Testing

TL3
Brent Knopp
Cole Vanderlaan
Khadeeja Mansour

What is testing?

- Checking if a program does what is is expected to
- Helps to find issues and limits of a program
- Can show the presence of bugs, but not their absence

Importance of testing

- Improves quality of the program to be more:
  - User friendly
    - Better experience
  - Efficient
    - Less errors later on
  - Professional
    - Reliable
    - Better performance

Levels of testing

- Unit testing
- Integration testing
- Functional testing
  - Boundary
- Performance testing
  - Stress
- User testing
  - Alpha
  - Beta
  - Acceptance

Unit testing

- Used for separately testing specific parts of a program to ensure they each work properly

Integration testing

- Tests how different parts of the program interact with each other
- May show issues with the program that did not appear with unit testing

Functional testing

- Used to show the program functions how it is supposed to
- Does the program meet the customer’s expectations?

Boundary testing

- A type of functional testing used to handle cases of an input being outside of a program’s boundaries
- System should not break, but gracefully find solution

Performance testing

- Used to see how a program performs with varying workloads

Stress testing

- A type of performance testing where stress is incrementally added to track where a program breaks

User testing

- Users and customers give advice to improve testing and the program

Alpha testing

- Users communicate with development team to see issues in the software

Beta testing

- The game is available for some users to test and give feedback on any issues

Acceptance testing

- Customers use the software and decide if it is ready for developers to accept it

Maximize testing performance

- Create tests as you code
- Create tests for each part of the program
- Make tests independent of each other
- Tests should be automated

Unity Testing

CS 383 Project Testing Requirements

- Each team member will be responsible for testing their feature:

  Initial Test Plan: 1 stress test and 2 boundary tests (due: October 19)

  Full Test Plan: Oral Exam (November 12 - November 17)

- TL3 role:

  Give tools and information to create individual Unity tests

  Facilitate a testing structure for each team’s project

  Oversee that adequate testing metrics are being met throughout the project

Initial Test Plan (Due October 19)

- **2 Boundary Test**

  A test to check if a value is in some specific range.

  Ex. checking to see if a player went off of the game board

- **1 Stress Test**

  A test that applies a stressor to a system in a incremental method until something breaks.

  Ex. spawning enemies until the game crashes

Unity Test Runner Tool

- The Unity Test Runner is a Unity tool that tests your code in both Edit mode and Play mode, and also on target platforms such as StandAlone, Android, or iOS.
- **Edit Mode:**
    - Tests that do not need an actual live scene. This is primary used for unit testing.
- **Play Mode:**
    - Tests allow you to exercise the actual game game simulation in real time and will verify that it runs as desired. Test code will be performed in an active unity scene.
- Play Mode is slower than Edit Mode

Edit Mode Test

- Used to make Unit Test, Integration Test, and Functional Test
- Recommended for a simple Boundary Test
- OverView:

  Edit Mode Tests run in the editor, and occurs when the game is not started. You can use this mode when you do not need the game to be running. The advantage is of Edit Mode Tests are significantly faster than Play Mode Tests.

- Edit Mode Test is simpler to construct compared to Play Mode Test.
- Uses a script with an assertion class to check to see if certain conditions have been met in the game.

Edit Mode Test Examples:

- Standard Unit Test when you want to test a single unit without the need to access anything else
- Integration tests that do not require running game
- Check Scenes, Game Objects and their Components
- Check Scriptable Objects or other files that you use
- Do not need an active scene to test

Play Mode Test

- A real time application test that allows your game code to be executed with a coroutine test.
- The main difference between the Play Mode test and the Edit Mode Test is that the play mode tests are executed in the application's runtime environment where the callback methods of MonoBehaviours are executed.

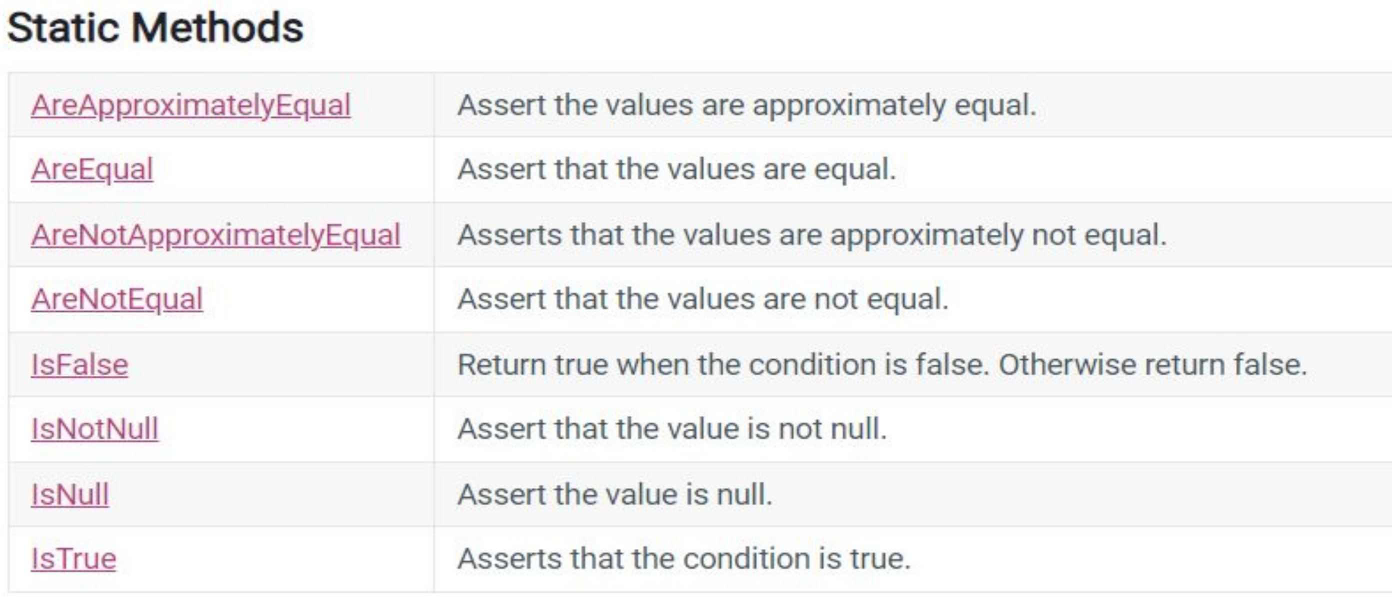
  Ex. check scripts that require runtime (Update, FixedUpdate, etc)

- Edit mode test can not use methods with MonoBehaviors.
- Both modes use the assertion class in a test script to evaluate code conditions.

Unity Assertion Class

- Assertion Class is used for writing Unity test

https://docs.unity3d.com//ScriptReference/Assertions.Assert.html

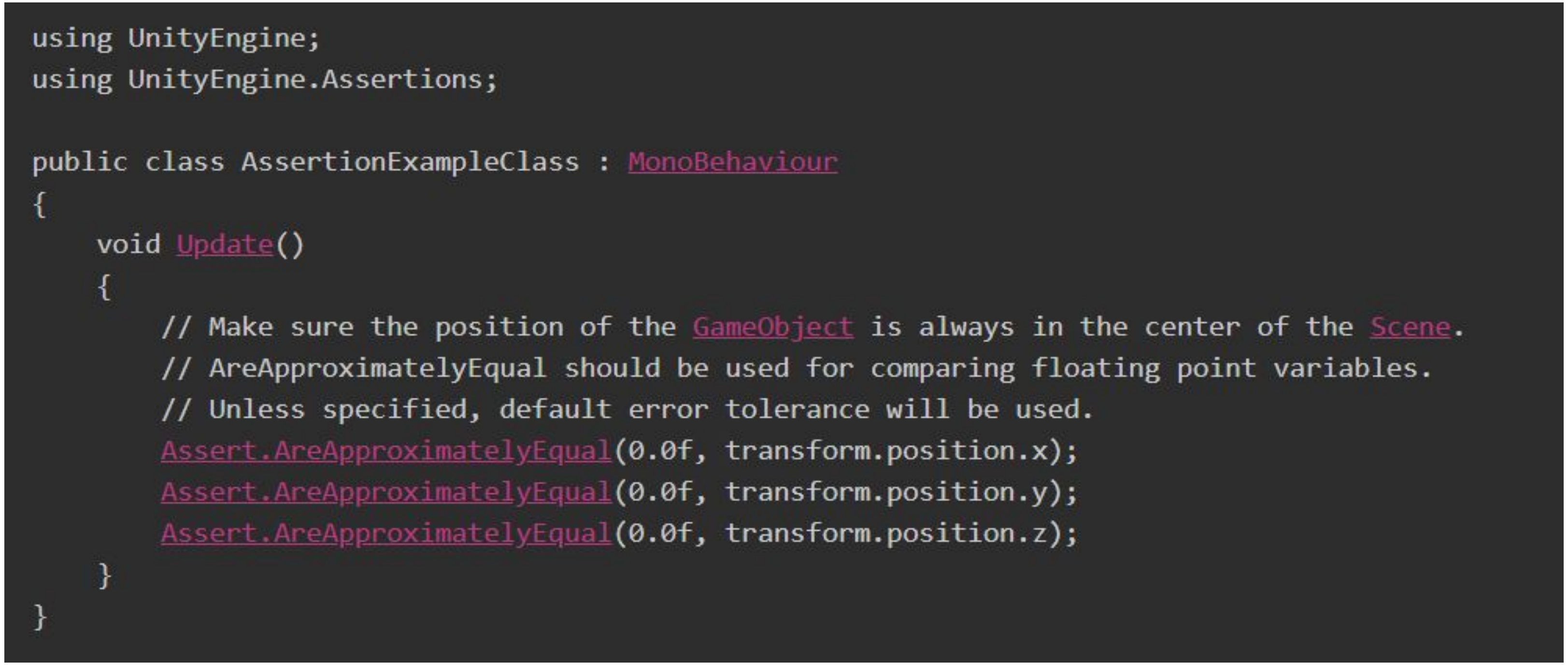


Static Methods

AreApproximatelyEqual	Assert the values are approximately equal.
AreEqual	Assert that the values are equal.
AreNotApproximatelyEqual	Asserts that the values are approximately not equal.
AreNotEqual	Assert that the values are not equal.
IsFalse	Return true when the condition is false. Otherwise return false.
IsNotNull	Assert that the value is not null.
IsNull	Assert the value is null.
IsTrue	Asserts that the condition is true.

Assert.AreApproximatelyEqual

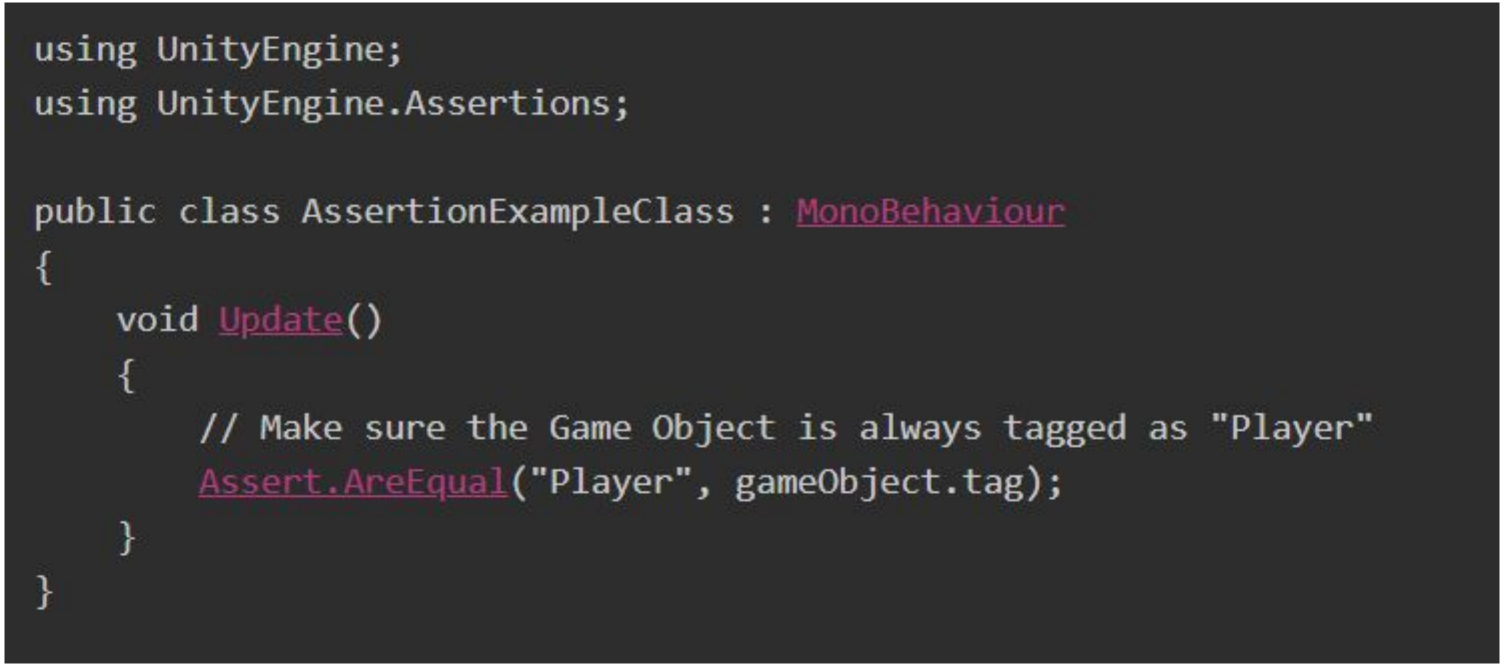
- Assert the values are approximately equal
- Default `tolerance` is 0.00001f.



```csharp
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    void Update()
    {
        // Make sure the position of the GameObject is always in the center of the Scene.
        // AreApproximatelyEqual should be used for comparing floating point variables.
        // Unless specified, default error tolerance will be used.
        Assert.AreApproximatelyEqual(0.0f, transform.position.x);
        Assert.AreApproximatelyEqual(0.0f, transform.position.y);
        Assert.AreApproximatelyEqual(0.0f, transform.position.z);
    }
}
```

Assert.AreEqual

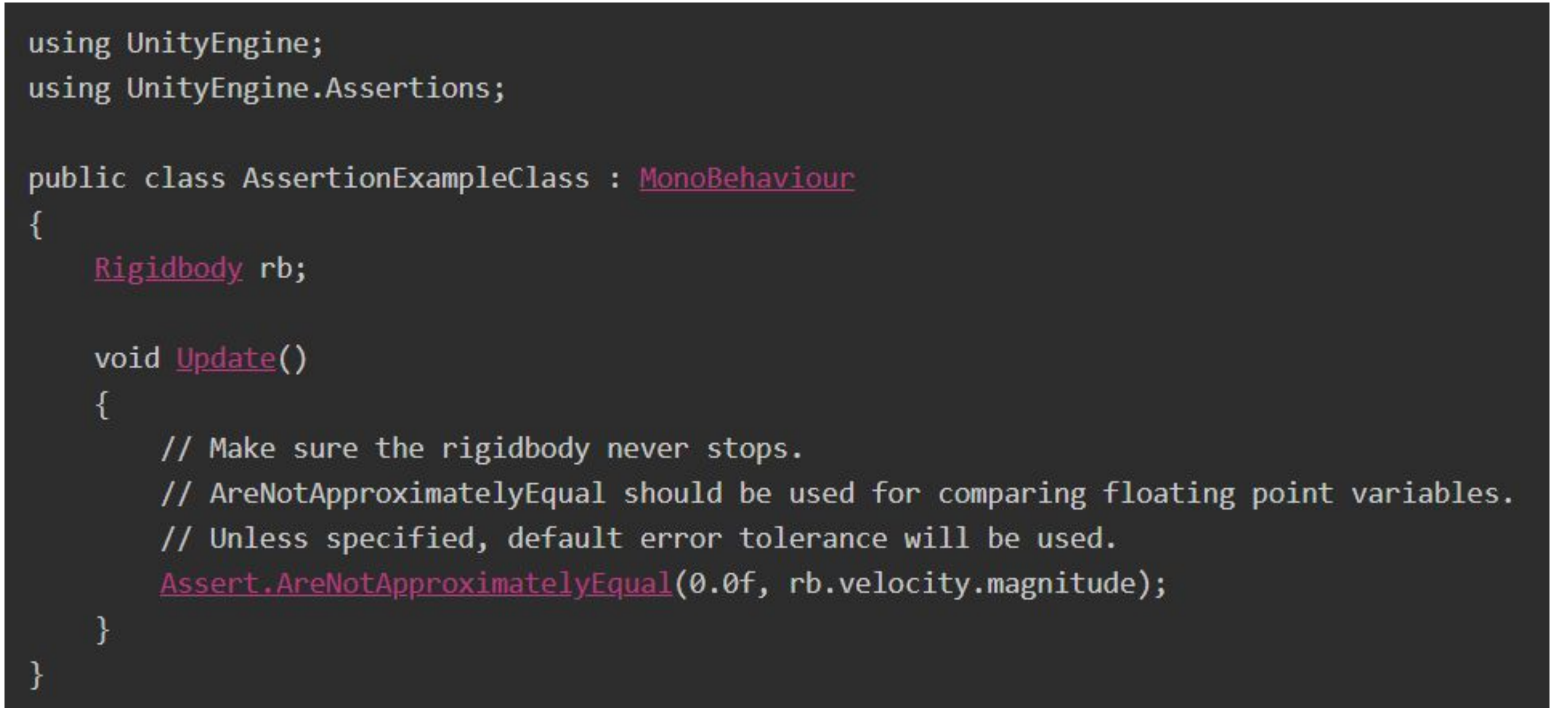
- Assert that the values are equal



```
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    void Update()
    {
        // Make sure the Game Object is always tagged as "Player"
        Assert.AreEqual("Player", gameObject.tag);
    }
}
```

Assert.AreNotApproximatelyEqual

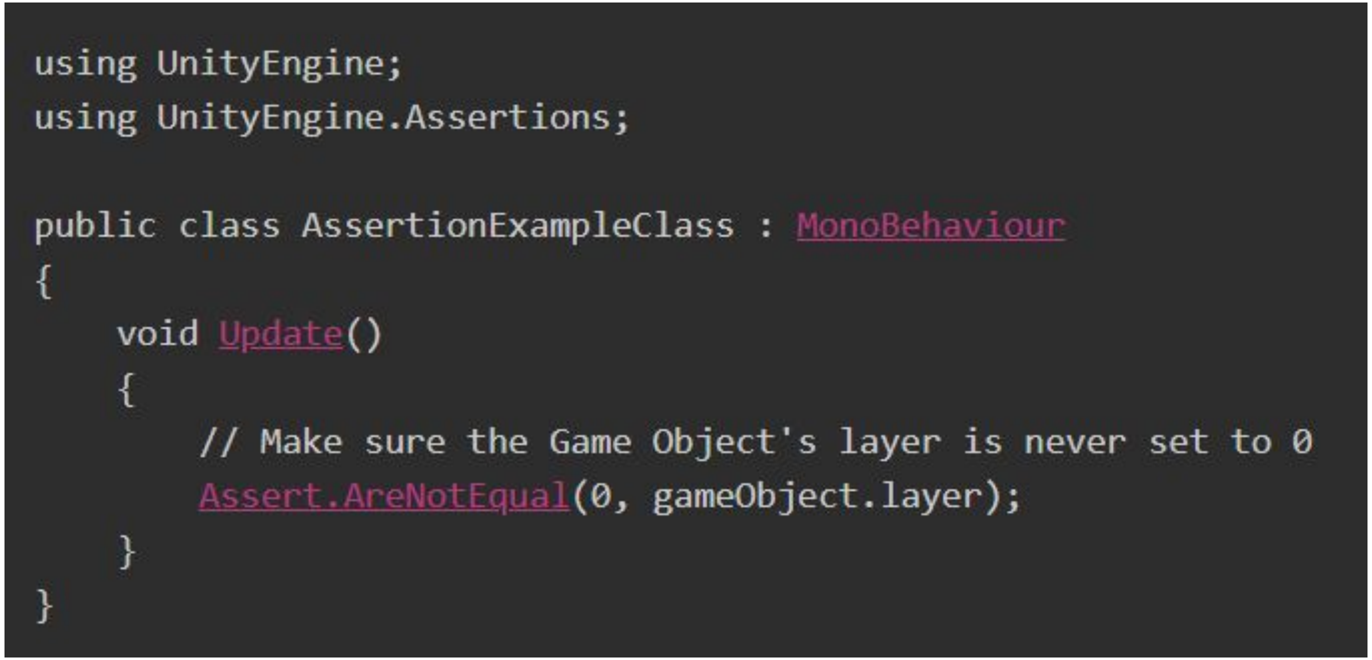
- Asserts that the values are approximately not equal
- Default tolerance is 0.00001f



```
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    Rigidbody rb;

    void Update()
    {
        // Make sure the rigidbody never stops.
        // AreNotApproximatelyEqual should be used for comparing floating point variables.
        // Unless specified, default error tolerance will be used.
        Assert.AreNotApproximatelyEqual(0.0f, rb.velocity.magnitude);
    }
}
```

Assert.AreNotEqual

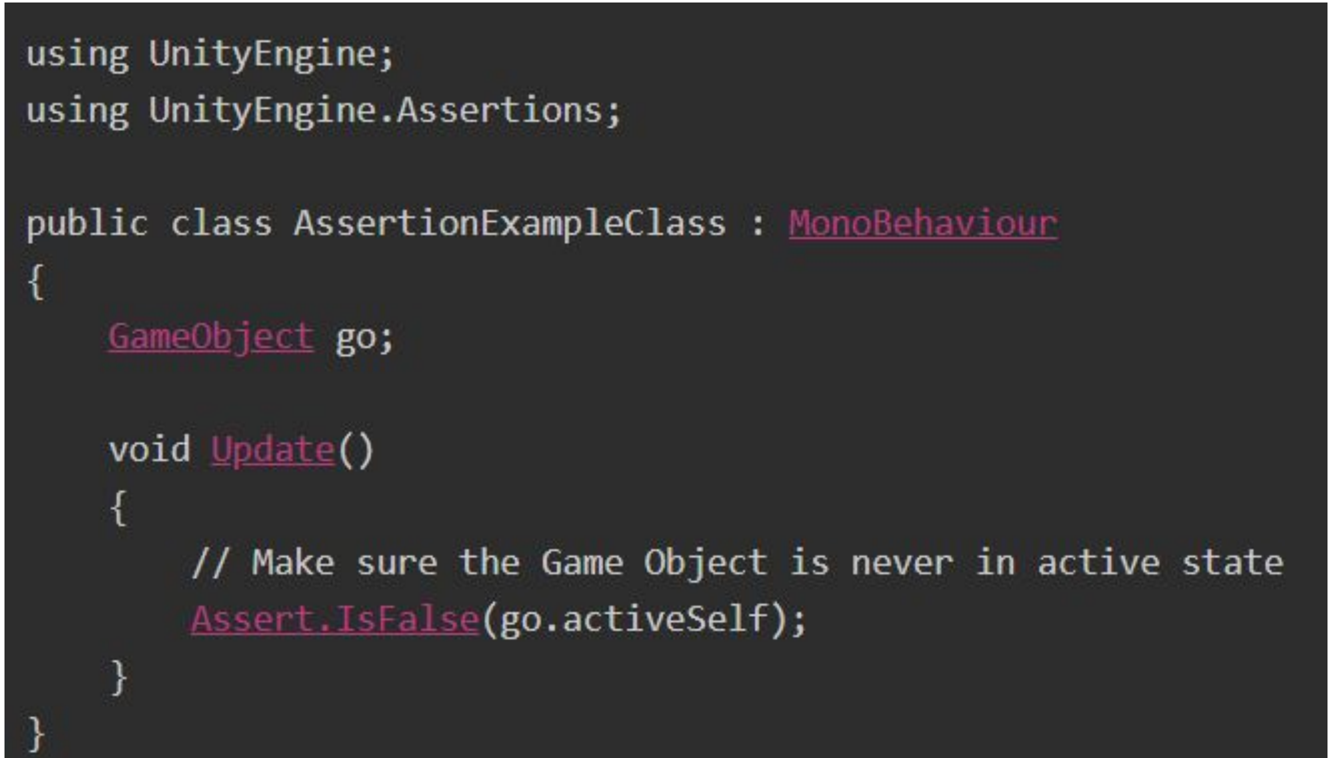
- Assert that the values are not equal



```
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    void Update()
    {
        // Make sure the Game Object's layer is never set to 0
        Assert.AreNotEqual(0, gameObject.layer);
    }
}
```

Assert.IsFalse

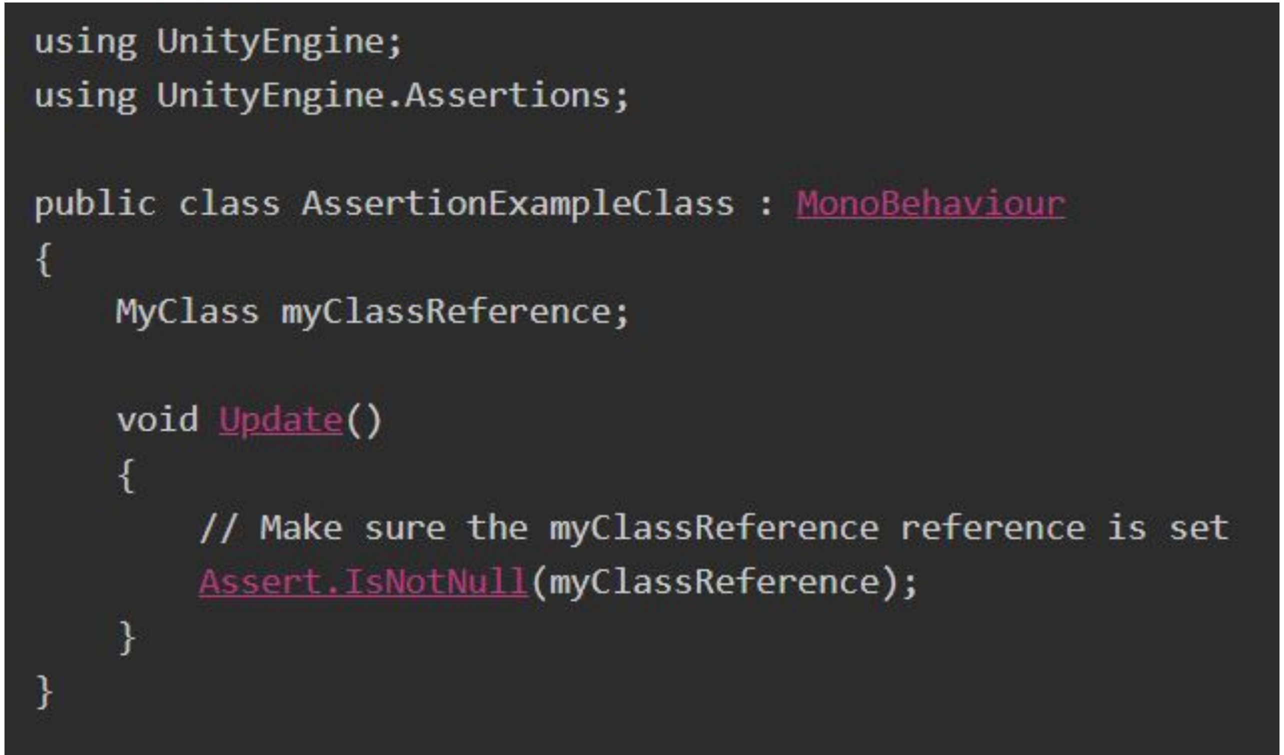
- Return `true` when the condition is false. Otherwise return `false`



```
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    GameObject go;

    void Update()
    {
        // Make sure the Game Object is never in active state
        Assert.IsFalse(go.activeSelf);
    }
}
```

Assert.IsNotNull

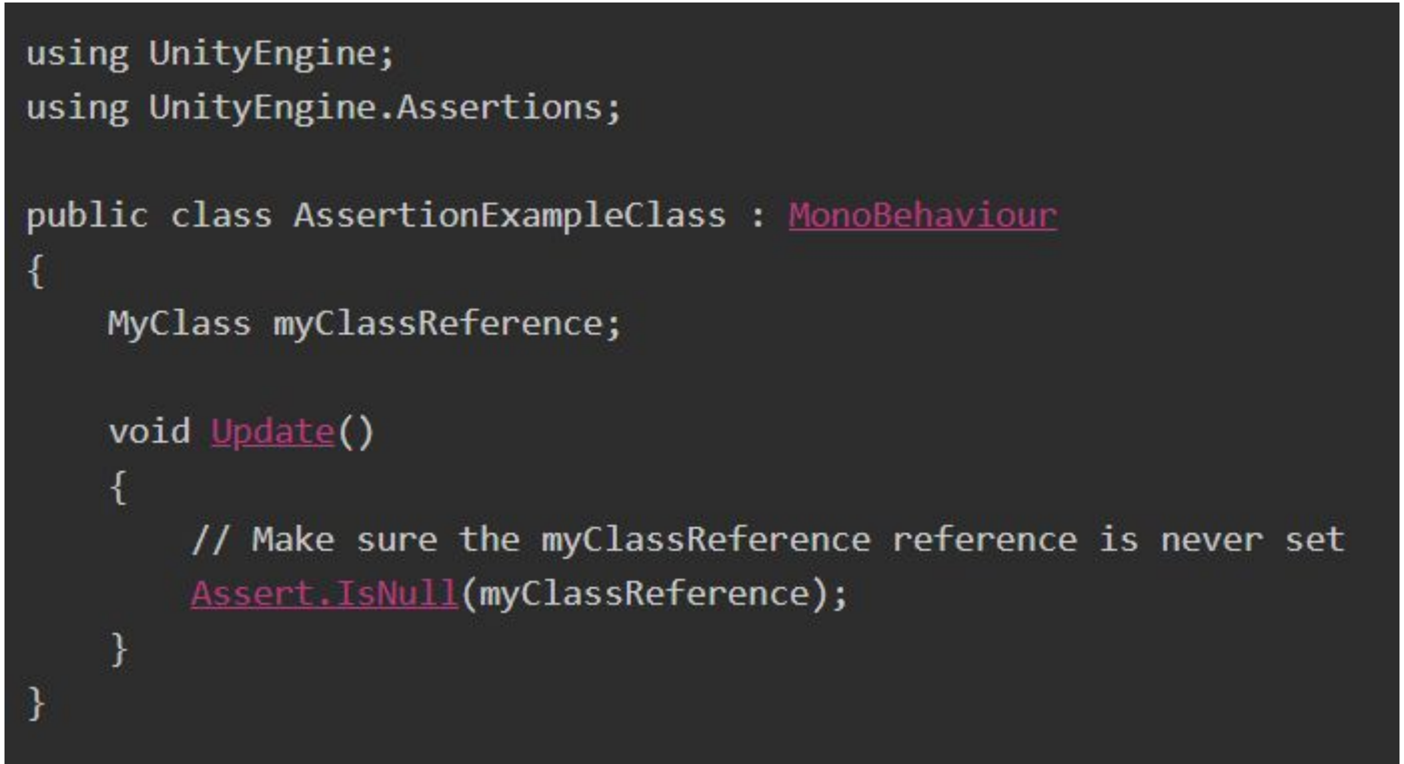
- Assert that the value is not null



```
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    MyClass myClassReference;

    void Update()
    {
        // Make sure the myClassReference reference is set
        Assert.IsNotNull(myClassReference);
    }
}
```

Assert.IsNull

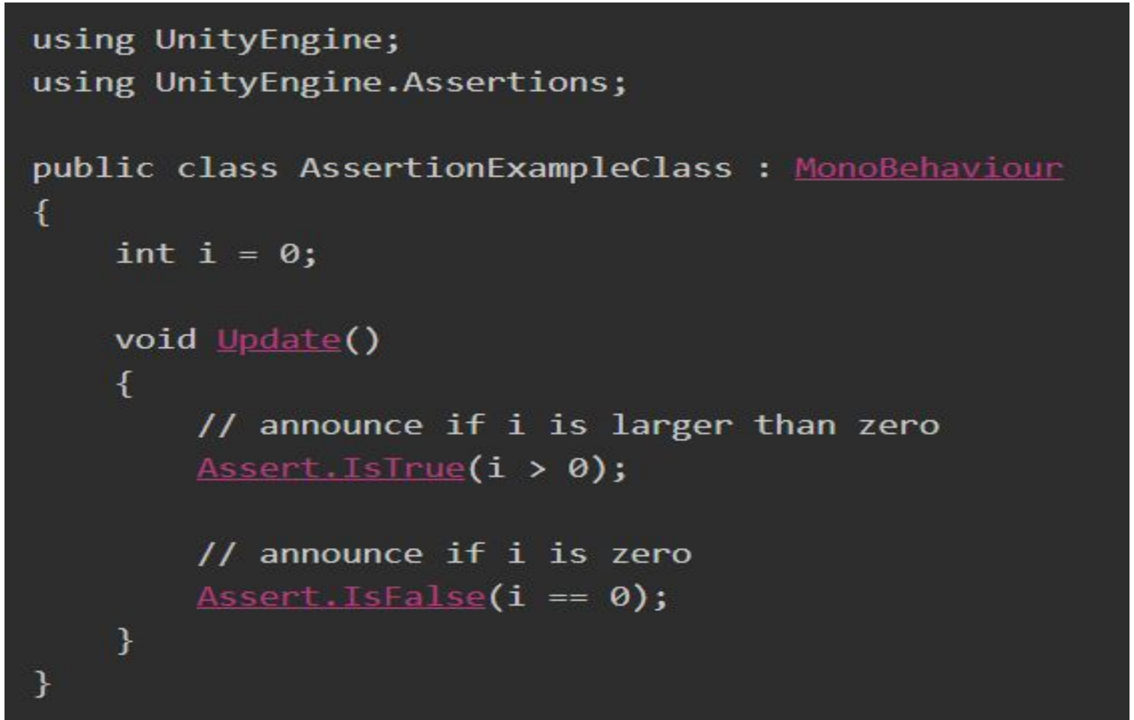
- Assert the value is null



```
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    MyClass myClassReference;

    void Update()
    {
        // Make sure the myClassReference reference is never set
        Assert.IsNull(myClassReference);
    }
}
```

Assert.IsTrue

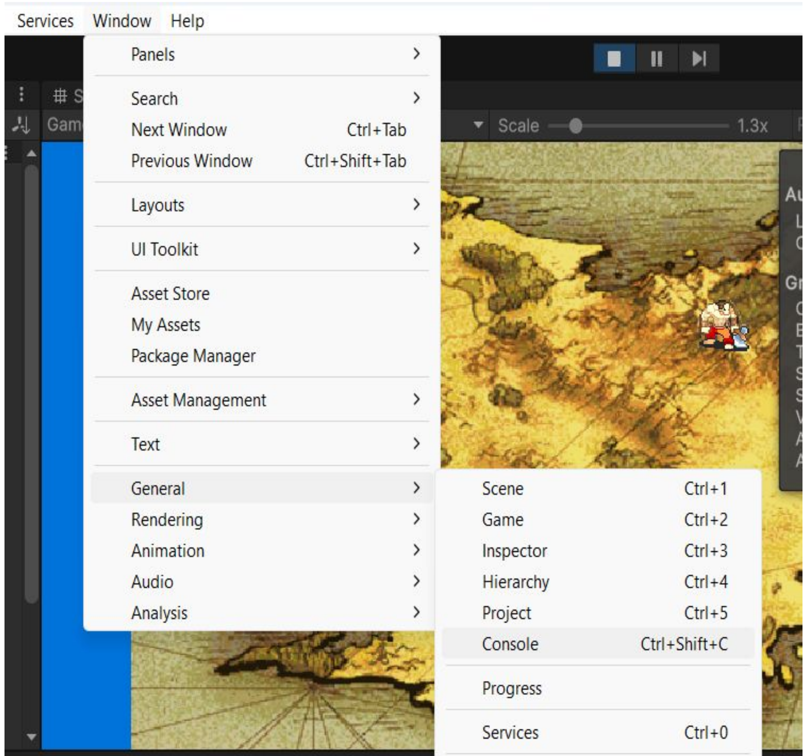
- Asserts that the condition is true

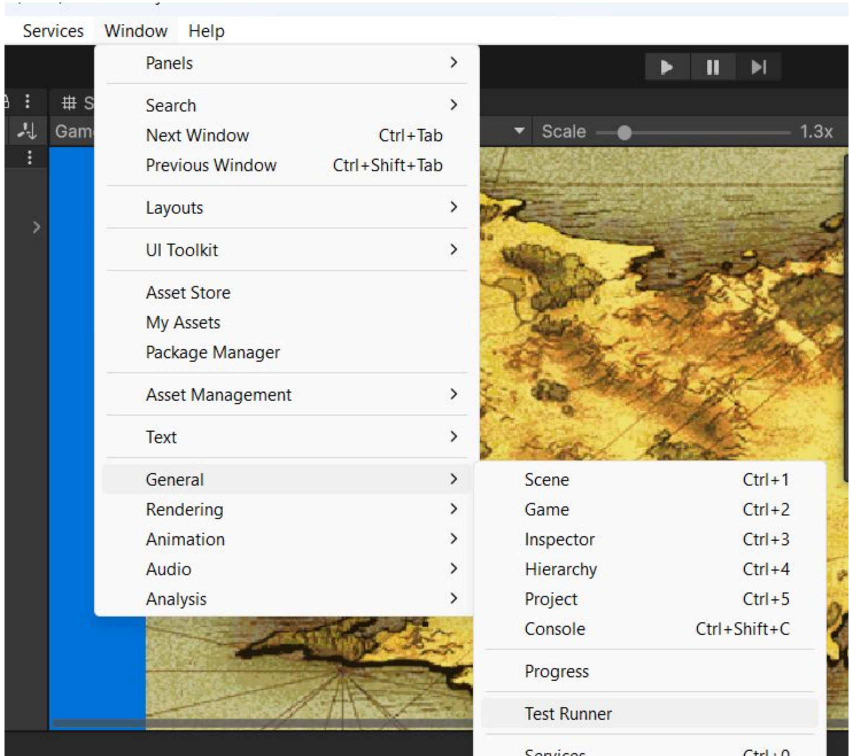


```
using UnityEngine;
using UnityEngine.Assertions;

public class AssertionExampleClass : MonoBehaviour
{
    int i = 0;

    void Update()
    {
        // announce if i is larger than zero
        Assert.IsTrue(i > 0);

        // announce if i is zero
        Assert.IsFalse(i == 0);
    }
}
```

Install Test Runner

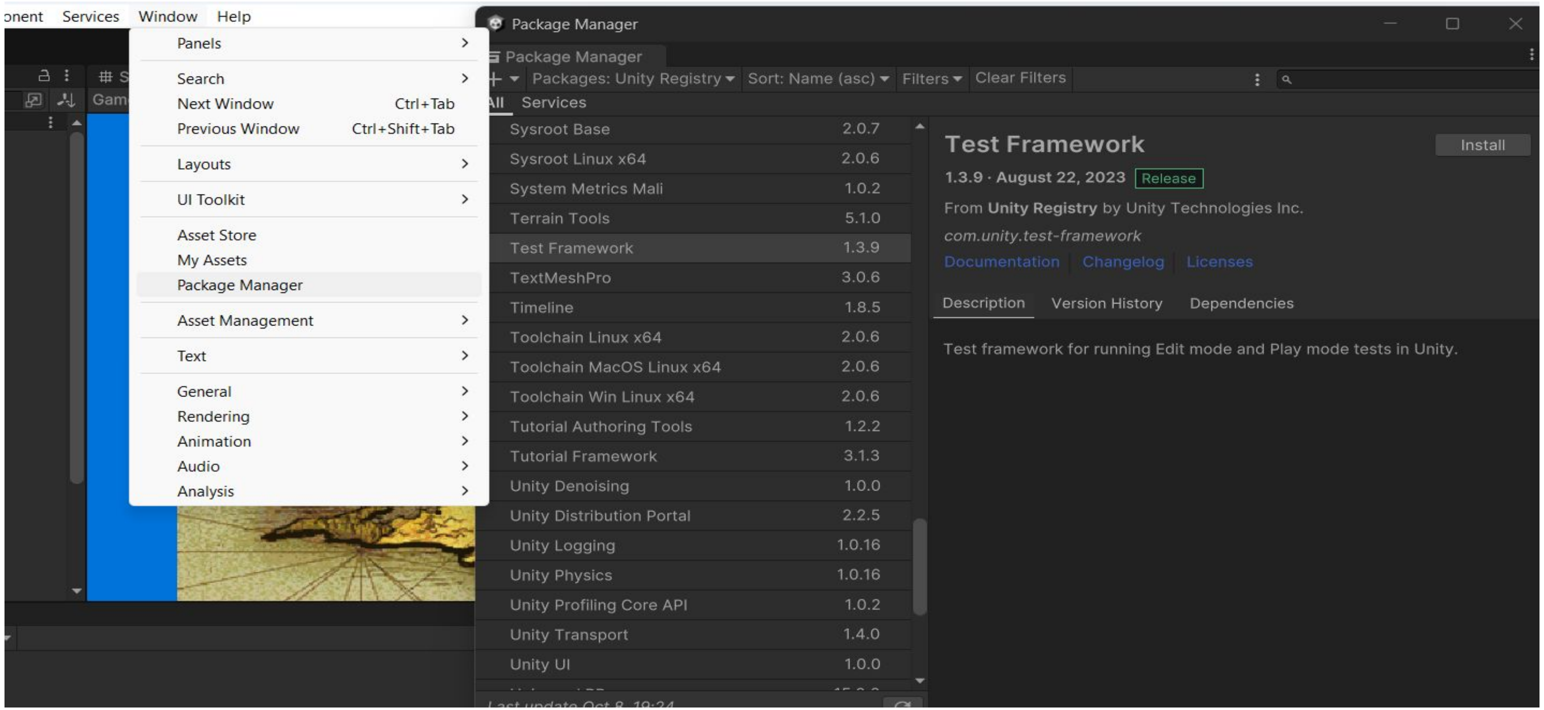
- Test Runner should appear under "General" window tab if installed





Install Test Runner

- Install Test Framework



Sysroot Base	2.0.7
Sysroot Linux x64	2.0.6
System Metrics Mali	1.0.2
Terrain Tools	5.1.0
Test Framework	1.3.9
TextMeshPro	3.0.6
Timeline	1.8.5
Toolchain Linux x64	2.0.6
Toolchain MacOS Linux x64	2.0.6
Toolchain Win Linux x64	2.0.6
Tutorial Authoring Tools	1.2.2
Tutorial Framework	3.1.3
Unity Denoising	1.0.0
Unity Distribution Portal	2.2.5
Unity Logging	1.0.16
Unity Physics	1.0.16
Unity Profiling Core API	1.0.2
Unity Transport	1.4.0
Unity UI	1.0.0

Test Framework

1.3.9 · August 22, 2023 Release

From **Unity Registry** by Unity Technologies Inc.

com.unity.test-framework

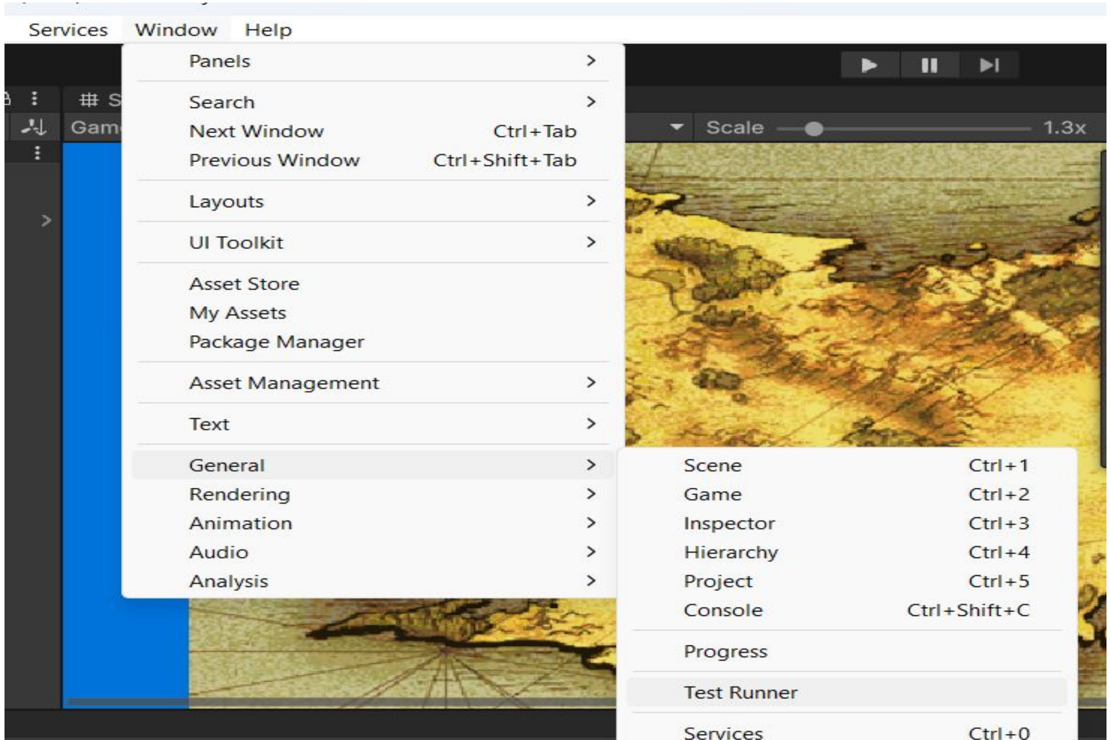
Documentation | Changelog | Licenses

Test framework for running Edit mode and Play mode tests in Unity.

Last update Oct 8, 19:24

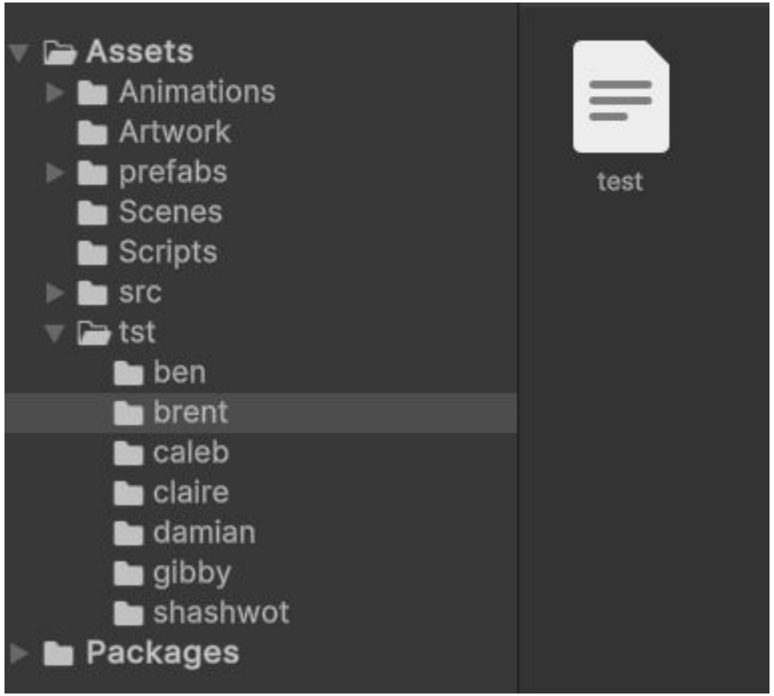
Install Test Runner

- Test Runner now appears under “General”



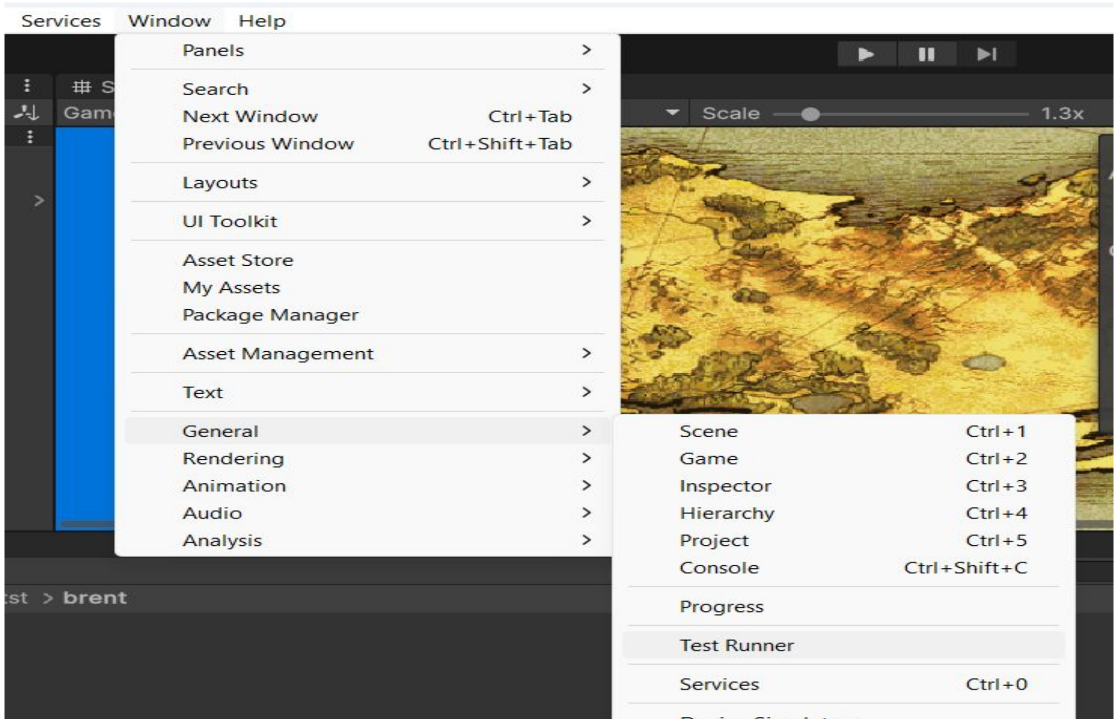
How To Setup A Test In Unity

- Step 1: Find your specific tst folder in Assets folder



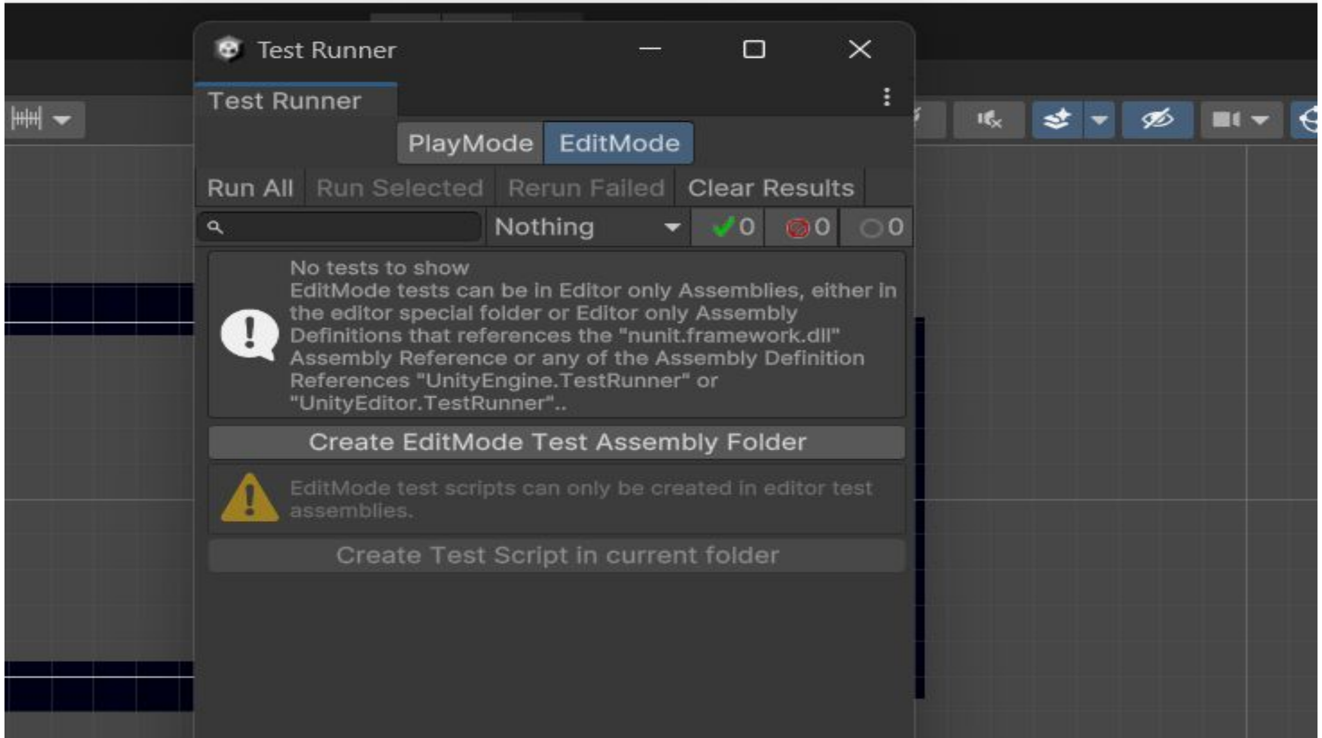
How To Setup A Test In Unity

- Step 2: Select test runner in unity in your test folder



How To Setup A Test In Unity

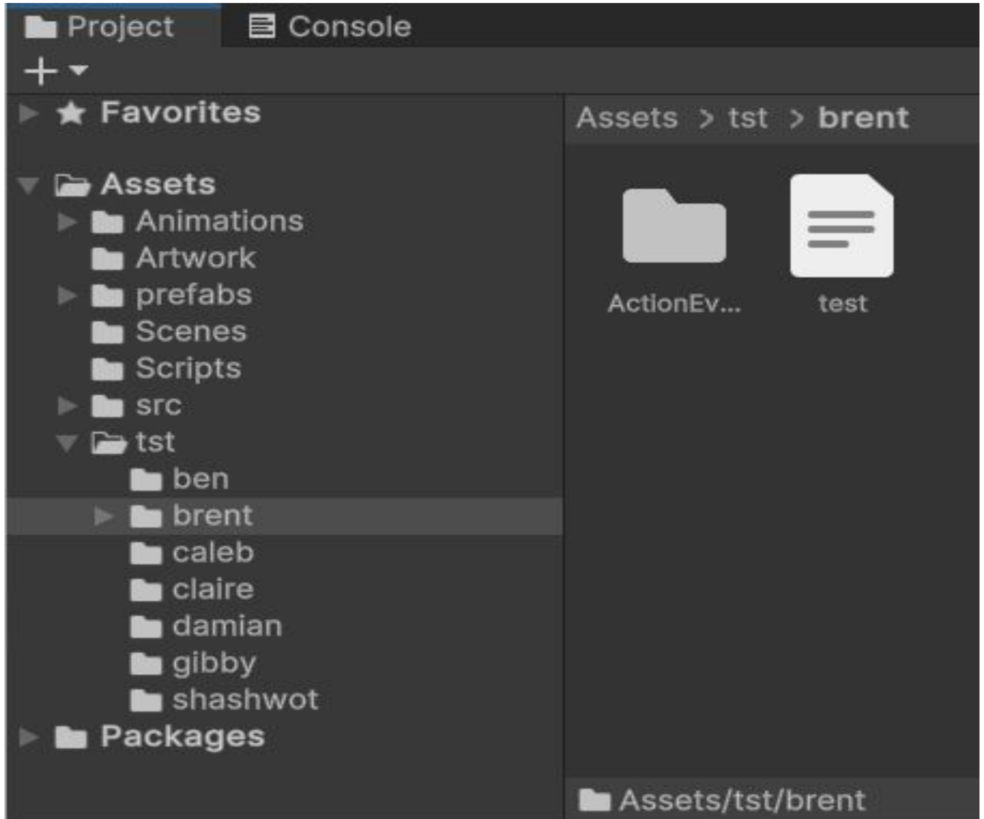
- Step 3: Select testing option and create test assembly folder



No tests to show
EditMode tests can be in Editor only Assemblies, either in the editor special folder or Editor only Assembly Definitions that references the "nunit.framework.dll" Assembly Reference or any of the Assembly Definition References "UnityEngine.TestRunner" or "UnityEditor.TestRunner"..

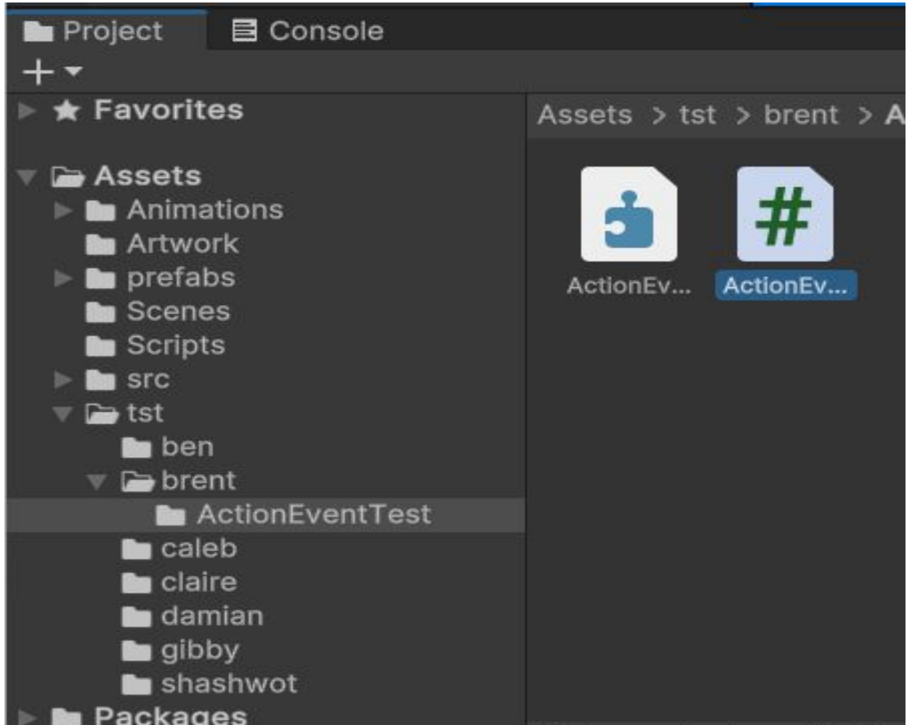
EditMode test scripts can only be created in editor test assemblies.

Assembly Folder



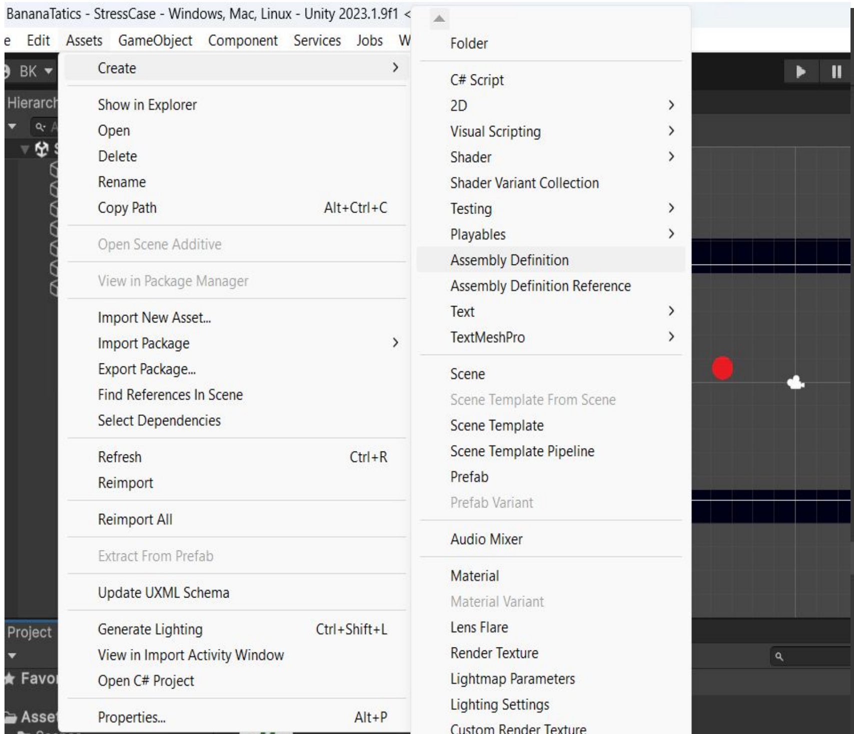
How To Setup A Test In Unity

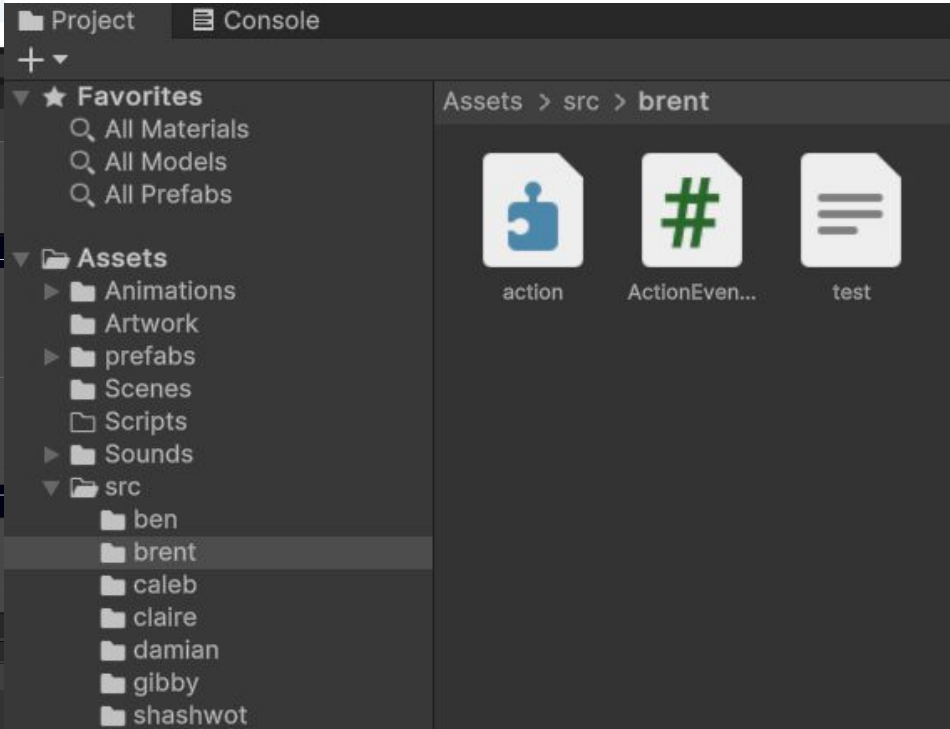
- Step 4: Create test script inside the assembly folder



How To Setup A Test In Unity

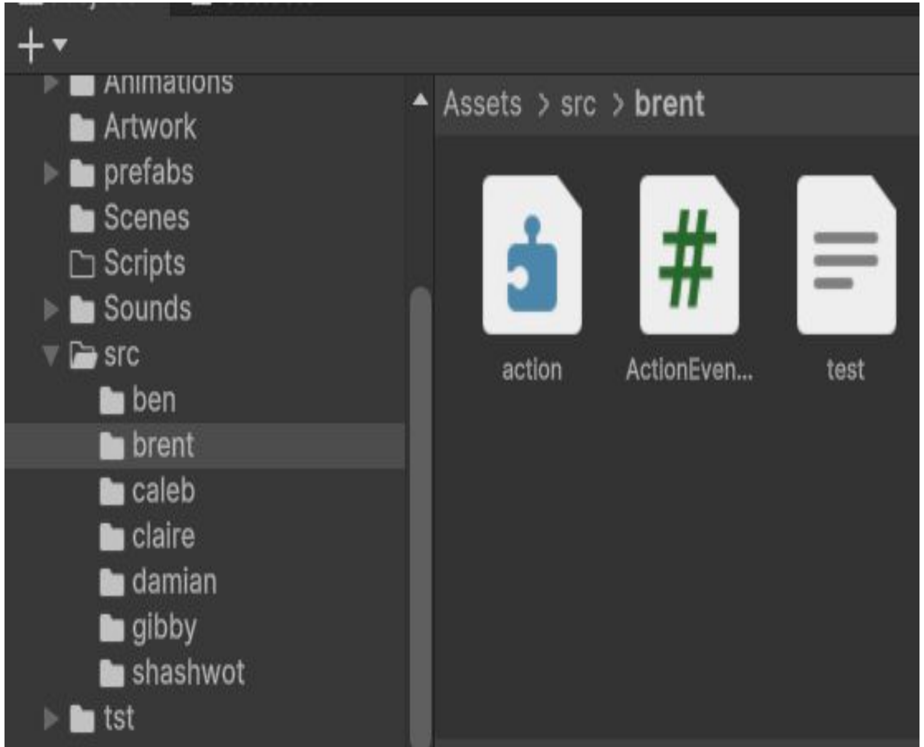
- Step 5: Create an assembly definition in src folder

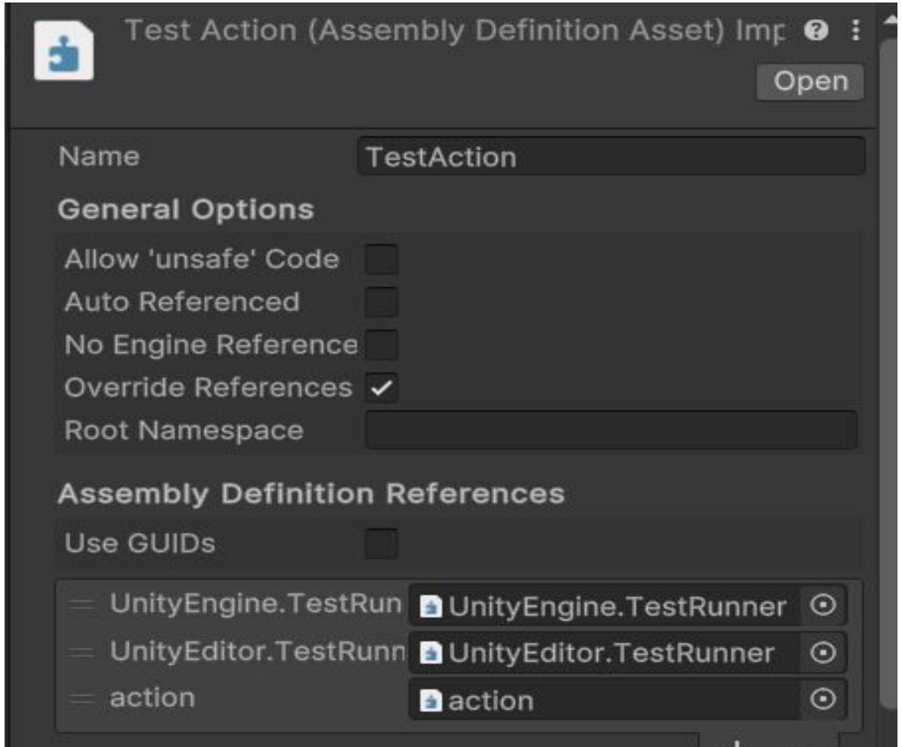




How To Setup A Test In Unity

- Step 6: Create a reference from test assembly definition to test script folder





Test Action (Assembly Definition Asset) Imp

Open

Name: TestAction

General Options

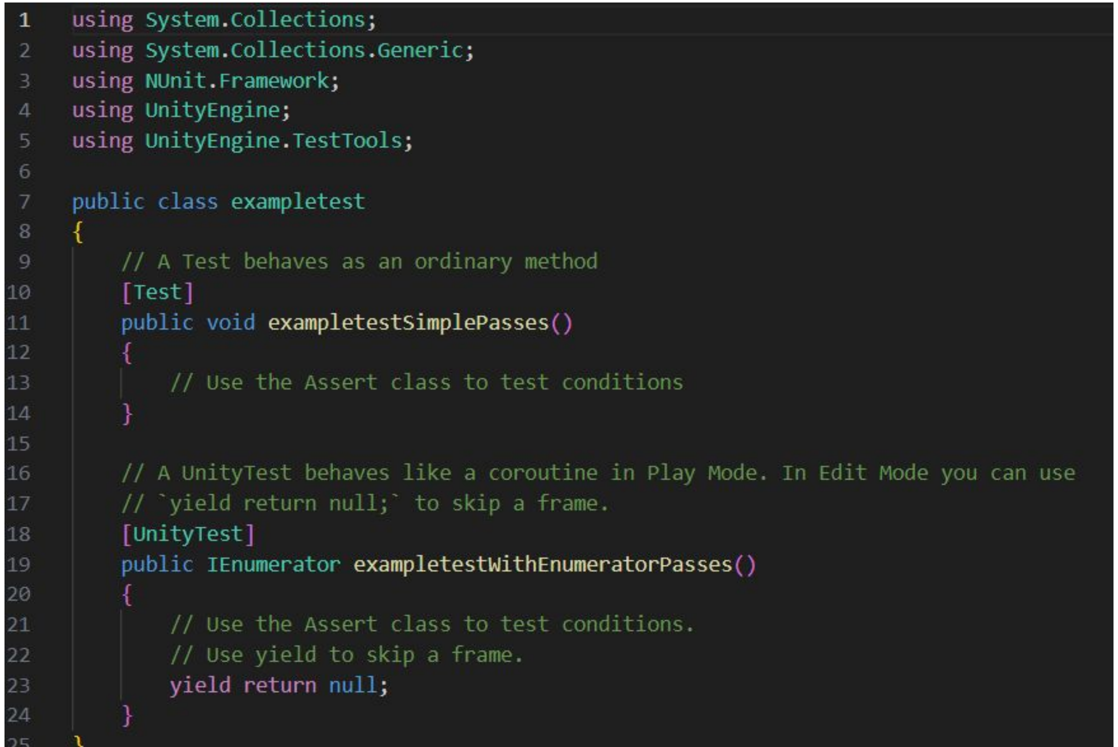
Allow 'unsafe' Code	
Auto Referenced	
No Engine Reference	
Override References	✓
Root Namespace	

Assembly Definition References

Use GUIDs	
UnityEngine.TestRun	UnityEngine.TestRunner
UnityEditor.TestRunn	UnityEditor.TestRunner
action	action

How To Setup A Test In Unity

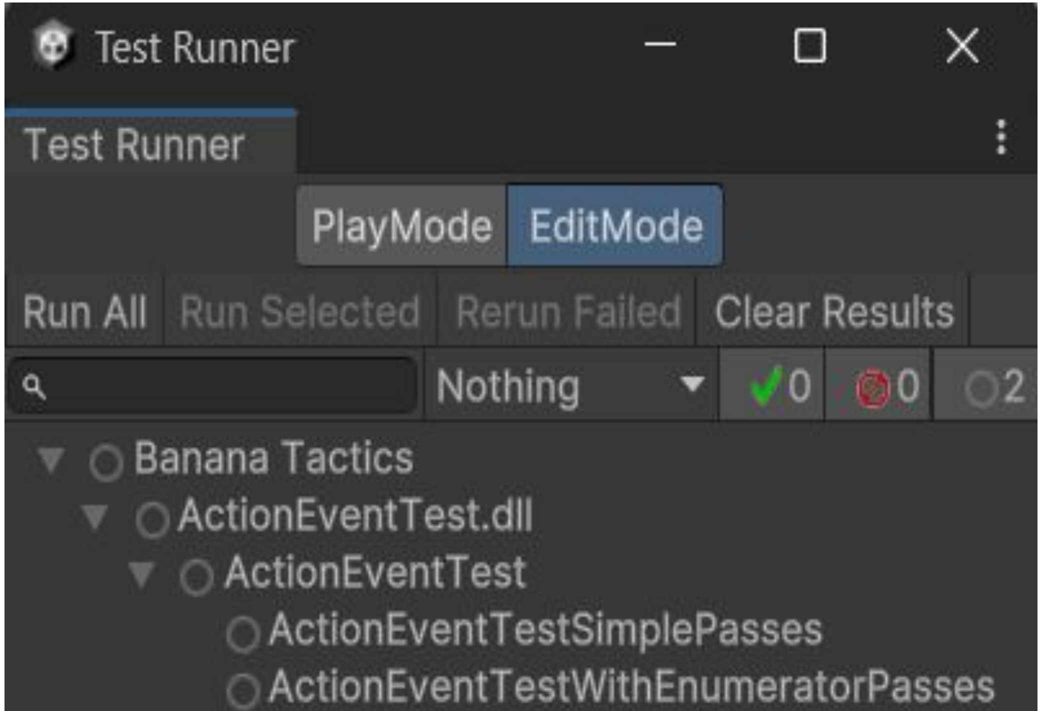
- Step 7: Open example test script template and write test (Template Script)



```
using System.Collections;
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;

public class exampletest
{
    // A Test behaves as an ordinary method
    [Test]
    public void exampletestSimplePasses()
    {
        // Use the Assert class to test conditions
    }

    // A UnityTest behaves like a coroutine in Play Mode. In Edit Mode you can use
    // `yield return null;` to skip a frame.
    [UnityTest]
    public IEnumerator exampletestWithEnumeratorPasses()
    {
        // Use the Assert class to test conditions.
        // Use yield to skip a frame.
        yield return null;
    }
}
```

How To Setup A Test In Unity

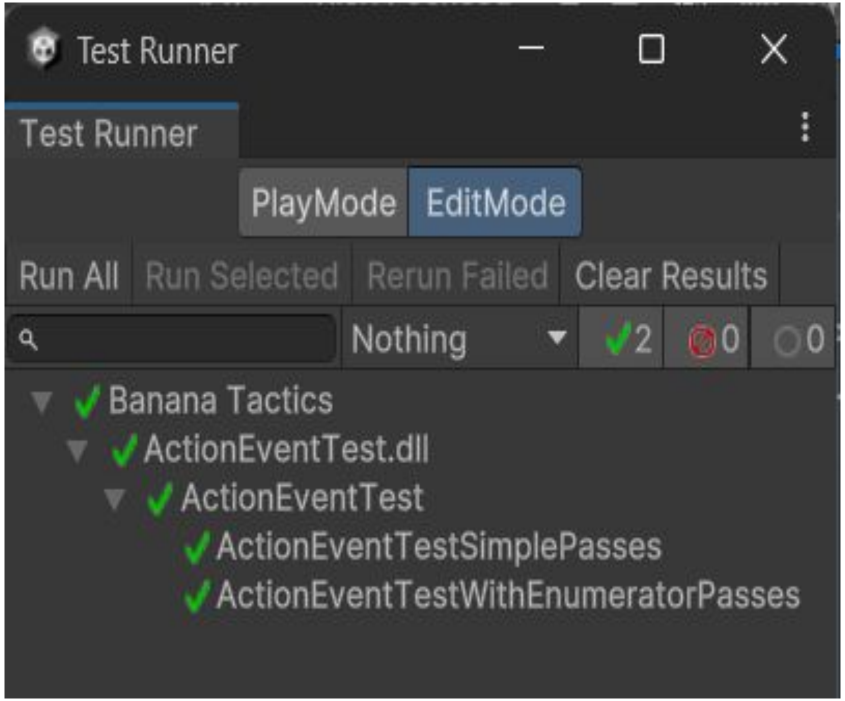
- Step 8: Open Test Runner and click on “ Run All ”



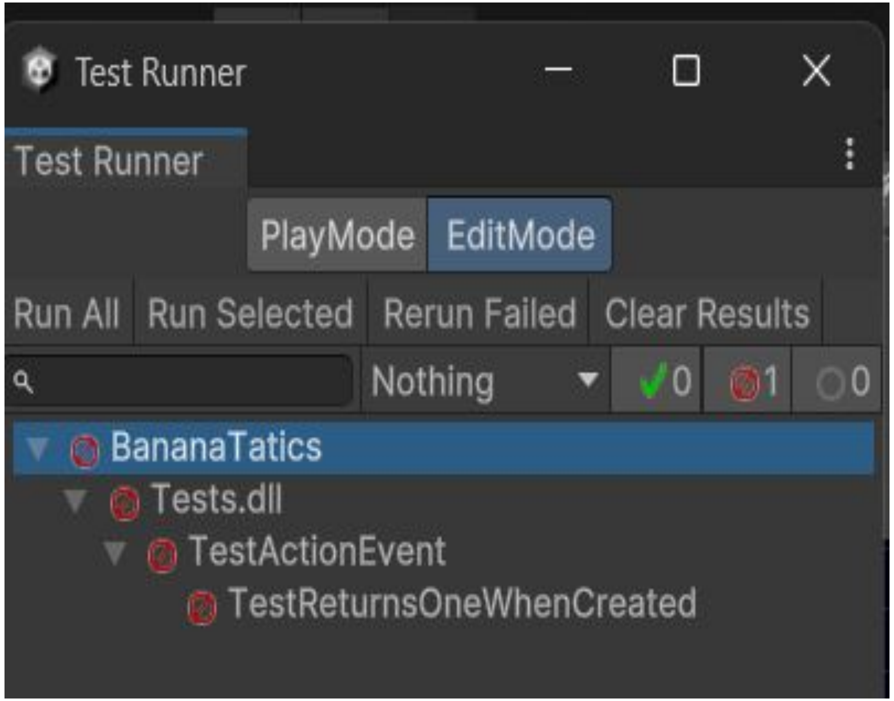
Banana Tactics
ActionEventTest.dll
ActionEventTest
ActionEventTestSimplePasses
ActionEventTestWithEnumeratorPasses

How To Setup A Test In Unity

Step 9: Finished Test And View Results: pass and failed example

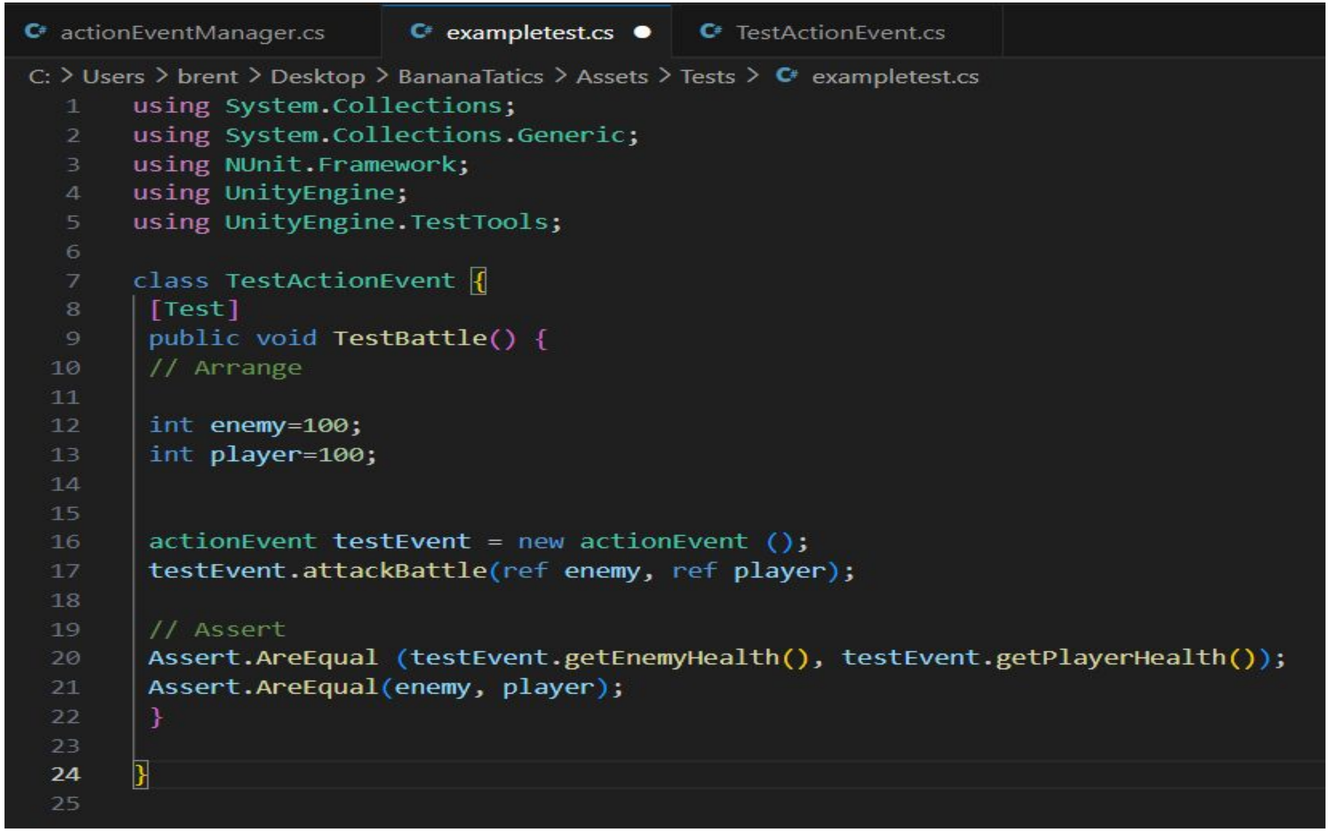


```
▼ ✓ Banana Tactics
  ▼ ✓ ActionEventTest.dll
    ▼ ✓ ActionEventTest
        ✓ ActionEventTestSimplePasses
        ✓ ActionEventTestWithEnumeratorPasses
```



```
▼ BananaTatics
  ▼ Tests.dll
    ▼ TestActionEvent
        TestReturnsOneWhenCreated
```

Example: Write Edit Mode Test



actionEventManager.cs
exampletest.cs
TestActionEvent.cs
C: > Users > brent > Desktop > BananaTatics > Assets > Tests > exampletest.cs

```
using System.Collections;
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;

class TestActionEvent {
 [Test]
 public void TestBattle() {
 // Arrange

 int enemy=100;
 int player=100;


 actionEvent testEvent = new actionEvent ();
 testEvent.attackBattle(ref enemy, ref player);

 // Assert
 Assert.AreEqual (testEvent.getEnemyHealth(), testEvent.getPlayerHealth());
 Assert.AreEqual(enemy, player);
 }

}
```

Testing Criteria For Unity

- All test will be fully automated and will not use a user’s input.
- All test must return a clear pass/fail value.
- Test should be design to pass a assertion statement.
- Test will be created in the Unity Platform with Test Runner.
- Stress test does not need to use Test Runner.
- Test need to be explicit and clearly show results.

How Boundary Test Relate To The Big Picture

- Instead of focusing only on the center of the data, boundary testing will detect errors occurring at the boundary edges of valid or invalid regions.
- A system will need a well define range of inputs that are acceptable to the scope of the project and error free.
- Without a accurate range set, the system can fail with invalid inputs that the user may think are valid.
- Will need to develop exception that can handle these invalid inputs.
- Developers must establish a valid input range before any exception can be made.

Traits Of A Boundary Test

- automated (runnable at the "push of a button")
- fast
- easy to read
- executed in isolation (tests should be independent from one another)
- assessed as either having being passed or failed
- relevant tomorrow
- consistent (the same results each time!)
- able to easily pinpoint what was at fault for each test that fails

Banana Tactics Case Example Of A Boundary Test

- Banana Tactics has a Battle calculator that take in 2 unit characters to battle each other
- They will do battle and their damage will be updated
- Initial health of a unit can not be less than 0 (makes no physical sense)
- Negative health or no health is a dead unit
- Must establish a boundary test to see if the minimum health of a unit is valid before a battle may commence
- Game will have game play error if a battle is allowed with a unit with 0 or negative health

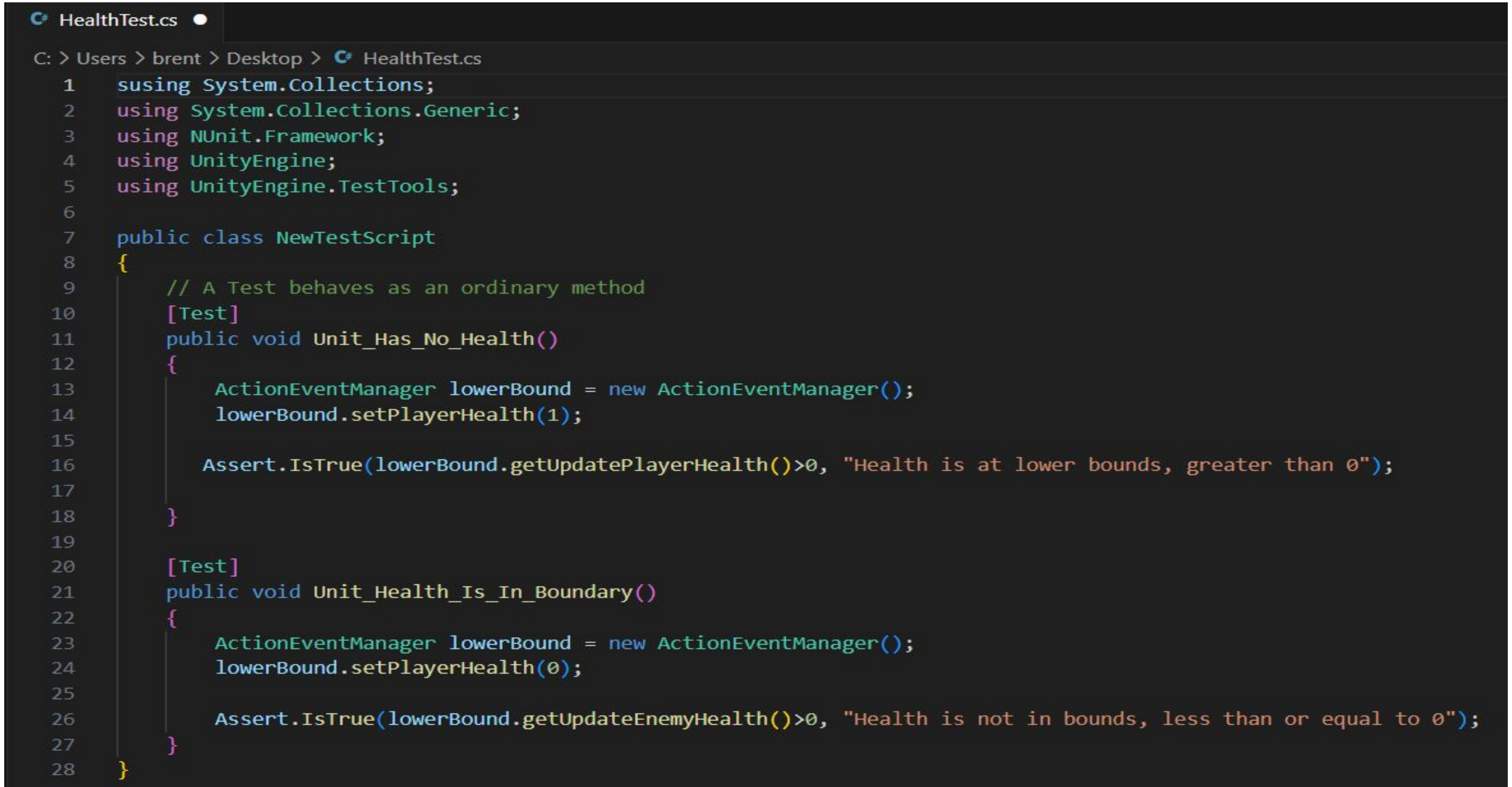
Setting Up A Boundary Test

- Will do a unit test in **Unity test Runner**
- Select two boundary condition
- One inside the boundary edge
- One outside the boundary edge

1. Select health that is the minimum health for an attack
2. Select health that is one less than the minimum health for an attack

- Prove that the boundary condition holds with the assertion class

Lower Boundary Health Test

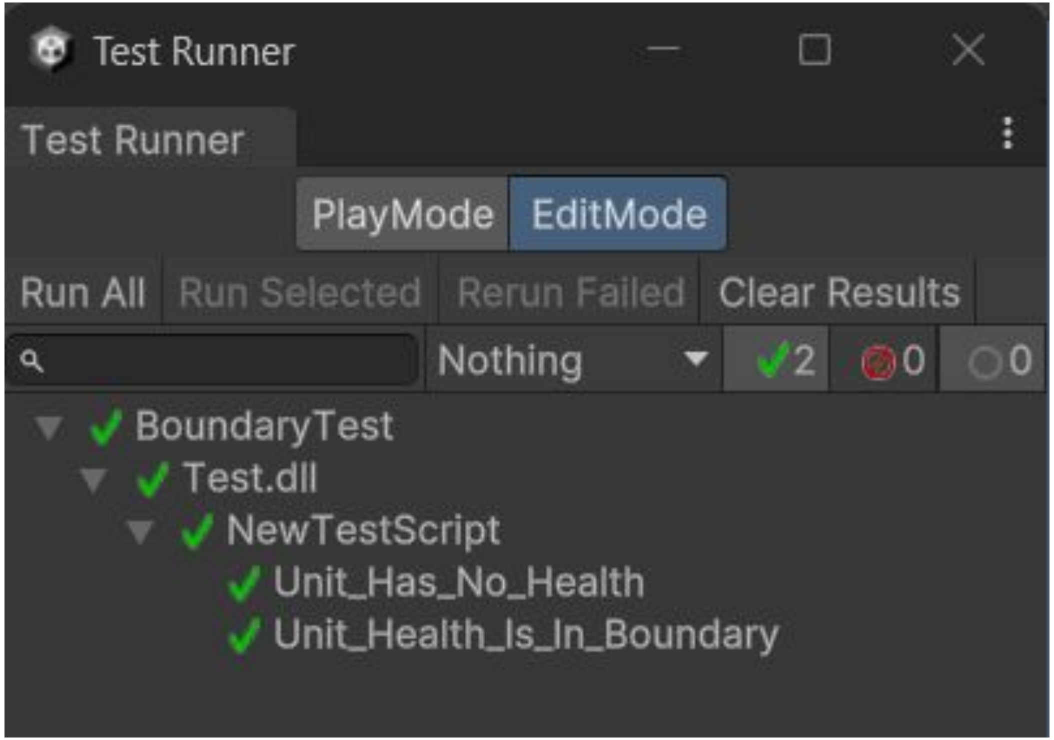


HealthTest.cs ●

C: > Users > brent > Desktop > HealthTest.cs

```
susing System.Collections;
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;

public class NewTestScript
{
    // A Test behaves as an ordinary method
    [Test]
    public void Unit_Has_No_Health()
    {
        ActionEventManager lowerBound = new ActionEventManager();
        lowerBound.setPlayerHealth(1);

       Assert.IsTrue(lowerBound.getUpdatePlayerHealth()>0, "Health is at lower bounds, greater than 0");

    }

    [Test]
    public void Unit_Health_Is_In_Boundary()
    {
        ActionEventManager lowerBound = new ActionEventManager();
        lowerBound.setPlayerHealth(0);

        Assert.IsTrue(lowerBound.getUpdateEnemyHealth()>0, "Health is not in bounds, less than or equal to 0");
    }
}
```

Lower Boundary Health Test Results



```
▼ ✓ BoundaryTest
  ▼ ✓ Test.dll
    ▼ ✓ NewTestScript
        ✓ Unit_Has_No_Health
        ✓ Unit_Health_Is_In_Boundary
```

How Stress Test Relates To The Big picture

- Must know upper thresholds of what the software will be able to handle
- Client may define some of these specifications
- Client is usually not an expert and does not know where the system is weak
- If we stress test small modular piece we can confidently construct them together
- Our class video game will just crash and is not a critical liability
- In the real world a failure could be extremely dangerous

  Ex. The software that is responsible for monitoring a nuclear reactor

Stress Test Requirement

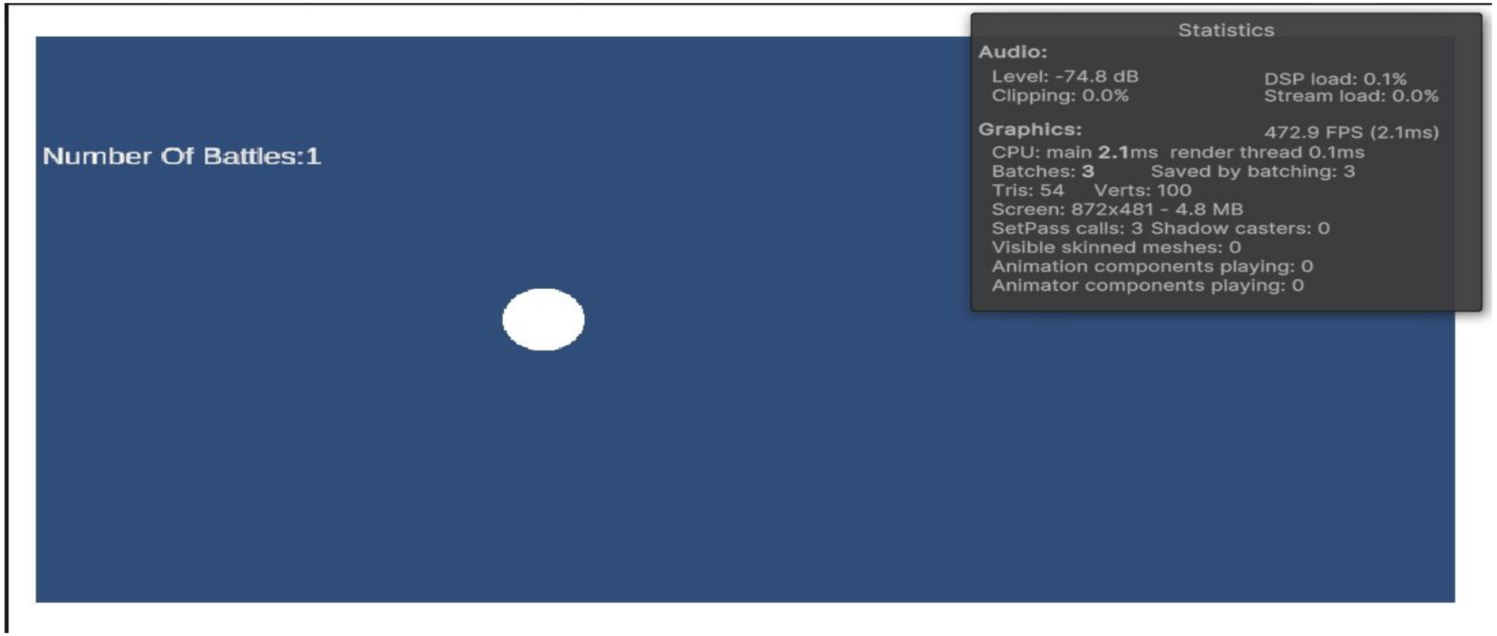
- Must use Unity
- Do not have to use Test Runner
- Must be related to your specific feature
- Must push unity’s limits and break the system
- Must be an incremental test (step by step increase)
- Have fun and be creative

Banana Tactics Case Example: Stress Test

- Unity Scenario - A battle calculator that calculates battle results in Banana tactics video game.
- Management wants to change the game to include multi-dimensional battles.
- Each battle must be finished before the next dimensional battle can start.
- **Question:** How many dimensions can we battle in 1 event before the platform can not handle the calculations and animation any more?
- Incrementally increase battle calculation until animation breaks down.
- Animation will break down when FPS is less than 60.
- FPS - frames per second

Basic Scene

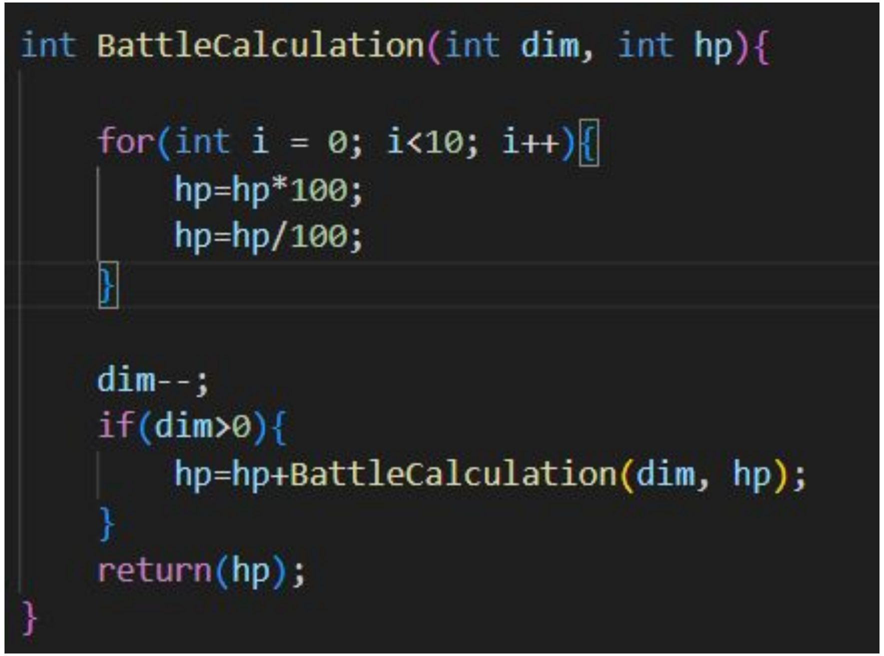
- Ball bouncing back and forth with clear animation



Number Of Battles:1

```
Statistics
Audio:
  Level: -74.8 dB                    DSP load: 0.1%
  Clipping: 0.0%                     Stream load: 0.0%
Graphics:                            472.9 FPS (2.1ms)
  CPU: main 2.1ms  render thread 0.1ms
  Batches: 3        Saved by batching: 3
  Tris: 54     Verts: 100
  Screen: 872x481 - 4.8 MB
  SetPass calls: 3 Shadow casters: 0
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

Basic Recursive Code For A Dimensional Battle



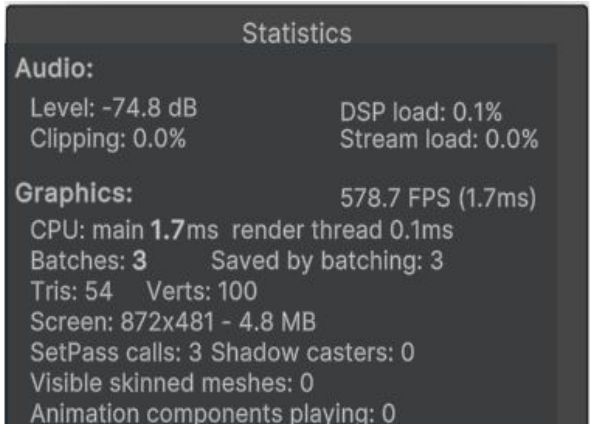
```
int BattleCalculation(int dim, int hp){

    for(int i = 0; i<10; i++){
        hp=hp*100;
        hp=hp/100;
    }

    dim--;
    if(dim>0){
        hp=hp+BattleCalculation(dim, hp);
    }
    return(hp);
}
```

Increase Dimensions Until Ball Cannot Render Correctly

- Will take a basic bouncing ball script and include a battle in the Update()
- Test Cases will start at 1 battle
- Incrementally increase battles by the power of ten

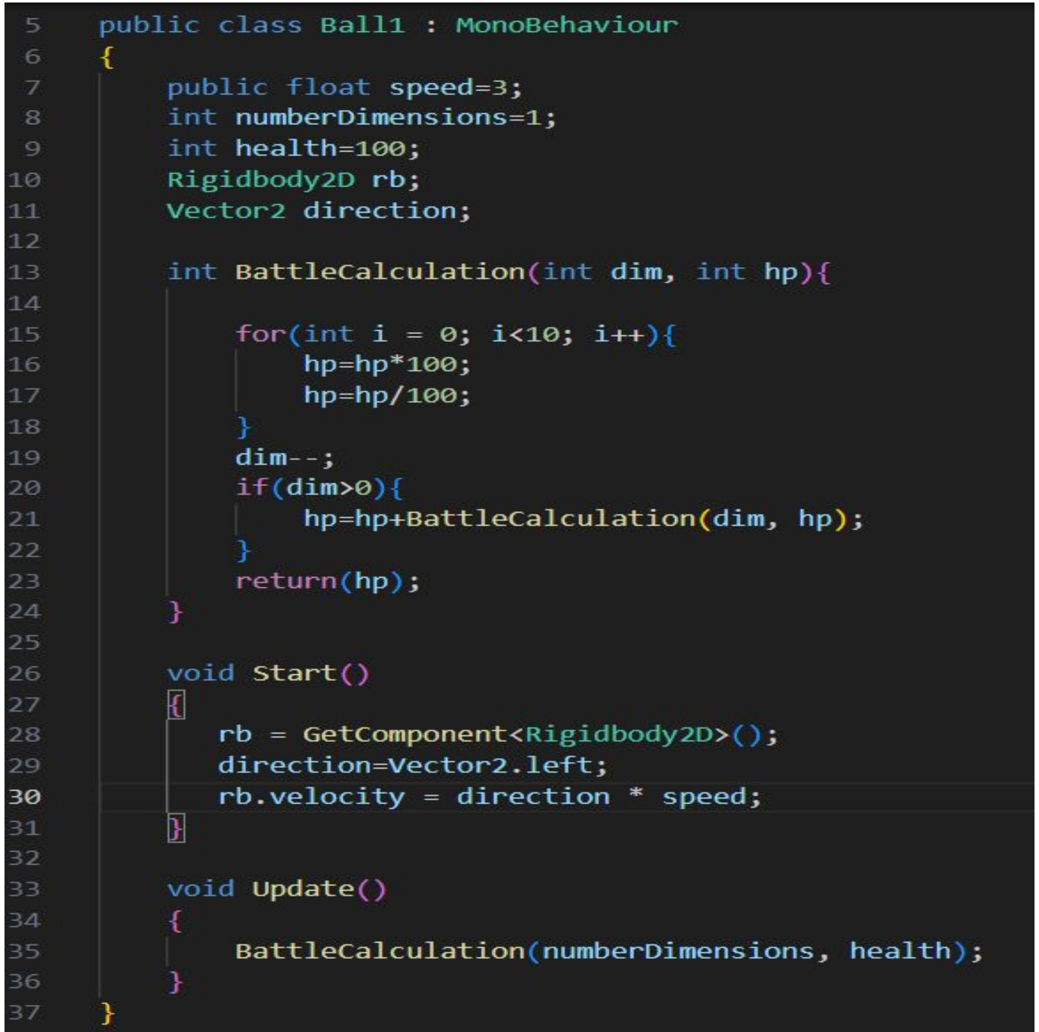
  Ex. 1, 10, 100, 1000, 10,000, 100,000……

- Determine at what battle count will the FPS fall below 60
- Unity has a statistic measurement tool



```
Statistics
Audio:
  Level: -74.8 dB                DSP load: 0.1%
  Clipping: 0.0%                 Stream load: 0.0%

Graphics:                        578.7 FPS (1.7ms)
  CPU: main 1.7ms  render thread 0.1ms
  Batches: 3        Saved by batching: 3
  Tris: 54    Verts: 100
  Screen: 872x481 - 4.8 MB
  SetPass calls: 3 Shadow casters: 0
  Visible skinned meshes: 0
  Animation components playing: 0
```

Complete Script:



```
public class Ball1 : MonoBehaviour
{
    public float speed=3;
    int numberDimensions=1;
    int health=100;
    Rigidbody2D rb;
    Vector2 direction;

    int BattleCalculation(int dim, int hp){

        for(int i = 0; i<10; i++){
            hp=hp*100;
            hp=hp/100;
        }
        dim--;
        if(dim>0){
            hp=hp+BattleCalculation(dim, hp);
        }
        return(hp);
    }

    void Start()
    {
       rb = GetComponent<Rigidbody2D>();
       direction=Vector2.left;
       rb.velocity = direction * speed;
    }

    void Update()
    {
        BattleCalculation(numberDimensions, health);
    }
}
```

Other Examples

1. **Cole**

- Boundary Test
- Stress Test

Questions about Test?

Cole Vanderlaan TL3

Team: Gallion Games

Design Patterns

Course Requirement

Every person needs 2 patterns and to know why they were necessary for implementation into your code.

You will also be expected to be able to replicate your chosen patterns class diagrams.

Make sure your pattern is solving a problem not creating one.

Creational patterns

-Creation of objects-

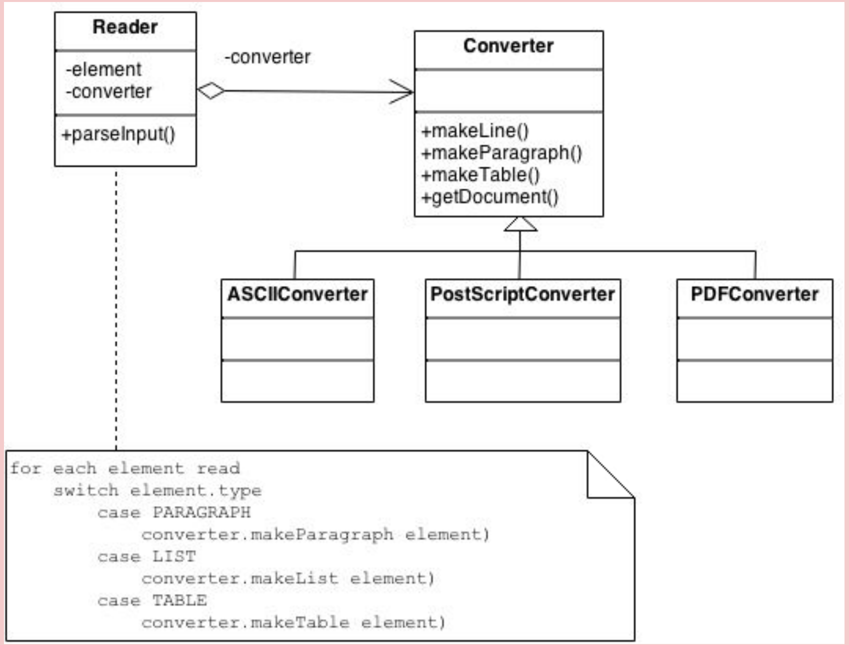
- **Abstract Factory**
- **Builder**
- **Factory Method**
- **Object Pool**
- **Prototype**
- **Singleton**

Often universally applicable

Generally lower complexity

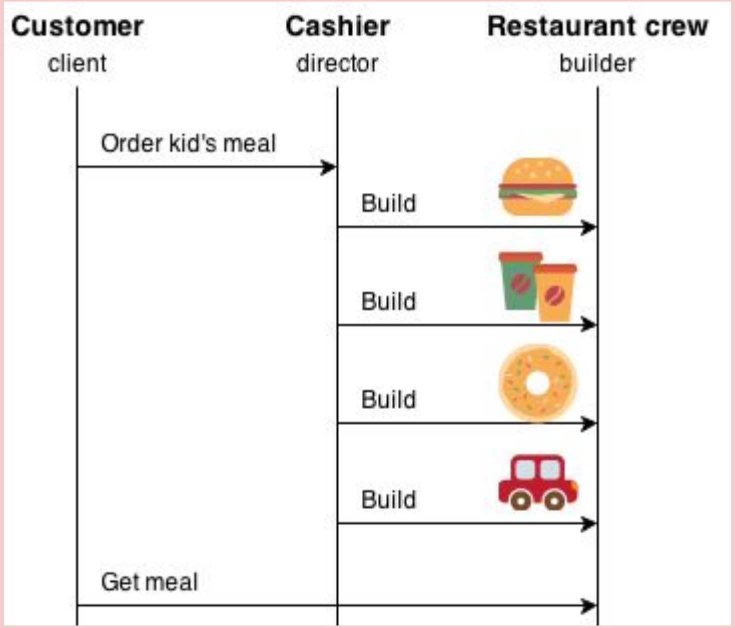
Useful for games with lots of objects

Builder



```
for each element read
    switch element.type
        case PARAGRAPH
            converter.makeParagraph element)
        case LIST
            converter.makeList element)
        case TABLE
            converter.makeTable element)
```

"Separate the construction of a complex object from its representation so that the same construction process can create different representations."



Potentially useful for restaurant game with complex orders being made

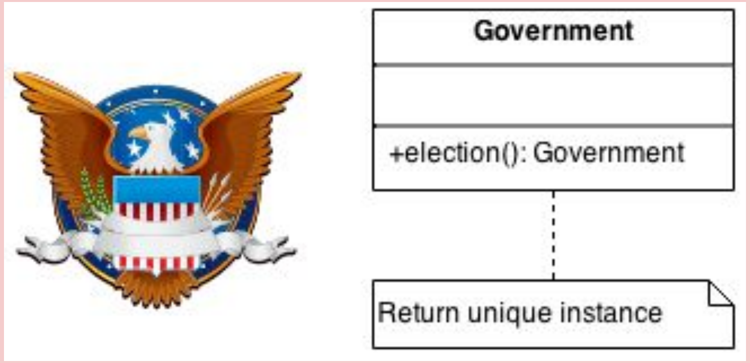
Singleton

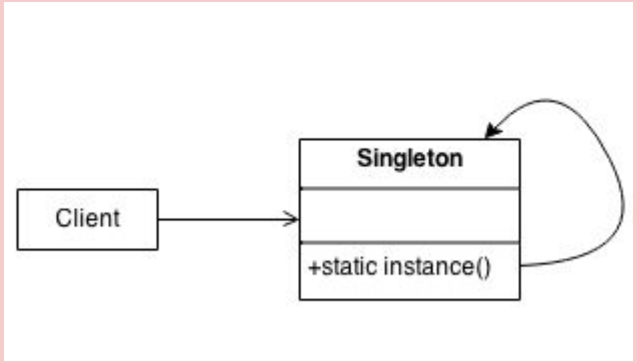
! Likely easiest to implement !

Instantiating only one instance of an object

Globally available

Ex: player character or text boxes





Modification: for multithreading in C#

Sealed in C# means you can not make a subclass of this class.

```
// Bad code! Do not use!
public sealed class Singleton
{
   private static Singleton instance=null;

   private Singleton() { }

   public static Singleton Instance
   {
     get {
       if (instance==null) {
         instance = new Singleton();
       }
       return instance;
     }
   }
}
```

What if right after the check for null the thread is interrupted and another thread requests this singleton?

Modification: for multithreading in C#

```
// Simple Thread safe Solution
public sealed class Singleton
{
   private static Singleton instance = null;
   private static readonly object padlock = new object();
   Singleton() { }
   public static Singleton Instance {
     get {
       lock (padlock) {
         if (instance == null) {
           instance = new Singleton();
         }
         return instance;
       }
     }
   }
}
```

Unfortunately, performance suffers as a lock is acquired every time the instance is requested.

Modification: for multithreading in C#

```
// Simple Thread safe Solution
public sealed class Singleton
{
  private static Singleton instance = null;
  private static readonly object padlock = new object();
  Singleton() { }
  public static Singleton Instance {
    get {
      if (instance != null) {
         return instance;
      }
      lock (padlock) {
        if (instance == null) {
          instance = new Singleton();
        }
        return instance;
      }}}}
```

This gets rid of the performance hit in the general case,

Modification: for multithreading in C#

```
// Thread safe and fully lazy
public sealed class Singleton
 {
   private Singleton() { }
   public static Singleton Instance {
     get {
       return Nested.instance;
      }
   }

   private class Nested {
     // Explicit static constructor to tell C# compiler
     // not to mark type as beforefieldinit
     static Nested() { }
     internal static readonly Singleton instance = new Singleton();
   }
}
```

Here, instantiation is triggered by the first reference to the static member of the nested class, which only occurs in Instance. This means the implementation is fullylazy, but has all the performance benefits of the previous ones.

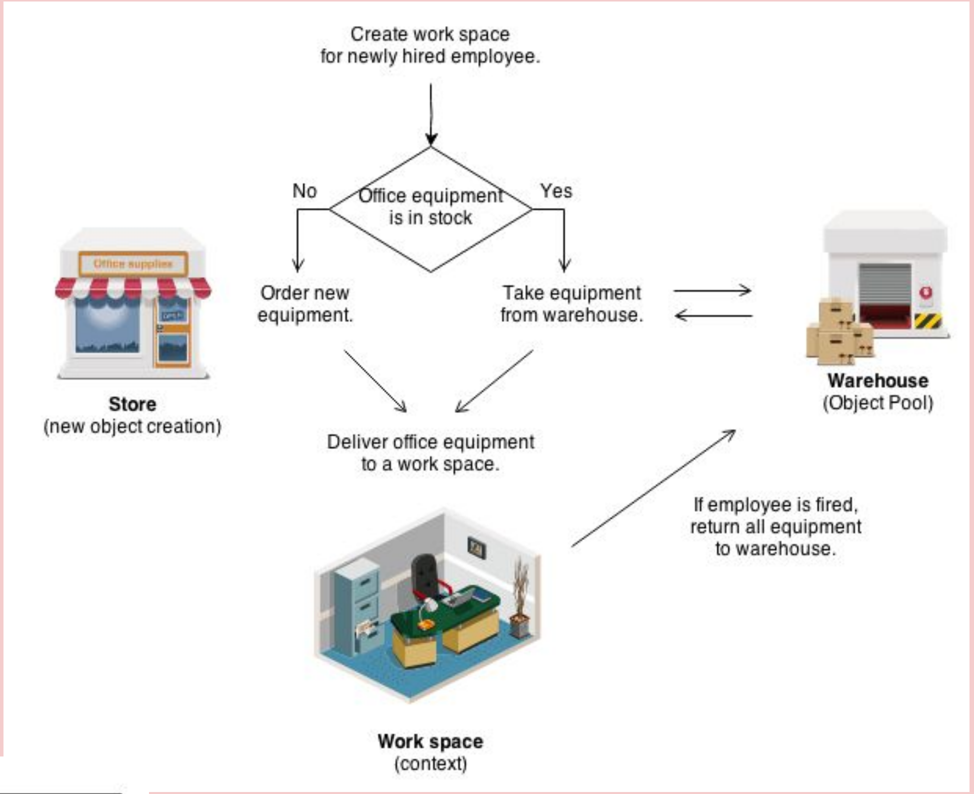
Note: nested classes have access to the enclosing class's private members,
the reverse is not true, hence the need for instance to be internal here.

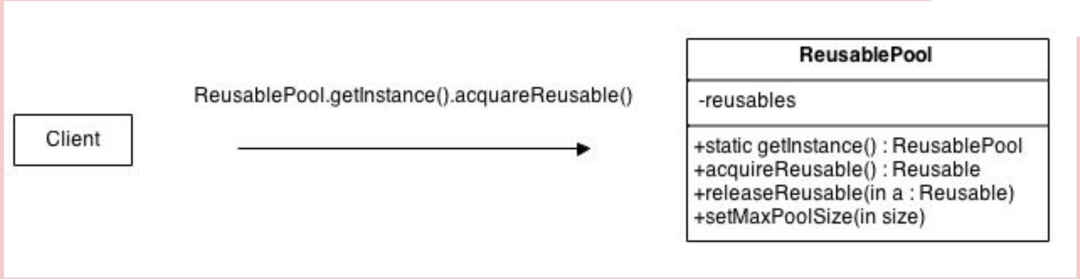
Object Pool

! Storage of Game Objects !

Spawners

Useful for storage of generic often reused assets.


Create work space
for newly hired employee.
No
Office equipment is in stock
Yes
Order new equipment.
Take equipment from warehouse.
Store
(new object creation)
Warehouse
(Object Pool)
Deliver office equipment to a work space.
If employee is fired, return all equipment to warehouse.
Work space
(context)


Client
ReusablePool.getInstance().acquareReusable()
ReusablePool
-reusables
+static getInstance() : ReusablePool
+acquireReusable() : Reusable
+releaseReusable(in a : Reusable)
+setMaxPoolSize(in size)

Ex: Instead of destroying and reinitializing the ships in between battles we can store them instead.

Structural patterns

-Relationships between objects-

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Private Class Data
- Proxy

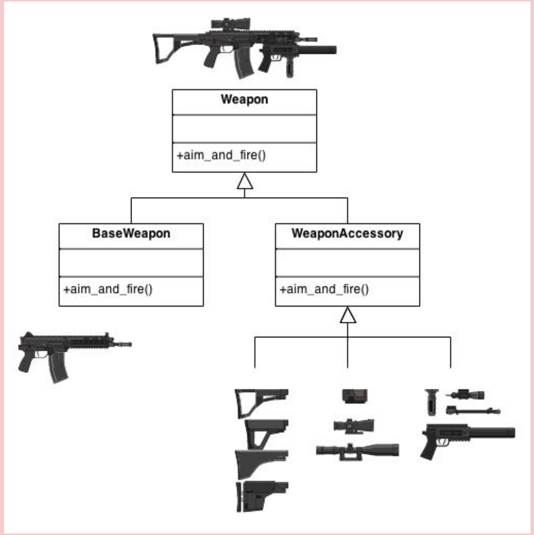
Decorator

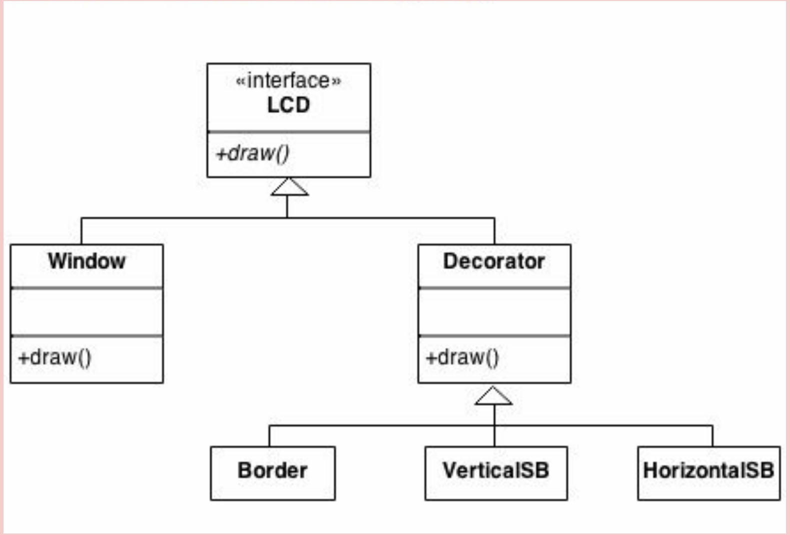
! Encapsulate object variation !

“Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”

If you are familiar with FPS games think weapon attachments

Attaching behaviors to items

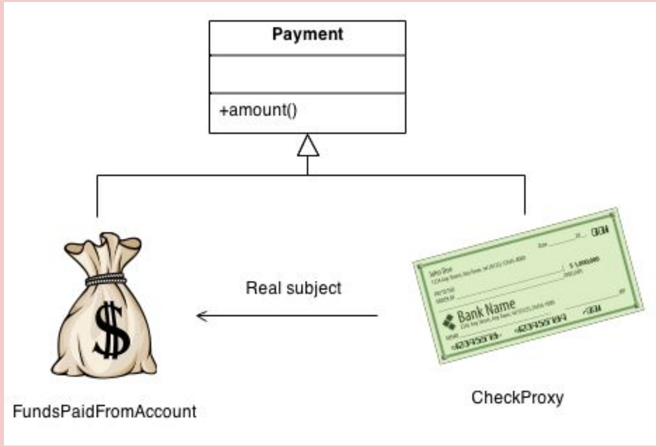


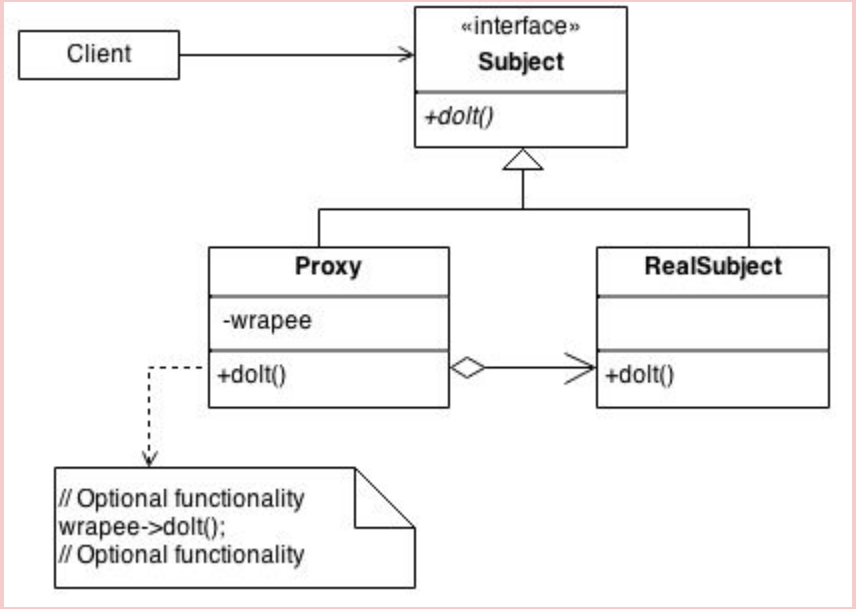


Proxy

! Placeholder for complex objects !

“Problem: You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.”





Could be useful for reducing strain on devices

Behavioral patterns

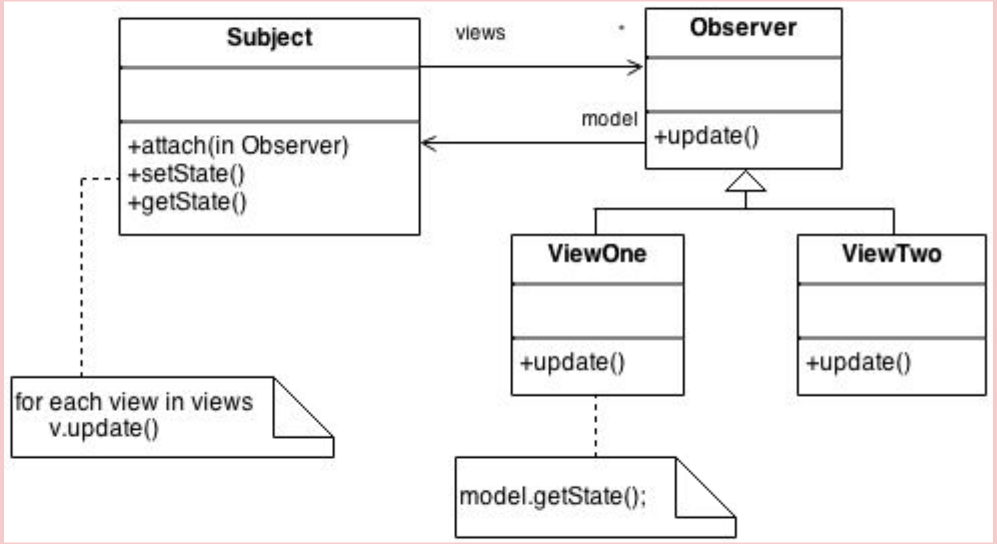
-Communication between operator and objects-

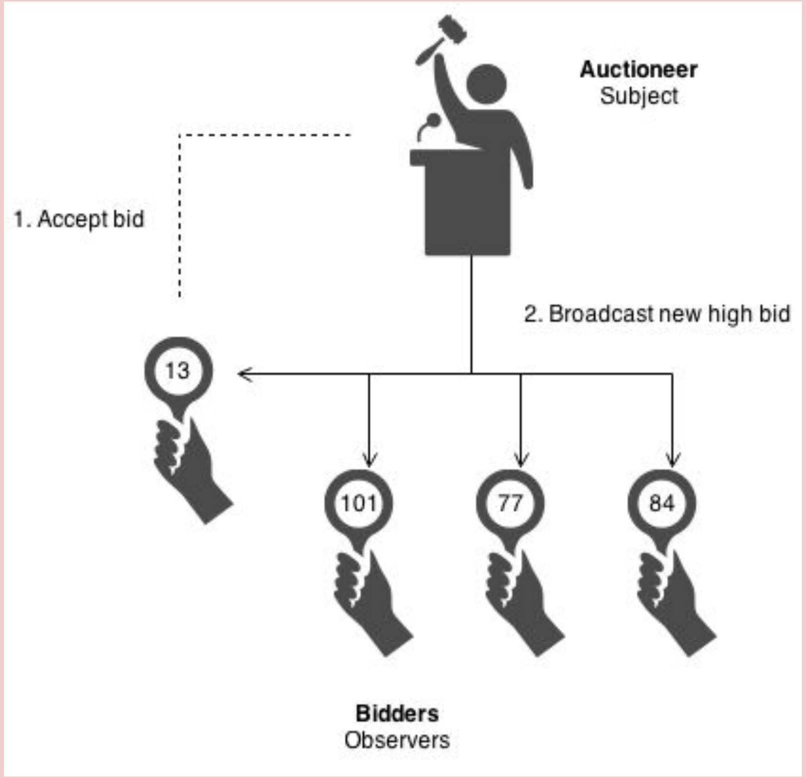
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Observer

! Automatic updates after state changes !

This is useful when you need to update a group of states when one of them changes



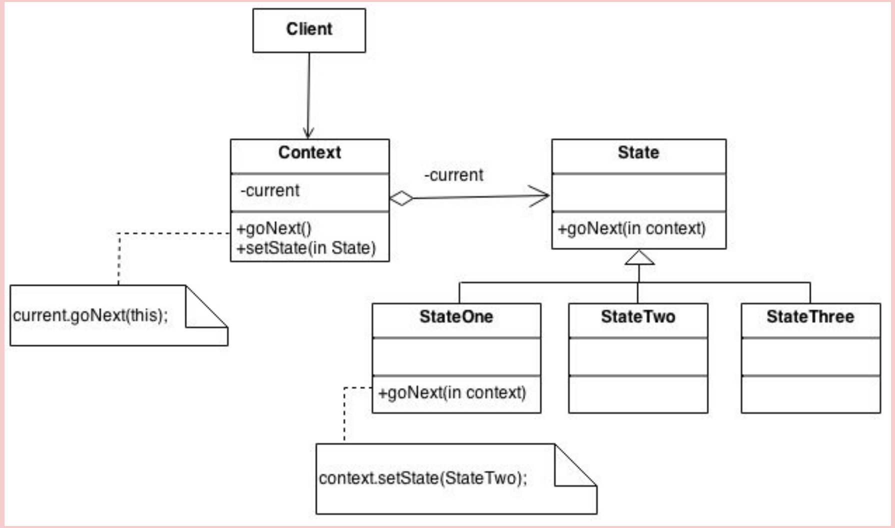


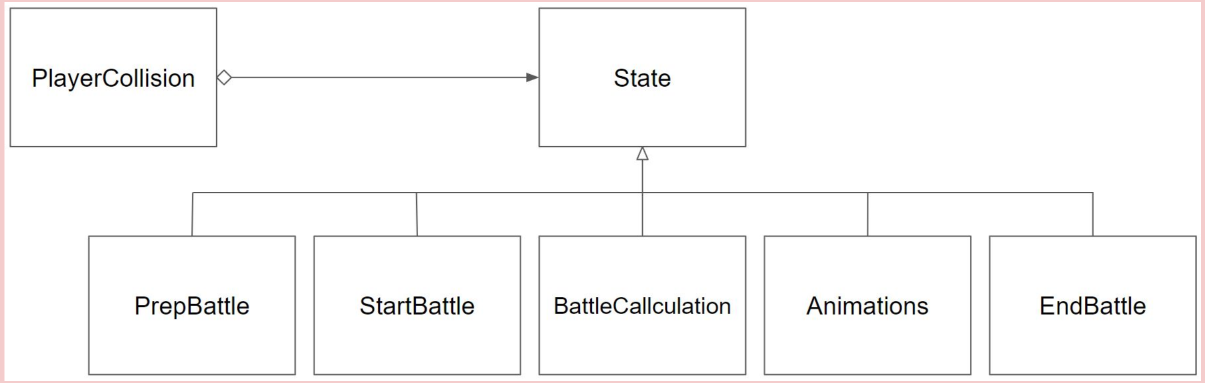
State

! Adaptable Functionality !

Potentially most useful for strategy games with turns and changing environment.

Transition of states depending on certain triggers





Resources

Design patterns and refactoring. SourceMaking. (n.d.). https://sourcemaking.com/design_patterns

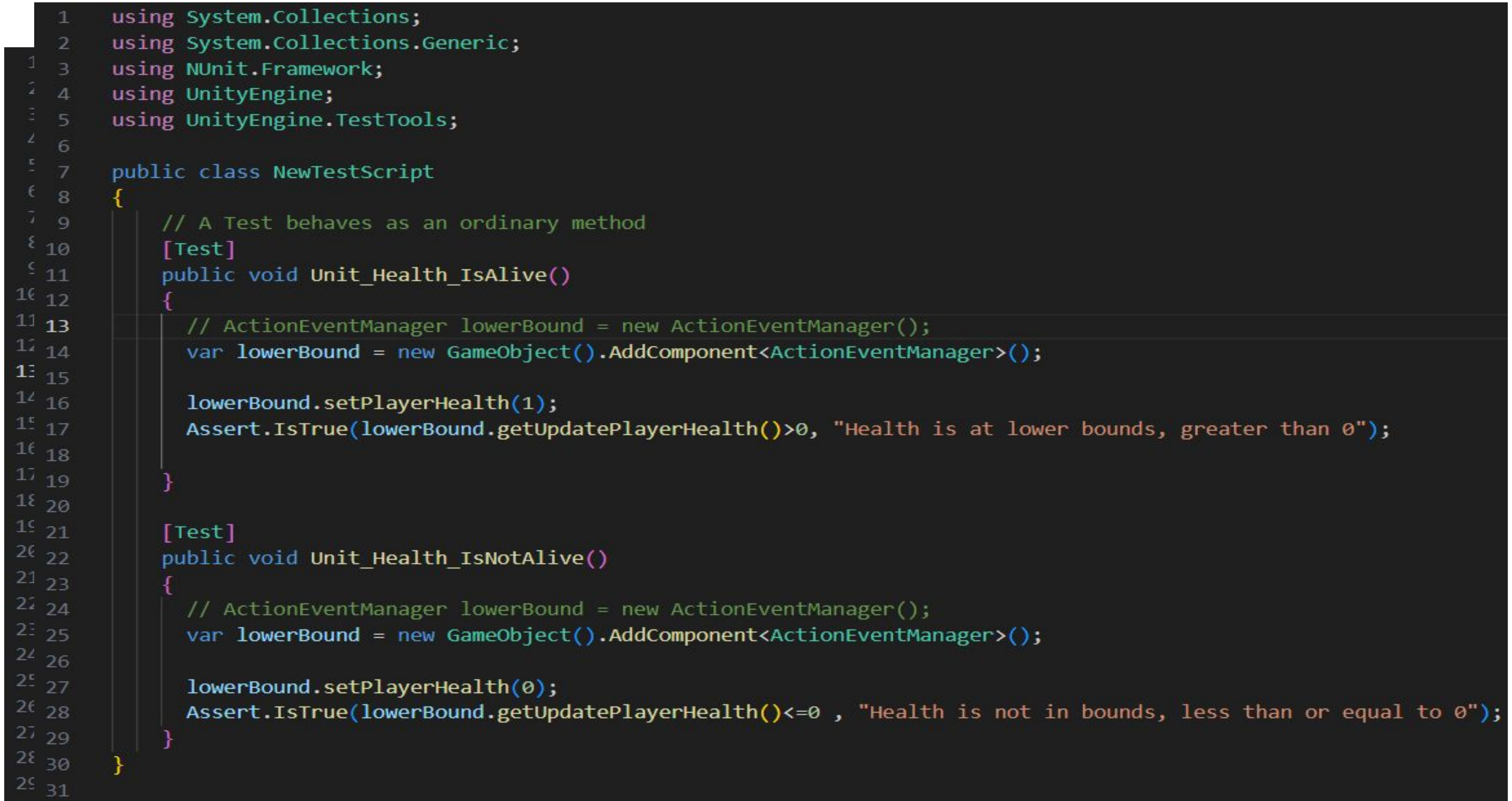
Technologies, U. (n.d.). *Testing and quality assurance tips for unity projects*.
https://unity.com/how-to/testing-and-quality-assurance-tips-unity-projects#performance-testing

CS 50 software design and implementation. Home. (2014, February 11).
https://www.cs.dartmouth.edu/~campbell/cs50/artoftesting.html

Brent Knopp TL3

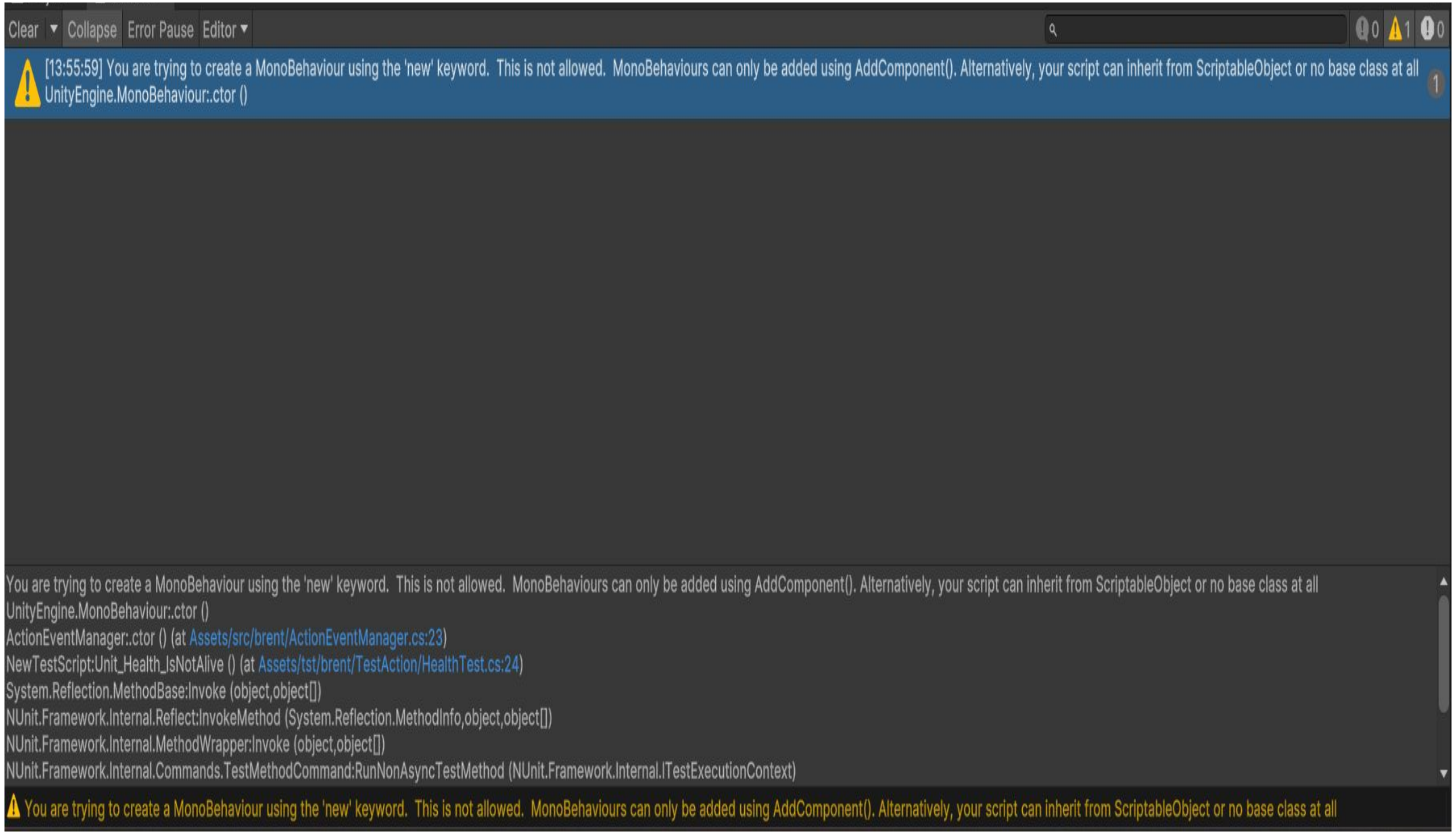
Team: Banana Tactics

Testing Script Update



```
using System.Collections;
using System.Collections.Generic;
using NUnit.Framework;
using UnityEngine;
using UnityEngine.TestTools;

public class NewTestScript
{
    // A Test behaves as an ordinary method
    [Test]
    public void Unit_Health_IsAlive()
    {
      // ActionEventManager lowerBound = new ActionEventManager();
      var lowerBound = new GameObject().AddComponent<ActionEventManager>();

      lowerBound.setPlayerHealth(1);
      Assert.IsTrue(lowerBound.getUpdatePlayerHealth()>0, "Health is at lower bounds, greater than 0");

    }

    [Test]
    public void Unit_Health_IsNotAlive()
    {
      // ActionEventManager lowerBound = new ActionEventManager();
      var lowerBound = new GameObject().AddComponent<ActionEventManager>();

      lowerBound.setPlayerHealth(0);
      Assert.IsTrue(lowerBound.getUpdatePlayerHealth()<=0 , "Health is not in bounds, less than or equal to 0");
    }
}
```

Test Warning

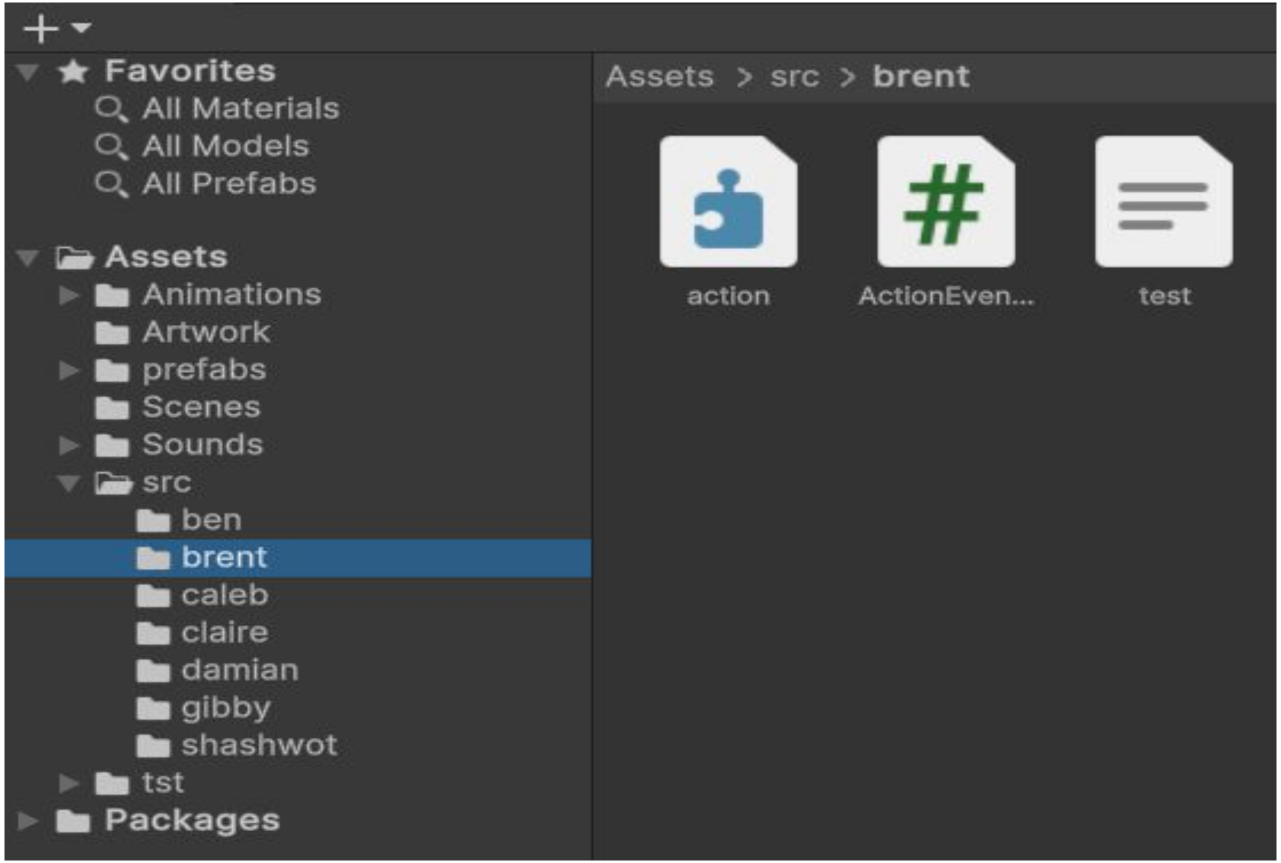


```
[13:55:59] You are trying to create a MonoBehaviour using the 'new' keyword.  This is not allowed.  MonoBehaviours can only be added using AddComponent(). Alternatively, your script can inherit from ScriptableObject or no base class at all
UnityEngine.MonoBehaviour:.ctor ()
1
```

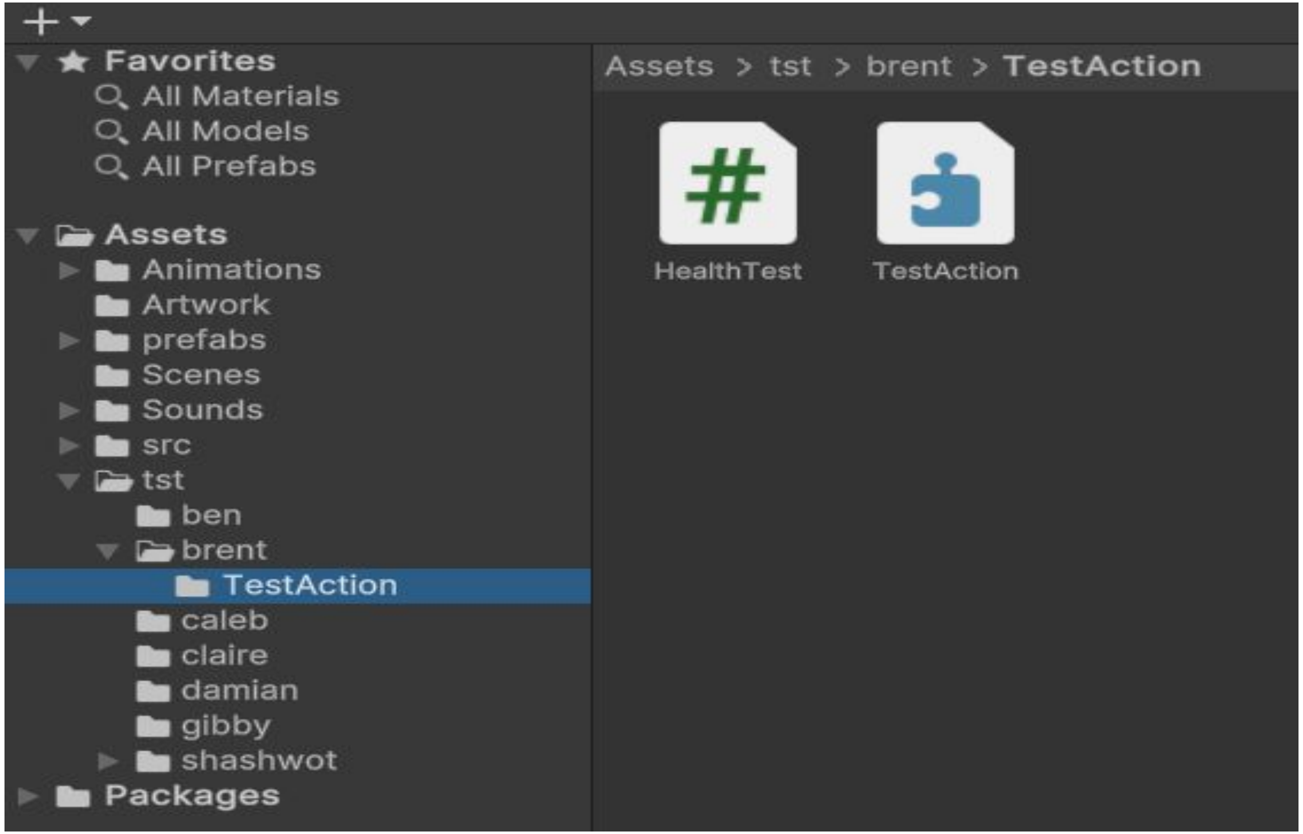
```
You are trying to create a MonoBehaviour using the 'new' keyword.  This is not allowed.  MonoBehaviours can only be added using AddComponent(). Alternatively, your script can inherit from ScriptableObject or no base class at all
UnityEngine.MonoBehaviour:.ctor ()
ActionEventManager:.ctor () (at Assets/src/brent/ActionEventManager.cs:23)
NewTestScript:Unit_Health_IsNotAlive () (at Assets/tst/brent/TestAction/HealthTest.cs:24)
System.Reflection.MethodBase:Invoke (object,object[])
NUnit.Framework.Internal.Reflect:InvokeMethod (System.Reflection.MethodInfo,object,object[])
NUnit.Framework.Internal.MethodWrapper:Invoke (object,object[])
NUnit.Framework.Internal.Commands.TestMethodCommand:RunNonAsyncTestMethod (NUnit.Framework.Internal.ITestExecutionContext)
```

```
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
```

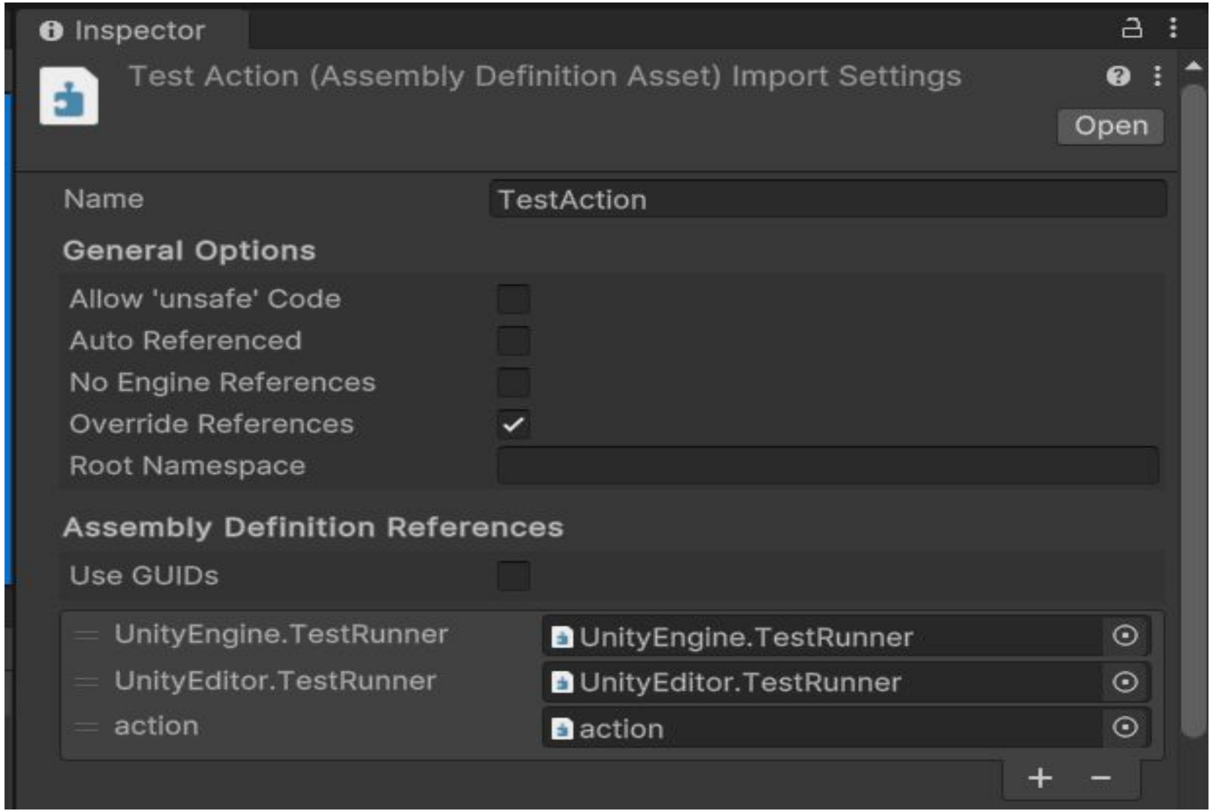
GitHub Testing Folder Structure



GitHub Testing Folder Structure



Inspector Window



Inspector

Test Action (Assembly Definition Asset) Import Settings

Open

Name: TestAction

General Options

Allow 'unsafe' Code: ☐
Auto Referenced: ☐
No Engine References: ☐
Override References: ✓
Root Namespace:

Assembly Definition References

Use GUIDs: ☐

UnityEngine.TestRunner	UnityEngine.TestRunner
UnityEditor.TestRunner	UnityEditor.TestRunner
action	action

Testing Question

Any Questions?