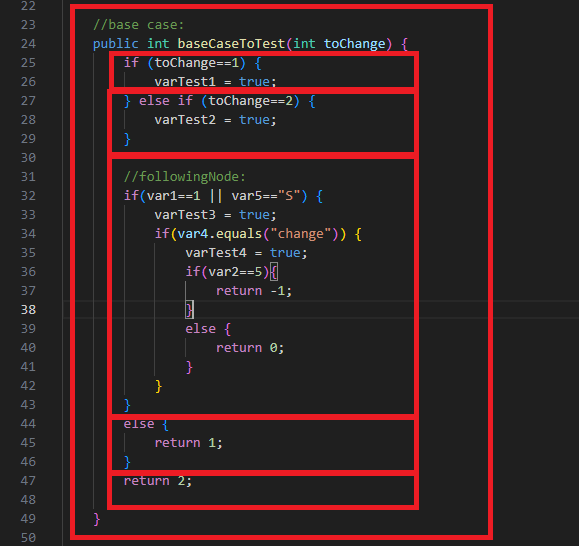
Logic of code and tests structure

In every java method it is possible to see a tree structure. Every conditional statement can be seen as a “node”. A node is some code that can or cannot be runned based on specifical conditions. Each node is composed by a single condition (mandatory) and a series of operations (optional) to do if the condition is true **[code inside curly brackets after the “if”]** and a series of operations to do if the condition is false [**code inside curly brackets after the “else”]** (optional) and a series of “children” nodes (optional).

In the figure every node is highlighted with a red rectangle. There is a “first node” (line 25) and all nodes except the first has some kind of relationship with a “parent” node. There are tree types of relationships:

1. **caseTrueNode:** A node can depend on the fact that the condition of the “parent” node is true (eg. the node starting at line 34 can be runned only if the condition of the node starting t line 32 is true)
2. **caseFalseNode:** A node can depend on the fact the the condition of the “parent” node is false (eg. the node starting at line 27 can run only if the condition of the node starting at line 25 is false)
3. **caseFollowingNode:** A node can run regardless the condition of the “parent” node is true or false (eg. the node starting at line 32 is not influenced by the return of the node starting at line 25). Note that if the “parent” node influence some variables used in the “children” node you should use a true/false node instead of a “following” node.



After defining a node, you can notice that if you want to cover all the node you need to write two jUnits:

For the “true” case:

1. Basic header of the method (@Test annotation, name of the test method, ecc.)
2. Code you need to enter in the node (so the code of the “parent” nodes)
3. Code you need to make the condition true
4. Code of the call to the method
5. Asserts and code you need to verify the method has runned correctly.

For the “false” case:

1. Basic header of the method (@Test annotation, name of the test method, ecc.)
2. Code you need to enter in the node (so the code of the “parent” nodes)
3. Code you need to make the condition false
4. Code of the call to the method
5. Asserts and code you need to verify the method has runned correctly.

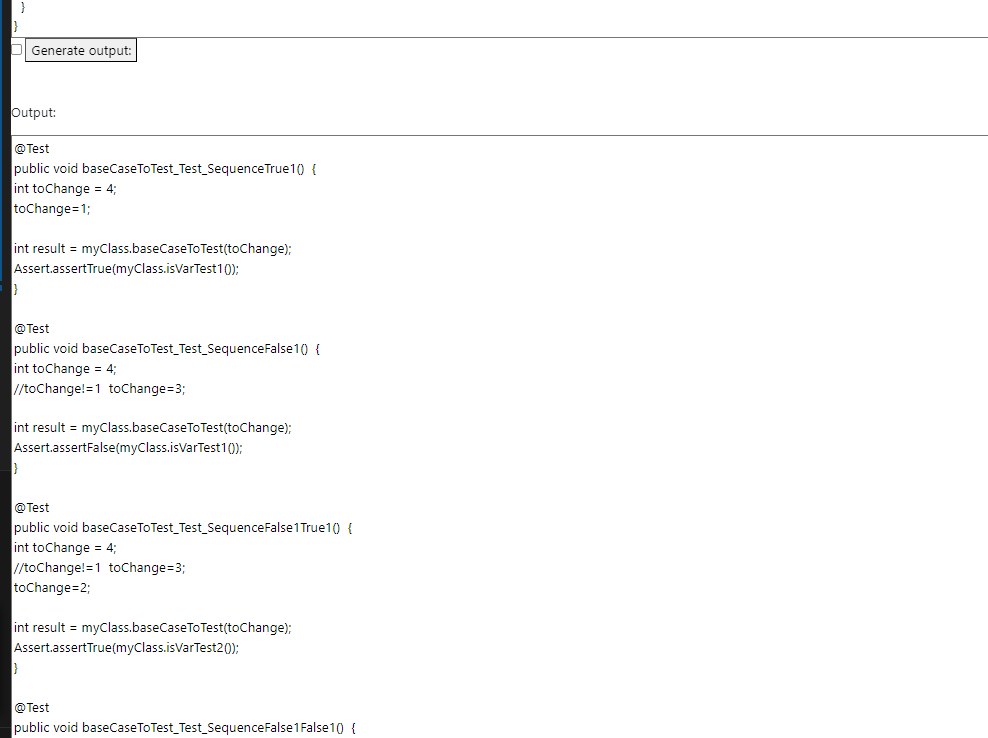
Using UnitTestsWriter, you will write only the “current” conditions for true or false and the asserts. All the remaining part will be automated by the script.

Basic generation logic

UnitTestsWriter generate code using a json. When a json is properly populated and the “Generate output” button is pressed, the code will be generated.

The json can be created both manually or using the visual interface proposed (reccomended). You can also use both: when you change the json the visual interface changes and vice-versa.

Remember if you put a json in the field, you can recover your work as if it is a “save”.

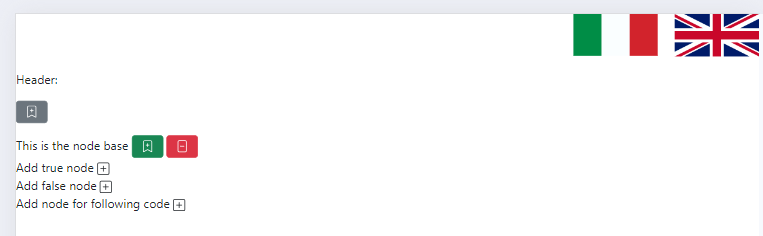
From the json, you will generate a set of method you need:  


If you want to directly modify the json, remember it should have 2 objects: header and first node and every new “node” should contains all the 8 fields as in the “json for example Methods”. A node can also contains another node.

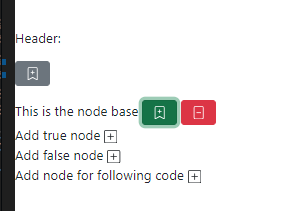
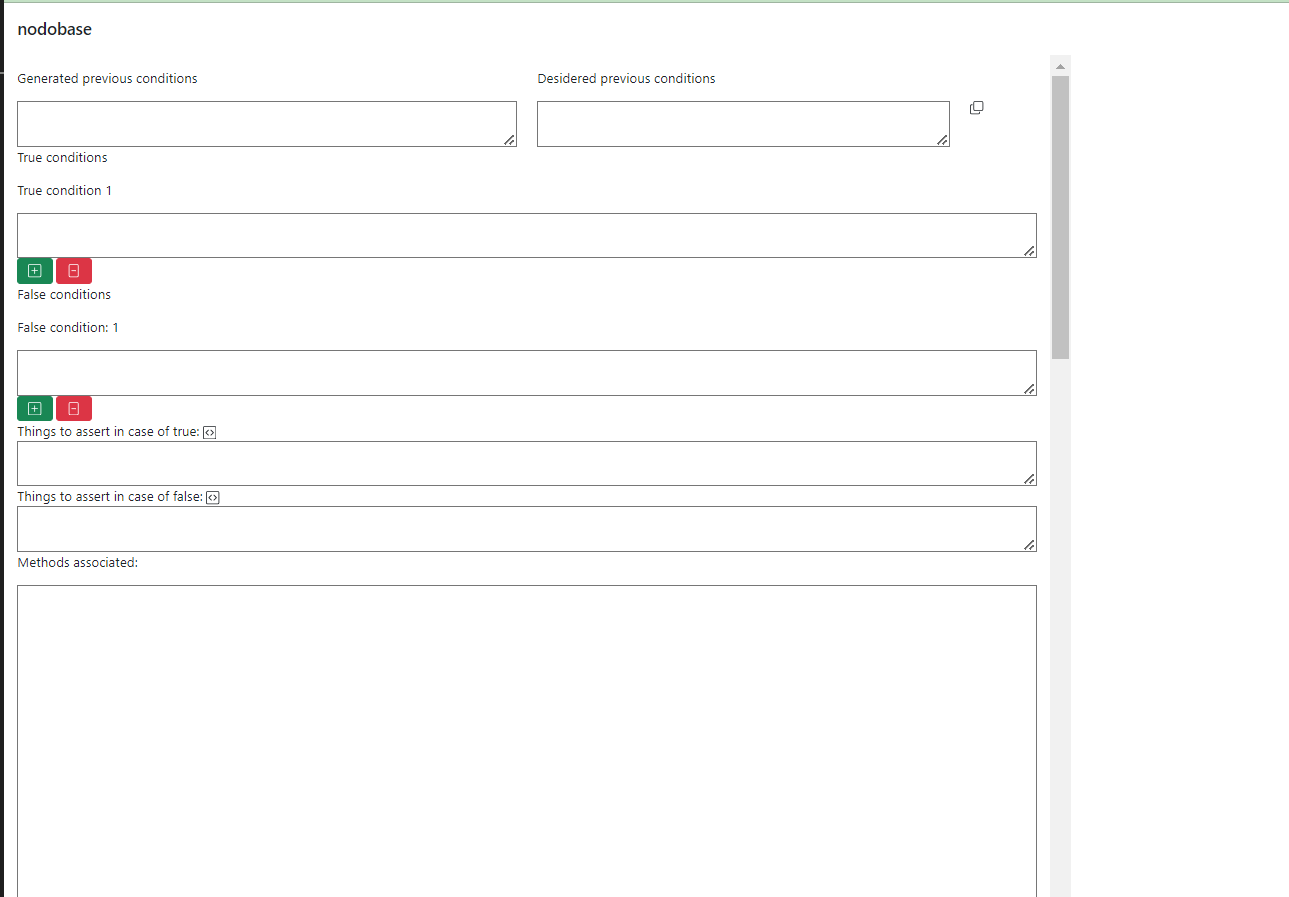
Graphic interface

There is a “graphic interface” that is used to generate the json. Note that any change in the json causes a change in the graphic interface and every change in the graphic interface causes a change in the json. So notice you can use the json also as a “save” system or you can edit in order to do things you want.

A “default” json is created at refresh and contains header and a first node.

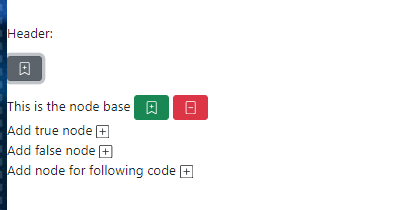


All the graphic interface is based on a “bootstrap canvas”. If you click to open the node, you will see all the node data:

# Header node details

Clicking to the “header icon”, the header node details are shown:





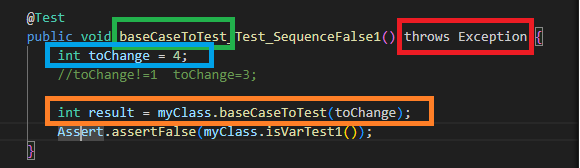
The data of these fields are the same inside the “header” object of the json.

The data contained are:

1. “commonHeader”: Field that will be put in every single test method at the begin
2. “methodName”: the name of the method without any other details. This field will be used to create the name of the testMethod: “@Test public void [nameMethod]path “.
3. “callOfTheMethodStr”: here you should write the method how you want to call in the methods.
4. “throwsStr”: Here you should write if something is wanted to be thrown.

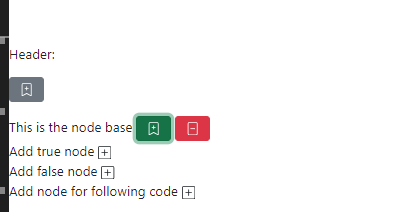
Here is one of the “output” methods created using the correspondant header:

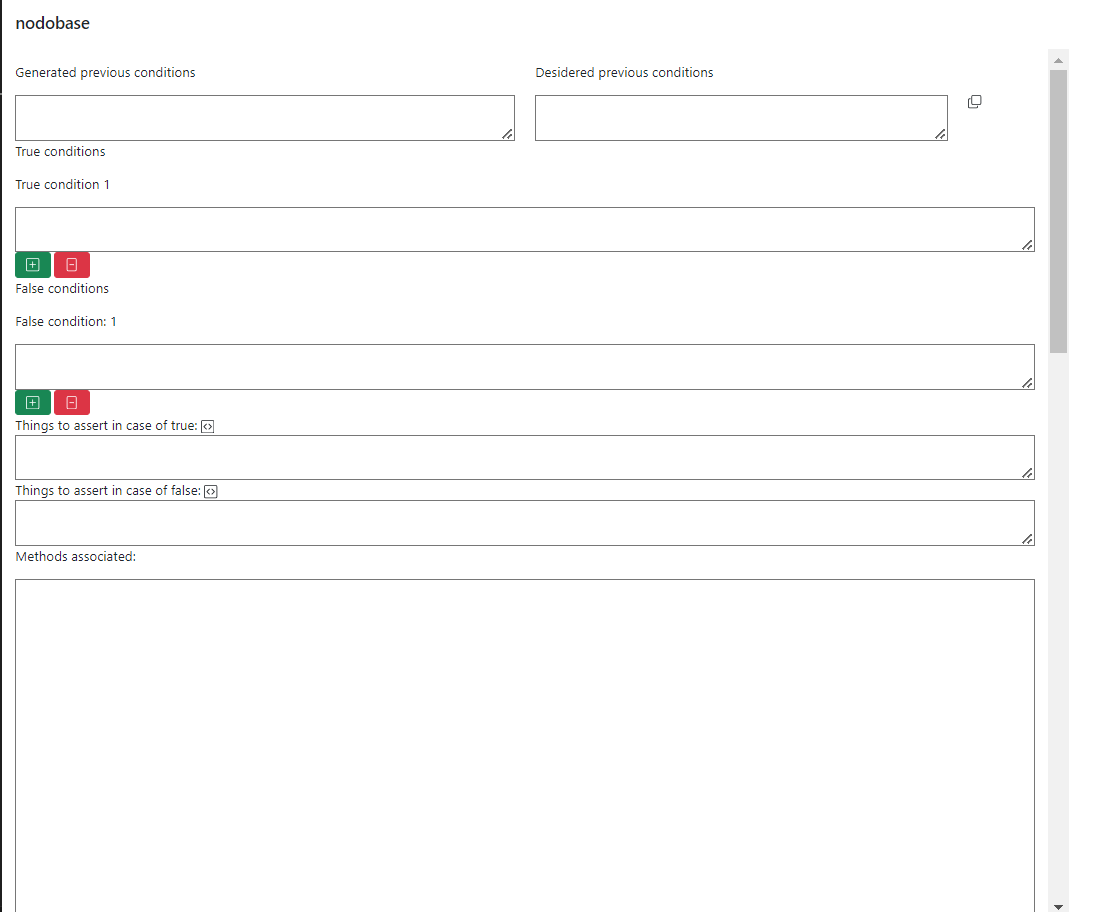




# Body node details

## Explanation

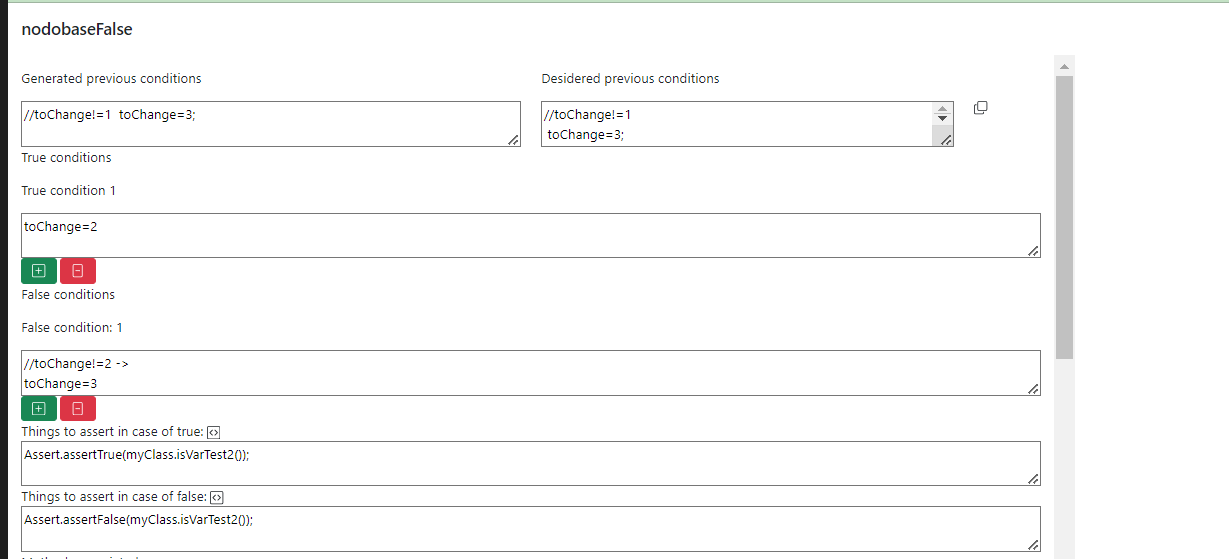
Clicking on the “node detail icon”, you will see all the details on the body and you will be able to edit them:  


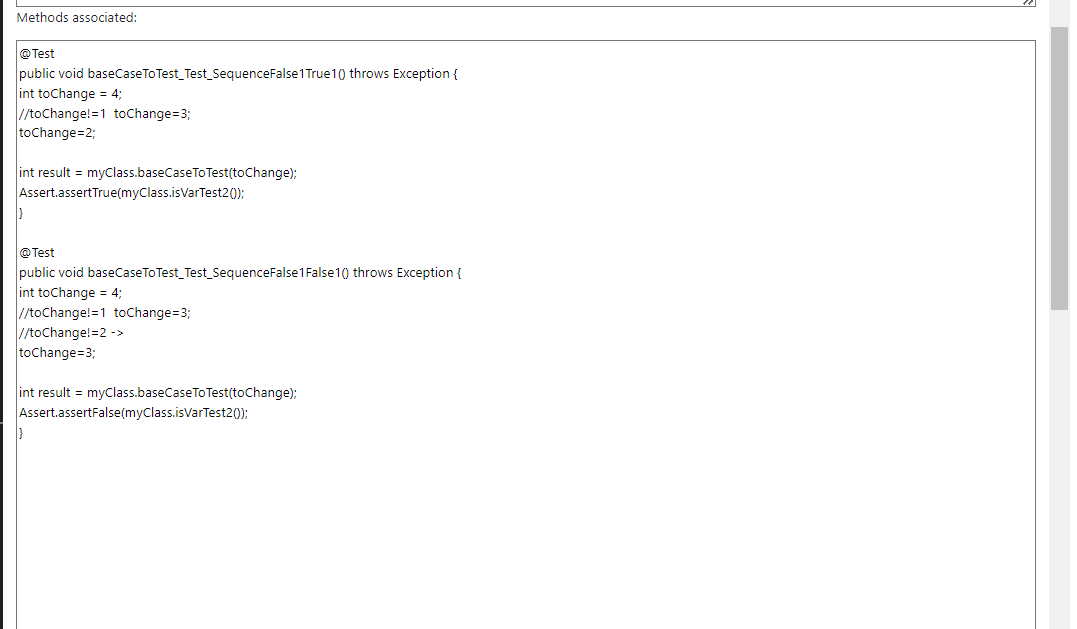
****

Most of these fields are the same as the ones in the corresponding node of the json.

1. Generated previous condition: null for the “baseNode”. For others can be one or more fields containing data “reported” from the other nodes. Those conditions should be necessary to be able to run the code of the node. They are computed and readonly.
2. “Desidered previous condition” is a field corresponding to “commonConditions” in the json. In the final method you will see this code immediately after the “commonHeader” fields. If you click the button on the right, you will copy the content from “Generated previous condition”. This action is recommended, but sometimes you may adjust this code.
3. “trueConditions/falseConditions”: it contains a string of instruction that must be put before the call of the method that manages the true/false case of the node. This is code necessary in order to run the code related to condition true/false. If there are “&&” or “||”, there are multiple valid conditions. Here is why the user can “add” or “remove” using “+” or “-”. If the user want to remove, an alert will show to ask a confirm.
4. “assertTrue/assertFalse”: It contains a string of instruction that must be put after the call of the method. This string should contains assert and other verifications that the method has been executed correctly. If you put some code of the method to test [eg. myClass.setVarTest2(true) ] and you click on the correspondent button, it might transform into the correct assert condition [eg. Assert.assertTrue(myClass.isVarTest2()) ]
5. “Methods associated”: a readonly output containing all the test methods you will generate with the specifical instructions given for that node. This code will be generated with all the other methods in the final output.

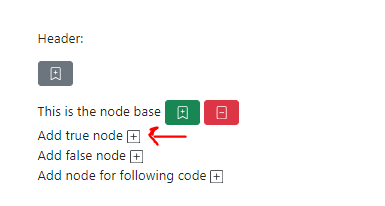
## Example

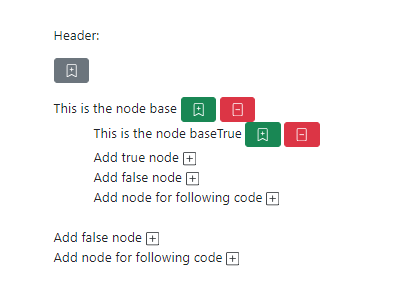
Passing the following code, you will get the following input:  




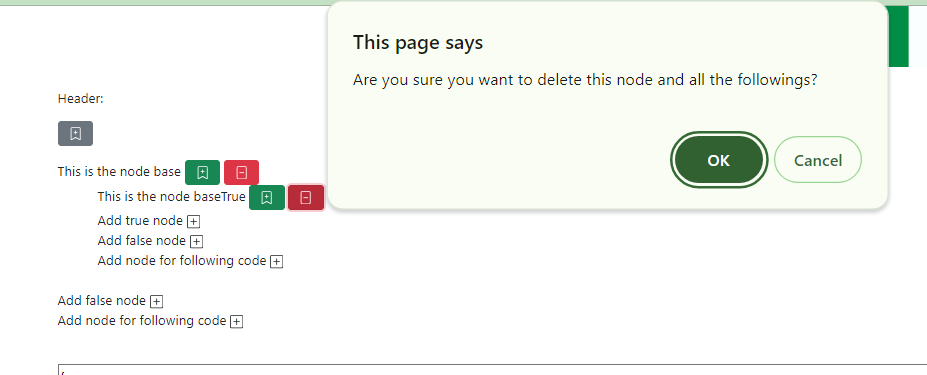
# Node creation and management

If the method contains more than 1 node, it is possible to create a correspondent new condition node with the correspondent button.





After adding a node, you can populate with the button. If a node is created by mistake, you can remove it with the red button. A window will ask for confirm.



# Final result example

After creating and population all the nodes, the json is ready. Clicking “generateOutput” button, all the test methods will be written in the output field.

In description an example of a json (putting it in the json field the all graphic interface will be generated) and an example of the tested methods and the generated tests. Note that “duplicate lines” will be automatically removed.

However, sometimes a revision is needed. The example test code is reported not manually corrected in order to help understand some limitations. A class with all data is created too.

# Languages

Default language is English. If the correspondent flag is clicked, the language will change. Other languages can be added