Automation Testing in Unity

Roland Zeng

CS199 Directed Research under Prof. Diana Ford

Summer 2015

Abstract

During the summer of 2015, I underwent research with Professor Diana Ford. The topic was to investigate automation testing in the Unity 3D Engine. Unity is the most popular game engine in the world, boasting a multi million user base consisting of almost half the world's game developers. This means the majority of aspiring game developers and coders will most likely encounter or develop in Unity. With the rise of modern trends in software development such as test-driven development (TDD), automation testing, and writing unit tests, I was interested in seeing the availability of test frameworks and tools available to Unity developers.

My search led me to Unity Test Tools, a testing framework created directly by the Unity development team. It offers testing features in the form of integration tests and a built-in unit testing framework. Under the guidance of Prof. Ford, I applied Unity Test Tools to write a wide set of tests to assert the robustness of one of my old Unity projects. I succeeded in writing integration tests, creating assertion components, and designing unit tests. However, I ran into many challenges, such as code refactoring and mocking limitations. This paper will describe the steps I took, the results I obtained, the challenges I faced, and my outlook on Unity Test Tools as a viable framework for introducing test-driven development for game developers.

Introduction

I had taken one of Prof. Ford's previous classes, 3D Real-Time Animation earlier in the year. In this course, we used the Unity engine to create a short game / animation over the course of ten weeks. Unity is a cross-platform engine used for developing desktop, mobile, and console games. As of now, Unity supports over ten platforms, with creations spanning from Android and iOS phone games, to Oculus Rift-compatible virtual reality games.[1] Furthermore, as of March 2015, Unity's newest version of the engine, Unity 5, was made free for all users. Because of its cross-platform capabilities and affordability, Unity is an extremely attractive option for indie developers and teams. As of August 2015, there were more than 4.5 million registered users, 1 million active developers, and over 600 million users of games that utilized Unity's engine.[2] Examples of popular Unity games include *Temple Run*, *Rust*, *Kerbal Space Program*, *Hearthstone*, and *Angry Birds*. It is no doubt that for many aspiring game developers, Unity will most likely be a first choice.

However, as many game developers find out, the larger a game gets, the more the chance of bugs and unwanted errors increases. In fact, this is a problem faced by not only game developers, but software developers in general. As projects become more advanced, they become more complex not only because more lines of code are written, but also because existing code is constantly being refactored and tuned. To account for this, many software developers turn to practices such as writing unit tests, which test individual code functionality, and designing integration tests, which test large components of code. These tests are usually run whenever changes to the code base are made, in order to catch new bugs before they

propagate through the software. This process is referred to as automation testing, and is a key feature of modern agile software development. Generally, some sort of framework is used to create sets of tests and assertions that can be executed with ease.

Unity Test Tools: An Overview

Because Unity is so popular, and because testing is a critical component of ensuring that a game runs perfectly, I wished to look into the viability of writing and running tests in Unity. I did not have to look far; Unity themselves offer a testing framework, Unity Test Tools, which provides three testing capabilities: Assertion Component, Integration Test Runner, and Unit Test Runner. I decided to download this framework and worked to incorporate tests into the Unity project I had created in Prof. Ford's course.

First off, I had to scrap my old project. Before taking 3D Real-Time Animation, I had no knowledge of Unity or C# scripting. Hence, as a beginner, my code was extremely messy and lots of bugs occurred. I decided to start with a fresh Unity project using concepts from my original game. This new game, called *UnitySnowmen* (see figures 1.1 and 1.2), was essentially my old Unity project from Prof. Ford's class minus a storyline, cutscenes, and visual elements. The player controlled a character that could fire bullets. There were enemy snowmen that would spawn, whose AI was programmed to shoot bullets of their own at the player. Hits on the player damaged the player's health bar, and hits on the enemy damaged the enemy's health bar. Deaths would lead to respawns for both the player and the snowman character.

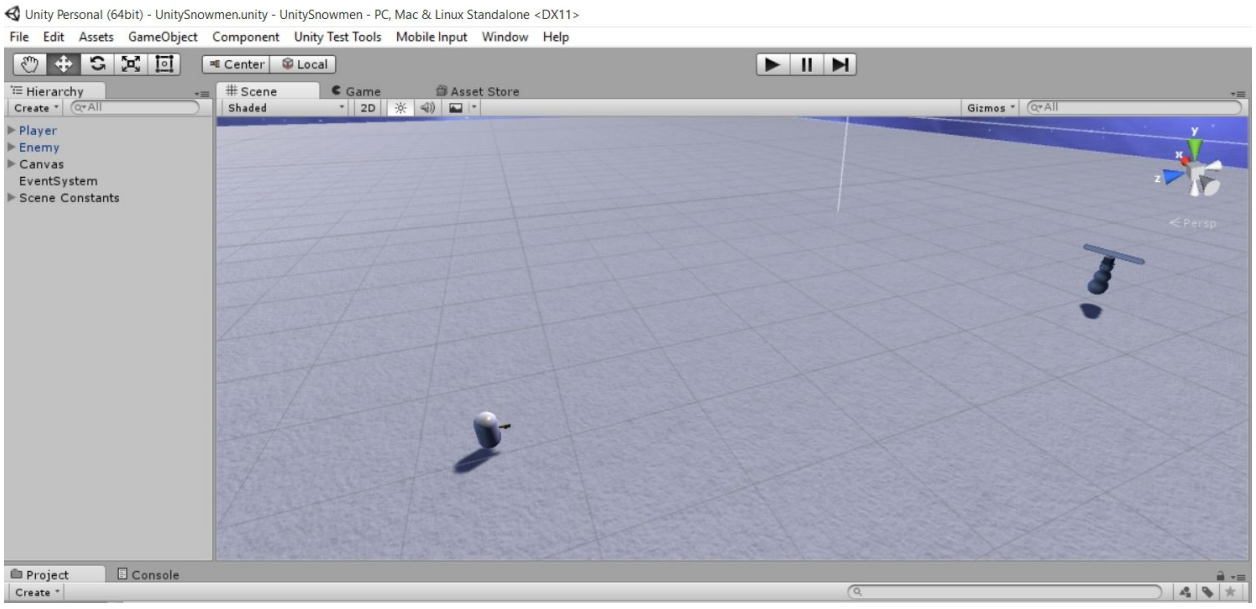

Figure 1.1: Unity Snowmen scene view

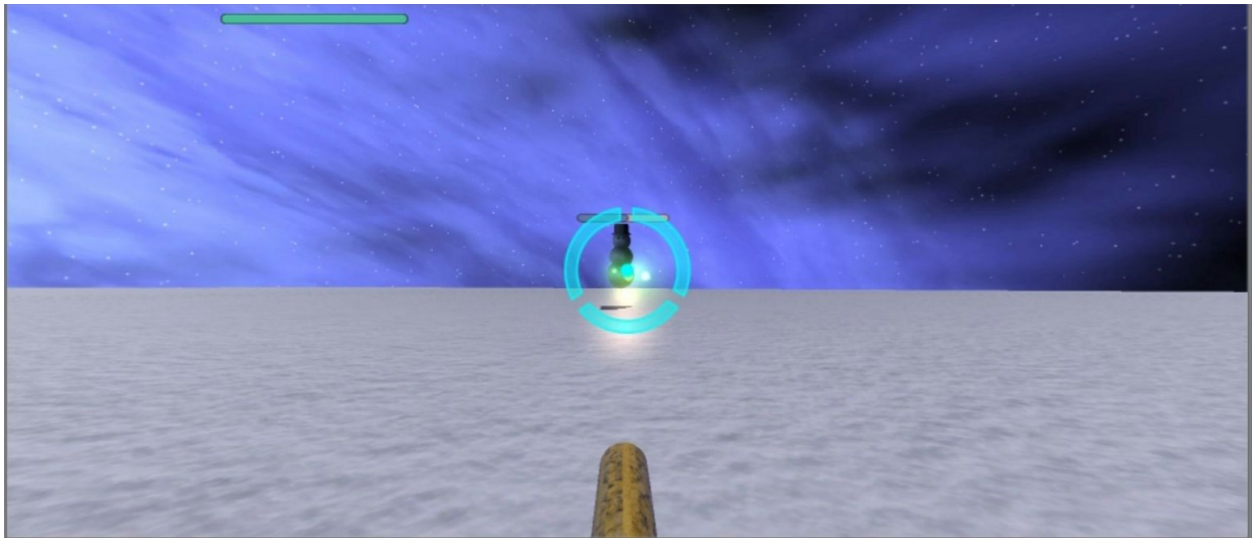

Figure 1.2: Unity Snowmen in-game screenshot

UnitySnowmen represents games in their beginning phase: game mechanics are implemented, with story and visuals soon to follow. This is the perfect stage to implement tests so that one can ensure the game continues to function even as more and more features get added.

Unity Test Tools: Assertion Component

The first feature available is the Assertion Component. In Unity, each object placed onto a scene has the ability to contain components. These components can be pre-built scripts such as Unity's FirstPersonController, which handles movement and player input, or self-coded scripts. For example, consider my "enemy snowman" game object (figure 1.3).

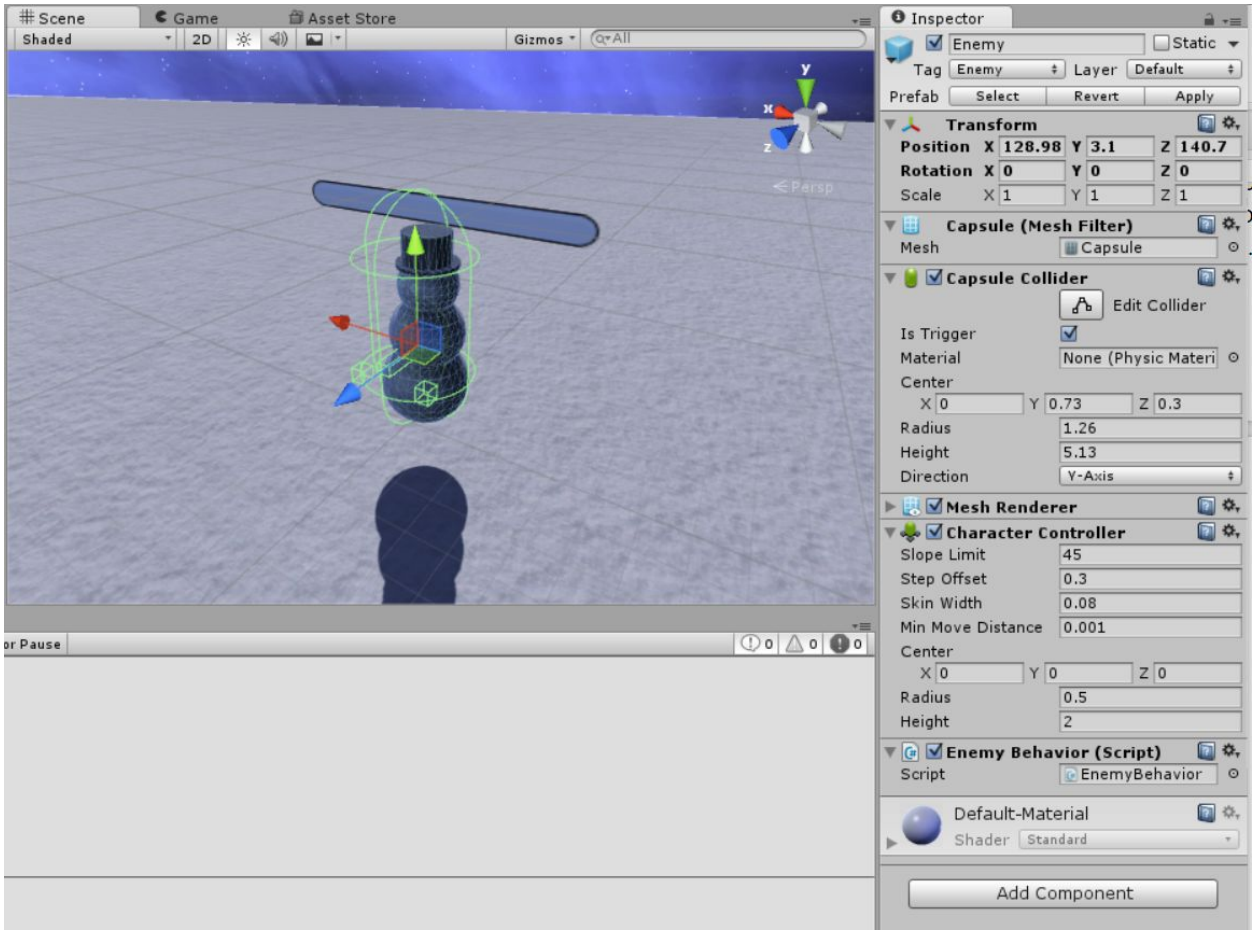

Figure 1.3: Enemy snowman game object & attached components

My enemy snowman object contains a “Capsule Collider,” which is a unity pre-built component that handles collisions so that I don’t need to write my own scripts. It also contains “Enemy Behavior,” a C# script I wrote on my own to tell the enemy “Patrol if player is far away or fire at player if close enough.” Likewise, Unity Test Tools provides the Assertion Component which can also be added onto any game object.

Consider an excerpt of my “Enemy Behavior” script (figure 1.4). It has a feature which, if the player is spotted, to send messages to its gun script to fire a bullet in the direction of the player. This means it must be able to interface with the gun object in the scene. To make sure the gun object actually exists and can be found by the enemy behavior script, I wrote an assertion statement on line 44. The assertion statement would have stopped game execution if the gun object was not detected.

```
30      // Use this for initialization
31      void Start () {
32          player = GameObject.FindWithTag("Player");
33          target = player.transform;
34          detectDistance = 35;
35
36          AIState = 0;
37          PatrolState = 0;
38
39          canFire = true;
40          bulletcounter = 0;
41
42          obj = transform.Find("EnemyKey/EnemyGunTip").gameObject;
43          obj2 = transform.Find("EnemyKey2/EnemyGunTip").gameObject;
44          Assert.IsNotNull(obj, "Error! Could not find enemy gun");
45
46          slider = GameObject.Find("Slider").GetComponent<Slider>();
47          Assert.IsNotNull(slider, "Couldn't find enemy slider!");
48          slider.minValue = 0;
49          slider.maxValue = 100;
50          enemyHP = 100;
51
52          respawnLoc = GameObject.Find("EnemyRespawn");
53          Assert.IsNotNull(respawnLoc, "Couldn't find respawn point!");
54          sw = true;
55
56      }
57
```

Figure 1.4: ***EnemyBehavior.cs*** *excerpt*

Unity Test Tools' Assertion Component is the non-code method of creating assert statements. To implement it, I clicked "Add Component" and found Assertion Component. Once it was added to my object, I am given a wide variety of comparers that I can test for (see figure 1.5).

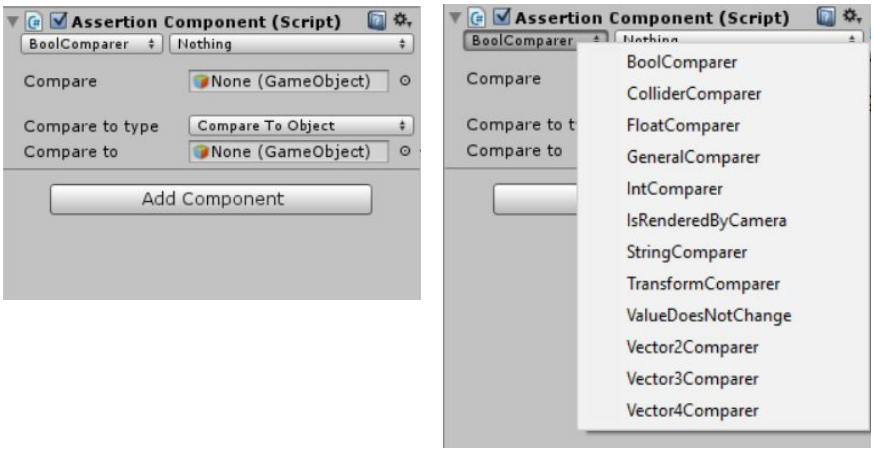

Figure 1.5: Assertion Component

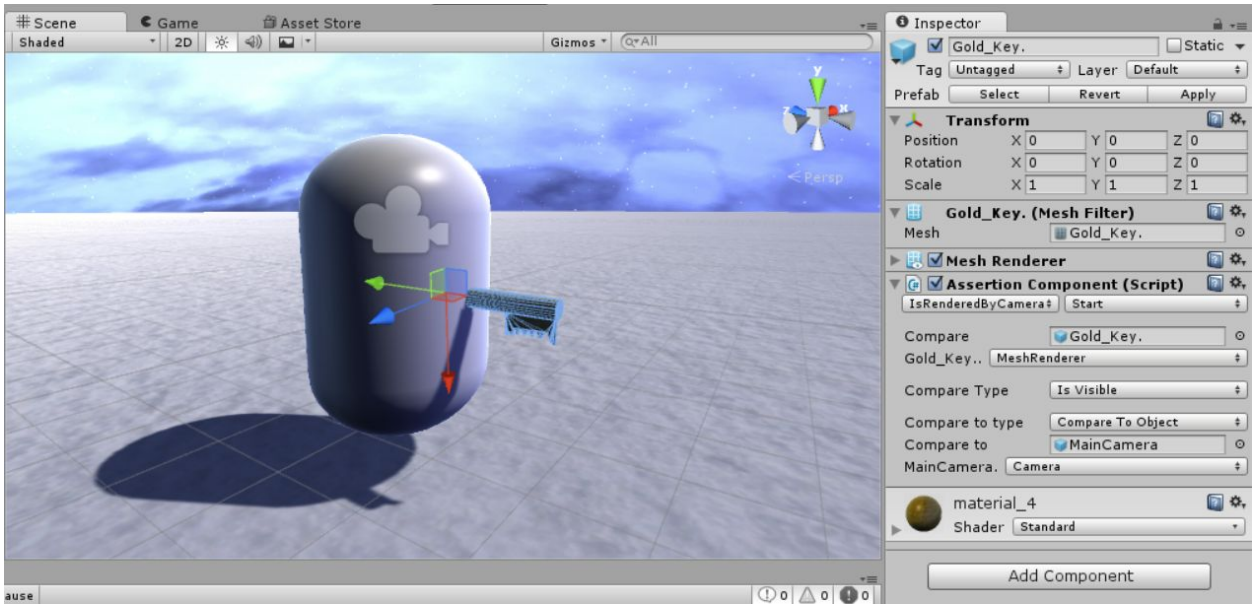

Figure 1.6: Assertion component to make sure the key is rendered by the player camera

In figure 1.6, I attached an Assertion Component to my player object, using the IsRenderedByCamera comparer, that checks if, the key (which represents the gun barrel of the player) is visible to the player's camera (MainCamera) at game start.

Now, instead of creating an Assertion Component to test if the key was active, I could have just used code to check if the key was active or not null at game start, similar to my method in figure 1.4. Why use Assertion Components then?

There are many answers. First off, when an assert component fails, it can pause the game at the exact moment of failure. This lets the developer view the entire scene and see what could have gone wrong. Secondly, less actual code needs to be written. This makes testing available to non-programmers such as artists or game designers. Also, because the tests are implemented as components, when the game is ready to be shipped, these components can be un-checked with ease instead of digging through and removing code. This saves space and performance in the final build, especially for games that require a lot of testing. Finally, Assertion components are a crucial aspect of the Integration Test Runner.

Unity Test Tools: Integration Test Runner

Integration testing refers to how software modules interact with one another. While unit tests (which we will cover later) test small individual units of code, integration tests make sure that groups of individual units work together on a higher level. In Unity's case, integration tests are usually used to make sure prefabs interact correctly with scene elements and other prefabs. These integration tests usually involve setting up a scene catered to isolating and testing a specific feature, running the scene and asserting that components function as intended, and

then tearing down the scene. Unity's Integration Test Runner does this using the Assertion Component from earlier. To start, I created a new scene, as the Integration Test Runner does not go in the regular scenes. In this scene, I created three tests, as seen in figure 2.1.

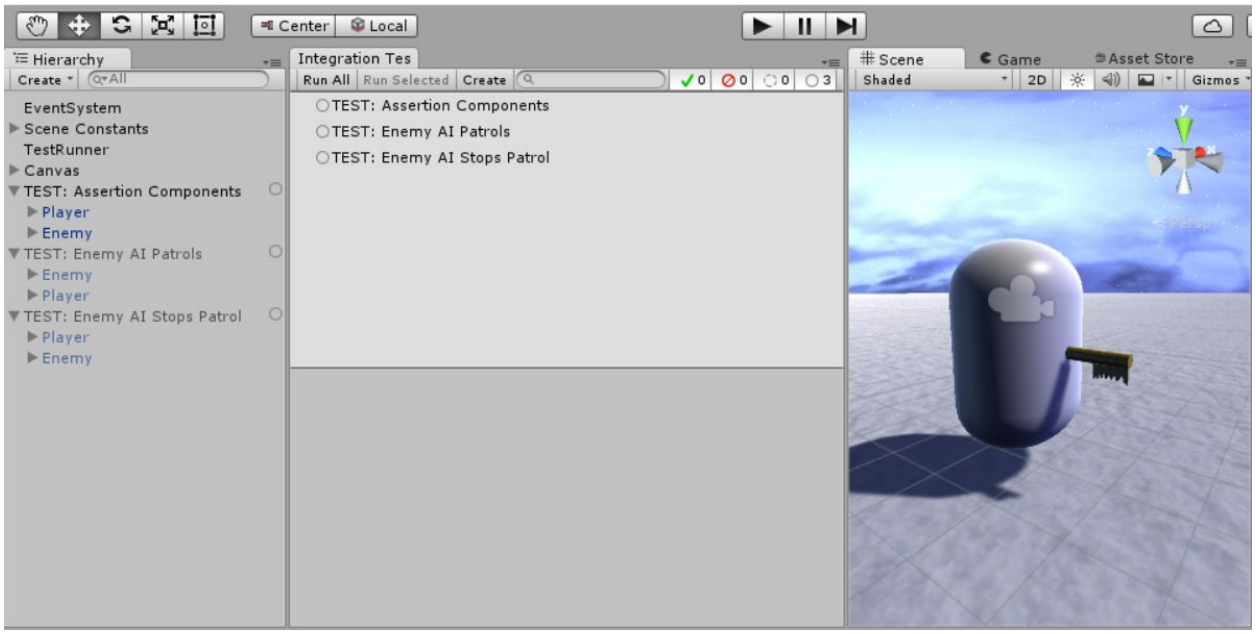

Figure 2.1: Integration Test Runner. New tests are created using the "Create" button in the Integration Tests window.

The first test, "Assertion Components," included the Assertion Components I demonstrated in the earlier section. The second test, "Enemy AI Patrols", contains an Assertion Component that checks whether the position of the enemy snowman changed since the start of the game. The third test, "Enemy AI Stops Patrol," checks if the enemy stops moving when it is attacking the player.

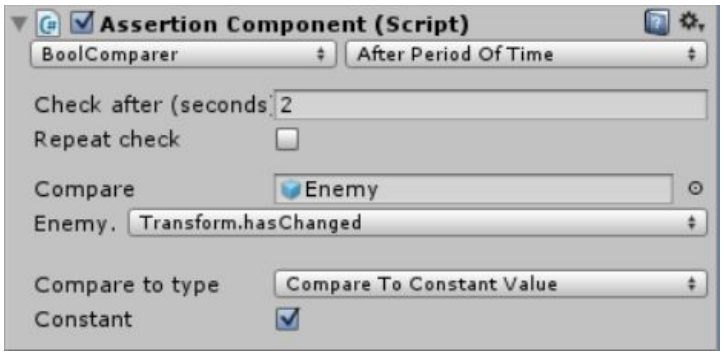

Figure 2.2: Assertion Component for TEST: Enemy AI Patrols. The test is set up such that the player is very far away from the enemy snowman. After 2 seconds of the test being spun up, make sure the enemy is patrolling by asserting that the transform (position) of the enemy has changed.

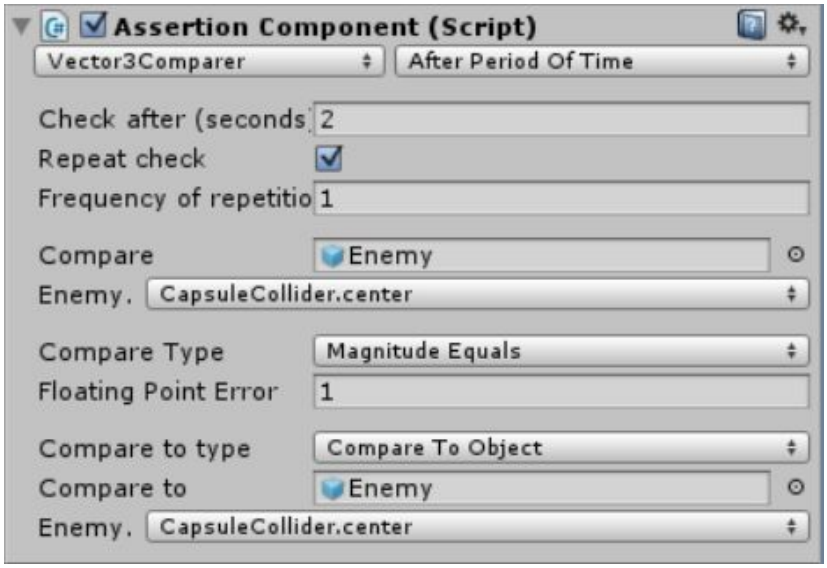

Figure 2.3: Assertion Component for TEST: Enemy AI Stops Patrol. The test is set up such that the player is within attack range of the enemy snowman. After 2 seconds of the test being spun up, make sure the enemy has not moved.

Note from figure 2.1 that when one test is highlighted, the rest are greyed out. This means each test has a distinct setup where I can choose to isolate and test certain variables. For example, in test 2, I chose to place the enemy and player far away from each other to test the enemy AI's "patrol when out of range" functionality. To run all the tests at once, I just need to press the "Run All" button in the Integration Tests window, as seen in figure 2.4.

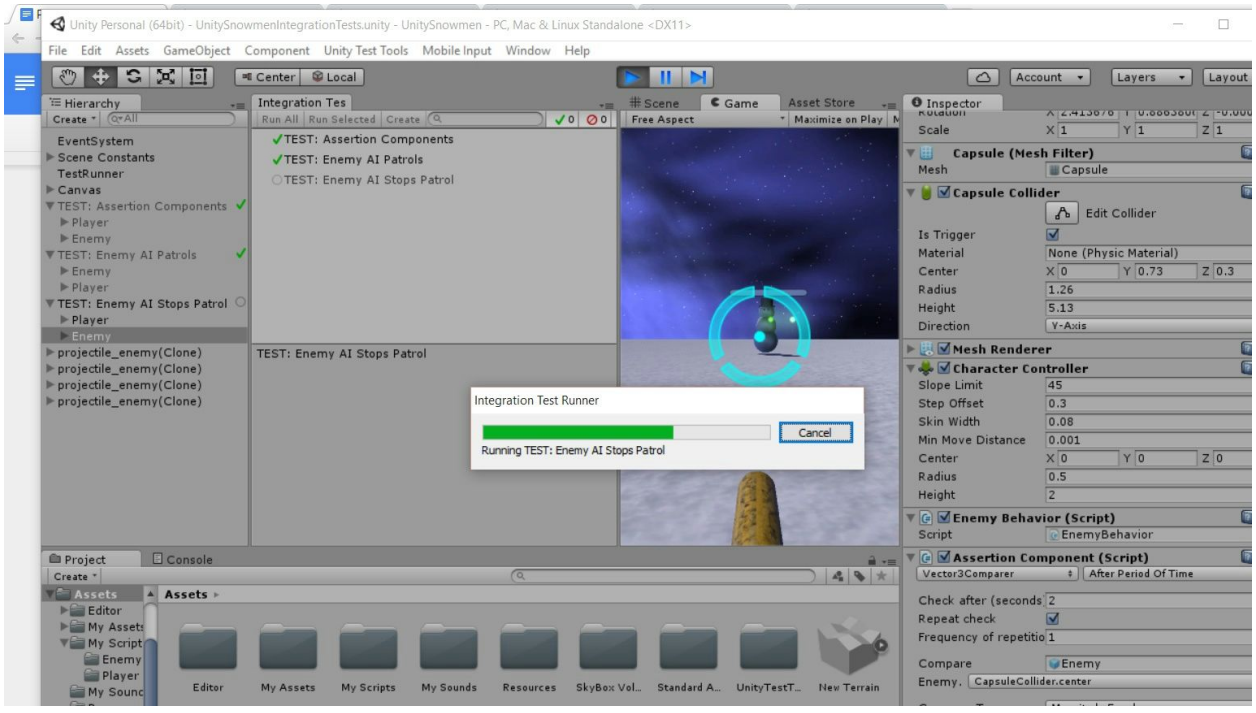

Figure 2.4: "Run All" execution of Integration Test Runner

Doing so runs all tests in sequential order, setting up new tests and tearing them down when completed. At the very end, a list of tests that succeeded and failed are displayed. The Integration Test Runner allows developers to create specific scenarios and ensure that modules behave correctly.

The integration test runner also allows one to implement test-driven development (TDD), a modern software practice. TDD consists of planning a feature, writing a test that fails without the feature, implementing the feature, and then asserting that the test passes.[3] TDD on the surface appears to be a very time-consuming task, due to the amount of effort required to write tests for every bit of functionality. However, it has been shown that TDD increases code quality by 15%-30% with only 10-20% longer development time. [4] Hence, I wanted to see just how easy it could be to apply test-driven development to my own project.

I decided I wanted to implement a feature where the player would take constant damage-over-time. In game terms, this could be because the player is cold and takes natural freeze damage. I wrote the following test:

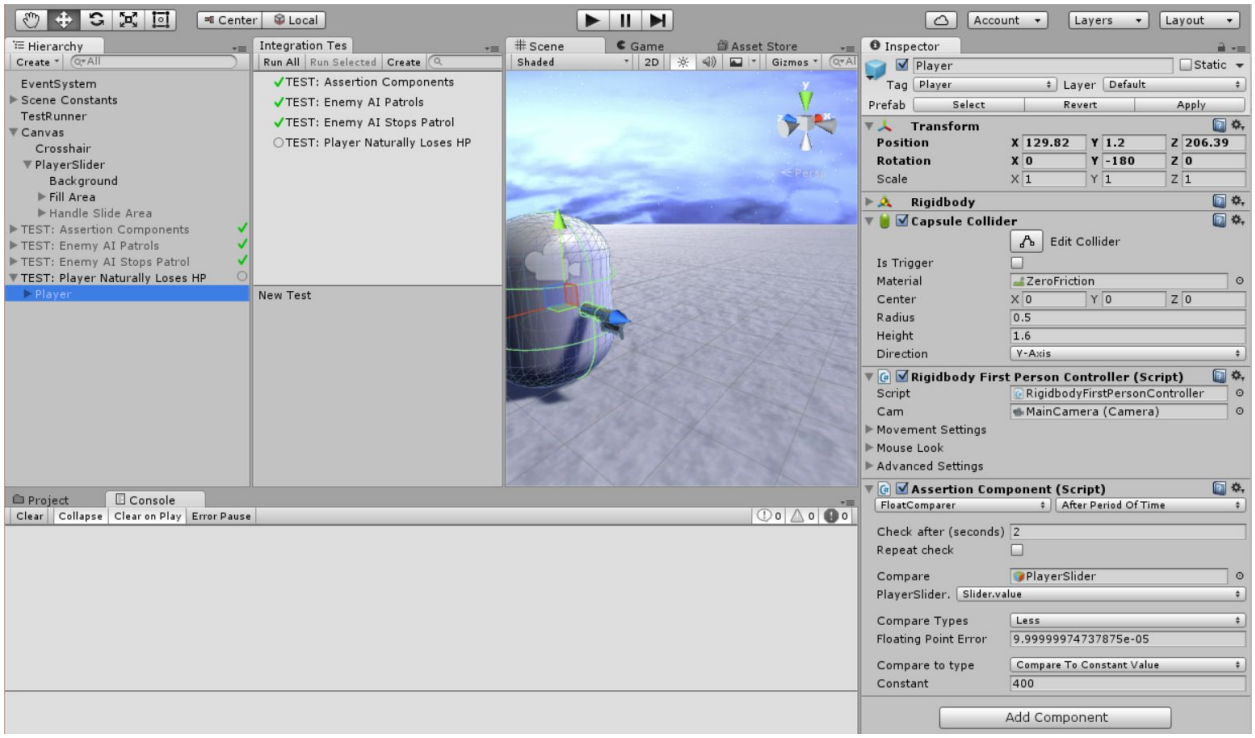

Figure 2.5: TDD test (pre-implementation)

In this test, I spawn a player with no enemy. Then, my Assertion Component checks if after 2 seconds the player's current health (represented by a slider value) is less than 400 (its max health) as a result of constant damage-over-time. Upon running the test, I received the following output:

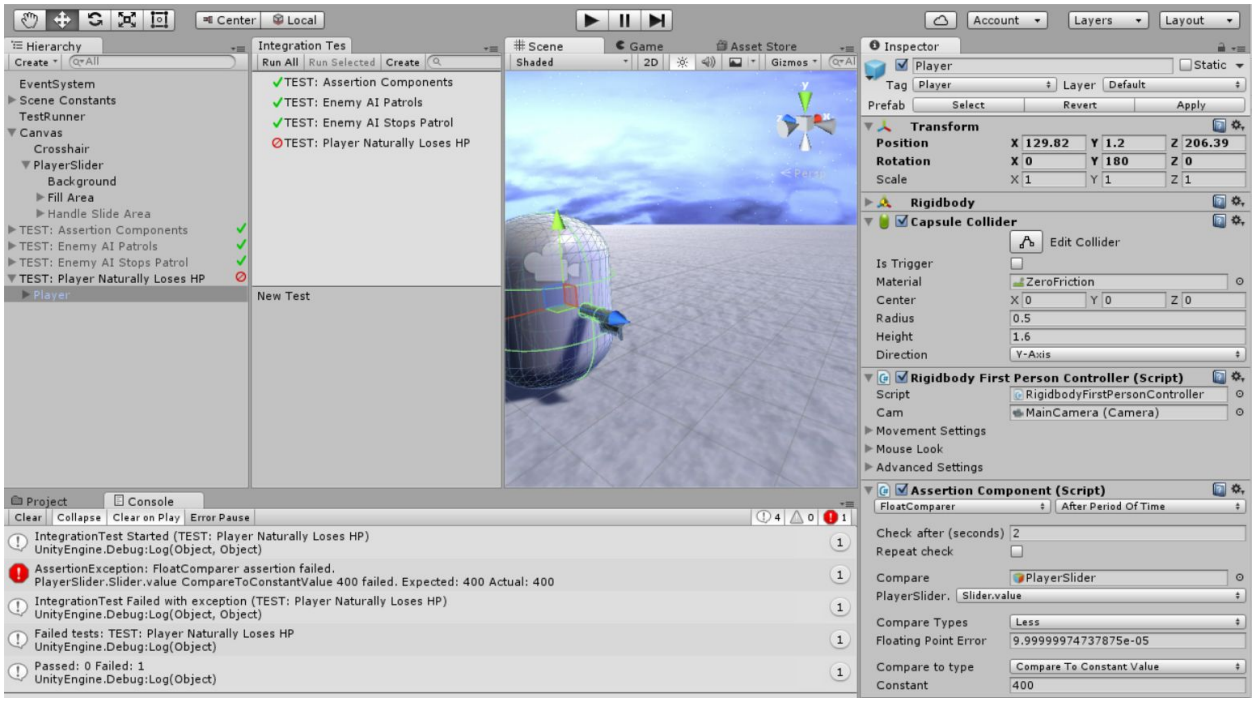

Figure 2.6: TDD test result (pre-implementation). Note that the assertion exception thrown doesn't make any sense because my assertion statement was to check that the slider value was below 400; I did not specify an actual value.

As seen, the test fails, because the player's health does not decrease. Next, I implemented the damage-over-time feature in my PlayerBehavior script.

```
// Update is called once per frame
void Update () {

    slider.value = playerHP;
    //print(slider.value);
    if (playerHP > 0 && canFreeze) {
        playerHP -= 0.1f;

    }
```

Figure 2.7: Implementation of TDD feature. Player loses HP every game tick. This HP value is reflected in the slider value. I created a boolean "canFreeze" so that in the future I could turn freeze on or off for future tests.

Re-running the previous failing test after implementing the freeze functionality resulted in:

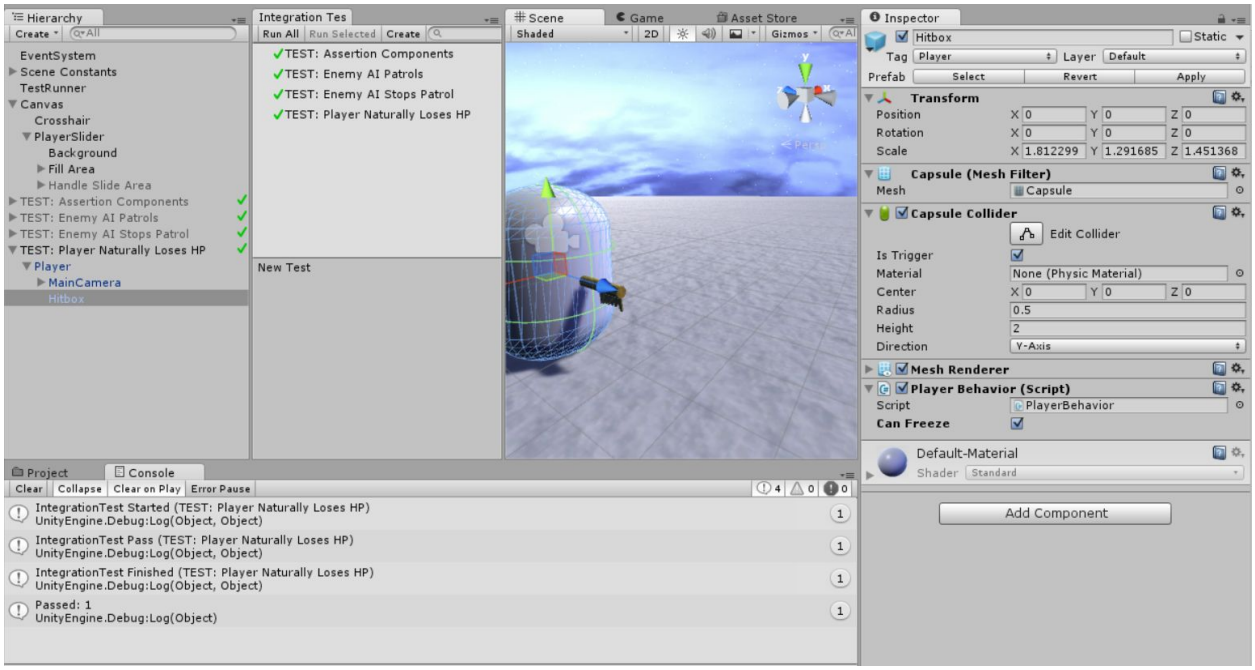

Figure 2.8: Test passes after functionality implemented

Test-driven development was a piece of cake. All I had to do was figure out which Assertion Component was needed to make the test fail or pass. Also, while the feature implementation required code, writing tests for the Integration Test Runner required no coding needed at all. This means game designers can design tests while programmers implement the features. Furthermore, because the Integration Test Runner exists in a scene independent from the main game scene, tests can be written without adding onto the load of the final build.

Unity Test Tools: Unit Test Runner

The final feature of Unity Test Tools is the Unit Test Runner. Unit tests are not for testing prefabs or modules; they directly test code functionality. Just like how an integration test asserts module integrity, a unit test must ensure that a single logical concept, like a function, executes reliably. To make this happen, Unity Test Tools provides NUnit, a unit testing framework, and Unit Test Runner, a feature similar to the Integration Test Runner, but for running unit tests. So, I set out to implement unit tests for *UnitySnowmen*.

Here is an example of two basic unit tests.

```
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Threading;
using NUnit.Framework;
using NSubstitute;

namespace UnitySnowmenUnitTests {

    [TestFixture]
    [Category("My Unit Tests")]
    public class MyUnitTests {

        [Test]
        public void PassingTest() {
            Assert.AreEqual(4, 2+2);
        }
        [Test]
        public void FailingTest() {
            Assert.AreEqual(3, 2+2);
        }


    }

}
```

Figure 3.1: ***BasicUnitTests.cs*** *-- Two basic unit tests.*

These tests do not test functions themselves, but rather basic arithmetic. I place this file inside the “Editor” folder so that it is not compiled into the final build (unit tests, just like integration tests, are not shipped with the final game). Then, by opening up the Unit Test Runner interface on Unity, selecting my unit tests, and pressing “Run All,” I am then showed which tests failed and which passed.

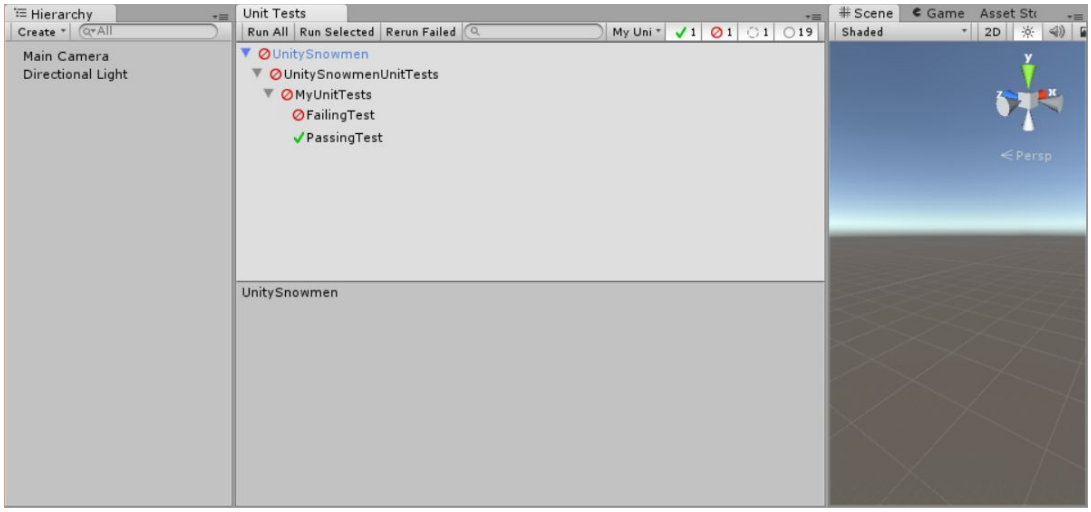

Figure 3.2: Unit tests displayed on the Test Runner interface.

I can click on individual tests to view what happened:

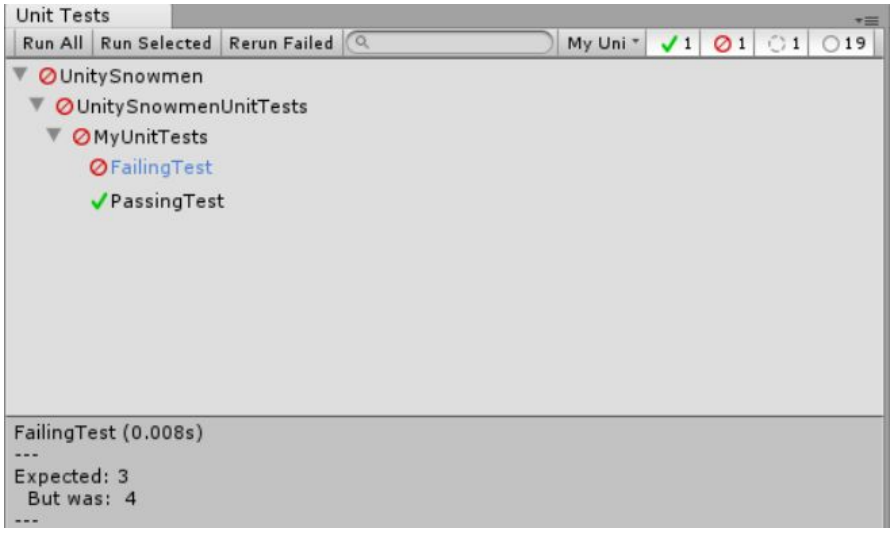

Figure 3.3: Failing Test. This was because I tried to assert that 2+2 expected to be equal to 3

As seen above, unit tests use assertions to test direct code, just like how Assertion Components and integration tests use assertions to test modules. However, unit tests offer slightly more than integration tests in that I can implement mocking to simulate player input. Remember that in the integration tests, I would set up my prefabs under certain circumstances and see how they interacted with one another. However, there was no way for me to create a test where “Fire player bullet and observe what happens.” With unit tests and NUnit, I can create unit tests that substitutes player input to observe if code behaves correctly.

To implement a mocking framework, I followed a Unity blog on unit testing. [5] The blog explained how to test MonoBehaviors. MonoBehaviors are a special class that almost every single Unity script inherits form. Each of my scripts inherits from the MonoBehavior class - For example, the header of EnemyBehavior.cs is shown below:

```
using UnityEngine;
using System.Collections;
using UnityEngine.Assertions;
using UnityEngine.UI;

public class EnemyBehavior : MonoBehaviour {
```

Figure 3.4: Each of my written C# scripts inherits from the MonoBehavior base class.

Unity Test Tools would throw warnings when I tried to test MonoBehaviors directly, because MonoBehaviors are meant to be added as components to existing game objects. Creating game objects as part of a unit test was also unfeasible, because the unit test would then involve creating and destroying game objects, which defeats the purpose of a single-purpose unit test. The Unity blog recommends solving this by using interfaces to extract code that does not use

the Unity API in a way known as the Humble Object Pattern. This is explained via example below.

I wished to test PlayerProjectileShooter.cs, the script responsible for firing player bullets. In this script, pressing spacebar would instantiate a bullet object. First, I created an interface called “IGunController” which referenced the “fire()” method from PlayerProjectileShooter. Then, I created a class in “PlayerController” which calls the “fire()” method from IGunController. Finally, in my unit test file, I use NUnit’s NSubstitute library to create an object that behaves like IGunController. My unit test creates a mock GunController object, executes its “fire()” method, and then checks to see if the fire() method was received by the interface. By using mocking objects and this pattern, the PlayerController class knows nothing about PlayerProjectileShooter (it can only invoke methods from IGunController) and hence we can avoid the issue of directly testing MonoBehaviors.

```
1  using UnityEngine;
2  using System.Collections;
3
4  namespace UnityTest {
5
6
7      public class PlayerProjectileShooter : MonoBehaviour, IGunController {
8
9          private GameObject prefab;
10         private float cd;
11
12         // Use this for initialization
13         void Start () {
14
15         }
16
17         // Update is called once per frame
18         void Update () {
19
20         if (Input.GetKeyDown(KeyCode.Mouse0)) fire();
21
22         }
23
24         #region IGunController implementation
25         public void fire() {
26             prefab = Resources.Load("projectile") as GameObject;
27             GameObject projectile = Instantiate(prefab) as GameObject;
28             projectile.transform.position = transform.position + Camera.main.transform.forward;
29
30             Rigidbody rb = projectile.GetComponent<Rigidbody>();
31
32             rb.velocity = Camera.main.transform.forward * 40; //40
33             Destroy(projectile, 3f);
34
35         }
36         #endregion
37     }
38
39 }
```

Figure 3.5: ***PlayerProjectileShooter.cs*** *-- To test the fire() method, which includes MonoBehaviors, I use a new interface IGunController (See line 7)*

```
using UnityEngine;
using System;

namespace UnityTest
{
    public interface IGunController
    {
        void fire ();
    }
}
```

*Figure 3.6: **IGunController.cs** -- Note that fire() corresponds to the method from PlayerProjectileShooter.cs*

```
using UnityEngine;
using System;

namespace UnityTest {


    [Serializable]
    public class PlayerController{

    private IGunController gunController;


    public void ApplyFire() {
            gunController.fire();
        }

    public void SetGunController (IGunController gunController) {
            this.gunController = gunController;
        }

    }

}
```

*Figure 3.7: **PlayerController.cs** -- This class, using public void ApplyFire(), calls the fire() method from the GunController interface without referencing any monobehaviors. Hence, we can run our unit test against this method.*

```
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Threading;
using NUnit.Framework;
using NSubstitute;

namespace UnityTest {

    [TestFixture]
    [Category("Mocking Unit Tests")]
    public class MockingUnitTests {


        [Test]
        public void GunFireWorks() {
            var gunController = GetGunMock();
            var motor = GetControllerMock(gunController);

            motor.ApplyFire();
            gunController.Received(1).fire();

        }

        private IGunController GetGunMock ()
        {
            return Substitute.For<IGunController>();
        }
        private PlayerController GetControllerMock ( IGunController gunController )
        {
            var motor = Substitute.For<PlayerController>();
            motor.SetGunController (gunController);
            return motor;
        }

    }

}
```

Figure 3.8: ***MockingUnitTests.cs*** *-- The actual unit test. Using NSubstitute, we create a mock GunController object and assert that ApplyFire() causes the controller to receive fire() once. This allows us to test the fire() method without having to deal with MonoBehaviors.*

In the unit test (figure 3.8), GetGunMock and GetControllerMock creates our mock object and then motor.ApplyFire() calls fire, which we then check if fire() was called on the GunController interface using the ".Received()" command. Following this same pattern, also known as "Arrange-Act-Assert," I created additional unit tests, this time mocking the PlayerBehavior class. When I was finished, all of my unit tests were displayed on the Unit Test Runner window.

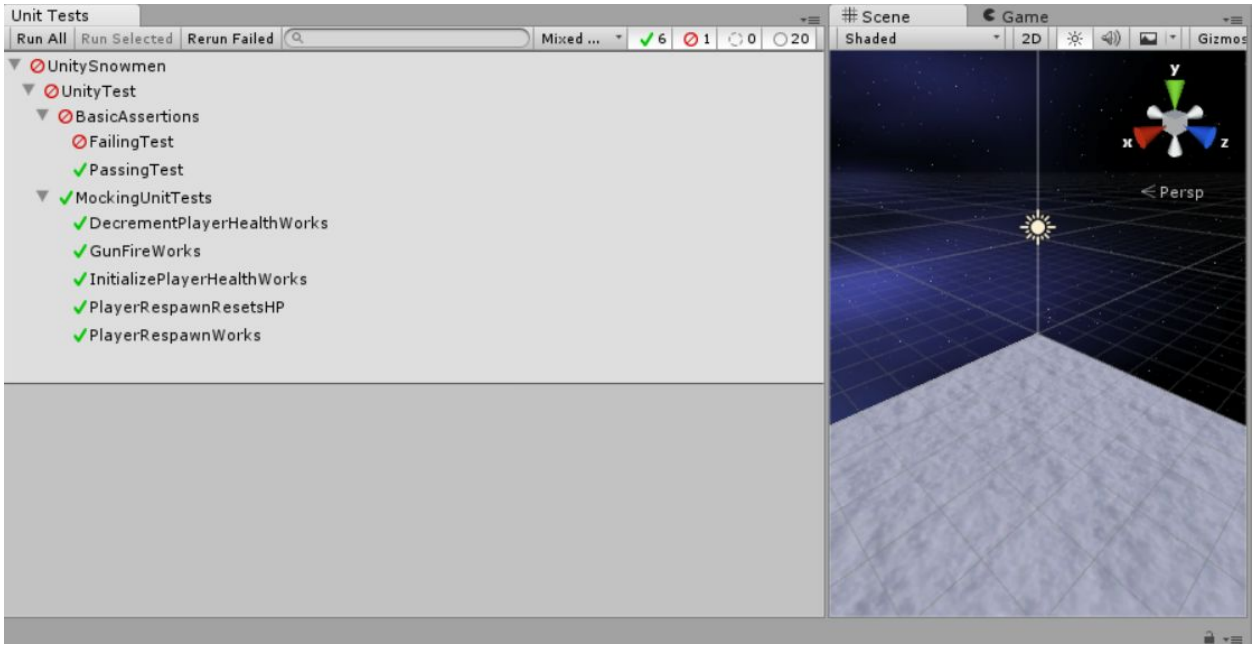

Figure 3.9: List of completed Unit Tests. BasicAssertions refer to the unit tests from figure 3.1.

In addition to checking if the fire() method in PlayerProjectileShooter works, I also wrote tests that asserted the respawn() and decrementPlayerHealth() methods in the PlayerBehavior script. These new tests also followed the “Humble Object Pattern” described above for mocking MonoBehavior functionality.

Overall, I was able to succeed in writing unit tests for *UnitySnowmen*. Although I had to learn how C# interfaces and classes worked, the Unit Test Runner documentation and official Unity blog posts guided me towards implementing my own substitutions for mocking MonoBehaviors. I definitely spent the most time implementing these unit tests as compared to inserting Assertion Components or assembling integration tests, but this allowed me to understand the importance of directly testing game code.

Conclusion

Unity Test Tools is extremely useful as a testing framework. Writing integration tests was extremely manageable given the many comparers and testing parameters offered by the Assertion Component. Despite having very little C# knowledge prior to this project, the Unity Test Tools documentation allowed me to successfully implement unit tests that bypassed MonoBehaviors.

Because Assertion Components and the Integration Test Runner do not require coding knowledge, Unity Test Tools opens up testing to all members of a game development team. For software engineers and coders who are accustomed to writing unit tests, the Unit Test Runner is a very solid framework for designing and running multiple unit tests. With testing and test-driven development becoming more and more crucial to software development, Unity Test Tools is perfect for developers and their teams to apply proper and cohesive testing practices to their games.

References

1. “Unity - Multiplatform - Publish your game to over 10 platforms.” https://unity3d.com/unity/multiplatform
2. “Unity - Fast Facts.” https://unity3d.com/public-relations
3. “Test-Driven Development.” https://en.wikipedia.org/wiki/Test-driven_development
4. “Unit Testing and Test-Driven Development in Unity3D”. Fray, Andrew. http://www.slideshare.net/tenpn/test-driven-development-25476304
5. “Unit Testing Part 2- Unit Test MonoBehaviors.” Paszek, Tomek. http://blogs.unity3d.com/2014/06/03/unit-testing-part-2-unit-testing-monobehaviours/

Project: *UnitySnowmen*

Public Repository: https://github.com/rzeng95/UnitySnowmen

All assets used are credited in *Credits List.txt*

The game itself is found in the scene: */Assets/UnitySnowmen.unity*

The integration tests are found in the scene: */Assets/UnitySnowmenIntegrationTests.unity*

The unit tests used are found in: */Assets/Editor*