**Testing and Test Automation in Game Development(Unity)**

In Unity, there are two types of tests:

* **Edit Mode** — running in the editor
* **Play Mode** — running in the game

You can find the Test Runner window under **Window** > **General** > **Test Runner**.

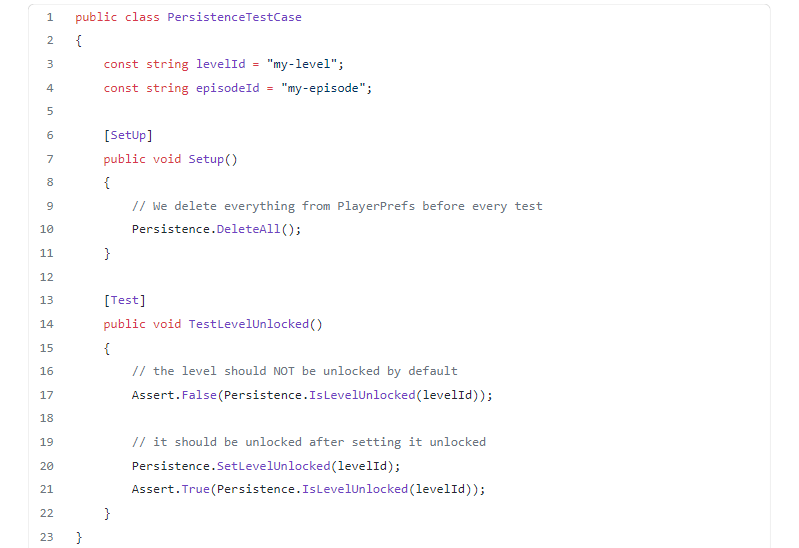
**Edit Mode Tests**

Edit Mode Tests run in the editor, and the game is not started. You can use them anytime when you don’t need the game to be running. The advantage is that Edit Mode Tests are significantly faster than Play Mode Tests. Some examples of usage are:

* Standard unit tests when you want to check a single unit without the need to access anything else, e.g., to verify that a function does what it should.
* Integration tests that do not require running game, e.g., accessing PlayerPrefs.
* Check Scenes, Game Objects and their Components.
* Check Scriptable Objects or other files that you use.

**Persistence**

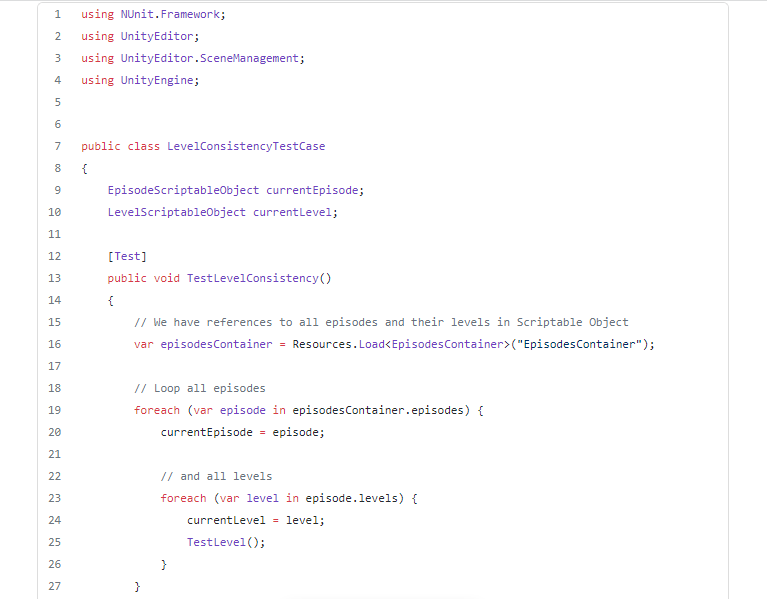
We have a class that serves as a facade for accessing PlayerPrefs. It is a good idea to write some tests, whether it works correctly. We use a SetUp method to clear the data before every test.



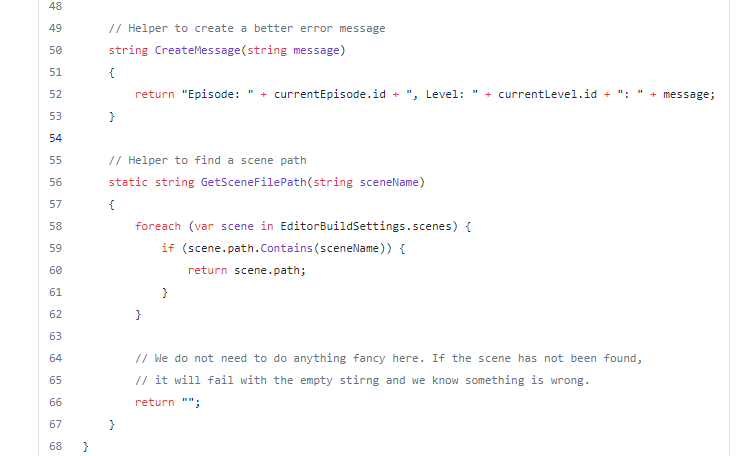
**Level Consistency**

We have a Scene and a Scriptable Object with additional metadata (used for example in Menu). It is easy to make a mistake there. However, we can check whether the metadata corresponds to the actual Scene.

One thing we have in our are levels are collectibles that player can collect. These are Game Objects in the level Scene. We have information about collectibles count in the Scriptable Object. So we can go through all the levels and check whether the number of collectible Game Objects corresponds to the count in the Scriptable Object.







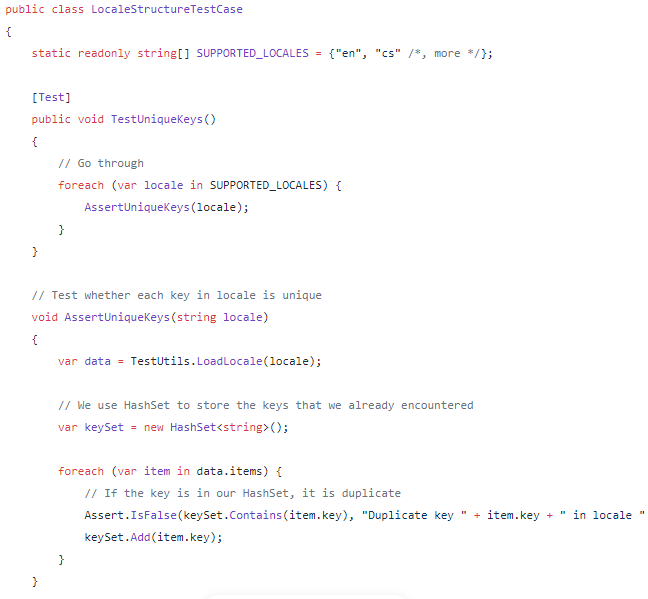
Note that if you load a Scene in Edit Mode Test, you need to use **EditorSceneManager.OpenScene(...)** which takes path instead of Scene name. You can find paths to Scenes in **EditorBuildSettings.scenes** array.

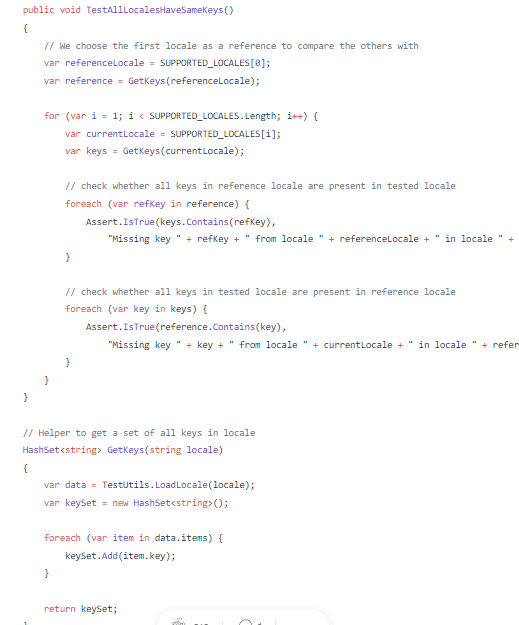
Of course, there are more things related to level consistency we test, for example:

* Are there all the expected Game Objects (e.g., the level end Game Object)?
* Do we have a correct Scriptable Object with metadata attached to the Scene?
* Do we have colliders on all blocks?

**Test Locale Files**

We have a JSON file for each language we want to support. It contains a key and a value for each phrase that should be translated. We can use tests to ensure that each key is there only once and also that all localization files contain the same keys.





**Play Mode Tests**

Play Mode Tests are slower because the runtime has to be started, scenes need to be loaded etc. They are also harder to write because you need to find out a way how to control your game or how to check expected results from the code.

We have some helpers to do the common tasks that we need during our Play Mode Tests.

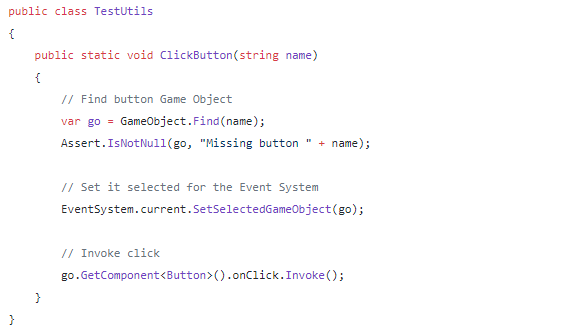
**Checking for Existing Components**

The typical scenario in our tests is that we do something and then we want to check whether a set of components is present and active in the Scene (e.g., some part of user interface appeared).



**Clicking a Button**

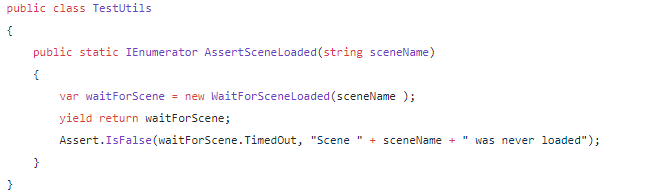
We have buttons in our user interface. If we want to simulate the player’s flow in the game, we need to click buttons from the tests somehow. We have a simple helper for that:



**Waiting for a Scene to Load**

In some tests, we change Scenes, and we need to wait until the Scene is loaded before we can continue with the test. The quick and easy solution is to use WaitForSecondsRealtime with a random number and hope for the best. If it fails, you can increase the number. However, this is nowhere near optimal. If it takes longer than expected for some reason, the test will fail even though there was nothing wrong. On the other hand, if the number is too big, the tests will run longer than necessary.

It waits till the Scene is loaded or the timeout is reached. Then we use it with a little helper method to check whether the Scene was actually loaded.



**Prepare the Environment Before the Test**

Use SetUp methods to prepare the test environment. You can use TearDown to clean up after the test. However, it is better to make sure everything is as expected before you run the test because otherwise, you rely on other tests to clean up after themselves. And if you forget to clean up somewhere, it might break a different test. It is then hard to debug the issue.

**Singletons**

Even though we all know that Singletons are usually considered a bad practice, we might still have some in our code. The problem is that they keep their state between tests. Therefore, it is essential to make sure they are set up as expected before we run the test.

**Time**

If you change **Time.timeScale** during one test, well, it is kept changed for all the tests that run afterwards. So make sure it is set up correctly. Otherwise, you might face tricky bugs in your tests.

We had a bug in our tests where we tested game pause. The problem was that during the test, Time.timeScale was changed to 0. After that test, we have a few more that were not affected by that change. And then we had one that got broken by that. If you run it separately, it worked. If you run all tests it failed, which was quite complicated to debug.

**Don’t Go Crazy with Tests**

It is important to realize that writing tests take time not only to write them initially but also to maintain them. When you change something in the game, you need to change the corresponding tests. It is better to have a few simple tests early when you are still changing a lot of things in the game. Once you are finishing the features, you can write more thorough tests.

Some tests might be too complicated, so writing and maintaining them would cause you to waste more time than they would save you. So always consider what and how much detailed you need to test.

**Running the Tests**

If you spend time to create automated tests, you need to run them. Otherwise, they are not useful. The best you can do is to run the tests every time you make any change (i.e., before committing the changes).

It is possible to run tests in Unity Cloud Build. You can event set to fail the build if tests don’t pass (we do that). We’ve connected Unity Cloud Build with Bitbucket to create a build and run tests for each pull request.