Unit Testing and Event Based Programming: A Guide to Best Practices in Unity

By Kurt Tikoft

1. Automatic Testing: What and Why?

1.1. Introducing Automated Testing

In agile game development, one of the most important aspects of development is consistent and iterative testing. As software, games are immensely complex, with huge systems and long lists of dependencies. Testing is always an important part of development for this reason, detecting bugs that emerge from the interactions of these complex systems. Game developers often rely on manual testing, conducted by human testers, to achieve their testing goals.

If you've never worked in non-game software development, though, it might surprise you to learn that this is not true of all software. In fact, programmers outside the game industry have been relying on automated testing tools for decades.

These tools use external software, separate to the software being developed, to execute tests automatically. There are different types of tests that can be executed, but in general they involve running code from the game, then comparing the outcome to an expected outcome.

These tests, in many cases, run outside the game logic and rendering, and can repeated throughout development as required.

Integrating these tests into the development process can save time and resources. When implemented proactively, they can be run each time a game builds to validate previously written code automatically. They are not a silver bullet for all forms of testing, but they can simplify the most repetitive software development tasks.

This guide is intended to assist developers who want to implement some of Unity's automated testing tools into their development cycle. It will cover what makes a game most suitable for automated testing in terms of design, how to engineer a game with automated testing in mind, and how to implement automated testing in Unity in a practical sense.

The information in this guide is based partly on academic research, and partly based on my experience implementing automated testing in my own game, *Your Holy & Virtuous Heretic* (*YHVH*). The game is a turn-based RPG with atmospheric exploration; I will provide necessary context into the mechanics, but if you would like to find out more, the game has a store page on Steam.

1.2. Does My Game Benefit from Automated Testing?

One reason that automated testing is less used in games, as compared to traditional software development, is that it is more situational. Developers must consider both the phase of production the game is in, and the design of the game in question.

Automated-friendly production

The most obvious weakness of automated testing is that it cannot evaluate a game for human enjoyment or fun. Manual testing is still required to understand the full effect of game mechanics on the subjective experience of the player. Lewis and Whitehead (2011) analysed video game software engineering practices in their research paper.

After interviewing game industry professionals, they found that game developers were not interested in automated testing technologies that couldn't answer these subjective questions (Lewis & Whitehead, 2011, p. p3). Speaking from personal experience, subjective questions do need to be asked on top of a codebase that is functional, so this acts moreso as an argument to include both automated and manual testing, but the point stands.

They found that the unpredictable and subjective output of game code means that it has emergent production properties, meaning that its goals and outputs are unpredictable; code can be changed up until near the end of development (Lewis & Whitehead, 2011, p. 2). This is a reality of the rapid iteration that games often demand but could result in automated tests that have been written being broken or made obsolete.

Because of these factors, automated testing should be implemented on aspects of a game that are unlike to be frequently overhauled or be implemented such that the tests are stable even if the underlying elements are changed.

That said, if a developer is leaving the implementation of automated testing for a later stage of development, they should still have its implementation planned and accounted for in the game's production and software engineering. As section 2 will detail, loosely coupled software architecture will greatly inform your success in automated testing.

Anecdotally, *Your Holy & Virtuous Heretic* was not originally developed with efficient software architecture, and I didn't know about automated testing until later in development. The game was essentially refactored and restructured to allow for it, which was a painful and arduous process. The result was better code that is more easily tested and maintained, but my lack of planning in this respect created a production roadblock.

Automation-friendly design

Game design that is most open to automated testing is, ideally, deterministic. Automated tests themselves must have reproducible results, and thus the elements that are being tested must also have reproducible results. The degree to which gameplay can be reproduced determines how much of it can be tested automatically, and how much must be tested manually.

For example, in a roguelike, it might be possible to test underlying gameplay systems such as attacks, item usage, or elemental damage systems. However, within the scope of the types of testing described in this document, it would be difficult to create meaningful tests related to levels or environment interactions as levels would be different every time.

The other important factor is the usefulness of the tests themselves. In his thesis about game testing, Cho recommends that automated testing is focused on tasks that need to be tested frequently, are time-consuming to test, or tedious to test (Cho, 2022, p. 76). A good example would be features that have a wide range of possible inputs and outputs across many contexts. Features that have a narrow range of inputs and outputs do not need to be tested as often, and therefore benefit less from automated testing.

Examples

Masella (2021) explains that *Sea of Thieves* was an ideal candidate for automated testing. This is because the game is an open world game with systemic elements that are complex, but ideally consistent (Masella, 2021). In this case, the nature of *Sea of Thieves* as a live service game with frequent updates also meant that new features would create unintentional, regressive bugs in old code, which would otherwise be time-consuming to test for (Masella, 2021).

As an example, in *Your Holy & Virtuous Heretic*, automated testing is not really used for the game's overworld exploration in most cases. This is because most elements in the overworld have a narrow range of inputs and outputs – puzzle elements can only be on or off, cutscenes are generally linear in nature.

However, automated testing is used extensively for the turn-based battle system. Enemies can have many different statuses, and the player's spells have many different effects, so automated tests are used to check all of the game's spells against all of the possible enemy statuses. If a new spell causes an issue with a previously created status, this system lets me find out very quickly.

Automated testing is also used for the game's negotiation system. Separate from cutscenes, *Your Holy & Virtuous Heretic* allows players to enter dialogue with enemies. The dialogue itself takes the form of a simple dialogue tree, but each enemy has a different tree, with different steps to achieve similar outcomes. Some dialogue choices also have different results depending on the context, such as how many other enemies are in the battle, but there is always a correct and consistent set of choices and an ideal state. Recreating all of these conditions would be time-consuming and difficult to test, so each of the game's demon conversations are automatically tested with a variety of contexts, saving a significant amount of time and allowing me to control the sale of this complex system.

Summary

Essentially, when considering if a mechanic, feature, or element of the game should have automated tests written for it, developers should consider:

- Is the expected output of the element consistent, or inconsistent?
- Does testing the element take up a lot of time and effort?
- Is the range of inputs and outputs narrow, or broad?

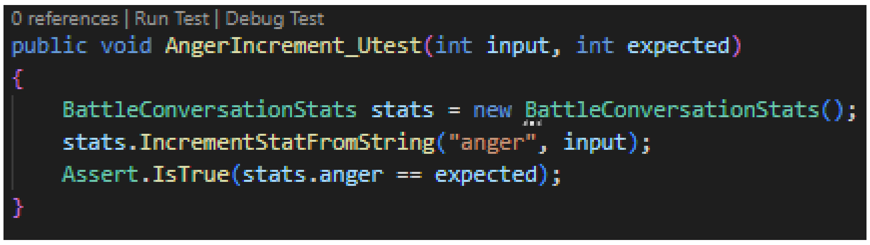
Elements that are consistent, but time consuming to test, with a wide range of inputs and outputs would absolutely benefit from automated testing.

2. The Unity Test Framework

2.1. Core Test Types

Automated testing is a broad field, and as such there are many different ways to design automated tests. This guide will cover two main types of automated testing that are possible with the Unity Test Framework.

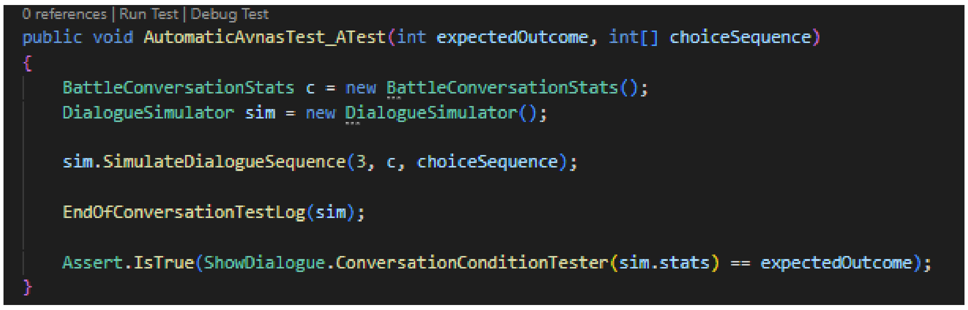
Unit Testing involves writing simple, short tests that test one "unit" of code, usually a single function or method. By executing many of these tests with multiple test cases, developers can diagnose simple bugs before they progress in the development pipeline.



```
0 references | Run Test | Debug Test
public void AngerIncrement_Utest(int input, int expected)
{
    BattleConversationStats stats = new BattleConversationStats();
    stats.IncrementStatFromString("anger", input);
    Assert.IsTrue(stats.anger == expected);
}
```

This is an example of a unit test used in *YHVH*. It tests a single function used by the game's negotiation mechanic to increment a stat from a given string.

Actor Testing uses more complex tests that create instances of game entities, without fully initialising the game scene or game loop, and then uses game functions to simulate their interaction.



```
0 references | Run Test | Debug Test
public void AutomaticAvnasTest_ATest(int expectedOutcome, int[] choiceSequence)
{
    BattleConversationStats c = new BattleConversationStats();
    DialogueSimulator sim = new DialogueSimulator();

    sim.SimulateDialogueSequence(3, c, choiceSequence);

    EndOfConversationTestLog(sim);

    Assert.IsTrue(ShowDialogue.ConversationConditionTester(sim.stats) == expectedOutcome);
}
```

This is an example of an actor test used in *YHVH*. It initialises only the necessary objects to test the game's negotiation mechanics and tests a given array of choices before evaluating the conversation state.

Other test types are possible with the Unity Test Framework. For example, integration tests are similar to actor tests, but involve running specialised test scenes within the full game logic with simulated player input. These can be useful, but I will be omitting these for the sake of brevity as they have their own realm of standards and practices. Their core requirements are identical to unit and actor tests.

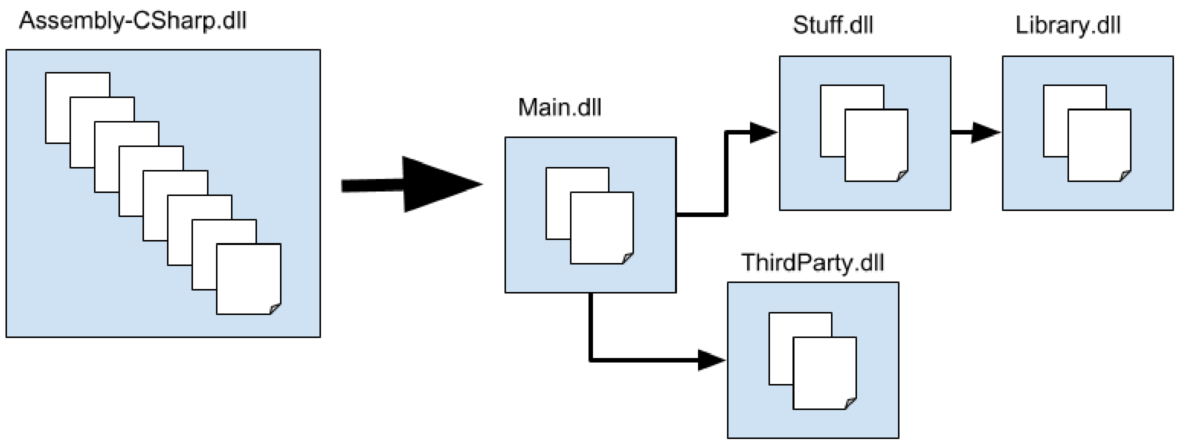
2.2. About UTF

Unity Test Framework (UTF) is, on paper, included in any version of Unity after 2019.2. However, getting your project to the point where a developer can write tests for it requires some considerations. UTF requires the use of a Unity software engineering feature called Assembly Definitions. To get the most out of both Assembly Definitions and UTF, users will need to implement Event Based Programming practices.

2.3. Assembly Definitions

When Unity compiles a project's scripts, it creates code libraries in .dll files. By default, all of the C# scripts in a project are compiled into a single default assembly, "Assembly-CSharp.dll". This is fine for some projects, but Unity has no knowledge of which scripts are dependent on each other.

An Assembly Definition defines all of the C# scripts in a given project folder as belonging to a separate assembly to the default assembly. Other assemblies can be defined as dependencies, as long as this doesn't result in an assembly being dependant on itself. Scripts in that assembly can only reference other scripts from the same assembly, or scripts from its dependencies.



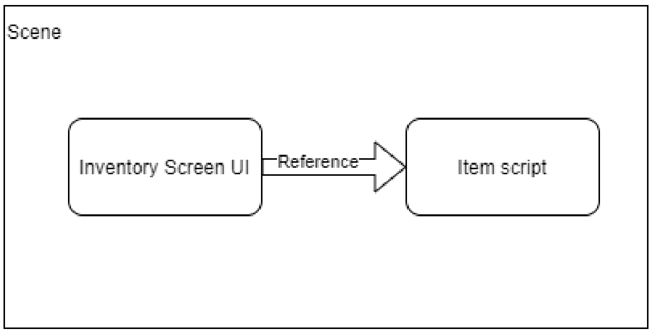
In this diagram provided in the Unity manual, the arrows represent dependencies. A script from 'Library.dll' can reference scripts in 'Stuff.dll' or 'Main.dll', but not 'ThirdParty.dll'. 'Stuff.dll' can reference scripts in 'Main.dll', but not 'Library.dll'. If 'Main.dll' defined 'Library.dll' as a dependency, this would cause a compile error, as it would make 'Main.dll' dependent on itself.

Assembly definitions encourage loosely coupled code, meaning that scripts aren't dependent on each other, therefore being easier to maintain. However, if code is separated this way, it could be difficult or impossible to create game elements such as UI, which needs to both send and receive information from the same components. The intended solution for this is event-based programming. While it is possible to implement assembly definitions and UTF without event-based programming, it is critical to understand these ideas to implement automated testing effectively.

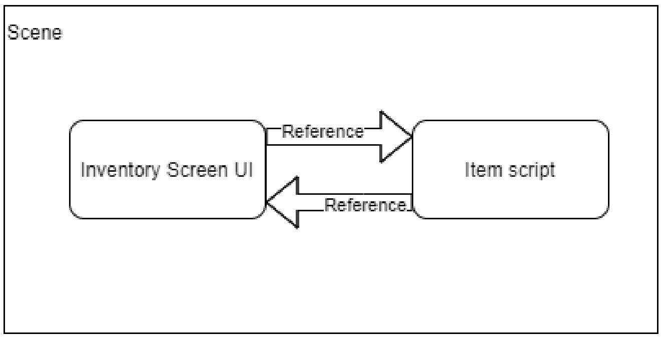
2.4. Event Based Programming

Event-based programming (also known as event-driven programming, or the Observer Pattern) is an approach to connecting scripts together without two-way references, as well as moving game logic out of the central Update loop.

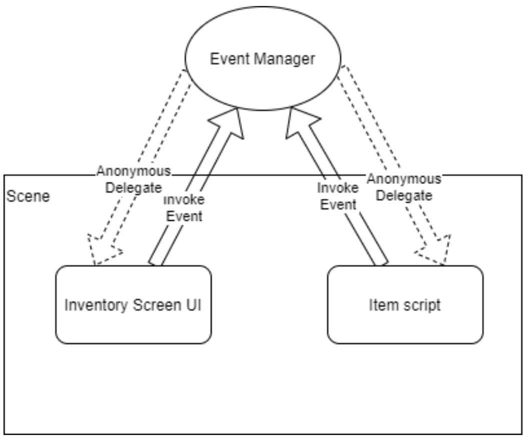
If a developer wants two scripts to communicate with each other in Unity (for example, an inventory screen UI script and an item script), the simplest way to do so is to create a script reference with a serialized field and assign the reference in the inspector.



This means that the script containing the reference is now dependent on the script being referenced. If information needs to return from the item back to the inventory screen later on, maybe after loading some data or completing an animation, then both scripts would need to have a reference with this approach, resulting in a circular dependency between these scripts.

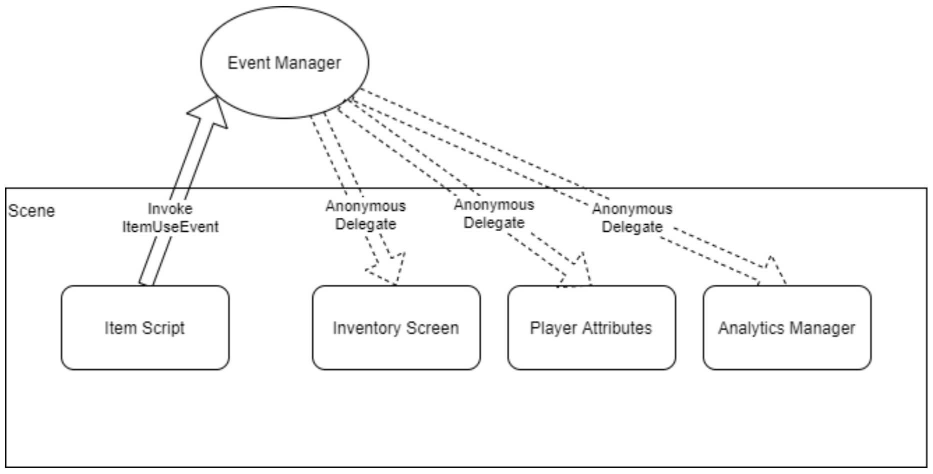


The solution to this is an event system. In this framework, the original script reference from the is replaced with a call to an event manager. Any method that would normally be referenced through this system is stored as an anonymous functional call, using a C# delegate or a Unity Action. Instead of the inventory screen calling the item script directly, it calls the event manager and invokes the event, and the event manager anonymously calls the item.



The result is that scripts that would be tightly coupled, or that would have circular dependency, are now mutually dependent on the event manager but not dependent on each other.

An Event Manager can also have multiple methods that are called when a single event is invoked, greatly simplifying the structure of complex code that affects multiple components.



This is called the “Observer Pattern”, because of how multiple components “observe” the event manager, rather than being directly called by the original component.

This is not a unique feature of Unity or C#, and as such there are many options that can operate as the event manager or event type in this case. C# has a native system using EventHandler and delegate objects, though it has some restrictions. Unity also has its own UnityEvent and UnityAction system, which is designed to cleanly integrate into its input and UI systems, although this can have a cost to performance. There are also third-party assets that replace the existing Unity and C# event managers.

The specifics of implementing event-based programming are well-documented, and ultimately differ between projects, and as such this guide will not detail the specifics of these approaches. However, it is important that one of these systems is in place to ensure your game’s codebase is loosely coupled, and therefore optimised for automated testing.

Why is this required for automated testing?

To use UTF, Unity defines a special assembly definition for tests only, called the "Test Assembly". Any scripts that a developer wants to run tests on needs to be defined in an assembly definition, which must be set as a dependency of the test assembly.

It's possible to recreate Unity's default compilation behaviour by simply creating an assembly definition encompassing all of the project's scripts and setting it as a dependency for the test assembly; some online resources about automated testing in Unity recommend this. Speaking from experience, this can have long-term problems.

Unit tests and actor tests, in particular, are designed to be highly compartmentalised; unlike integration tests, they do not run in normal scenes and are meant to be repeated many times with consistent outputs. They should ideally be initialised and executed in code, without rendering. Lacking event-based programming and/or proper use of assembly definitions can result in issues with initialisation with any scripts that require non-static references.

If the project structure uses a lot of references, not making use of events, initialisation becomes complex. If the method being tested relies on any scripts that would normally be set in the inspector, the developer must implement a way for the references to be set in code. This can quickly make the "Setup" phase of the test complicated, and possibly introduce bugs into the test itself, defeating the point of automated testing.

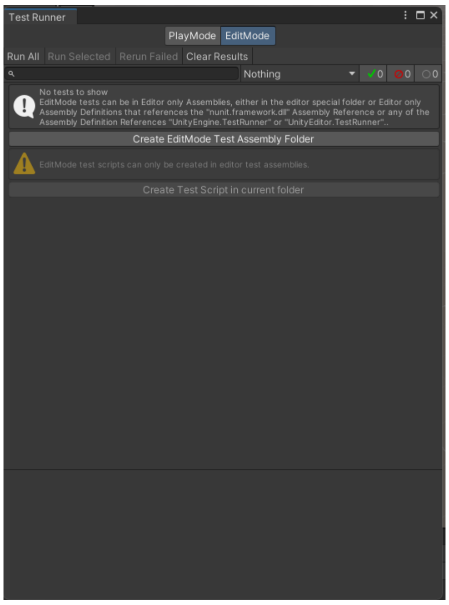
Even with event-based structure, running the Unity Test Framework on a single large assembly without defined dependencies has a tendency towards instability. Reports on the Unity Discussions forum report errors coming from UTF that require Unity to be completely uninstalled and reinstalled, even when running code from official tutorials. Speaking to personal experience, when I first attempted to utilise UTF without structured assembly definitions, I could only occasionally run unit or actor tests without Unity crashing. With better implementation of assembly definitions, this behaviour disappeared completely, and UTF has been stable since then. As of 30/10/2024, Unity has not addressed this directly.

In essence, the implementation of assembly definitions and event-based programming structures makes automated testing both more stable and simpler. It reduces complexity in both writing and running tests.

3. Writing Tests, Step By Step

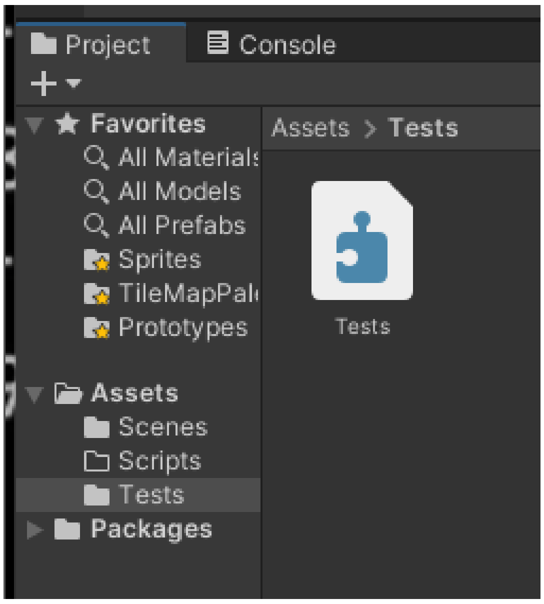
3.1. Setup

Creating the Test Assembly is as simple as opening the Test Runner window (Window > General > Test Runner). From the project view, navigate to the folder you wish to store your automated test files in, then click “Create EditMode Test Assembly Folder”. By default, this will create a folder called Tests, and an assembly definition called Tests.

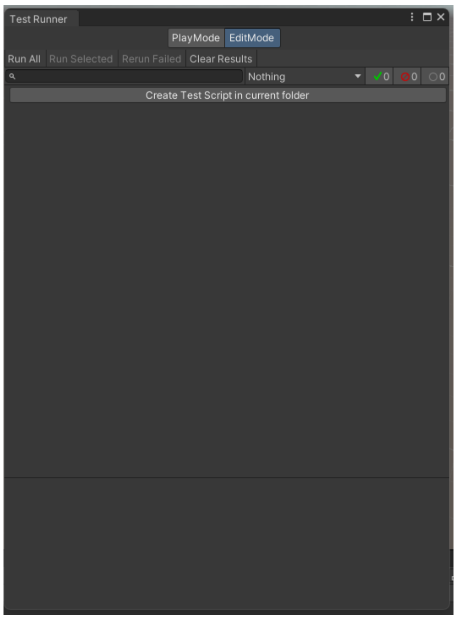


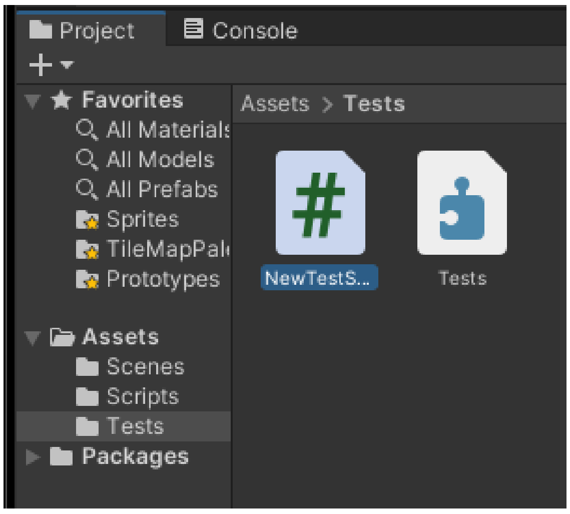
No tests to show
EditMode tests can be in Editor only Assemblies, either in the editor special folder or Editor only Assembly Definitions that references the "nunit.framework.dll" Assembly Reference or any of the Assembly Definition References "UnityEngine.TestRunner" or "UnityEditor.TestRunner"..

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



The window will now have a button labelled “Create Test Script in current folder”. Clicking this will create a stub unit test script.

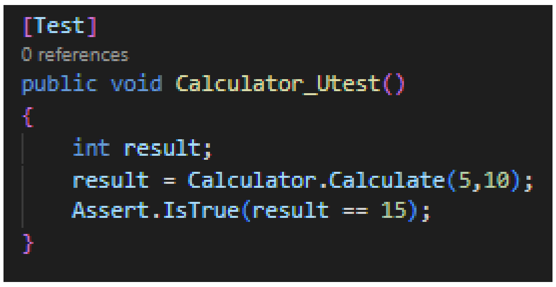




3.2.Writing an automated test

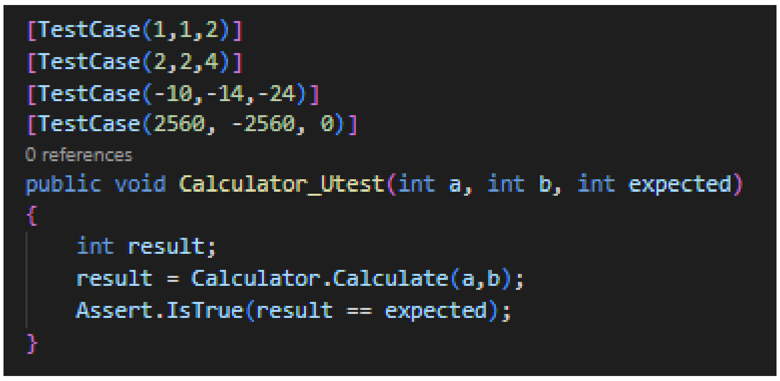
Tests are written in plain C#, with some additions. The first is test attributes before the method, which define how the test will be run.

The [Test] attribute defines the test as an NUnit test, the basic test type that UTF extends from. These tests must have the return type "void", and all test logic will be executed in one single frame.



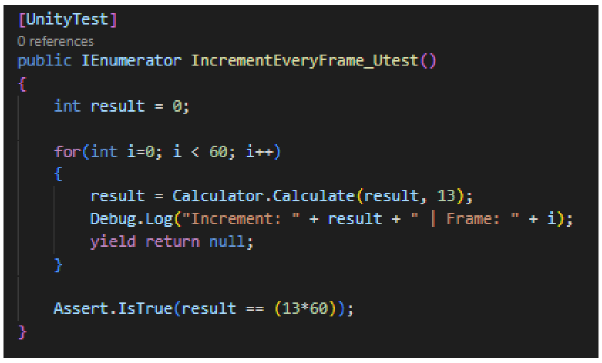
```
[Test]
0 references
public void Calculator_Utest()
{
    int result;
    result = Calculator.Calculate(5,10);
    Assert.IsTrue(result == 15);
}
```

NUnit tests can also have [TestCase()] attributes. This allows a developer to add parameters to the test method, then specify values for these parameters in attributes as test cases. Each test case will be run as a separate test under the same method. By doing this, developers can reuse code, ensuring stable and consistent tests and saving time.



```
[TestCase(1,1,2)]
[TestCase(2,2,4)]
[TestCase(-10,-14,-24)]
[TestCase(2560, -2560, 0)]
0 references
public void Calculator_Utest(int a, int b, int expected)
{
    int result;
    result = Calculator.Calculate(a,b);
    Assert.IsTrue(result == expected);
}
```

The [Test] attribute can be replaced with the [UnityTest] attribute, defining the test as a UnityTest. These tests are run as IEnumerators, meaning that they can simulate the iteration of frames with the line 'yield return null'. This means that they can test functions that affect the game state in the following frame, such as GameObject.Destroy(). Any yield instruction can be used; yield return new WaitForSeconds() could be used to test over seconds of real world time.

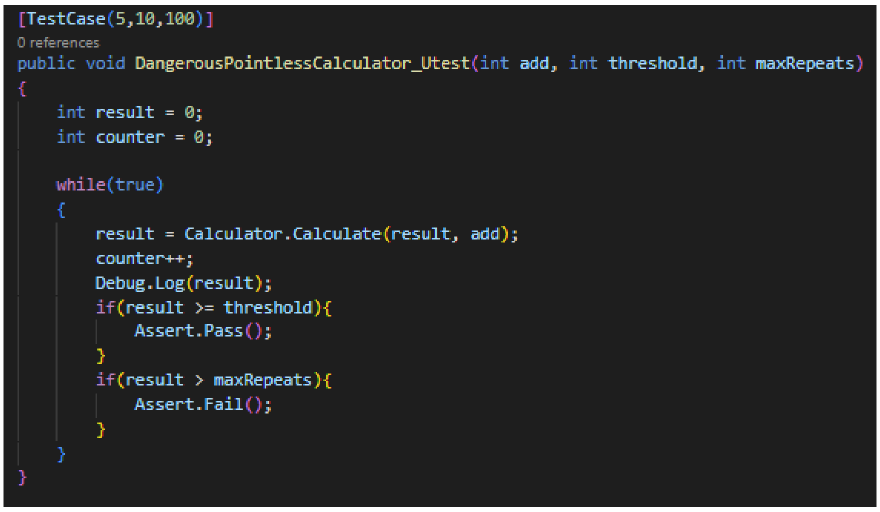


```
[UnityTest]
0 references
public IEnumerator IncrementEveryFrame_Utest()
{
    int result = 0;

    for(int i=0; i < 60; i++)
    {
        result = Calculator.Calculate(result, 13);
        Debug.Log("Increment: " + result + " | Frame: " + i);
        yield return null;
    }

    Assert.IsTrue(result == (13*60));
}
```

Lastly, each test will always be reported as successful unless they contain a method call from the Assert class. These test the expected game state at the end of the test. Assert.IsTrue is the simplest, testing if a boolean variable or statement is true. Other methods allow for more complex logic; for example, Assert.NotNull can test for successful initialisation of an object, Assert.Pass() and Assert.Fail() allow the test to be terminated early with successful or failed results. The full list of Assert methods is available here.



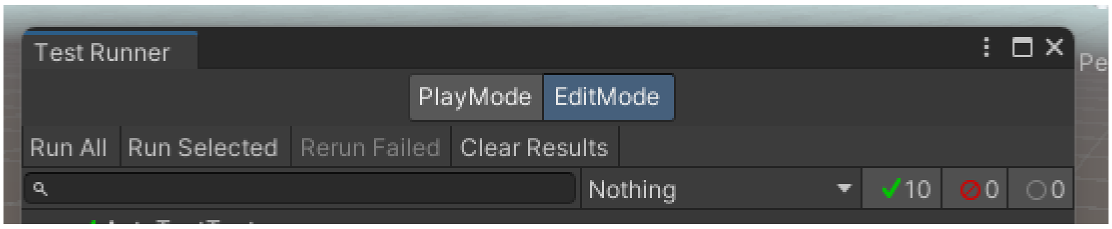
```
[TestCase(5,10,100)]
0 references
public void DangerousPointlessCalculator_Utest(int add, int threshold, int maxRepeats)
{
    int result = 0;
    int counter = 0;

    while(true)
    {
        result = Calculator.Calculate(result, add);
        counter++;
        Debug.Log(result);
        if(result >= threshold){
            Assert.Pass();
        }
        if(result > maxRepeats){
            Assert.Fail();
        }
    }
}
```

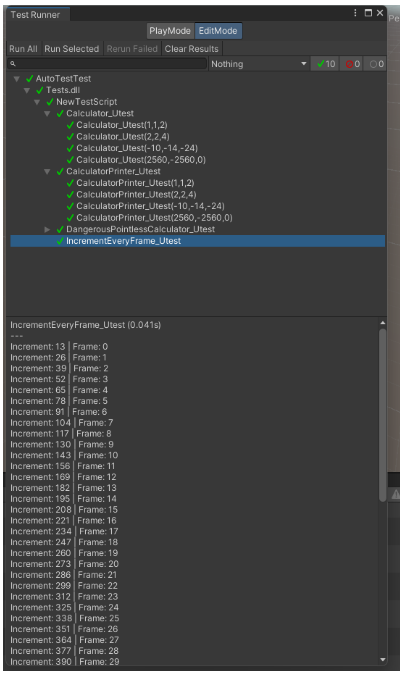
3.3. Running tests

To run your brand-new tests, return to Unity's Test Runner window. Unity separates the tabs in this window depending on the context they run in. Unit and Actor tests only need to be run in the Editor mode, so they should appear under the EditMode tab. Here, tests marked with the [Test] or [UnityTest] attribute will appear in a hierarchy tree.

Simply clicking "Run All" will run every test in the hierarchy. "Run Selected" will run only the test you have highlighted in the inspector. Any tests that fail can easily be ran again with "Rerun Failed".



If your test method prints to the console at any point, the console output will also be written to a results window below the hierarchy. This is useful for mid-test status updates, and any supplemental information that contextualise your tests.



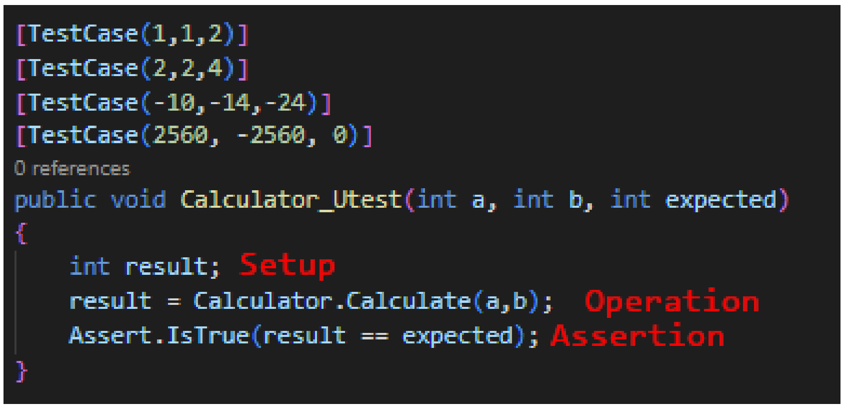
```
▼ ✓ AutoTestTest
  ▼ ✓ Tests.dll
    ▼ ✓ NewTestScript
      ▼ ✓ Calculator_Utest
          ✓ Calculator_Utest(1,1,2)
          ✓ Calculator_Utest(2,2,4)
          ✓ Calculator_Utest(-10,-14,-24)
          ✓ Calculator_Utest(2560,-2560,0)
      ▼ ✓ CalculatorPrinter_Utest
          ✓ CalculatorPrinter_Utest(1,1,2)
          ✓ CalculatorPrinter_Utest(2,2,4)
          ✓ CalculatorPrinter_Utest(-10,-14,-24)
          ✓ CalculatorPrinter_Utest(2560,-2560,0)
      ▶ ✓ DangerousPointlessCalculator_Utest
        ✓ IncrementEveryFrame_Utest
```

```
IncrementEveryFrame_Utest (0.041s)
---
Increment: 13 | Frame: 0
Increment: 26 | Frame: 1
Increment: 39 | Frame: 2
Increment: 52 | Frame: 3
Increment: 65 | Frame: 4
Increment: 78 | Frame: 5
Increment: 91 | Frame: 6
Increment: 104 | Frame: 7
Increment: 117 | Frame: 8
Increment: 130 | Frame: 9
Increment: 143 | Frame: 10
Increment: 156 | Frame: 11
Increment: 169 | Frame: 12
Increment: 182 | Frame: 13
Increment: 195 | Frame: 14
Increment: 208 | Frame: 15
Increment: 221 | Frame: 16
Increment: 234 | Frame: 17
Increment: 247 | Frame: 18
Increment: 260 | Frame: 19
Increment: 273 | Frame: 20
Increment: 286 | Frame: 21
Increment: 299 | Frame: 22
Increment: 312 | Frame: 23
Increment: 325 | Frame: 24
Increment: 338 | Frame: 25
Increment: 351 | Frame: 26
Increment: 364 | Frame: 27
Increment: 377 | Frame: 28
Increment: 390 | Frame: 29
```

3.4. Test Design and Philosophy

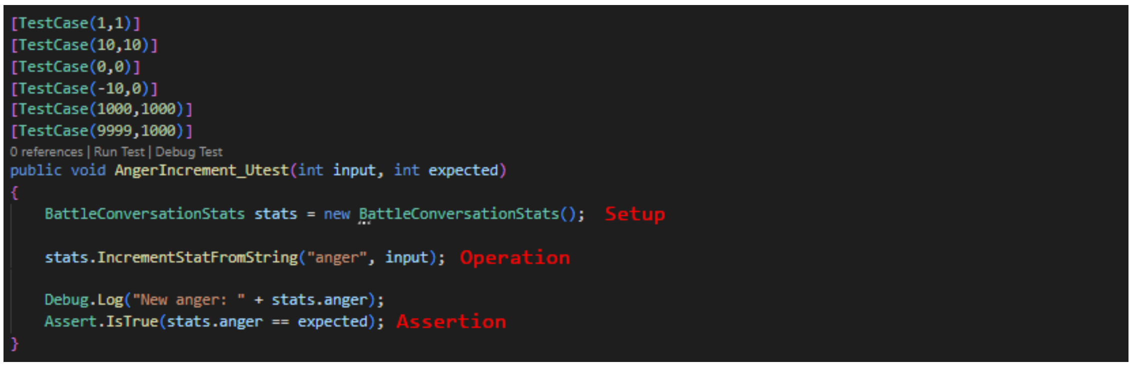
Structure and Usage

Each test has three phases: setup, operation, and assertion. **Setup** initialises the test variables, **operation** executes the code being tested, **assertion** defines the intended output. This test example could be condensed into one line, but I've expanded and labelled it for clarity.



```
[TestCase(1,1,2)]
[TestCase(2,2,4)]
[TestCase(-10,-14,-24)]
[TestCase(2560, -2560, 0)]
0 references
public void Calculator_Utest(int a, int b, int expected)
{
    int result; Setup
    result = Calculator.Calculate(a,b);  Operation
    Assert.IsTrue(result == expected); Assertion
}
```

Here are some more representative real-world examples from *Your Holy & Virtuous Heretic*. This first test is a unit test, validating a basic function of the game's negotiation system.

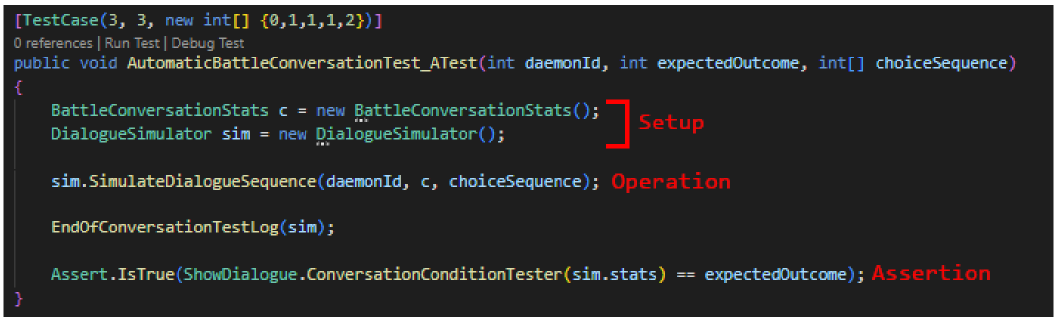


```
[TestCase(1,1)]
[TestCase(10,10)]
[TestCase(0,0)]
[TestCase(-10,0)]
[TestCase(1000,1000)]
[TestCase(9999,1000)]
0 references | Run Test | Debug Test
public void AngerIncrement_Utest(int input, int expected)
{
    BattleConversationStats stats = new BattleConversationStats();  Setup

    stats.IncrementStatFromString("anger", input);  Operation

    Debug.Log("New anger: " + stats.anger);
    Assert.IsTrue(stats.anger == expected);  Assertion
}
```

The second is an actor test, validating the critical path of an enemy's dialogue tree. In *Your Holy & Virtuous Heretic*, conversations with enemies can be initialised with only an enemy ID; the test case attribute determines the enemy to test, the expected return from the dialogue system, and an array of choices to make during the conversation.



```
[TestCase(3, 3, new int[] {0,1,1,1,2})]
0 references | Run Test | Debug Test
public void AutomaticBattleConversationTest_ATest(int daemonId, int expectedOutcome, int[] choiceSequence)
{
    BattleConversationStats c = new BattleConversationStats();  ] Setup
    DialogueSimulator sim = new DialogueSimulator();            ]

    sim.SimulateDialogueSequence(daemonId, c, choiceSequence); Operation

    EndOfConversationTestLog(sim);

    Assert.IsTrue(ShowDialogue.ConversationConditionTester(sim.stats) == expectedOutcome); Assertion
}
```

Technically, tests can contain any code the developer wants, as long as it has the appropriate attributes and an Assert method. The above structure simply ensures that tests are single-purpose and easy to debug – automated tests that are too convoluted and themselves need to be debugged defeat the point of automated testing.

When to run tests

Automated tests should be run as often as it is practical to do so. For most developers, this means that automated tests act as a validation layer before creating a new build of the game. Should a test succeed in one build, and fail in the next, it acts as a signal that something in the code has had regressive, possibly unintentional changes.

Assembly definitions lower Unity's compile times considerably, and can be set to run programmatically, and as such some developers have them automated to run after every compile. In the majority of cases, this will slow down development too much.

Cho emphasises that, no matter how and when automated tests are performed, it is important for writing and maintaining them to be factored into the production process (Cho, 2022, p. 76). Ensuring that it is

Where to go next?

UTF builds on the popular NUnit framework for unit tests, and there is a range of tools that can make automated testing with it more streamlined. For example, NSubstitute (linked here) can make "mock" instances of classes with predetermined outputs without needing to instantiate them. This may not always be ideal for actor testing, as this is less comprehensive than fully initialising relevant classes, but can make it easier to quickly create unit tests for complex behaviour.

As alluded to earlier in the guide, there are forms of automated testing that UTF makes possible that I have not covered. Integration tests allow developers to efficiently test complex interactions – they can also have their own issues, but Masella's GDC talk explores best practice for mitigating them (Masella, 2021). With the new Unity input system, simulating user input is more easy than ever, making this type of test more accessible.

There are also alternative types of automated testing that are not built on unit tests but could be beneficial to your production pipeline. Provinciano discusses how he used an "instant replay" system that simulated inputs to perform automated testing (Provinciano, 2017). While this does not require UTF, it could be a part of an integration testing setup.

4. References

Cho, J. (2022). *Bughunting on a Budget: Exploring Quality Assurance Practices and Tools for Indie Game Developers*. [Master's thesis] University of Alberta.

Lewis, C., & Whitehead, J. (2011). The whats and the whys of games and software engineering. *GAS '11: Proceedings of the 1st International Workshop on Games and Software Engineering* (pp. 1-4). Honolulu: Association for Computing Machinery.

Masella, R. (2021, September 24). *Automated Testing of Gameplay Features in 'Sea of Thieves'*. Retrieved from Youtube: https://youtu.be/X673tOi8pU8?si=hrhUv0NtNnAGPLfs

Provinciano, B. (2017, July 24). *Automated Testing and Instant Replays in Retro City Rampage*. Retrieved from Youtube: https://youtu.be/W20t1zCZv8M?si=uDNULfJ1Cj5E0uqF