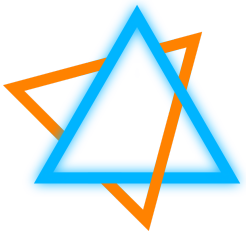
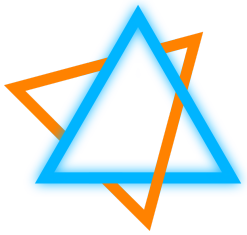


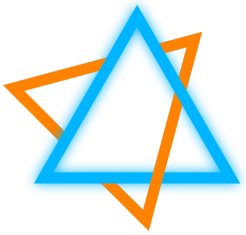
HudDimension
www.huddimension.co.uk

BDD Extension Framework Documentation

Welcome to ***BDD Extension Framework for Unity Test Tools Documentation*.**

In these sections, you can learn all about the uses and the features of **BDD Extension Framework**.

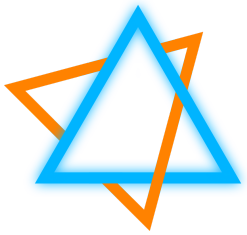




Support needed?

Do you need help?

Please, send an email: info@huddimension.co.uk, you will get an answer you as soon as possible.

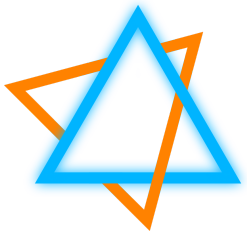


What is BDD?

BDD stands for Behavior Driven Development.

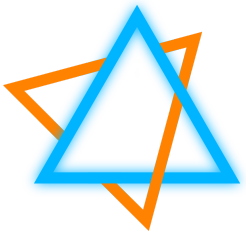
Like TDD (*Test Driven Development*) it is a software development approach that allows you to start your coding session from the tests.

Unlike TDD, the Behaviour Driven Development lets you think regarding how the user expects to interact with your software. This strategy can help you to design the software accurately, without using artefacts that could make your software slower and uneasy to maintain, avoiding the over design.



Quick start

If you want to start just now and try BDD Extension Framework, have a look at this section. Otherwise, if you want to learn all the potentials of Unity3D BDD Extension, jump to the next one.



Install the right assets

Before starting to use BDD Extension Framework, just be sure you have already setup all the instruments you need:

1. First of all, be sure to have not any build errors in your scripts. If so, fix them before continuing.
2. Do you have already the asset “Unity Test Tools” in your project? Just jump to the step 4, otherwise, download and install it from Unity Assets Store.
3. If you did not have the asset “Unity Test Tools” in your project, make sure to close and start Unity3D again after the installation before continuing.
4. If you want to verify if the “Unity Test Tools” are correctly installed, search for a new menu item on Unity main menu called “Unity Test Tools”. Usually, it is close to “Window” menu. If it did not appear, probably there is some error during the build of your project. If the errors are related to the test examples of the Unity Test Tools class, just delete those tests.
5. Now download from the “BDD Extension Framework” from Unity Asset Store.

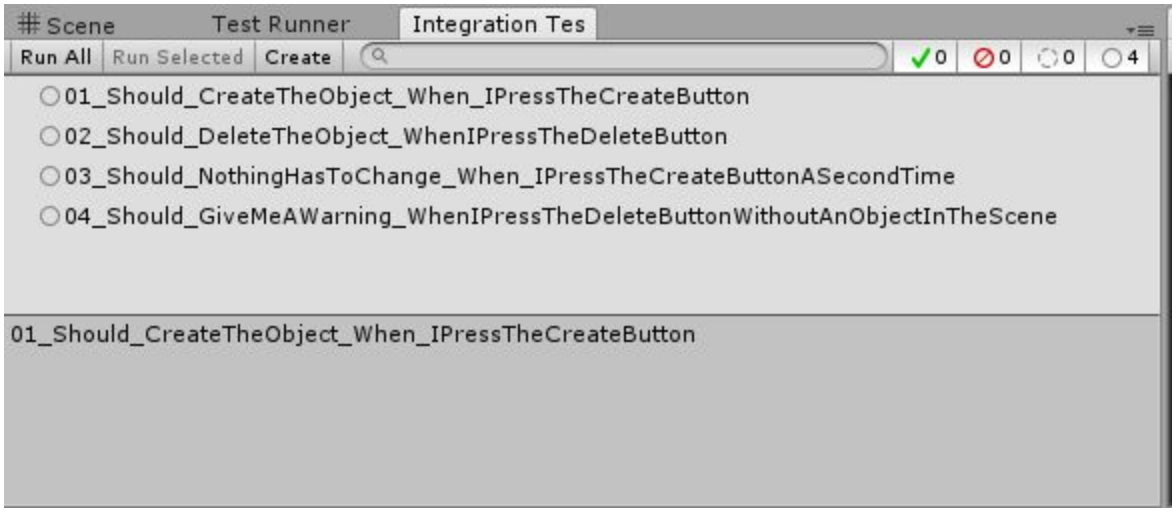
The setup should be almost complete now.

If you have any trouble completing the previous tasks, please email us to info@huddimension.co.uk and write down in detail your issues. Make sure to indicate the version of Unity3D you are using, the version of the “Unity Test Tools” and the version of the “BDD Extension Framework”. Indicate also the details of the building errors, if you have any.

Configure the layout

To use BDD Extension Framework, you need to open the **Integration Test Runner window** for the Unity Test Tools: you can open it by choosing the submenu "Integration Test Runner" under "Unity Test Tools" menu. If nothing happens, it could be you have the window already open: please, search for a window called "*Integration Tes*", it could be docked as tab everywhere in your Unity layout.

If you have not any integration test already configured, you could open the scene "**Creation of a GameObject Dynamic Scenarios.unity**". You can find it in the folder **"Assets/BDDExtensionForUnityTestTools/Examples/Scenes**" under your project.



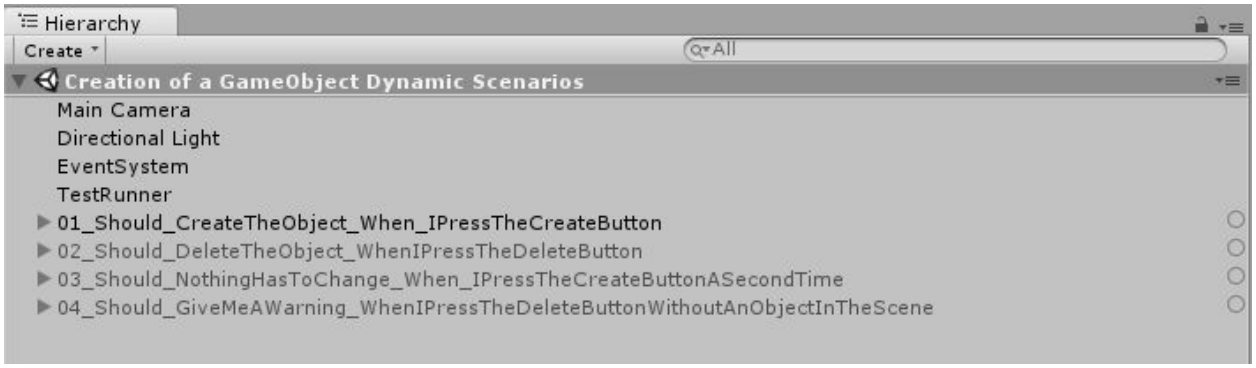
In the Integration Test Runner window, you can see the integration tests already configured for your scene. The buttons on the top of the window let you, in order, to "**Run All**" your integration tests, just "**Run Selected**" tests or "**Create**" a new integration test.

If you have many integration tests and you cannot locate the one you need, you can just write a word in the search field in the middle of the top window row.

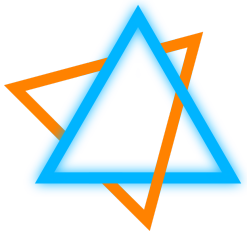
If you want to filter the tests regarding their result, you can check or uncheck the green button for having or not having in the list the successful tests.

If you want to filter the failed tests, you can check or uncheck the red button.

Every test you have in the list of the integration test runner window is also present in the **Hierarchy window** as GameObject.



To manage an integration test, you have to work on the corresponding GameObject.



Unity Test Tools: base information

To use BDD Extension Framework you have first know some base information about the **Unity Test Tools**. In this section, we are going to explain just what you need to know about them for using BDD Extension Framework.

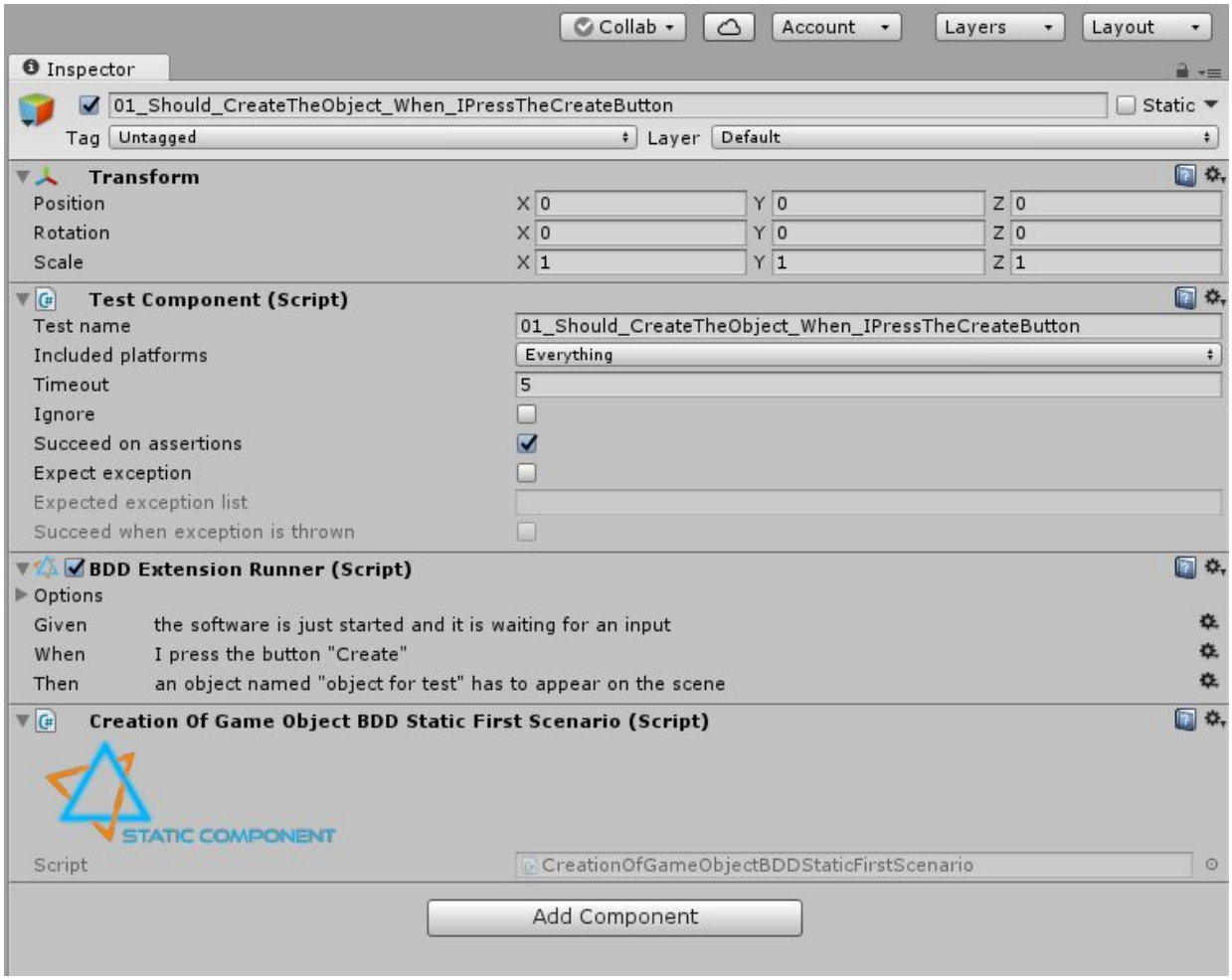
To make the explanation easier, please open the scene "**Creation of a GameObject Static Scenarios.unity**". You can find it under "**Assets/BDDExtensionForUnityTestTools/Examples/Scenes**".

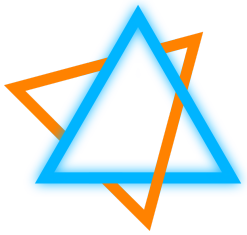
In the scene, there are two buttons: "**Create**" and "**Delete**". We are not going now into what the scene has to do: what you need to know now is that the integration tests configured in this scene are made to test the various interactions among the objects in the scene.

If you have a look at the *Integration Test Runner window*, you can notice that there are four integration tests already configured. If you look into the *Hierarchy window*, you can see four GameObjects named as the same. Managing each of these objects you are going to take control to the corresponding integration test.

The inspector

Now you can select the GameObject named "**01_Should_CreateTheObject_When_IPressTheCreateButton**" and look at its *inspector*.





You can see the following components:

- **Test Component**
- **BDD Extension Runner**
- **Creation Of Game Object BDD Static First Scenario**

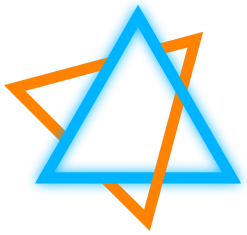
The first component, called "**Test Component**", is the *core* of the *Unity Test Tools Integration Tests*. It is automatically added to the GameObject when you create it by pressing the button "Create" at the top of the Integration Test Runner window.

The second component, called "**BDD Extension Runner**" is the *core* if the *BDD Extension Framework*. If you want to use the features of the BDD Extension Framework, the first thing you have to do after creating the integration test is to add this component to it.

The third component, called "**Creation Of Game Object BDD Static First Scenario**", is an example of what usually you are going to create and develop during your BDD coding session. Don't be afraid: you have not to develop for inserting the image, it automatically appears when you set up the component.

What you need to know about the Test Component are the following properties:

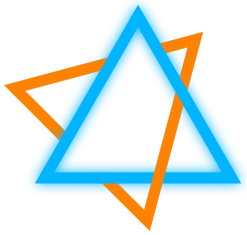
- **Test Name**: It is the name of the Integration test; any change of the name here is going to affect the name of the GameObject and the name of the integration test itself.
- **Included platforms**: you could leave it with the value "Everything", but if you want to test a specific environment, just select the platform you want to test.



- **Timeout**: It is the number of seconds that the component waits until giving an error of timeout if the test does not return a result. You are going to check and change this value more times during your coding session because if the value is too small, the test could not have time enough to finish its execution. However, if the value is too big and if your test is going into a loop, you could have to wait much time before getting the control of Unity again.
- **Ignore**: you can check this flag if you are going to hit the "Run All" button but you want to run just the other tests but this one.
- **Succeed on assertions**: It does not matter now to know how to use this flag. You have just to know that the BDD Extension Framework manages this flag automatically, so you can just leave it checked.

We are not using the other properties so that we can ignore them for now.

Regarding the other two components, we are going to explain soon how to use them; for now, just keep in mind they are components of the BDD Extension Framework.



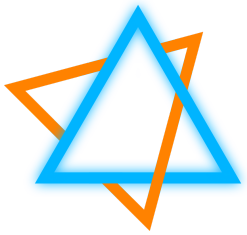
The scene

One of the most important guidelines for using the Unity Tools Integration Tests is about the scenes: you have not to put integration tests inside the scene that you are going to publish; you have to, instead, **have a separate test scene** where you are going to code your integration tests.

For this reason, you are going to use prefabs to assure that your test scene has the same objects of the original one.

Creating at least one Integration Test by pressing the button "Create" in the Integration Test Runner window, you are going to have one additional object in the scene, called "**TestRunner**". It is added by the Unity Test Tools to configure the scene properly for the execution of the integration tests: just ignore it, you are not going to use it directly.

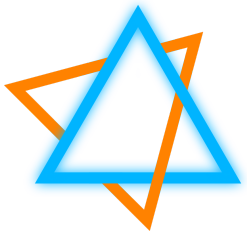
One of the most important concepts about the scene in the integration tests is the ***scope*** of the objects inside the scene: In fact, if you have more than one integration test in the scene, and you execute more than one integration test at a time, **you could experiment some unexpected behaviour**.



We can have **two scopes**:

- **Internal**: every object inside the hierarchy of an integration test GameObject is internal to it. Every modification of the objects inside its hierarchy is made inside a sandbox. When the execution of that particular integration test finishes, every change is reset to the original state.
- **Global**: every object outside the hierarchy of an integration test GameObject has a Global scope. Every change of the objects with this scope during the execution of an integration test still remains during the execution of the other integration tests, and it is reset to the original state when the last integration test execution finishes.

It is important avoiding to put objects outside the integration tests that might change their status during the execution of any integration test because it can cause unexpected behaviours of the other integration test executions. You can usually put fixed objects in the scene, like walls or the terrain. If you put an object outside the integration test that could, for example, change position, you have to be careful.



How to run an Integration Test

If you want to run a specific integration test you can:

- Select it in the **Integration Test Runner window** and press the button "**Run Selected**"
- Select it in the **Integration Test Runner window**, press the right mouse button and select "**Run**" from the **context menu**.
- Select it in the **Hierarchy window**, press the right mouse button and select "**Run**" from the **context menu**.

You can run more than one integration test of your choice at a time, just select them together holding the **CTRL button** on the keyboard.

If you want to run all the integration test in the scene you can:

- Press the button "**Run All**" in the **Integration Test Runner window**.

BDD Extension Framework

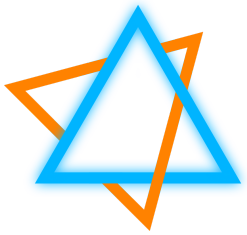
We are going to have an overview of the **BDD Extension Framework**.

With the BDD Extension, you are going to develop your test using the ***Given-When-Then scenarios***. If you are not used to the concepts of the Given-When-Then scenarios, **you can read here a short explanation**.

To create a new BDD Test, just choose from the menu: "Unity Test Tools" -> "BDD Extension Framework" -> "Create BDD Test": the test is going to appear in your Hierarchy window with the name "New Test".

Every part of the BDD Scenario is identified by one or more methods inside a particular class: this class is called **BDD Component**. Every BDD Component can have many methods: the methods that are going to be executed for each part of the scenario, and the order of their execution, can be decided **programmatically by code** or **dynamically via inspector**. When a method is selected to be executed in a scenario is called **Step Method**.

If a Unity script is a BDD Component, on its inspector is present a particular logo, indicating the type of the BDD Component.

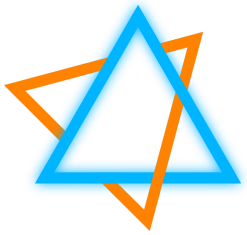


The BDD Methods and the Step Methods

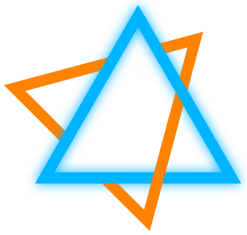
A method inside a BDD Component can be an ordinary C# method or a BDD Method: the BDD Methods are those that have declared an **attribute Given/When/Then** or, at least a **GenericBDDMethod** attribute. If a method does not have these attributes is a **generic C# method**. It can be invoked by a BDD method, but it cannot be managed directly by the BDD Extension Framework.

The methods that have declared a Given/When/Then attribute can be used as a main part of the Given/When/Then scenario. Inside the attribute, there is **a mandatory string parameter** that has to contain the **descriptive sentence** of that part of the BDD Scenario. When a Given/When/Then method is chosen for being a part of the scenario, it is called **Step Method**.

The methods that have declared a GenericBDDMethod attribute cannot be used as Step Method. However, they can become an "invisible" part of a scenario if they are set to be invoked before a Step Method, usually as a prerequisite, by the declaration of another attribute on the Step Method: the **CallBefore** attribute.



There are **two ways** to choose which BDD Methods have to take part of the scenario: whichever is the way, they have to be contained in a **Static BDD Component** or a **Dynamic BDD Component**.

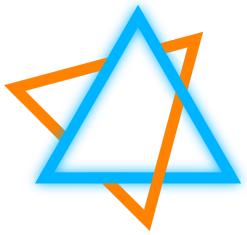


BDD Components: Static or Dynamic

There are two types of BDD Components: **Static** and **Dynamic**.

A **Static Componen**t is a C# class derived from the class called **StaticBDDComponent**. There can be **only one Static Component** for each integration test, and it defines and describes the whole BDD Scenario. Inside a Static Component, there are all the methods that compose the BDD test Scenario. The choice of the Step Methods and the order of their execution are set programmatically by the declaration of the Given/When/Then attribute: if one or more methods, for example, declare a Given attribute, they are going to build the Given method **chain of execution**. The order of the execution is set by the first parameter of the attribute declaration: lower is the value of the numeric parameter, earlier is the execution of the method.

A **Dynamic Component** is a C# class derived from the class called **DynamicBDDComponent**. There can be **one or more Dynamic Components** for each integration test. The BDD Scenario can be composed by **picking out** the Given/When/Then methods **from every Dynamic Component**. The choice of the Step Methods and the order of their execution are set using the inspector of the component called "**BDD Extension Runner**".



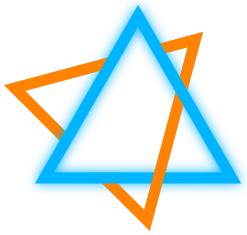
Static BDD Methods signature

Every Static BDD Method has to have the following signature:

```
public IAssertionResult MethodName()
```

The return value has to be a standard implementation of the interface IAssertionResult: It can be "**AssertionResultSuccessful**", "**AssertionResultFailed**" or "**AssertionResultRetry**". You have to return one of these objects if, in order, the test is successful, the test is failed, the test needs to be executed again.

In the signature of a Static BDD Method, **the parameters are not allowed**.



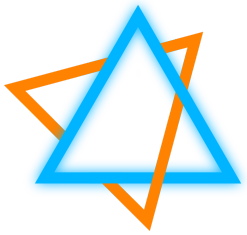
Dynamic BDD Methods signature

Every **Dynamic BDD Method** has to have the following signature:

```
public IAssertionResult MethodName(Type1 parameter1, Type2 parameter2, ...)
```

Like the Static BDD Methods, they have to return a standard implementation of the interface IAssertionResult.

They **can also have parameters**: you can read about the allowed types of the parameters and how to use them properly here.

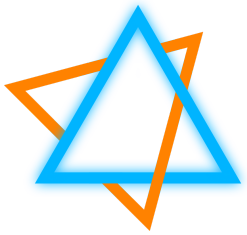


Error checking

Every **BDD Component** has an **inbuilt error checking**. After each build in Unity3D, or after a BDD Component is attached to the integration test, the component runs the checks for errors. If an error is detected, **it appears inside the inspector** of the component.

The **BDD Extension Runner** has an **inbuilt error checking** too, and it runs after every built in Unity3D or every time a BDD Component is attached or removed. If an error is detected, **it appears inside the inspector** of the BDD Extension Runner.

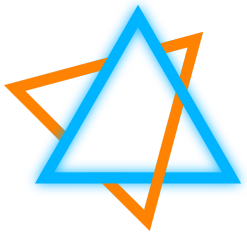
For a detailed list of the errors and their possible fixes, read here.



The BDD Extension Runner options

At the top of the BDD Extension Runner inspector you can have access to two options:

- **Rebuilt settings**: sometimes, during the coding session and the refactoring, the settings of the BDD test could be not aligned anymore with the actual settings of the BDD Components. If it happens, the BDD Extension Runner shows the associated errors. For some error, it could be necessary to overwrite the previous settings with the new ones. Rebuilding the settings can make lost the values of the parameters previously inserted.
- **Run under Fixed Update**: by default, every Step Method is executed during a **Unity Update cycle**. If you need to run them during the **Fixed Update cycle**, check the option.



The BDD Extension Framework in details: What are we going to learn

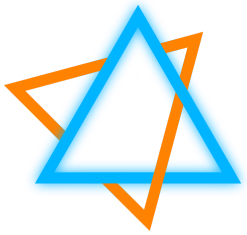
Here there is a list of **questions and answers** that will explain the most common doubts about the automated tests, the Behaviour Driven Development, the use of the BDD Framework in Unity and the best practices for testing the software using this framework. ***This guide covers just the aspects of the BDD relating the coding***.

Q: What are the differences with the old manual testing approach?

The manual test method forces the developer and the tester to interact with the software using their hands and their eyes to verify the software behaviour. This type of test takes an enormous amount of time and, unless there is a written list of tests to execute, covers only a little part of the tests the software should need. For assuring the complete absence of errors, ideally, all the tests have to be done for each change of the code.

Using a framework that lets you **automate the testing process** help you to run the tests in a faster way and, most important, there is not the temptation to skip some planned test for lack of time.

One of the most important differences is the possibility to code your software using the Behaviour Driven Development approach.



The **Unity Test Tools** give you the right instruments to automate your tests.

The **BDD Extension Framework for Unity Test Tools** gives you to the right tool to develop using the **Behaviour Driven Development** approach and to formalise your tests in a **more readable way**.

Q: This is not Unity Test Tools. What is the difference?

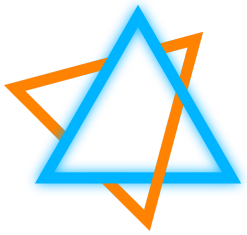
The **BDD Extension Framework** is an **Extension** of the standard **Unity Test Tools**. It uses them and gives to you an **additional way** to develop and run your tests. The presence of the Unity Test Tools Asset in your

project beside the BDD Extension asset is essential.

This extension allows you to use the **Given-When-Then scenarios**, particularly used in the **Agile Development Processes**. It lets you write down your automated integration tests in a more readable form. They are particularly useful when you have to **understand the tests coverage** or to understand what the test is going to verify, especially when it fails. You could already meet the Given-When-Then scenarios if you have used software like *SpecFlow, Cucumber or Selenium*.

Q: Can I continue to use the Unity Test Tools?

This framework extends the Unity Test Tools Integration Tests. That means you will continue to use the Unity Test Tools but in a newer and simpler way. You will still able to develop your integration tests in the standard way.

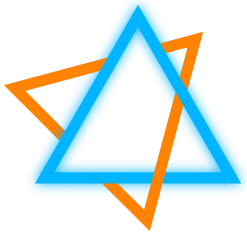


Q: I do not follow any Agile Framework, I do not use User Stories, and I barely understand what they are. Is that a problem?

If you don't work in an agile environment, you could not be used to the concept of the **User Stories**. Using BDD is not directly related to the User Stories, even if it is **more natural** to develop using BDD if you work under a user story. In this guideline, we try to not bind the concepts to any Agile Framework and User Stories. Instead, if you work in an agile environment and you are using the User Stories all the time you will be able to make the right equivalences between the "requirements" of the software and the user stories.

Q: I read about Acceptance Tests, Integration Tests, Regression Tests but I don't know what they are. Could you explain them?

- **Acceptance Tests** are written to assure to meet the software requirements. For example, if in your software an object has to move to a specific position after touching another object, you code your test to verify this behaviour.
- **Integration Tests** are written to assure that your different modules in development for your software are running correctly together.
- **Regression Tests** are written to assure that all the changes made on the software are still compatible with the other parts of the software or, in general, to verify the software behaviour after the changes.



One of the most frequent questions from the developers about these types of tests is: How can I recognise if a test is an Acceptance Test, an Integration Test or a Regression test reading the code?

The answer is: Usually you can't. The difference is regarding the scope of the test. You can sense if the test is one of them reading its description, but after its development, it has the same form of the other types of test.

There are several ways to code them. BDD Extension Framework helps you to write and maintain all these kinds of tests using the Given-When-Then scenarios.

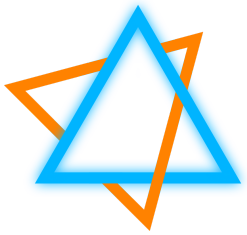
Q: I don’t know anything about Given-When-Then scenarios. Can you explain me those in detail?

Given-When-Then scenario is a method of proven efficiency for describing the software behaviour, given a previous state and an action. It is based on three steps:

- **Given**: The description of the software state at the beginning of the test.
- **When**: The description of the action that has to occur on the software.
- **Then**: The description of the expected behaviour of the software.

Describing a scenario using this format has the following benefits:

- It **helps the user**, the **analyst**, the **stakeholder**, the **product manager** to describe the tests in a more efficient and clear way.
- It **helps the developer** to understand easily the scope of the test you are going to code.



- It contributes to **maintaining the readability** of the test: after its development, you can understand its scope easily.

An example of a Given-When-Test scenario is:

- *Given I am on the first page of the control panel of my software*
- *When I click on the "Back" button*
- *Then the software has to close the control panel.*

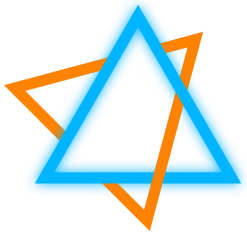
As you can see, the first sentence "*Given I am on the first page of the control panel of my software*" gives you the **state** of the software **before** the action you have to test.

The second sentence "*When I click on the "Back" button.*" is the description of the action you have to perform to obtain a result to test.

The third sentence "*Then the software has to close the control panel.*" is the **expected behaviour**.

One of the most frequent questions is: This test seems to be incomplete because it does not say what has to appear after the software closes the control panel. The answer to this question is pretty simple: there will be other scenarios that will describe what the software has to show after closing the control panel. The behaviour could change depending on the initial state. In our example, the description of the initial state of the software does not mention anything about what the software showed before opening the control panel. It means that every time I am on the control panel, and I click on the back button, the software has to close the control panel, with no exceptions.

One of the most frequent problems for the developers for understanding the Given-When-Then scenarios is to understand the association from the description of a

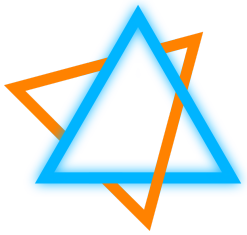


scenario and the actual code that tests it. The explanation of this concern is: some frameworks associate the Given-When-Then sentences to the code you have to write. The BDD Extension is one of them, specific for Unity.

Q: Are there some guidelines for developing using BDD?

You can read the section BDD Extension Framework Guidelines. However, there are some rules (among the others) you have to know right now to understand correctly what are you going to learn:

- **Test First - Code After principle**: The Behaviour Driven Development is based on this principle. It means that you are going to write and code the test before you develop the code you need to test.
- **Test only a single behaviour at a time**: Each test should verify just one behaviour. If you test more than one behaviour at a time, you could bind these behaviours to each other.
- **Develop just what is strictly necessary to pass the test**: Developing some extra code could indicate an over design of your software.
- **Do not insert extraneous code during your coding**: If you put just a line in your code that is not specifically requested by the test scenario, that line could stay not tested until you release your software.



Q: I am already using TDD. Can I continue to use it?

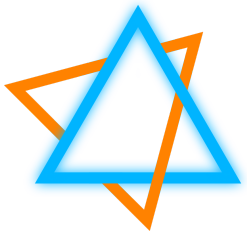
TDD stands for *Test Driven Development*, and it is based on the **unit tests**. BDD stands for *Behaviour Driven Development*, and it is based on the **Unity Integration Tests**. They are not exclusive. Actually, **they are complementary**. The Behavior Driven Development approach allows you to design your software from a **higher point of view**. Using it, you will design your classes closer to their natural use. However, when you are developing your classes in a higher detail, the TDD process will help you to focus your effort only on what you really need to develop.

Q: Am I forced to do TDD also?

You are free to develop your software in the ways you are more comfortable. However, if you are thinking to use BDD, it could mean that you are looking for a more efficient, organised and clear way to develop your software so that TDD could be the natural addition to your development process.

Q: Are the examples developed using TDD?

For simplicity, the examples are not developed using TDD. It is intentional: we did not want to overload the explanation with other concepts that might confuse you.



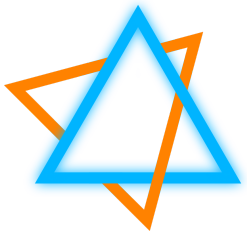
Q: I want to increase the test coverage of my software. Could BDD help me?

Of course! BDD and TDD follow the principle “Test it first”. It means that before you start to write down your code, you write the tests that verify the code you are going to write. If you follow this approach strictly, **it lets you have only code already tested**.

Q: The tests written using BDD are enough to consider my software completely tested?

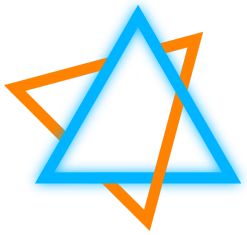
The answer is: *usually they are not*. It depends on how much detailed they are. During a BDD session normally, you develop the tests that **verify the user requirements and the acceptance criteria**. Sometimes they could also involve the coding of the **Integration Tests**. Usually, they **do not include the Regression Tests**. You should code them at a later stage.

If we have to talk about a complete test coverage, we have to consider a lot of other aspects, like the behaviour of your software on particular devices or circumstances. In large enterprises, there is usually a separate team called **Quality Assurance Team (QA)**: the members of this team have the right expertise to analyse and prepare the automated tests for cover those other aspects of the software testing.



Q: I have already a software. Can I start to use the BDD Extension Framework on it?

You can use the BDD Extension in more than one way. It is a bit late to develop in BDD for the code you have already written unless you are going to rewrite it. However, you can use BDD for writing the new code for your software, adding, for example, new features. However, other than that you are still able to write the Acceptance Tests, Integration Tests and Regression Tests also on the code you have already written.



Getting started

Are you ready to learn how to use BDD Extension Framework? Let's start with a simple example.

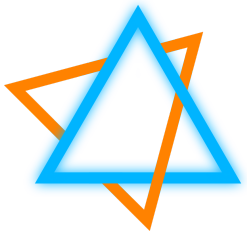
What tools do you need:

For having a look at the demo scenes you need **Unity**, of course, the standard **Unity Test Tools** and the **BDD Extension Framework** imported as assets in your project. After that, you need **your favourite C# editor** for viewing the code. For the setup of your project, please have a look at the Quick start section.

What do you have to know before:

The **supported language is C#**. The test classes are C# classes derived from **MonoBehaviour**. There is some use of the **C# attributes** for the classes, the methods and the properties. These attributes are very easy to use, you can just learn how to and when to use them from this guide, even if you do not fully understand the basics of the attributes in C#.

...Moreover, obviously, *you have to know how to develop in Unity* to be able to take control of your program and write your tests.



Dynamic BDD Test: Creation of a GameObject

[*File*: **Creation Of A GameObject Dynamic Scenarios.unity**]

You can now open the scene "**Creation Of A GameObject Dynamic Scenarios.unity**". This simple scene has two buttons, called "Create" and "Delete". For now, we are going to understand how to use the integration tests for testing the software.

We are going to test four behaviours:

1. **When I press the button "Create" I expect to see a new object in the scene.**
2. **When I press the button "Delete" with the object in the scene, I expect to destroy the object.**
3. **When I press the button"Create" when there is already an object in the scene, I expect that nothing is going to change in the scene.**
4. **When I press the button "Delete" when there is not an object in the scene, I want to be warned by a message on the screen that there is not an object to destroy.**

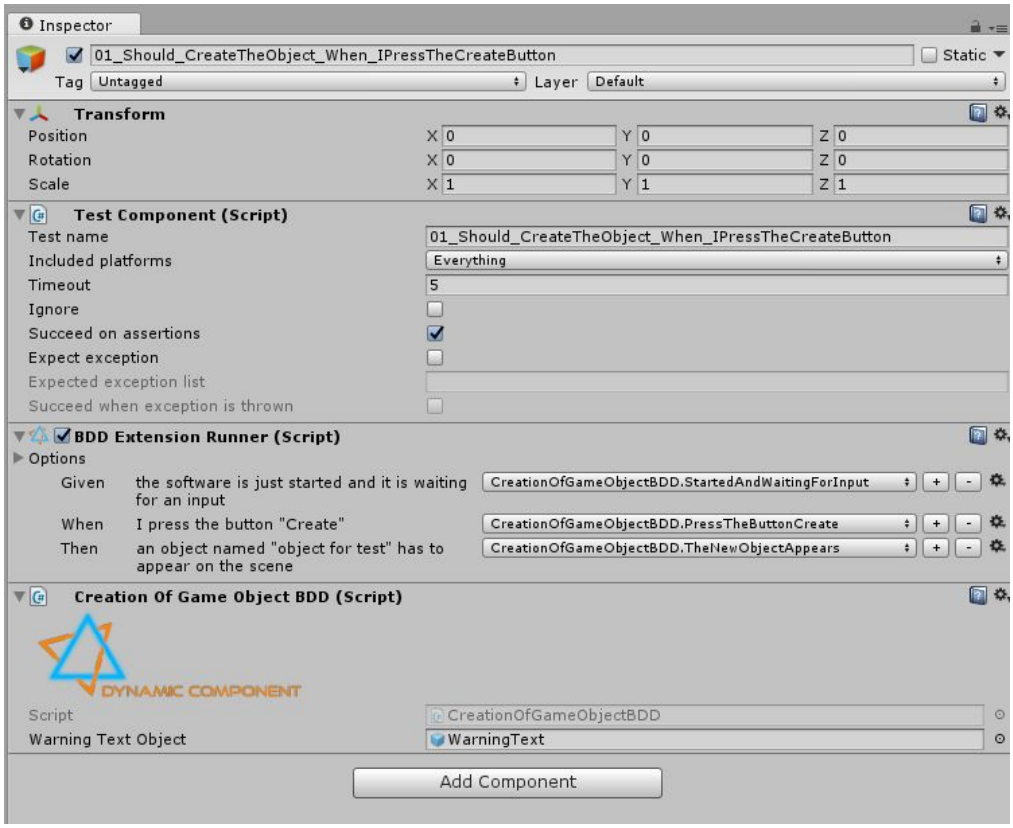
For each behaviour, there is an integration test in the Integration Test Panel.

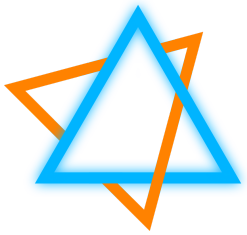
Each behaviour can be seen as an Acceptance Criteria, a requirement for the software.

Unity Integration Test 01

We are going to analyse our **first Given-When-Then scenario**. If you are not used to this type of test, please consider reading one of the previous paragraphs to be sure to understand.

When you click on the first test, you can see its structure in the inspector panel.





Like a standard Integration Unity Test, there is the component called “**Test Component**”. It is the very same component you are used to having in your Unity Integration Tests. You can just change the Timeout value and the “**Included platforms**”. Leave the other values unchanged.

The second component is called “**BDD Extension Runner**”. It is the **core** of the **BDD Extension Framework**. It allows you to configure the test properly. It runs the scenarios when you execute the test.

The third component is called “**Creation Of Game Object BDD**”: It is the class that **contains the methods** that describe the various scenarios. When you are going to develop your own tests for your software you will create a new class where you are going to put your test methods.

How can we **describe our test scenario**? Using the Given-When-Then form:

- **Given the software is just started and it is waiting for an input.**
- **When I press the button “Create”**
- **Then an object named “object for test” has to appear on the scene.**

This exact scenario is shown under the “BDD Extension Runner” inspector. Each line corresponds to a specific step of our scenario and a specific method inside the class “Creation Of Game Object BDD”. This scenario is going to implement the first requirement: “*When I press the button “Create” I expect to see a new object in the scene.*”

Let's analyse in details the inspector for the BDD Extension Runner:

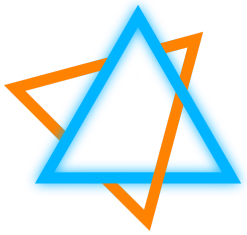
- Marked with the red numbers from 1 to 3 we have the standard labels Given, When, Then. They are a reference point for what we are going to configure.
- With the red numbers from 4 to 6, we have the description of the scenario. This description is loaded by the framework reading the Step Methods.
- With the red numbers from 7 to 9, we have the combo boxes where we can choose the method to use for each step of the test scenario. In this particular case, for example, for the "Given" step we are using the method called "StartedAndWaitingForInput" of the BDD Component CreationOfGameObjectBDD (marked with the number 7).
- With the red numbers 10 and 11, we have the buttons that let us to add or remove additional steps for the basic steps "Given, When, Then". The new line is going to be marked with a label "and", joining the colloquial writing of the preceding step.
- Marked with the red number 12 we have the comment for opening the IDE on the method selected in the corresponding combo box.

How does the class **CreationOfGameObjectBDD** looks like? We are going to explore it right now. You can open it easily using the "**Edit Script**" function (if your favourite IDE is already configured):

Otherwise, you can open it manually locating it under the "**example**" folder.

First of all, we would like to spend some words about the **naming conventions**.

Using a naming convention for class names and methods names is a good practice. In these examples, ***we are not using a particular naming convention***, just for keeping the names easy to understand without overloading the explanation with other concepts.



- **Class declaration**

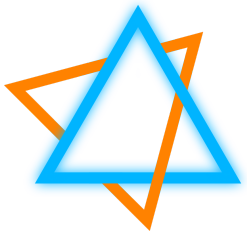
```
public class CreationOfGameObjectBDD : DynamicBDDComponent

{

[...]
```

Just have a look at the class declaration. It inherits the class **DynamicBDDComponent**. This particular base class **is a MonoBehaviour component**, so it let you add the CreationOfGameObjectBDD class as a component of the integration test.

Secondly, it makes the CreationOfGameObjectBDD class a **BDD Component, recognisable by the BDD Framework**.

Furthermore, it contains the definitions of other features that we are going to explore soon.

You can choose, of course, the name of the class you prefer: just think about to use a particular name convention, so you can easily understand what part of the software the class is going to test.

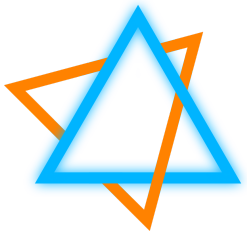


- **Methods declaration**

```
[Given ("the software is just started and waiting for an input")]
  public IAssertionResult StartedAndWaitingForInput()
  {
[...]
[When ("I press the button \"Create\"")]
  public IAssertionResult PressTheButtonCreate()
  {
[...]
 [Then ("an object named \"object for test\" has to appear on the scene")]
  public IAssertionResult TheNewObjectAppears()
  {
[...]
```

For each method declaration, we can notice **two important elements**: The **attribute** and the **return value** of the method. It is the **standard declaration** of a method you can use with the BDD Framework.

As you can see, each method has declared a different attribute, depending on its role in the scenario. The methods with the attribute Given can be used only in the Given step. Opening the **corresponding combo box for Given steps** on the Runner Component



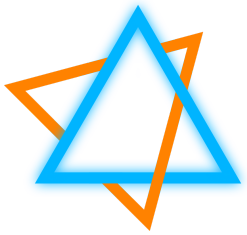
on the inspector, it lists only the methods marked with the Given attribute. Same for the When attribute and the Then attribute.

Inside the attribute declaration, there is **the description of the step in the scenario**. This text appears on the BDD Runner when the method is selected.

The return value of the method is an **interface** called **IAssertionResult**. The method has to **return an object** implementing this interface for telling the framework the result of the method execution.

There are three possible implementations you can use as the return value:

1. **AssertionResultSuccessful**(): When the execution of the method ends **without any error**, you have to return this object. You do not need to give to the framework other information: it knows that it can continue to run the next scenario step. It is the common return value of the Given and When steps and, if the test described by the scenario is passed it is also the return value of the Then step.
2. **AssertionResultFailed(string text)**: Returning this object, you are telling the framework that **something went wrong** during the execution of the step. It is conventionally used only in the “Then” steps, to communicate the failure of the test execution (if it is failed, of course). The string text passed to the constructor is the description of the error.
3. **AssertionResultRetry(string text)**: Sometimes you cannot be sure about the validity of the test execution until the software reaches some conditions. The execution of all the code in Unity not always is made between an “Update” call



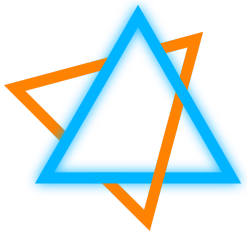
and the following one, so sometimes you have to perform the test only when you are sure that your **software has reached the right state**. Inside your methods, you can put the code to verify if your software has reached the right state. If you have to wait some time, you can return the object AssertionResultRetry to tell the framework to **try again** the execution of the method. After a **default timeout of 3 seconds** (but shortly we are going to see how to personalise this timeout), the framework stops the iterations on this methods, **using the return value as a failure result**. The text passed to the constructor is the description of the error that is shown in a case of timeout.

Inside the methods, you can just write your code as you are used, dealing with the objects in the scene, checking the components and their public properties and so on. The only thing you have to **keep in mind is to return the right IAssertionResult object**, depending on the behaviour you are expecting from the framework.

Inside the class, you can put other methods in addition to the step methods marked with the “Given” ”When” “Then” attributes. These methods are not directly used or even recognised by the BDD Framework, and they are there just for your convenience.

Q: I was expecting to use Asserts in the code to have the result of my test. Why not?

We have considered implementing the framework in a more standard way. However, some test in the Unity World cannot be reduced to a single comparison between two



simple values. Furthermore, some features in the framework could not be used properly just managing Assert constructs, like, for example, the Retry feature.

Q: I can see other “Given” “When” “Then” methods in the class. What are they?

They are used by the other three integration tests. We are going to analyse them soon.

Q: Do I have to return always an object? Can the return value be null?

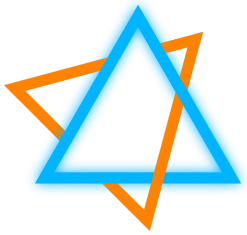
You have to **return always one of those three types**. If you return null, the framework interprets it as a coding mistake.

Q: How do you suggest to code the tests? Are there some guidelines?

Yes, sure, there is the section BDD Extension Framework Guidelines. There you can read how to organise your tests and how to code them more efficiently.

Q: Is the code on the CubeManager class written using BDD?

Yes, it is, but there is just a line in the code that is not developed using the BDD guidelines, we are going to analyse it shortly. Consider it like an Easter Egg.

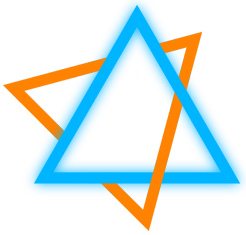


Q: Why do I have to return an AssertionResultRetry object? Can I just use a Thread.Sleep or just a cycle inside my step method?

Every step method is executed during a **Unity Update** or **Fixed Update cycle**. If you put some code inside your method that waits for something, your software is going **to be frozen inside that method**, stopping your software to run.

Q: Do I always have to compose the test choosing the methods in the inspector? Can I write a single class for each test and write inside it what I want to run?

No, you do not have to. **There is another way** to code your tests that do not imply the use of the inspector. You can read the chapter Static BDD Test: Creation of a GameObject for learning about it. However, we suggest finishing to read this chapter, so you can learn the basic information you need.



Unity Integration Test 02

In this test, we are going to learn how to **reuse the code to speed up** our development session.

We are going to implement the requirement: *“Pressing the “When I press the button “Delete” with the object in the scene, I expect to destroy the object.*”

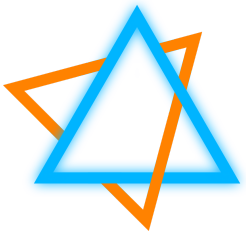
The scenario for this test is:

- **Given there is a cube in the scene called “object for test”**
- **When I press the button “Delete”**
- **Then the software has to destroy the object named “object for test”.**

We can see straightaway that the first step is actually what we have already done during the first test. The “Given” step could be implemented using the methods Given, When and Then from the first test scenario. The scenario could also be written in this way:

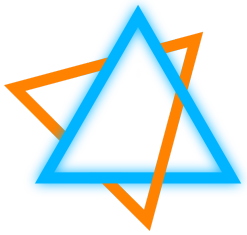
- **Given the software is just started and waiting for an input**
- **and I press the button “Create”**
- **and an object named “object for test” appears on the scene**
- **When I press the button “Delete”**
- **Then the software has to destroy the object named “object for test”.**

It could be implemented by **adding a new attribute “Given”** to the previous “When” and “Then” methods so that we can also have them in the “Given” combo box:



```
[When("I press the button \"Create\"")]

[Given("I press the button \"Create\"")]

  public  IAssertionResult PressTheButtonCreate()

   {

[...]

[Then("an object named \"object for test\" has to appear on the scene")]

[Given("an object named \"object for test\" has to appear on the scene")]


   public IAssertionResult TheNewObjectAppears()

   {

[...]
```

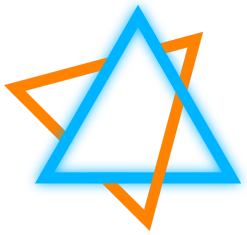
Using this feature, the methods that were once visible only in the When and Then combo boxes are now visible in the Given combo box so that we can reuse them. Unfortunately, **we are losing the legibility of the scenario**.



Fortunately, there is another **cleaner way** to do this: using the [**CallBefore**] attribute.

```
[Given("There is a cube in the scene called \"object for test\"")]
[CallBefore(1, "StartedAndWaitingForInput")]
[CallBefore(2, "PressTheButtonCreate")]
[CallBefore(3, "TheNewObjectAppears")]
public IAssertionResult ThereIsACubeInTheScene()
{
    return new AssertionResultSuccessful();
}
```

The [**CallBefore**] attribute tells the framework to **execute the methods indicated** by the string parameter **before** executing the method ThereIsACubeInTheScene(), in the order indicated by the first numeric parameter. In this case, we have just made a step method using three existing step methods. The main method itself has nothing to do because the other methods make all the job. Every method in the [CallBefore] stack returns its IAssertionResult object: the **result of the call chain** is **the outcome of every single execution** so that if there were a failure in the call chain, it would be correctly interpreted by the BDD Framework. For this reason, the main method "ThereIsACubeInTheScene()" can just return a successful state. Coding the test in this



way or in the first way is exactly the same for the framework, but we are saving the scenario legibility.

Q: Do I have to put only methods from the same scenario step in the CallBefore attribute?

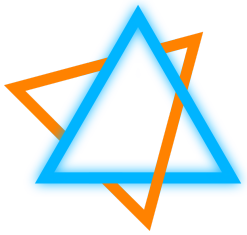
No, you do not have to. In the CallBefore attribute, you can choose every step method you have in your class.

Q: Can I use ordinary methods in the CallBefore attribute?

No, you cannot. If you want to call conventional C# methods, just do it in the body of the method you are coding.

Q: Can I use step methods from another BDD Component in the CallBefore attribute?

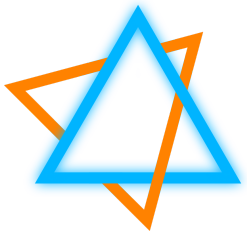
No, you cannot. If you take the decision to develop another BDD Component, mainly it is because you want to keep the code clear, but you can, of course, use the inheritance to bring those step methods in your class. Keep in mind that using the inheritance you are going to see all the step methods coded in the base class on the combo boxes: this behaviour could be wanted or not. Please, read the chapter “BDD Extension Framework Guidelines” for more information.



Q: I cannot see my “When” methods in the “Given” combo boxes, but I want to use it. How can I do it?

It is because every combo box filters only the right type of method. In the “Given” combo box you can only see the methods with a [Given] attribute.

Sometimes the same code you wrote for the “When” and “Then” methods for a previous scenario are useful for building the first steps of a new scenario. In this case, you can add another attribute, different from the original one, for indicating that you are going to use the method in another Step Type. Usually, the colloquial sentences in the scenario should be a little different so that you have to write the right sentence for the new scenario Step Type. Keep in mind that only one attribute of the same step type is allowed for each method.



Unity Integration Test 03

We are going to implement the requirement: "*When I press the button"Create" when there is already an object in the scene, I expect that nothing is going to change in the scene.*"

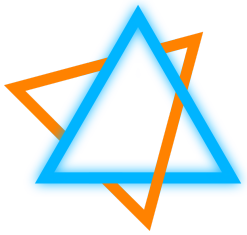
The scenario for this test is:

- **Given there is a cube in the scene called "object for test"**
- **When I press the button "Create"**
- **Then nothing is going to change in the scene.**

We can notice that the "Given" step could be the same of the last scenario so we could just reuse the same method. However, if we analyse the "Then" scenario, we are being asked to check that nothing in the scene is changed after the second hit of the button "Create". It means that, somehow, we have to **memorise** the state of the scene just before pressing the button "Create" the second time, when the scene is in a stable state. For this reason, there is **another "Given" method** called **ThereIsACubeInTheSceneAndStoreItForAFollowingUse()**: it has the same scenario text of the method ThereIsACubeInTheScene() but there is one more CallBefore attribute:

[CallBefore(4, "StoreTheListOfCubesInTheScene")]

This new method stores an array containing the cubes objects in the scene; Its declaration is different to the other Step Methods declarations:

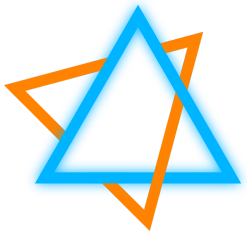


52

```
[GenericBDDMethod]
public IAssertionResult StoreTheListOfCubesInTheScene()
{
```

In this case, we are using the attribute [**GenericBDDMethod**] instead of *“Given” “When” “Then”* attributes. This particular form of Step Method declaration is useful when we want a method that could be usable as a [**CallBefore**] method, but it is not a proper Step Method with its own scenario text description.

The "When" step is also already used in the first scenario, used again as “When” step. We are reusing it like in the first step, but we want to bring to your attention an important point. We are already using it in a [CallBefore] attribute in the “Given” step. It means that the method **PressTheButtonCreate() is called twice** during the execution of the whole scenario. In this case, the method is well coded, so there is nothing to be afraid. However, sometimes you could be tempted to reuse a method that modifies the state of some property in the BDD Component, like the previous method StoreTheListOfCubesInTheScene(). In this case, you have to be very careful, because the second execution of the method could have an unexpected behaviour. It is better to **use always stateless step methods** unless that behaviour is intentional and the scope of the method is clearly to modify the state of some component property.



In the "Then" method declaration we can see **the new parameter "Delay"**:

```
 [Then("nothing is going to change in the scene.", Delay = 1000)]
   public IAssertionResult OnlyACubeInTheScene()
   {
[...]
```

The parameter "Delay=1000" is used to obtain a delay of 1000 milliseconds (1 sec) before the execution of the method. In this case, we are using the delay to be sure that the software has finished all the operations for the “probable” modification of the scene. According to your software behaviour, you can surely find a better way to develop this kind of algorithm. It was just a pretext for showing you the Delay parameter.

Unity Integration Test 04

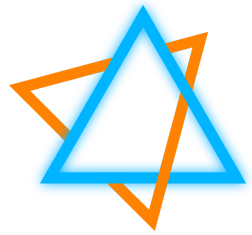
We are going to implement the requirement: "*When I press the button "Delete" when there is not an object in the scene, I want to be warned by a message on the screen that there is not an object to destroy.*"

The scenario for this test is:

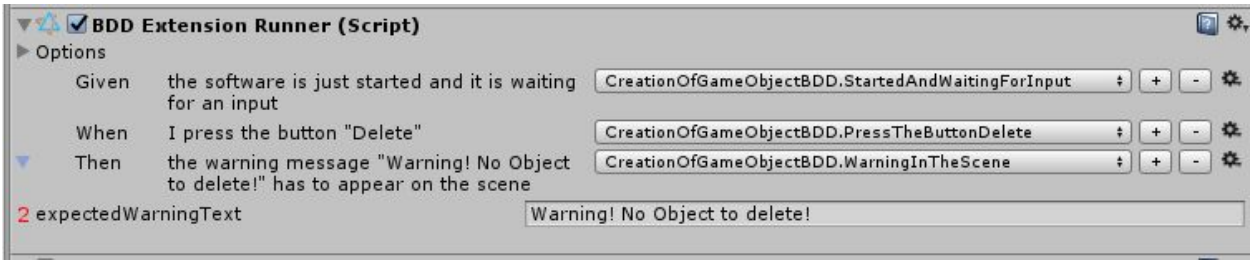
- **Given the software is just started and it is waiting for an input**
- **When I press the button "Delete"**
- **Then the warning message "Warning! No Object to delete!" has to appear on the scene.**

The first two steps have not something new to explain; they are using the same features we already found.

Watching the Inspector for this test, we can see something different, compared to the other tests:



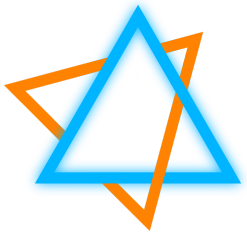
Looking at the image, we can notice that there is an indication that the **row is expandible** (marked with the red number):



Expanding the row, we can see another line: it is **a parameter for the selected method**. In this row, we can insert the value of the parameter. Let's analyse the code in details:

```
[Then("the warning message \"%expectedWarningText%\" has to appear in the scene", Delay = 1000f)]
public IAssertionResult WarningInTheScene(string expectedWarningText)
{
[...]
```

Here we can see that the method, in this case, has a string parameter called "expectedWarningText". This parameter is used inside the method body. The value of the parameter **is passed by the framework** using the value inserted on the inspector.



The framework stores that value inside a **particular structure inherited from the base class DynamicBDDComponent**.

In this example, the type of the parameter is a **System.string**. The BDD framework can store values for all the types that Unity3D can manage as serialised property in the inspector, **even if the property is custom**. However, the BDD framework can natively manage just the Unity3D predetermined base types (see the chapter "Simple field types that can be serialized" on the Unity3D Documentation).

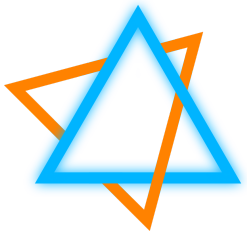
There is a way to add this feature for the others serialised types: for doing it, please read Methods Parameters Management.

Analysing in details the [Then] attribute you can see we are using the parameter name inside the step scenario description.

```
[Then("the warning message \"%expectedWarningText%\" appears in the scene", Delay = 1000f)]
```

With this syntax, you are telling the framework to **substitute** the code **%expectedWarningText%** with the **parameter value**. You can see its effect in the previous image. Please, compare the step scenario description showed in the image with the sentence declared in the [Then] attribute to see the result.

Another aspect you could find interesting about this method is the use of the BDD class public property called WarningMessage. It is, of course, a pointer to a GameObject in the scene, passed by the inspector, as usual in the Unity development. It was just a reminder that the BDD Component is a Unity Component attached to the Integration



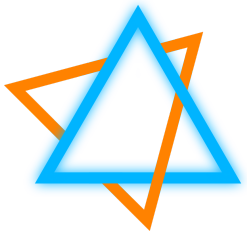
Test, so you can develop your tests keeping in mind you can still use all your experience in Unity development.

Q: I have made a custom inspector for my serialised property. Does the framework recognise that?

Yes, of course. The framework uses the Unity property fields to draw the parameters on the inspector, so every update you make on your custom inspector is going to be reflected instantly on the BDD Runner view.

Q: What if I want to use a parameter that cannot be managed by the inspector?

In this case, the better way is to create that object programmatically inside you test code. You can put as parameter the data you need to build your object.

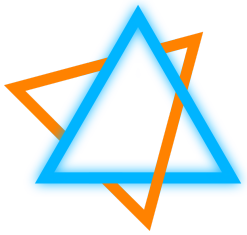


BDD in action: Having a look at CubeManager class

After analysing the integration tests for our little software, just have a look at the **result**.

Previously we said that the Behaviour Driven Development is used, like TDD, to develop only the code we need and to design our classes in a way closer to the final use.

Just for example, let's see how the method **OnCreate()** was developed between the several steps.



Test 01

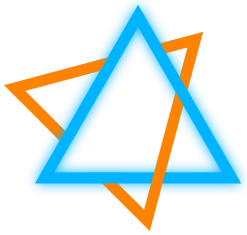
During the development of this test, we created the “Create” button, the cube prefab and the component CubeManager. Inside the component, we just added the public property CubePerefab and the method OnCreate():

```
public void OnCreate()
  {
        Transform cube = Instantiate(CubePerefab, transform);
        cube.name = "object for test";
  }
```

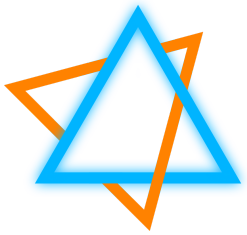
If you what to compare this version of the method with the final version, you can see that **something is missing**. It is because, during the development of the first test, the only requirement was that pressing the “Create” button “ an object named “object for test” has to appear on the scene”.

The corresponding Acceptance Criteria was “*When I press the button “Create” I expect to see a new object in the scene.*”

In this requirement, **there is no clue** about the need to avoid to have more than one object at a time. Yes, usually we already know all the acceptance criteria at once, and the third one asks us to limit the creation of the cube to one object at a time, but we have to keep in mind one of the objectives of this type of development: we have to code

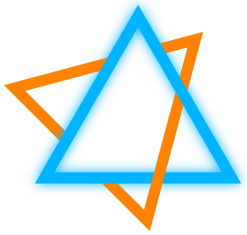


only what the requirement ask. One of the most insidious pitfalls for a developer is to lose the scope of what he is developing, indulging in developing a big infrastructure because it “could” be useful in the future.



Test 02

The method OnCreate() is unchanged. We created the method OnDelete(). Just because we focus our attention only on the OnCreate() method now, we are not describing the development process of the other part of the software.



Test 03

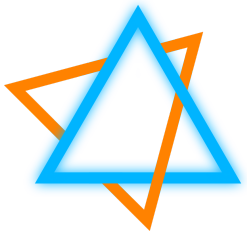
Here we are adding the code to avoid more than a cube at a time.

```
public void OnCreate()
    if (GameObject.FindWithTag(CUBE) == null)
    {
        Transform cube = Instantiate(this.CubePerefab, transform);
        cube.name = "object for test";
    }
}
```

As you can see, the code is a little different from the final version of this method. In fact, in the final version there is one more line:

WarningMessage.SetActive(false);

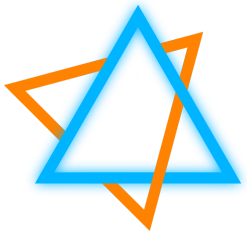
However, **there are not other Acceptance Criteria** that ask us to modify the behaviour of the “Create” button. **Why that line appears in the final version of the code?**



Test 04

During the development of this test we coded another part of the OnDelete() method and here we discover that we developed the **warning message behaviour**. During this development, the developer noticed that after the first warning, the sentence keeps staying visible on the scene, so he implemented a way to make it disappear. It could be a good idea at first glance but, he violated some principles of BDD:

- **There is not a requirement for that behaviour**: people who give the requirements and write the acceptance criteria have a big picture of the project. At first glance, it seems to be an omission but could also be a proper intent. Perhaps there is going to be a new requirement the week after that could ask to make disappear the warning message pressing a third button called “Clear”. In this case, the developer could not remember the little line he already added the week before (or it can be another developer, of course), and the development of the new requirement could be tricky because the developer has to know that he has to delete some previous code. If the developer is not sure if there is an omission, the right way is to ask for an explanation. If there is an omission, he receives the additional requirement.
- **The most important fact is**: that line of code is not tested. There is not a test to verify that behaviour. It is not enough that during the execution of the existing tests that line of code is executed. In some particular cases, you could find an unexpected behaviour of your software just because of that line.



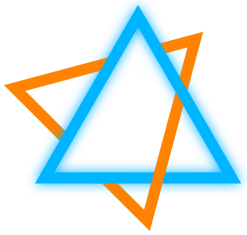
Static BDD Test: Creation of a GameObject

[*File*: **Creation Of A GameObject Static Scenarios.unity**]

The use of the inspector to choose the step methods for composing your scenario **is not the only way** to code your test scenario. You can also do it using the **Static Components**.

You can open the example scene "**Creation Of A GameObject Static Scenarios.unity**". In this scene, you can see **the same four tests** we have already discussed. If you have a look of the inspector of the first one you can see the very same scenario for the first one of the previous scene, but in this case, **you cannot choose the method steps using the inspector**. All the information are **coded in the BDD Component** called "**CreationOfGameObjectBDDStaticFirstScenario**".

There is just few information to add about this BDD Component compared to the previous one.



- **Class declaration**

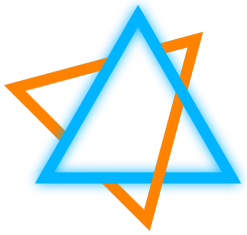
```
public class CreationOfGameObjectBDDStaticFirstScenario :
StaticBDDComponent
{
[...]
```

The first difference is the class that our component inherits: **StaticBDDComponent**. In this case, we are telling the framework that all the information about the building of the scenario is coded into the class. It means that the inspector does not show the combo boxes and the “add/remove” buttons: **you cannot modify the scenario using the inspector**.

- **Methods declaration**

```
[Given(1, "the software is just started and it is waiting for an input")]
  public IAssertionResult StartedAndWaitingForInput()
  {
[...]
```

The declaration of the methods is not so different from the previous one: we have just to **add a numeric parameter**. This parameter tells to the framework the **order** of the methods it has to run during the **execution** of the test scenario. This parameter has to **start from 1**, and the **numbering** has to be **sequential**.



- **Methods Parameters**

The methods defined in a Static Component **cannot have parameters**. This is because the static way for coding you test is meant to be used also if you are not sure to store the information in the inspector.

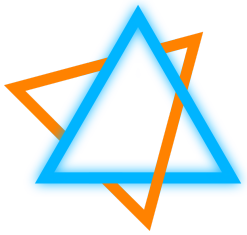
- **Number of BDD components**

If in a Unity Integration Test there is a component with a **StaticBDDComponent** Class, all the non-Static components are ignored, and you cannot use them. Furthermore, **you cannot have more than one Static Components** in the same Unity Integration Test.

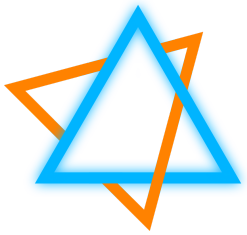
Q: Why do I have to choose the StaticBDDComponent classes instead of the Dynamic ones?

The use of the Dynamic Components let you reuse the same code, especially if all the test has the Given and When steps in common. With the Static Components, you have to duplicate your code for each class that describes a scenario.

All the information you put in the inspector for the Dynamic Component is stored in special arrays inside your component and inside the base class, stored using the inspector. If you do not pay attention, **you could lose the setting** previously stored in



your integration test (please, read the section BDD Framework: how to use it and avoid mistakes for more information). If you use the Static Components, it is not going to happen.



Methods Parameters Management

Dynamic Components can have **methods** with **parameters**. You insert their values via the inspector during the Dynamic Scenario configuration.

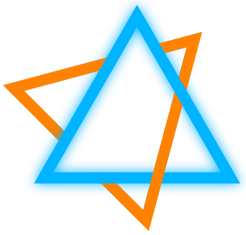
BDD Extension Framework can manage **many types of parameters natively**:

C# base types: bool, byte, sbyte, char, decimal, double, float, int, uint, long, ulong, short, ushort, string.

Unity3d: GameObject, Transform, Component,

Vector2, Vector3, Vector4, Rect, Quaternion, Matrix4x4, Color, Color32, LayerMask, AnimationCurve, Gradient, RectOffset, GUIStyle.

However, **you can add other managed types** to your own Dynamic Component. To do that, you have to **declare a ParametersValuesStorage** field inside your **Dynamic Component** as follow:



```
69

[HideInInspector]

[ParametersValuesStorage]

[SerializeField]

protected CustomType[] customFieldName;
```

Where **CustomType** is the type of the parameter you want to manage, and **customFieldName** is the name of the array field that is going to contain the values of your parameters. **You have not to initialise the field**; it is completely managed by the BDD Framework.

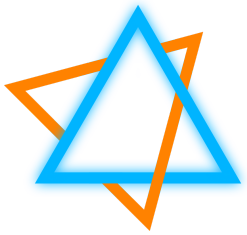
We want to bring your attention on the field declaration: it has to be, always, an array. Do not forget to make it an **array** or you will receive an error on the component inspector.

A Dynamic Component can have **only one ParametersValuesStorage for each Type**. If you declare two ParametersValuesStorage fields with the same element type, you will receive an error on the component inspector.

You cannot manage every type of classes. Unity has rules about that:

see the chapter "Simple field types that can be serialized" in this page on the Unity3D Documentation.

As you can read on Unity documentation, **you cannot serialise multilevel types**. Our field is **already an array**, so if you try to manage a container type, you will have unexpected behaviours.



If a **custom type** has its own **custom inspector**, the framework is going to use it to show the parameter properly on the inspector.

Q: How can I verify if I can use a type?

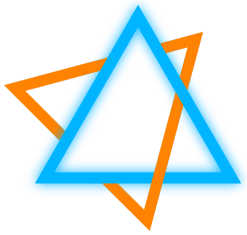
You can, initially, **make the field public** and delete the attribute **[HideInInspector]**. If the inspector of the component shows the field, the type can be used for your parameters.

Q: What if I want to manage a List<T> or an array?

The quick solution could be to declare a normal **public field** in your Component. Obviously, the type of the elements has to be one of those that Unity can serialise.

Q: If I make the ParametersValuesStorage field public and I delete the [HideInInspector] I can see the field in the component inspector. Is it right?

Yes, it is, but we suggest always to hide the field to avoid the temptation to modify its values manually. You cannot know how the BDD Framework is going to manage your changes.

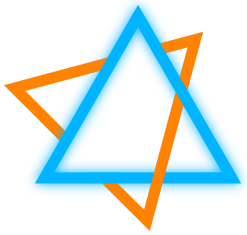


Headless running (batch mode)

Unity Test Tools can run in batch mode.

BDD Extension Framework is compliance with *Unity Test Tools*. The output of the integration tests is very detailed to allow the developer to analyse the error without issues even if the integration test fails in batch mode.

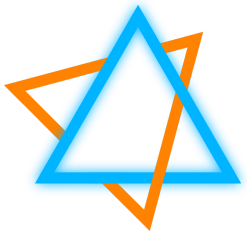
To discover how to run your integration tests in batch mode, please, read the chapter Headless running (batch mode) on the Unity Test Tools manual pages.



BDD Extension Framework Guidelines

There are some guidelines for using the BDD Extension Framework at the best of its potential.

You may decide to follow them or choose which of them you want to follow.



Behaviour Driven Development: how to organise the scenes

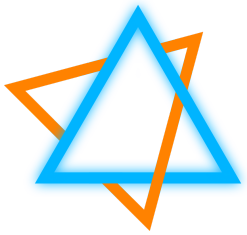
One of the guidelines for using the integration test of the Unity Test Tools is to **develop the integration tests in a different scene**, dedicated to the integration tests. This guideline is fully adopted by the BDD Framework.

If you are developing in BDD you should have a **clean scene** with only the pre-existing objects you need for the development process. You are going to create all the new objects in this scene during the development process and only the **strictly necessary things** that the requirements are asking for. We call this scene "**Development Scene**".

Because the creation of the GameObjects is made in a different scene than the "official" scene, you have to work with **prefabs**. Every GameObject you are going to create in the BDD scene has to be brought back to the original scene, once you finish the development process.

So, the development scene should contain only the **objects strictly involved** by the requirements. At the first time **do not add all the objects** you think should be involved by all the requirements: **add them once at a time**, only when you are going to develop a requirement that asks explicitly to interact with it. This behaviour allows you to avoid a "crowded" scene with useless objects.

If you develop your classes using **TDD**, you should use **mocked objects** to develop your code and simulate the behaviour of classes that interact with your new code. The



main example of mocked object is the **persistence layer**. It means that you develop your code in a protected and simulated environment.

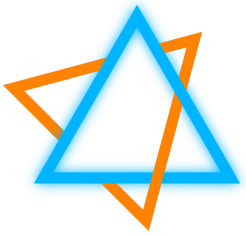
In this case, you should need to develop and run the "**Integration Tests**" for your new code. This time "Integration Tests" stands for the tests with the scope to assure that your new code is going to work properly using the real objects that you were mocking during the development process.

We suggest developing the "Integration Tests" **in the same development scene** you were using for the BDD. This because the "Integration Tests" use the same scenarios already implemented during the BDD session. You **could just copy** the original tests and modify them for using the real objects.

So in our Development Scene, we have the BDD tests, the Integration Tests and, perhaps, the Unit Tests if you were coding using TDD.

What about the Regression Tests?

The regression tests are, by definition, tests that interact with all the software, in real and **not simulated conditions**. The scene where you are going to develop your regression tests will be an actual copy of the real scene you are going to release. You have to develop your regression tests inside the copy of the real scene.

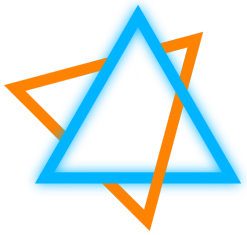


Behaviour Driven Development: how to organise the BDD Components

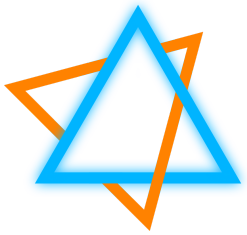
There are several approaches, depending on the environment you are working into.

If you are working in an agile environment, your code session could be driven by the **user story** you are implementing. In this case, each BDD Component could contain all the BDD Methods that you are going to use during the development of the acceptance tests for a single user story. This approach allows you to keep your classes less “crowded” from a huge pool of methods. It perhaps could mean you have to duplicate some BDD Methods throw the BDD Components because much code could be in common with each user story, especially for the “Given” methods. In this cases, if you are developing using Dynamic Components, you could use a **second BDD Component**: a generic one that is not specifically related to any user story. In this component, you can put all the BDD Methods that you are going to use frequently, across the user stories you are going to implement. After that, you have to add as a component the generic BDD Component plus the BDD Component you are developing for your user story. The BDD Framework can recognise both the components, allowing you to configure your test using the BDD Methods from both classes.

If you are not coding using the help of the user stories, but just coding some generic or specific requirement without clear acceptance criteria (development typical of waterfall processes), you may have not a guide for separating your BDD Components properly.

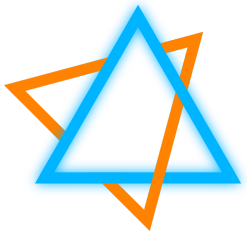


In these cases, you are going to have huge BDD Components, full of methods, without any suggestion how to split them into smaller classes. There is something you can do for containing the problem. For example, you could identify some areas in your software and split you BDD Components following that type of aggregation. An example could be the options panel with multiple pages: In this case, each page could identify a different BDD Component.



Behaviour Driven Development: Experimenting

The main purpose of the **Unity Integration Tests** is to help you to set your objects in the scene and develop your scripts also by **experimenting the results** of your work. It is right, all the developers have to learn new concepts and technologies all the time, and experimenting with the process “Try and Error” is a natural way to learn. However, this type of development is not related to the objectives of the Behaviour Driven Development. **Experimenting new technologies** is a fundamental part of your job, but **you cannot develop in BDD using this approach**. It does not mean that you have to give up with your experiments. You could start you developing session with BDD, but sometimes, during your coding, you could find some technology you do not know perfectly. If it happens, you could start to experimenting during your BDD process. In this case, **we recommend paying close attention**. During your experiments, you are going to write much code that is going to be useless (dead code) or not optimised, often very messy. During your experiments you could lose the focus to make a clean design, that is one of the main objectives of the BDD approach.

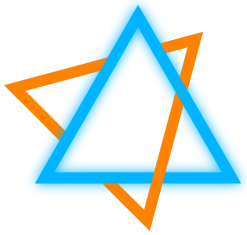


At the end of your coding session, when your code works properly, **you could take three different approaches**:

- **Refactoring**: You should have a refactoring session every time, even if you are not experimenting. It is an important part of the process. It allows you to clean your code and organise it in a better way. However, sometimes you could lose the design scope of BDD. If you think your code does not match anymore with the natural behaviour of the users of your software, you could consider the second approach.
- **Develop your code again using BDD**: Sometimes your code is so messy that you could think it is irrecoverable: you could try to write your code again using the new skills you just acquired about the new technology. This approach allows you to get your thoughts together and write better code.
- **Both the approaches**: the better way could be to take both the approaches. It means to make some refactoring of the core classes that use the technology you are learning and using BDD to design a clearer pool of classes that use the refactored ones.

You could feel to have to do extra work: **Why do I have to write again a code that took me a week to work fine?**

The answer is simple and significant. **You have an objective** that is not only to code a working software: **writing code that is clean, readable, maintainable, reusable.**



BDD Framework: how to use it and avoid mistakes

The BDD Extension Framework gives you the possibility to develop your tests with the Given-When-Then approach. It makes possible to use the Behaviour Driven Development process for developing your software. How can you use it to have the **best results**?

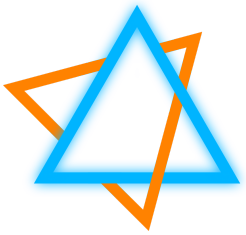
BDD Framework: How to create your own dynamic test

A BDD integration test is an actual Unity Test Tools Integration Test.

You are going to start the creation of your test. You can do it in two ways:

- Select from the menu: "Unity Test Tools" -> "BDD Extension Framework" -> "**Create BDD Test**": the test is going to appear in your Hierarchy window with the name "New Test".
- Creating a normal integration test in the Unity Test Tools Integration Panel. It creates a new object in your Hierarchy window named “New Test”. Its inspector shows just the Test Component script attached to the game object. To make this integration test a BDD Integration Test **you have to add the component named “BDD Extension Runner”**

After adding it, the framework **checks the Integration Test for errors**. At the very beginning, we are going to have an empty inspector saying: **"Please, add your BDD Components and enjoy BDD."**. It means, obviously, that we have to add a BDD

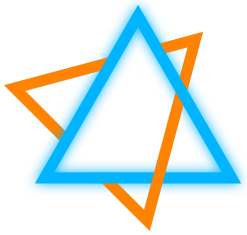


Component before starting to configure the scenario. If you have not yet the right class to add, you first have to create it. You can just use the empty **template** called "**TemplateDynamicBDD.cs**" by copying it in your script folder and renaming the name of the file and the class name.

Now you have to create your BDD Methods. First of all, you have to have the **test scenario** written in the **Given-When-Then** format. After that, you can replicate the sentences in your scenario into the BDD Methods.

Now you can add your new script to the integration test:

After adding the component, **the combo boxes** on the runner inspector **are not set to any value**, for two reasons: first there is not yet a BDD Component in your test, and second, the BDD Framework cannot know which methods you want to set. Instead of the scenario sentences, the BDD Framework shows the writing "**### Please, choose a method or delete the row. ###**". It is just a reminder to set up your test properly, otherwise, you cannot run it.



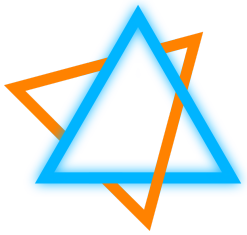
BDD Framework: How to create your own static test

Similar to the Dynamic Tests, you have to create first the BDD Test by selecting the menu element "Create BDD Test" from the menu: "Unity Test Tools" -> "BDD Extension Framework": the test is going to appear in your Hierarchy window with the name "New Test".

After that, you have to create your **Static Component**. You can use the empty **template** called “**TempateStaticBDD.cs**”: copy it in your script folder and rename the filename and the class name.

When you create your BDD Methods be sure to set the **numeric parameter** order for all the methods that have to be **Step Methods**.

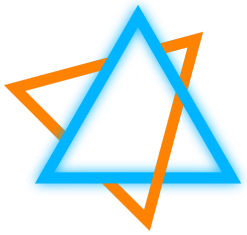
When you think you are ready, **you can add the Static Component** into the integration test: the framework recognises it automatically and populates the scenario.



BDD Framework: How to avoid to lose information

All the information set using the inspector are stored in **particular arrays**. Usually, the framework hides them, but you can consider them as public properties of components attached to the integration test object, so they follow the same rules as the standard component properties:

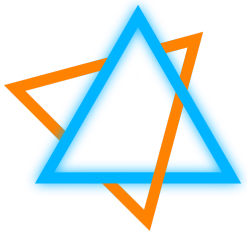
- **Parameters**: The values of the parameters for the Dynamic Components Methods are stored in the ParametersValuesStorage fields. If you **delete**, **rename** or **change** their **types**, you are going to **lose** all the **data stored** inside because the inspector could lose the pointer to them. Furthermore, if you **delete**, **rename** or **change** the **type of a parameter** inside a method signature, **the framework loses the pointer** to the parameter and the value stored for it.
- **Dynamic Step Methods**: The values you can read on the combo boxes for the Dynamic Tests are stored as strings inside the BDD Extension Runner component. If you **rename** a **method** or a **class**, **the runner loses the pointer** to the methods. When it happens, the runner shows an error, asking you to fix it.
- **Do not remove the Dynamic Components from the integration test**: if you do that, **all the related data will be deleted** by the inspector.
- **Do not reset your components or the BDD Extension Runner component**: if you do that, **all the related data will be deleted** by the inspector.



Q: What can I do if I notice I lose data from the Inspector?

If the action that made you lose the data was made on the Inspector, just use the **Undo feature**.

If you have deleted or renamed something inside your BDD Component, t**he Framework usually tries to protect the data** stored inside the BDD Extension Runner giving you an error and letting you fix it. Unfortunately, if the data lost were stored inside the component (the values of the parameters) you could have lost them forever.

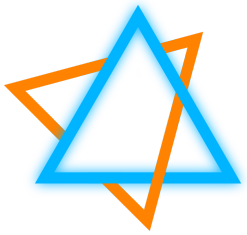


BDD Framework: Error and problems handling

Here there are the **common errors** that can occur during the BDD Framework usage. **They appear on the inspector** of the component that generates the error.

There are two types of errors: **Component errors**, and **Runner errors**.

Most of the component errors are reflected also into a runner error, for avoiding to miss the error messages in case of the component that generates the error is located outside the scroll area of the inspector.



Components Errors

Message: "There are more than one BDD Methods with the name componentName.methodName. You cannot have two or more methods with the same name."

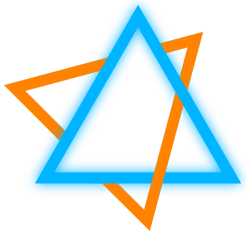
Reason: A BDD Component cannot have more than one BDD Method with the same name. The overloading by changing the parameters is not permitted.

How to fix it: Rename the methods.

Message: "The field componentName.fieldName is not an array."

Reason: You are trying to configure a ParametersValueStorage field, but the type of the field is not an array.

How to fix it: add the array declaration on the field or delete the attribute [ParametersValuesStorage]



Message: "The component componentName has more than one ParametersValuesStorage for the type typeName".

Reason: You are trying to declare another ParametersValuesStorage field for the same type.

How to fix it: Delete one of the declaration. If one of them is placed inside the DynamicBDDComponent class, delete the other declaration. It could be that you are trying to declare a field using an alias for another type already declared (Ex. int, System.Int32).

Message: "The field componentName.fieldName is not public but it has not the [SerializedField] attribute. The inspector is not going to manage it."

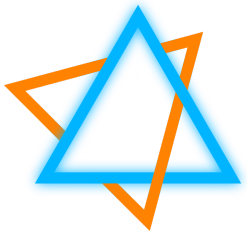
Reason: You forgot to add the [SerializedField] attribute on the field declaration.

How to fix it: add the [SerializedField] attribute to the field declaration.

Message: "There is not ParametersValuesStorage for the type ###type for the parameter parameterName for the method componentName.methdoName".

Reason: BDD Extension Framework cannot locate a ParametersValuesStorage field for the indicated type in the BDD Component.

How to fix it: Check the declarations of your ParametersValuesStorage fields. They have to have the attribute [ParametersValuesStorage] and the attribute [SerializeField]. Create the ParametersValuesStorage field.



Message: "The method componentName.methodName does not return an IAssertionResult value."

Reason: "The return value of the indicated method is not "IAssertionResult".

How to fix it: Change the return value into "IAssertionResult".

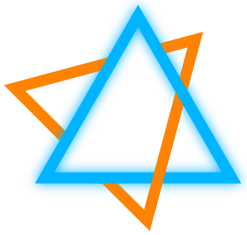
Message: "Method componentName.methodName not found. It is referenced in a CallBefore attribute for the method componentName.otherMethodName".

Reason: The BDD Extension Framework cannot locate the method indicated inside the BDD Component. It could be:

- a typo
- you are indicating a method that is not a BDD Method
- you are indicating a method the does not belong to the same BDD component.

How to fix it:

- correct the spelling of the name of the method
- correct the declaration of the method
- copy the method and its body from another component.



Message: "The method componentName.methodName has a CallBefore attribute but it is not a BDD Method."

Reason: The method decorated with the attribute [CallBefore] is not declared as BDD Method.

How to fix it: change the declaration or delete the [CallBefore] attribute.

Message: "The method componentName.methodName has a wrong value for CallBefore.ExecutionOrder: n. It must be >0".

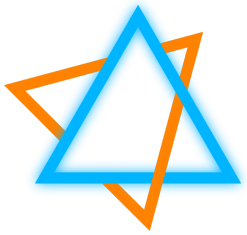
Reason: The value indicated is 0;

How to fix it: change it with a value>0

Message: "The method componentName.methodName has duplicated CallBefore.ExecutionOrder: n".

Reason: For the same BDD Method there are at least two [CallBefore] attribute with the same value of the parameter.

How to fix it: change the values. They have to be unique.



Message: "The method componentName.methodName has a missing CallBefore.ExecutionOrder: n".

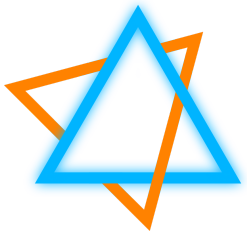
Reason: There is not a [CallBefore] attribute for the indicated method with the indicated value, but there is another [CallBefore] attribute with a greater value.

How to fix it: change the values of the parameter of the [CallBefore] attribute. Their numbering has to be sequential.

Message: "The method componentName.methodName has a recursive call. Recursive calls are not allowed. Call chain: TxTxTxTxTx."

Reason: The method indicated can obtain another call of itself following the nested calls of its [CallBefore] attributes. The recursive calls are not permitted.

How to fix it: Read the error text to find where the recursion is placed and correct the [CallBefore] attributes to avoid recursions.



Message: "The CallBefore chains declared for the method componentName.methodName have a non-unique identification for the parameters of the method identified by the key: componentName.methodName.parameterName.Ids. Please, use the Id property in the CallBefore attributes for making them unique."

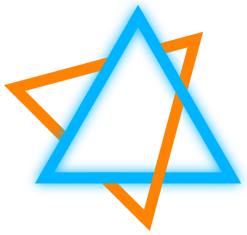
Reason: If a BDD Method with parameters is called more than one time using the [CallBefore] attributes by a BDD Method, the BDD Extension Framework has difficulties to manage the values of the parameters.

How to fix it: Add the property Id in each [CallBefore] attribute using different arbitrary string.

Message: "The [Given|When|Then] declaration for the method componentName.methodName has a wrong ExecutionOrder value: it must be >0".

Reason: The value for the executionOrder parameter has to be >0 for being interpreted correctly by the BDD Extension Framework.

How to fix it: change the value of the parameter with a numeric>0.



Message: "The [Given|When|Then] declaration for the method componentName.methodName has a duplicate ExecutionOrder value. Check the others [Given|When|Then] methods.";

Reason: for each Step Type (Given, When or Then) the numbering of the parameter ExecutionOrder has to be sequential without repetition.

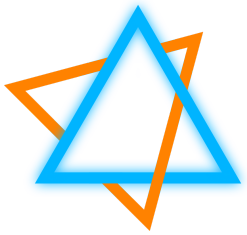
How to fix it: change the values of the ExecutionOrder parameters to make them unique.

Message: "The component componentName has a missing [Given|When|Then].ExecutionOrder: n."

Reason: There is not a Step Method for the type indicated that has the ExecutionOrder parameter set to the value n, but there is another ExecutionOrder that has a value greater than n.

How to fix it:

- Change the values of the ExecutionOrder parameters to make the numbering without any jump.
- Add a Step Method with the value indicated.



Message: "The component componentName has not [Given|When|Then] components".

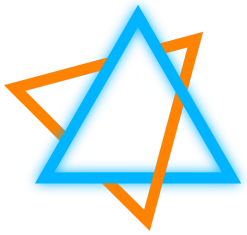
Reason: The BDD Extension Framework cannot locate a method inside the indicated component with the [Given|When|Then] declaration. There must be at least one Step Method for each Step Type.

How to fix it: Create a BDD Method for the indicated Step Type.

Message: "The method componentName.methodName is not allowed to have parameters".

Reason: The Static Components are not allowed to have BDD methods with parameters.

How to fix it: delete the parameters from the signature of the indicated method.



Runner Errors

Message: "Please, add your BDD Components and enjoy BDD."

Reason: The BDD Extension Framework cannot locate a BDD Component attached to the Integration Test.

How to fix it:

- Add a BDD Component
- Change the declaration of an already attached component
- Delete the BDD Extension Runner.

Message: "The component componentName is duplicated."

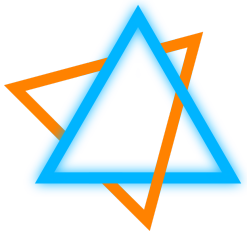
Reason: The BDD Extension Framework finds more than one BDD Component with the same name. You can have only one component at a time with the indicated name.

How to fix it: Delete the other component with the name indicated.

Message: "There is more than one Static BDD Component. Only one is allowed".

Reason: For a Static Scenario, the BDD Extension Framework needs to have one Static Method only.

How to fix it: delete the unused Static Methods.



Message: "Incomplete settings detected on [Given|When|Then] methods at position n".

Reason: For a Dynamic Scenario there is a combo box not set yet with a Step Method.

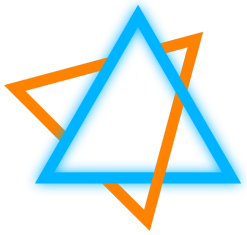
How to fix it: Choose a Step Method or delete the row.

Message: "Method componentName.methodName not found on [Given|When|Then] methods at position n".

Reason: An already configured Step Method points to a method that is not present in the indicated component anymore. It could happen if the original method is renamed, deleted, or its BDD Declaration is changed (for example if it is changed from Given to When).

How to fix it:

- Rename back the method to the original name.
- Create a BDD Method again with the name indicated.
- Fix the BDD Declaration of the method.
- Choose another method.



Message: "The parameter componentName.methodName.parametername is not found in [Given|When|Then] methods at position n".

Reason: After configuring the Step Method, the parameter indicated has been renamed or deleted.

How to fix it:

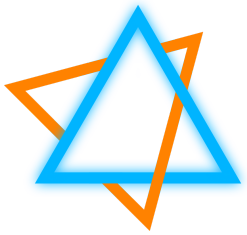
- Rename back the parameter.
- Create another parameter with the same name and type.
- Execute the "Rebuild Settings" option if you want to delete the previous information about the parameter indicated.

Message: "The parameter componentName.methodName.parametername has a wrong type in [Given|When|Then] methods at position n. Previous type: type1 Current type type2".

Reason: After configuring the Step Method, the parameter indicated has changed its type.

How to fix it:

- Change back the type or the parameter indicate: you are going to save the value previously inserted via inspector.
- Execute the "Rebuild Settings" option if you want to overwrite the previous information about the parameter indicated: you are going to lose the previous value inserted via inspector.



Message: "The component for the method componentName.methodName is not found in [Given|When|Then] methods at position n".

Reason: After configuring the Step Method, the component that contains the method has been removed from the Integration Test or has been renamed.

How to fix it:

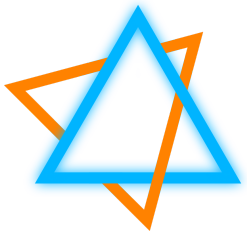
- Add the component again.
- Rename back the component.
- Chose another Step Method or delete the row.

Message: "The ParametersValuesStorage field fieldName for the parameter componentName.methodName.parameterName is not found in [Given|When|Then] methods at position n".

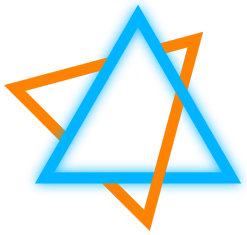
Reason: After configuring the Step Method, the BDD Extension Framework cannot locate the indicated field. It could be renamed or deleted.

How to fix it:

- Rename back the indicated field.
- Create the field again.



- Execute the "Rebuild Settings" option if you want to overwrite the previous information about the field indicated: you are going to lose the previous value inserted via inspector for the parameter indicated.



The BDD Framework in detail: the code reference

DynamicBDDComponent

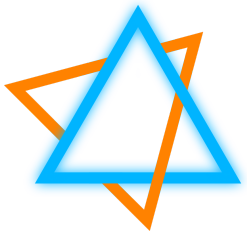
Every component has to inherit from the class DynamicBDDComponent to became a Dynamic Component. When it does, and it is attached to an Integration Test, in its inspector appears the logo of the Dynamic Component. It any logo appears, probably there is some build error, or the component does not inherit from DynamicBDDComponent class.

When a Dynamic Component is attached to an integration test, it is detected by the BDD Extension Runner that scans the content of the component looking for the Given-When-Then BDD Methods to show in the combo boxes in the inspector. There can be more than one Dynamic Components attached to an integration test.

StaticBDDComponent

Every component has to inherit from the class StaticBDDComponent to became a Static Component. When it does, and it is attached to an Integration Test, in its inspector appears the logo of the Static Component. It any logo appears, probably there is some build error, or the component does not inherit from StaticBDDComponent class.

When a Static Component is attached to an integration test, it is detected by the BDD Extension Runner that scans the content of the component looking for the



Given-When-Then BDD Methods that compose the BDD Scenario to show in in the inspector. There can be only one Static Component attached to an integration test. If there is also a Dynamic Component, it is going to be ignored.

[**HideInInspector**]

This attribute is the standard Unity attribute that tells to Unity to not show the following public property on the inspector. It is used mostly for hiding the arrays with the property [ParametersValuesStorage]

[**SerializeField**]

This attribute is the standard Unity attribute that tells to Unity to use the following field as serialised property of the component. It has to be used if the field is not explicitly declared public.

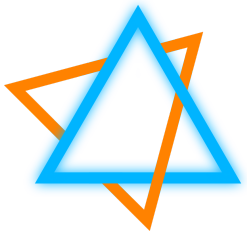
[**ParametersValuesStorage**]

This attribute tells the framework to consider the following array as a storage for the methods parameters values inserted via the Unity inspector.

There has to be only one array for a type in the whole chain of inheritance.

There has to be one array for each type used in the parameters of the methods.

You cannot use this attribute for the Static Components.



protected Type[] arrayName;

It is the official declaration of a ParametersValuesStorage field. The type of the elements of the array has to be the same of the parameter you want to manage.

It has to be protected or private, and it has to be preceded by the [SerializeField] attribute for let the inspector to use it.

The possible types of the elements of the array have to follow these guidelines.

After inserting parameters values on the inspector, DO NOT rename or delete the array: you are going to lose all the values inserted via the inspector in all the integration tests and in all scenes you used it.

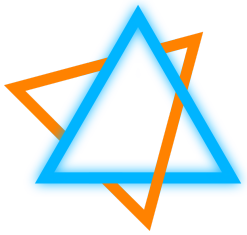
[Given|When|Then(string scenarioText, uint delay, uint timeout]

This attribute tells the framework to consider the following method as BDD Method for a Dynamic Scenario.

string scenarioText: mandatory: it is the text that describes the scenario, using a colloquial sentence that follows the words “Given”, "When", "Then" or "and".

Example: Given(“there is only one object in the scene”)

scenarioText: can contain the code %parameterName%, where parameterName is the name of a parameter in the method signature. The framework substitutes the code with the ToString() value of the parameter.



uint delay: optional: default value=0: it is the value in milliseconds of the delay request to run method.

uint timeout: optional: default value=3000: it is the value in milliseconds of the timeout that can be reached for the Retry execution of the method. The timeout is interpreted by the BDD framework as an upper limit of time for the Retry operation for the execution of the Method. If the timeout is reached during the execution of the method, the method keeps running until it's natural end.

There can be only one attribute of the same type for each method in the BDD Component.

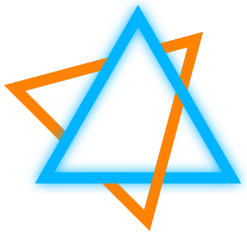
There can be more than one BDD methods with the same attribute type in the BDD Component.

[Given|When|Then(uint executionOrder, string scenarioText, uint delay, uint timeout]

This attribute tells the framework to consider the following method as BDD Method for a Static Scenario.

uint executionOrder: mandatory: it codes the order of the execution of all the Step Methods of the same step type (Given, When or Then). It has to start with the number 1 and the numbering has to be sequential.

string scenarioText: mandatory: it is the text that describes the scenario, using a colloquial sentence that follows the words "Given", "When", "Then" or "and".



Example: Given(“there is only one object in the scene”)

uint delay: optional: default value=0: it is the value in milliseconds of the delay requested to run method.

uint timeout: optional: default value=3000: it is the value in milliseconds of the timeout that can be reached for the Retry execution of the method. The timeout is interpreted by the BDD framework as an upper limit of time for the Retry operation for the execution of the Method. If the timeout is reached during the execution of the method, the method keeps running until it’s natural end.

There can be only one attribute of the same type for each method in the BDD Component.

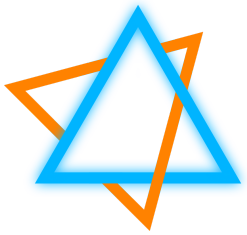
There can be more than one BDD methods with the same attribute type in the BDD Component.

[CallBefore(uint executionOrder, string method, uint delay, uint timeout, string id)]

This attribute tells the BDD Framework to execute another BDD method action before executing the main BDD method.

There can be more than one [CallBefore] attributes for each BDD method.

A BDD method may not have a [CallBefore] attribute.



uint executionOrder: mandatory: it codes the order of the execution of all the methods of a group of a [CallBefore] attribute. It has to start with the number 1 and the numbering has to be sequential.

string Method: mandatory: it is the name of the BDD method that the BDD Framework has to execute. It has to be a valid BDD method defined inside the same BDD Component.

uint delay: optional: default value=0: it is the value in milliseconds of the delay requested to run method.

uint timeout: optional: default value=3000: it is the value in milliseconds of the timeout that can be reached for the Retry execution of the method. The timeout is interpreted by the BDD framework as an upper limit of time for the Retry operation for the execution of the Method. If the timeout is reached during the execution of the method, the method keeps running until it's natural end.

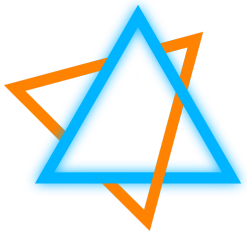
string id: optional: an arbitrary string to assure the unique identification of the parameter in the group of the [CallBefore] attribute.

public IAssertionResult MethodName(type1 parameterName1, type2 parameterName2, …)

This is the standard declaration of a BDD method for Dynamic Components.

It has to be public for letting the framework to access to it.

The return value has to be a standard implementation of the interface IAssertionResult.



The return value cannot be null.

The types of the parameters have to respect the rules described in this section.

After inserting values into the parameters via the inspector do not rename the parameters: you lose the values inserted previously.

A BDD method can have only one attribute [Given] at a time.

A BDD method can have only one attribute [When] at a time.

A BDD method can have only one attribute [Then] at a time.

A BDD method can have only one attribute [GenericBDDMethod] at a time.

A BDD method can have all the three [Given] [When] [Then] attributes at a time or a subset of them.

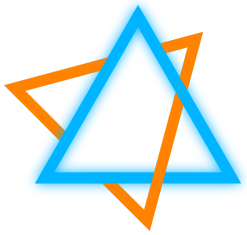
A BDD method has to have at least one of [Given] [When] [Then] [GenericBDDMethod] attributes.

A BDD method can have one or more [CallBefore] attributes.

A BDD method may not have a [CallBefore] attribute.

AssertionResultSuccessful()

It is one of the three implementations of the interface IAssertionResult. A BDD method returns it if the result of the test performed is successful.



AssertionResultFailed(string text)

It is one of the three implementations of the interface IAssertionResult. A BDD method returns it if the test is failed or if there is some unexpected error during the execution of the method. The text string has to describe the error occurred and it is included in the integration test log. It rises a standard fail assertion for the Unity Test Tools.

AssertionResultRetry(string text)

It is one of the three implementations of the interface IAssertionResult. A BDD method returns it if the method has to wait for a pre-determined state of the software. If a timeout of 3 seconds, by default, or after a timeout defined with the timeout property is reached, the BDD Framework interprets the AssertionResultRetry as an AssertionResultFailed object. The text string has to describe the error occurred and it is included in the integration test log. If treated as an AssertionResultFailed object, it raises a standard fail assertion for the Unity Test Tools.