Creating unit tests in Unity

by

Matthew Priem
Amanda Long
Michael Johnson
Blake Johnson

Minnesota State University, Mankato

Spring 2021

Contact us at
matthew.priem@mnsu.edu
amanda.long@mnsu.edu
michael.johnson-10@mnsu.edu
blake.johnson@mnsu.edu

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

When it comes to unit testing in the Unity Engine, there are some problems you will more than likely encounter on your initial attempt. The first thing to consider, which is probably the most important condition, is that unit tests cannot be run on MonobBehaviours. For those unfamiliar with MonoBehaviour, it is the base class from which every Unity script derives1. The MonoBehaviour class provides the framework which allows one to attach their scripts to a GameObject in the Unity Editor2. GameObject is a class which represents the actual physical objects which are placed within a Unity scene, that make up the contents of game environments. The main reason why unit tests cannot be run is essentially because there is no way to instantiate MonoBehaviuors, which would prevent any tests from running completely. Because unit tests cannot be run directly on MonoBehaviours, there are some additional steps which need to be taken when designing the unit test.

CHAPTER 2

Setup for tests in Unity

2.1 Setup for tests in EditMode

We are going to show how to setup tests in EditMode. In addition, we are going to show how to setup NSubstitute within Unity in order to run all of the example tests in later chapters. Setting up NSubstitute is not mandatory to run tests in unity, but in later chapters we will see the benefits of doing so.

In order to download NSubstitute navigate to the nuget page Here, and download version 2.0.3. Table 2.1 show a complete list of versions that work with Unity. Once downloaded, extract the files using an unzip app (such as 7zip). Follow the path nsubstitute.2.0.3/lib/net35 to locate the NSubstitute.dll file. In order to NSubstitute in Unity in *must* be in a folder titled Plugins. Go to Assets folder and create a subfolder named Plugins. Drag and drop the NSubstitute.dll file from your explorer directly into the Plugins folder. If done correctly it should look similar to Figure 2.1.

EditMode tests *must* be in a folder named Editor. This does not need to be a child of the Assets folder, but it must be a decedent. Create a folder named Editor. We will create tests here later. If done correctly, it should look similar to Figure 2.2.

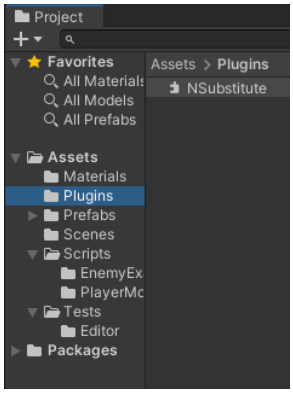


Figure 2.1: Assets/Plugins/NSubstitute

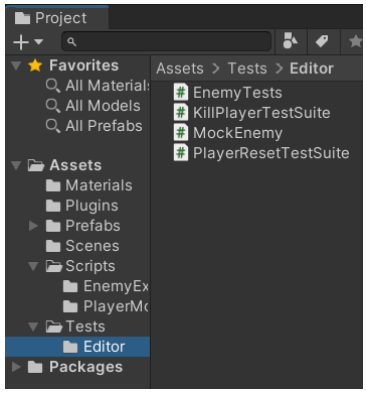


Figure 2.2: Assets/Tests/Editor

Finally, we will open the Test Runner window in Unity. To do so select Window → General → Test Runner. That's it. We will be ready to start creating and running unit tests in EditMode in the next chapter.

2.2 Setup for tests in PlayMode

asdf

Version		Works with Unity
2.0.3	net35	✓
2.0.3	net40	?
2.0.3	net45	?
2.0.3	netstandard 1.3	?
4.2.2	net45	✗
4.2.2	net46	✗
4.2.2	netstandard 1.3	✗
4.2.2	netstandard 2.0	✗

Table 2.1: Versions of NSubstitute that work with Unity
https://www.nuget.org/packages/NSubstitute/2.0.3

CHAPTER 3

CREATE TESTS IN UNITY

3.1 Create a test in EditMode

3.1.1 The problem

As mentioned in Chapter 1, we cannot directly instantiate a MonoBehaviour. Let's look at an example that illustrates that the issue with testing a class that inherits from MonoBehaviour. In Figure 3.1 we have a class named Enemy that moves a gameobject along a path. There is nothing wrong with this script and will work as intended. The issue comes when trying to test the MoveEnemy method. If we try to write a test such as in Figure 3.2, we would find that the enemy on line 37 of Figure 3.2 will always be null. In fact, the Test Runner will give the warning, "You are trying to create a MonoBehaviour using the 'new' keyword. This is not allowed. MonoBehaviours can only be added using AddComponent(). Alternatively, your script can inherit from ScriptableObject or no base class at all." Since enemy is Null, then enemy.gameObject.transform.position in line 43 of Figure 3.2 will also be null, and the assertion will fail.

```
Unity Script | 2 references
public class Enemy : MonoBehaviour
{
    float timer = 0;

    Unity Message | 0 references
    void Update()
    {
        timer += Time.deltaTime * 4;

        MoveEnemy(timer);

        if (timer >= 8 * 4)
            timer = 0;
    }

    1 reference
    private void MoveEnemy(float timer)
    {
        if (timer >= 0 && timer <= 8)
            transform.position = new Vector3(timer - 4, 1.58f, -4);
        else if (timer > 8 && timer <= 16)
            transform.position = new Vector3(4f, 1.58f, timer - 4 - 8 * 1);
        else if (timer > 16 && timer <= 24)
            transform.position = new Vector3(-(timer - 4 - 8 * 2), 1.58f, 4f);
        else if (timer > 24 && timer <= 32)
            transform.position = new Vector3(-4f, 1.58f, -(timer - 4 - 8 * 3));
    }
}
```

Figure 3.1: Enemy script

```
33      [Test]
        0 references
34      public void NonworkingTest()
35      {
36          //Arrange
37          Enemy enemy = new Enemy();
38
39          //Act
40          enemy.MoveEnemy(4f);
41
42          //Assert
43          Assert.AreEqual(new Vector3(0f, 1.58f, -4f), enemy.gameObject.transform.position);
44      }
```

Figure 3.2: Test that will not work

3.1.2 Humble Pattern

One way to fix this issue is with the humble object pattern. The humble object pattern is designed to decouple the logic we want to test from the dependencies that are difficult to test. In our example, we want to decouple the movement logic from the MonoBehaviour. We will make a humble object that inherits (has dependencies) from MonoBehaviour and implements a class that controls the component we want to test.

We can see from Figure 3.3 that the test does have dependencies from MonoBehavior. However, in Figure 3.4 actual logic that we want to test (EnemyMover) is no longer aware of MonoBehaviour.

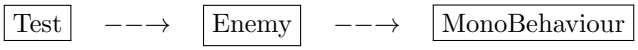


Figure 3.3: Class diagram without Humble Pattern

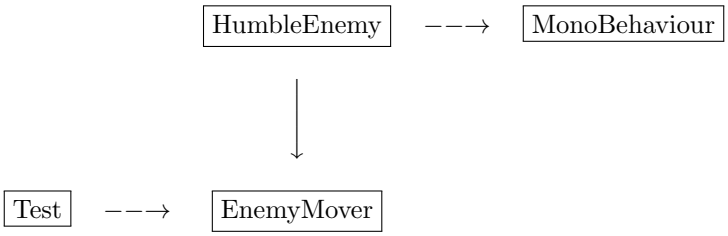


Figure 3.4: Class diagram with Humble Pattern

Applying the logic from the humble pattern class diagram (see Figure 3.4), we will extract the MoveEnemy method from the Enemy class to its own class. Let's call this class EnemyMover. We will then rename the Enemy class to HumbleEnemy. The result is that the Update method in HumbleEnemy (Figure 3.5) is calling the MoveEnemy method in EnemyMover (Figure 3.6), thus decoupling the move logic from MonoBehaviour.

```
17      void Update()
18      {
19          timer += Time.deltaTime * 4;
20
21          enemyMover.MoveEnemy(timer);
22
23          if (timer >= 8 * 4)
24              timer = 0;
25      }
```

Figure 3.5: Update method of HumbleEnemy

```
        0 references
7       public void MoveEnemy(float timer)
8       {
9           if (timer >= 0 && timer <= 8)
10              transform.position = new Vector3(timer - 4, 1.58f, -4);
11          else if (timer > 8 && timer <= 16)
12              transform.position = new Vector3(4f, 1.58f, timer - 4 - 8 * 1);
13          else if (timer > 16 && timer <= 24)
14              transform.position = new Vector3(-(timer - 4 - 8 * 2), 1.58f, 4f);
15          else if (timer > 24 && timer <= 32)
16              transform.position = new Vector3(-4f, 1.58f, -(timer - 4 - 8 * 3));
17      }
```

Figure 3.6: MoveEnemy method in the EnemyMover class

This is almost testable. Notice in Figure 3.6 there is an error in the MoveEnemy method. By decoupling MoveEnemy from MonoBehaviour, we have broken the reference to the transform (as transform derives from MonoBehaviour). We can fix this with an interface and mocking.

Notice that we are trying to change the transform.position, but MoveEnemy no longer has reference to a gameObject (since it is has not been passed one and cannot be attached to one in the editor). The HumberEnemy will be attached to a gameObject which has a transform. However, it inherits from MonoBehaviour, so we cannot instantiate it directly in our test. Notice that the MonoBehaviour and therefore HumberEnemy has many dependencies that we do not need for our test. All we need is a transform.postion, which is a Vector3. We can pass that Vector3 from an interface, implement the interface in the HumberEnemy, and mock the interface in the test. This can be seen implemented in Figure 3.7, Figure 3.8, and Figure 3.9.

```
5    public interface IPosition
6    {
         9 references
7        Vector3 Position { get; set; }
8    }
9
```

Figure 3.7: Interface for Vector3 i.e. transform.position

```
4
     Unity Script | 0 references
5    public class HumbleEnemy : MonoBehaviour, IPosition
6    {
7        EnemyMover enemyMover;
8
9        float timer = 0;
10
         Unity Message | 0 references
11       private void Start()
12       {
13           enemyMover = new EnemyMover(this);
14       }
15
         9 references
16       public Vector3 Position
17       {
18           get { return transform.position; }
19           set { transform.position = value; }
20       }
21
         Unity Message | 0 references
22       void Update()
23       {
24           timer += Time.deltaTime * 4;
25
26           enemyMover.MoveEnemy(timer);
27
28           if (timer >= 8 * 4)
29               timer = 0;
30       }
31
```

Figure 3.8: HumbleEnemy class

```
9 references
public class EnemyMover
{
    IPosition _enemyPosition;

    4 references
    public EnemyMover(IPosition position)...

    4 references
    public void MoveEnemy(float timer)
    {
        if (timer >= 0 && timer <= 8)
            _enemyPosition.Position = new Vector3(timer - 4, 1.58f, -4);
        else if (timer > 8 && timer <= 16)
            _enemyPosition.Position = new Vector3(4f, 1.58f, timer - 4 - 8 * 1);
        else if (timer > 16 && timer <= 24)
            _enemyPosition.Position = new Vector3(-(timer - 4 - 8 * 2), 1.58f, 4f);
        else if (timer > 24 && timer <= 32)
            _enemyPosition.Position = new Vector3(-4f, 1.58f, -(timer - 4 - 8 * 3));
    }
}
```

Figure 3.9: MoveEnemy method with interface

Let us quickly review how we successfully decoupled our code. The transfrom.position of the HumbleEnemy is got and set using the interface. The EnemyMover then changes the values of the transform.position through the interface. The only dependency of HumbleEnemy that EnemyMover knows about is the interface. This means that if we want to test the MoveEnemy method inside the EnemyMover class, all we need to do is implement that interface in our test instead of MonoBehaviour.

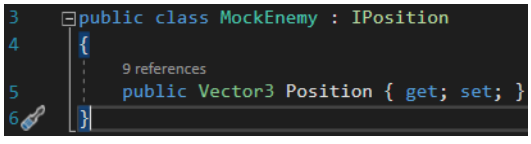
One problem still remains. We cannot instantiate an interface directly. For example, if we write

$$\text{IPosition enemyPosition} = \text{new IPosition}()$$

it will fail. We can fix this with mocking. We will look at two methods for mocking. The first is mocking by writing a stub that implements the interface. The second method will be using NSubstitute.

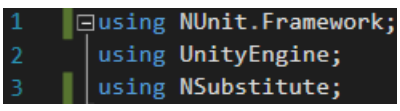
3.1.3 Mocking

To mock the interface by writing a stub, we can write a class that implements the interface, and hard code the values. In our example we will set the values in the test using the interface's setter. Since this stub is mocking the transform.position (i.e. Vector3) of the enemy, we will call it MockEnemy (see Figure 3.10). If we wanted to test more components of our enemy, we could add them here, and use this MockEnemy for multiple tests. However, we'll cover that more in–depth later.



```
public class MockEnemy : IPosition
{
    9 references
    public Vector3 Position { get; set; }
}
```

Figure 3.10: MockEnemy



```
using NUnit.Framework;
using UnityEngine;
using NSubstitute;
```

Figure 3.11: Using statements for tests

We can now write our first unit test. We will replace new IPosition() with new MockEnemy(), instantiate an EnemyMover class, move the enemy, and see if our test works. The actual code for this can be seen in Figure 3.12. Note: in order to use the [Test] attribute we will have to add the using statement NUnit.Framework (see Figure 3.11 line 1).

```
8       [Test]
        0 references
9       public void HumblePatternTest()
10      {
11          //Arrange
12          IPosition enemyPosition = new MockEnemy();
13          EnemyMover enemyMover = new EnemyMover(enemyPosition);
14
15          //Act
16          enemyMover.MoveEnemy(4f);
17
18          //Assert
19          Assert.AreEqual(new Vector3(0f, 1.58f, -4f), enemyPosition.Position);
20      }
```

Figure 3.12: Testing MoveEnemy()

In the HumblePatternTest() (see Figure 3.12) we are testing the MoveEnemy method from the EnemyMover class. We are passing the value 4f to the function and are asserting that this should result in the Vector3 equal to (0f, 1.58f, -4f). If our assertion is true, the test will pass. If it is false, our assertion will fail. However, in either case we have a functional unit test.

Alternatively, we can mock the IPosition using NSubstitute. We do so with line 26 of Figure 3.13. The rest of the this test can remain the same.

```
22      [Test]
        0 references
23      public void NSubEnemyTest()
24      {
25          //Arrange
26          IPosition enemyPosition = Substitute.For<IPosition>();
27          EnemyMover enemyMover = new EnemyMover(enemyPosition);
28
29          //Act
30          enemyMover.MoveEnemy(4f);
31
32          //Assert
33          Assert.AreEqual(new Vector3(0f, 1.58f, -4f), enemyPosition.Position);
34      }
```

Figure 3.13: Test using NSubstitute

We will show how to run these tests using the Unity Test Runner in the next chapter.

3.1.4 Benefits of NSubstitute

Based on the two previous examples, it may not be clear the benefits of using NSubstitute. Suppose our previous enemy had an internal state that determined the how quickly it moved. For instance, standing could have a speed of 0, walking could have a speed of 1, and running could have a speed of 3. There might be a method in the IPosition interface that, when implemented in HumbleEnemy, returns the speed of the enemy. If we wanted to write a MockEnemy stub with a hard–coded value for the speed, we would need to write 3 different stubs, one for each each speed. We would have three different tests, each with a different MockEnemy.

We could consolidate the three stubs into one using the returns() function in NSubstitute. In each of our tests we could write something similar to

```
enemyPosition.GetSpeed().returns(0);
enemyPosition.GetSpeed().returns(1);
enemyPosition.GetSpeed().returns(3);
```

We will still need 3 tests, one for each speed, but we no longer have to create individual stubs for specific speeds. The general idea is that as the code becomes more complex, writing individual stubs may become repetitive. One use of NSubstitute allows for reduction of repetitive code.

3.2 Create a test in PlayMode

Available for further contribution.

3.3 Test vs UnityTest

Available for further contribution.

CHAPTER 4

Run tests in Unity

4.1 Run tests in EditMode

To run a test in EditMode, first select it, and then push the Run Selected Button. That's it. If a test fails it will show a red circle with a line through it. If a test passes, it will have a green check mark. Additionally, all tests can be run at once by selecting Run All.

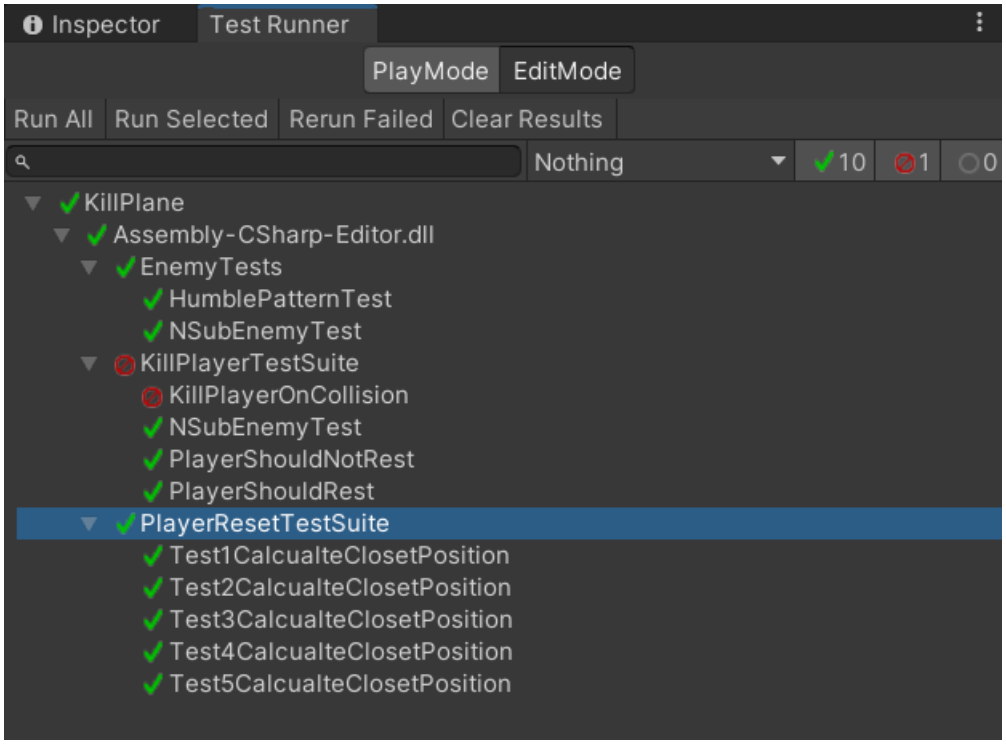


Figure 4.1: EditMode Test Runner

4.2 Run tests in PlayMode

CHAPTER 5

The End

5.1 Conclusion / Sum–up

Now you should be able to write your first test in Unity.

5.2 Continuing the documentation

5.2.1 Further exploration

Benefits of NSubstitute can be further flushed out.

[Test] vs. [UnityTest]

1. Describe difference.
2. Give example. Show a test that works as a UnityTest and fails as a Test. Possible examples could be
   (a) Destroy a gameObject. Hypothesized reason UnityTest will pass and Test will fail:
      i. Test is of type void and will execute code immediately.
      ii. UnityTest is of type IEnumerator which will yield return. This means we can use something similar to yield return WaitForEndOfFrame() as part of the Act.
      iii. Unity waits until the end of a frame to destroy a gameObject, so it is hyposthsized that the Test will fail since all the code in Test will run prior to the frame ending. This is also why the UnityTest will likely pass. It can wait to assert until after yield return WaitForEndOfFrame() has ended the frame.

5.2.2 Questions we don't know the answers to

What happens if two components trigger at the same time when the game is running?
ex. what if two colliders enter a OnTriggerEnter() on the same frame. How does Unity decide which methods run first, and will that effect the outcome of the tests?

Which versions of NSub work with Unity? If they don't work, why not? Is it missing assembly references? If so, which ones? Can those be easily fixed? What are the differences between versions of NSub? Would there be benefits in creating the missing assembly references, so newer versions can be used? If so, what are they?

In this document we are testing scripts that derive from MonoBehaviour. Scriptable objects can be instantiated directly, so some of the above steps may not be required. We don't know if this documentation will work the same for scriptable objects.