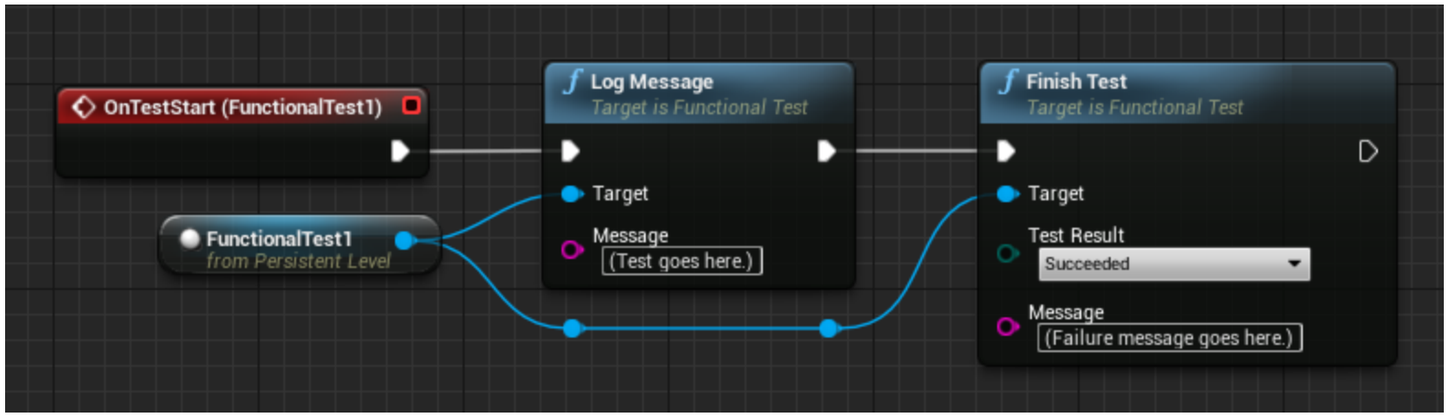
Functional Testing

Framework for Functional Testing.

Creating a Test

Setting up a test is done by placing a **Functional Test** **Actor** in a Level. That Actor can then be scripted to run a set of tests by the **Level Script** in the Level Blueprint. The test itself can be built into the Functional Test itself (as a child code class or Blueprint), or assembled directly in the **Level Script**.



A Level Blueprint running a placeholder Functional Test.

Functional Test Class Features

The Functional Test class provides the following important functions:

Function Name	Description
`PrepareTest`	The `PrepareTest` function can be overridden in a code or Blueprint child class. This function is the first to run, along with the `OnTestPrepare` delegate, and can be overridden to perform the initial setup required for your test. If this setup requires multiple frames, for example, if the setup requires streaming data to be loaded into the Level, building pathing data, or connecting to a server, this function should start those processes.
`IsReady`	This function is called on every tick after the initial `PrepareTest` call, and will continue to be called until `OnTestStart` is run. By default, this function returns `true`, which permits `OnTestStarted` to be called immediately. If the initialization processes set in motion by `PrepareTest` are incomplete, this function should return `false` to prevent the main test code from starting prematurely.

Function Name	Description
`OnTestStart`	This delegate is called when the **Functional Testing Manager** begins the test. Bind this to your test functionality, and be sure to call `FinishTest` at the end.
`OnTestFinished`	Once the test ends, this delegate will be called. Because testing often affects the Level or Actors within the Level in a way that impacts subsequent tests, using this opportunity to clean up can be vitally important to maintaining a usable testing environment.

The Functional Test class also provides the following support features:

Function or Property Name	Description
`OnAdditionalTestFinishedMessageRequest`	Implementing `OnAdditionalTestFinishedMessageRequest` is useful as a way to log additional information into the test summary.
`RegisterAutoDestroyActor`	Actors passed to this function will be destroyed automatically when the test ends. This is a good way to clean up Actors that were spawned as part of a test.
`LogMessage`	This function logs the text you provide to the `LogFunctionalTest` category in the **Output Log**. It is intended as a way to log progress while a test is running.
Observation Point	If an Actor is assigned to this Property, the Player will be teleported to its location and rotation when the test starts.
Enabled	This variable can be set to `false` to disable the test.

Testing Via Level Script

To run a Functional Test in the Level Script, first place a Functional Test Actor in the Level. With the Functional Test selected, open the Level Script and place the `OnTestStart` delegate and a reference to your Functional Test. By dragging off of the Functional Test's pin, you can create one or more `FinishTest` nodes. At this point, you can build your test so that it connects the `OnTestStart` delegate to the `FinishTest` node (or nodes, if your test contains branches). You may also create the `OnTestFinished` function if your test has done anything that requires cleanup. When this setup is complete, the Automation System will be able to run this test properly. This method is acceptable for simple tests, generally those that require little to no setup, and don't need to be run multiple times or on more than one Level.

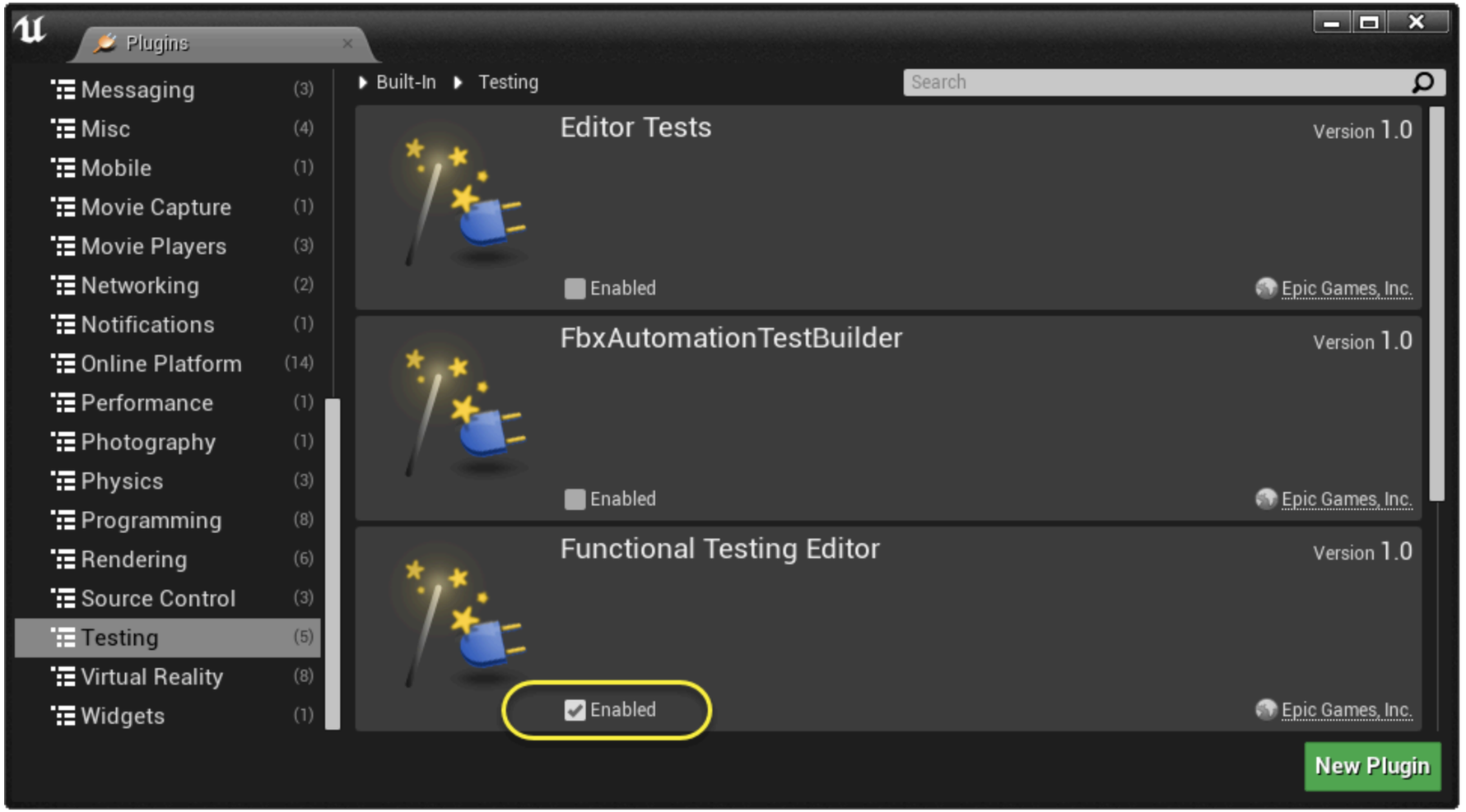
Adding events or references to an Actor (such as your Functional Test Actor) within the Level Script requires that the Actor in question is selected in the **Level Editor** or **World Outliner**.

Testing Via Child Class Method

If a Functional Test requires a more complex setup, or is intended to run multiple times (either in a single Level, or on multiple Levels), overriding `AFunctionalTest` is the recommended method. Extending the base Functional Test class in code or with Blueprints grants the ability to use `PrepareTest` and `IsReady` functions, which are important for running more complex or inter-dependent tests, or tests with setup times longer than one frame. Implementation of these tests via the Level Script is the same as before, although the bulk of the test code can now be contained in the new Functional Test class itself, rather than the Level Script, making the test easy to use in multiple Levels, or multiple times within the same Level.

Creating And Using Expected Test Results

Some tests have results that are large, complex, precise, or otherwise unsuited to manually-written solutions. In cases like this, it is usually helpful to run a single test, personally verify that the results are correct, and then save those results for comparison against the results of future tests. This concept is implemented as **Ground Truth Data** in the **Functional Testing Editor** plugin.



Using Ground Truth Data requires enabling the Functional Test Editor plugin.

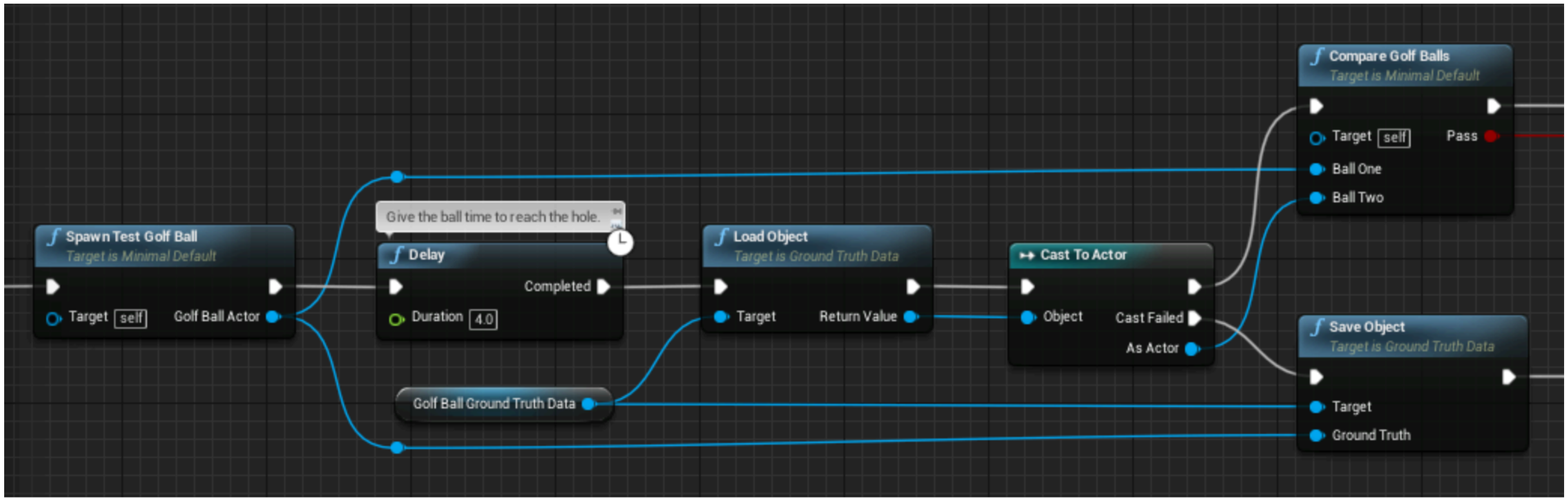
The concept of **Ground Truth Data**, implemented in the `UGroundTruthData` class, is used to store and compare these results. A Ground Truth Data Object stores an Object of your choosing as the "correct" result of whatever test you are performing. This Object can then be compared in any way you like to a

corresponding Object in your live simulation, and your logic can then determine if this constitutes passing or failing the test. For example, if your game requires precise, dependable physics simulation, such as a golf game, you might have a test where a simulated player hits a hole-in-one. Your Ground Truth Data could contain the golf ball as the Object, and you could compare the location of your live simulation's golf ball to the saved one (which, presumably, has landed in the hole), allowing for only a very small error without failing the test. A test like this could be designed as follows:

- Spawn a "test" golf ball at a certain location and initial velocity. Give it a fixed amount of time to travel. At this point, your golf ball should be in the ideal position. You can now attempt to load the "correct" golf ball from the Ground Truth Data for this test.
- If the Ground Truth Data does not return an acceptable Object (the returned Object is null or the wrong class, for example), then save the test golf ball we created to the Ground Truth Data as the expected result. If it turns out that the golf ball didn't land where it should have, the Ground Truth Data can always be reset by manually setting its `ResetGroundTruth` variable to `false` in the Blueprint Editor.

  > After checking the `ResetGroundTruth` box in the Blueprint Editor, it will automatically uncheck itself, but your data will be reset.

- If the Ground Truth Data does return an Object which can be cast to the desired class (in this example, Actor is sufficient), a comparison between that Object and the test golf ball can be made to evaluate whether or not the test should pass.



This test checks for consistent location of a "test golf ball" that has been given four seconds of travel time.

Tests set up in this manner can run the test and set the expected test result with the same script. The expected result can be reset by editing the Ground Truth Data in the Editor.