplying a scalar on the lefthand side of the equation, and a vector on the right-hand side.

Now, we'll tidy up the `Update()` method, then refactor the Physics tests.

Change the `Sphere.Update()` method to the following, and verify that the tests still pass:

```
        public void Update(double deltaTime)
        {
            Vector acceleration = new Vector(this.force.X / mass,
                this.force.Y / mass, 0.0 );
            this.velocity += acceleration * deltaTime;
            this.location += this.velocity * deltaTime;
        }
```

Note that as we've overloaded the + operator, += has also automatically been implemented in terms of our operator. Unfortunately however, we can't implement this ourselves; the overloaded += operator creates code semantically the same as the following:

vector1 = vector1 + vector2;

This means that the reference held in vector1 is replaced by a new instance, leading to more allocations than necessary. Even though allocation is cheap in .NET,
It may be better performance wise not to make unnecessary allocations, the garbage collector works best with either very short or long lifetime objects. It may give better performance to add a method `AddToMe(Vector rhs)` that adds the values in the rhs `Vector`'s instance to our instance. But we'll look at these performance implications later, we'll build our code to work, and be simple, and once we've got something to benchmark we'll look at these kinds of issues.

Of course, if the `Vector` was a struct this wouldn't be an issue, and it may give better performance, but we'd first see whether the vector should have value rather than reference type semantics. Also, looking at performance is something that will have to happen much later after we build some benchmarks; we want to see the performance effects of various optimizations in a realistic app; it may be that a change that relies on caching of *some stuff* may work well in a small benchmark, but perform worse in a larger app under memory pressure

Refactoring Tests for 3 Dimensions

Now let's go back and refactor our `TestSphereNewtonLaws` class. We created tests to update the velocity and location of a sphere, in one dimension, and refactored the tests into helper methods. We then added the `Vector` into the mix, but only in the x dimension. We drove the development of the `Vector` class by adding a test in the y dimension, but we couldn't use any of our helper methods as they were hardcoded to the x dimension.

We want to tidy our tests to allow them to run in each dimension, and we want the same tests to run in each dimension – we want the test methods to call a helper method with the vector class to access, and it will access either the x, y or z axis dependant on a member set in the fixture's constructor. This should be clearer as we implement this.

Create a new class, `TestSphereNewtonLawsDimensionSpecific`, but don't mark it with the `TestFixture` attribute. We want to derive from this class to allow the derived fixtures to set the dimension to be called. Make the class public, and add a `using` statement in for the `NUnit.Framework` namespace.

```
/// <summary>
/// Tests that are dimension specific.
/// The contained test cases will be ran for the three
/// dimensions.
/// </summary>
public class TestSphereNewtonLawsDimensionSpecific
{
}
```

Move all of the test methods apart from `TestCreateASphereWithZeroMass`, `TestCreateASphereWithNullLocation` and `TestCreateASphereWithNullVelocity` into the new fixture class. Also move all of the private helper methods. Change

`TestCreateASphereWithZeroMass` and the null check methods to create a `Sphere` directly:

```
[Test]
[ExpectedException(typeof(ArgumentException),
    "mass cannot be zero")]
public void TestCreateASphereWithZeroMass()
{
    Sphere sphere = new Sphere(new Vector(0.0, 0.0, 0.0),
        new Vector(0.0, 0.0, 0.0), 0.0);
}
```

Note that creating the `Spheres` directly goes against the work we did in introducting `CreateSphere()`, but that method is going to become dimension specific in the other tests. We could create a new version of the method in `TestSphereNewtonLaws`, but we'll go and do that if we need to.

You can see that we've only got the constructor argument tests running now, we've lost all of the checks for the position and velocity, that we just moved into `TestSphereNewtonLawsDimensionSpecific`. Create a new class, deriving from `TestSphereNewtonLawsDimensionSpecific`, called `TestSphereNewtonLawsDimensionX`. Add the `TestFixture` attribute to the class, and we've got the tests appearing in NUnit again.

Add a new private int member to `TestSphereNewtonLawsDimensionSpecific` called `vectorAxis`. This will indicate which axis on the `Vector` we will use. We could actually implement this as an indexer on the `Vector`, such that we could access all the dimensions as `vectorInstance[0]` for the x axis, `vectorInstance[1]` for the y axis etc. The `X`, `Y` and `Z` properties are a specialization of a multi-dimensional vector for only 3 dimensions. But our current requirements are only for 3 dimensions, and there's no requirement for an indexer other than for one method in the tests, and we'd have to add supporting tests for functionality not used elsewhere.

Note that we have to move all of our tests into the base class, and derive from this class with 3 classes decorated with the `TestFixture` attribute, as NUnit works by looking for attributes to describe tests. I've been blogging on taumuon.blogspot.com about how NUnit 2.4's extensibility feature lets us create tests programatically.

Now, in `TestSphereNewtonLawsDimensionSpecific`, we'll add two new methods that will be used to set and retrieve the current vector value depending on whether the x, y, or z dimension fixture is running:

```
/// <summary>
/// Retrieves the value stored in the current fixture's
///  dimension out of the provided Vector.
/// </summary>
private double GetVectorValue(Vector vector, int dimension)
{
    switch (dimension)
```

```
            {
                case 0 :
                    return vector.X;
                case 1 :
                    return vector.Y;
                case 2 :
                    return vector.Z;
                default :
                    throw new ApplicationException
                        ("Unrecognised dimension: " +
                        dimension.ToString());
            }
        }

        /// <summary>
        /// Stores the value into the Vector given fixture's
        /// current dimension.
        /// </summary>
        private void SetVectorValue(Vector vector, int dimension,
            double value)
        {
            switch (dimension)
            {
                case 0:
                    vector.X = value;
                    break;
                case 1:
                    vector.Y = value;
                    break;
                case 2:
                    vector.Z = value;
                    break;
                default:
                    throw new ApplicationException
                        ("Unrecognised dimension: " +
                        dimension.ToString());
            }
        }
```

Let's go ahead and make use of these two helpers everywhere that we currently use on of the `Vector`'s accessors. Change the `CreateSphere()` overload to create the vectors at the origin, and to then set the required dimension.

```
        private Sphere CreateSphere(double location,
            double velocity, double mass)
        {
            Vector locationVector = new Vector(0.0, 0.0, 0.0);
            SetVectorValue(locationVector, vectorAxis, location);
            Vector velocityVector = new Vector(0.0, 0.0, 0.0);
            SetVectorValue(velocityVector, vectorAxis, velocity);
            return new Sphere(locationVector, velocityVector, mass);
        }
```

Change `UpdateSphereAndCheckState()` to be:

```
        private void UpdateSphereAndCheckState(Sphere sphere,
            double expectedVelocity, double expectedLocation,
            double appliedForce, int numberOfIterations)
        {
            const double time = 10.0;
            Vector force = new Vector(0.0, 0.0, 0.0);
            SetVectorValue(force, vectorAxis, appliedForce);
            sphere.ApplyForce(force);
            for (int i = 1; i <= numberOfIterations; ++i)
            {
                sphere.Update(time / numberOfIterations);
            }
            const double tolerance = 1e-10;
            Assert.AreEqual(expectedVelocity,
                GetVectorValue(sphere.Velocity, vectorAxis),
                tolerance, "Velocity not as expected");
            Assert.AreEqual(expectedLocation,
                GetVectorValue(sphere.Location, vectorAxis),
                tolerance, "Location not as expected");
        }
```

Now our tests (apart from the hardcoded test in the y dimension) are accessing the vector only via the Get and Set methods that we've added. Let's go ahead and add the tests for the y dimension. Add a new fixture as follows:

```
[TestFixture]
public class TestSphereNewtonLawsDimensionY
    : TestSphereNewtonLawsDimensionSpecific
{
}
```

Run the tests. We've got a green bar, this isn't necessarily what we want. It's always a bit dangerous when refactoring tests to make sure that the tests still are checking the right thing, similar to the reasons we want a red bar before a green bar. In this case, the tests aren't doing what we want – we haven't set the `vectorAxis` in this fixture to indicate the y direction, so the fixture is just a repeat of the x dimension.

Let's break the tests – change the `Vector` class's `Y` and `Z` getters to return zero. Now the vector tests obviously fail, but we still haven't got our red bars everywhere. Add a protected setter, called `VectorAxis` to the base class that sets the `vectorAxis`. Add a constructor to `TestSphereNewtonLawsDimensionY` that calls this property to set the `vectorAxis` to 1;

This is good, we're getting the expected red bars. Add a fixture for the z axis tests, similar to the y axis tests, setting the `vectorAxis` to 2, and you'll get the same results for the z axis.

Fix the y, but not the z setter in the `Vector` class, as we want to see that we get a green bar for the y tests, but check that the z tests aren't an inadvertent duplication of the y tests (as happened earlier).

We can now remove the y specific test `TestMassTwoKilogramsNonZeroLocationYDimension()` from the base class, as we're convinced that it's adequately covered by the other tests. As the tests are getting quite complicated, it may be worth keeping at least one hardcoded test for each dimension (maybe the most complicated test case) in case of any errors in the logic of our test harness.

Now, change the `Z` vector accessor to return the value of the z dimension. The tests fail as the Sphere's `Update()` method calculates the acceleration as:

```
Vector acceleration = new Vector(this.force.X / mass,
    this.force.Y / mass, 0.0 );
```

Change the final component to be

```
this.force.Z / mass
```

Again, this hints that maybe we should overload the / operator (but we can just pass the recipricol of the mass into our multiply by scaler overloaded * operator), but we'll do that if we find that we are following this pattern in more than one place.

Conclusion

That's it, we've got green bars. We've got a basic working physics engine.

Notice that we've kept things as simple as possible, and put in just enough code to get our tests working. We could have gone ahead and wrote tests for the vector dot product and cross product, and we didn't even implement a subtraction operator, we only implemented what we needed for our tests to pass. The reasoning behind this is that we may decide to use another `Vector` class, or we may want to write the class in C++/CLI for performance reasons.

We didn't even implement the subtraction operator on the `Vector`, and that is something we would definitely do if we intended this to be used as a third party and as the engine matures, but for now it's needless work. Of course, the first time we need the operator in our engine we'll implement it.

You may be wondering why I didn't develop the code in the first tutorial using TDD. Well, the simple answer may be that I should have, but in that tutorial I wanted to show how to use the Tao Framework with OpenGL. The truer answer may be that I choose not to as there isn't that much logic in that tutorial, and most of the code there is is to do with drawing to the screen, and there is not much value to writing tests for it. I could test the output by either obtaining the handle of the drawing surface and saving the drawn output to the file, to check for changes (assuming that the drawing happens on the same graphics card with the same drivers the output should stay constant), or by introducing an interface betweent the OpenGL calls to test that I'm making the same calls (but that would be a performance killer), and a visual check of the output would probably be good enough. I should definitely put some tests in for the code that is in the 3D project (apart

from testing the OpenGL calls), especially as it grows more complicated to include serialisation, and some kind of scene graph. We'll start adding tests gradually as we add more functionality.

Hopefully this tutorial has given you some insight into the processes behind Test-Driven Development. There's a lot of text for not much code, but that's to cover many of the little refactorings. During development they don't take that long, and feel much safer than putting in great swaves of poorly tested code. In the next tutorial we'll get the ThreeD engine to make use of the physics code.