b
UNIVERSITÄT
BERN

TestView Plugin

A Nautilus Plugin to facilitate Unit Testing

Bachelor Thesis

Dominic Sina
from
Zollikofen BE, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

05. Februar 2015

Prof. Dr. Oscar Nierstrasz
Boris Spasojević

Software Composition Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

The purpose of this bachelor project is to improve the Pharo environment by making it more unit test friendly. Instead of writing a new system browser we chose to realize this as a Nautilus plug-in since this makes it easy to set up and builds on established parts of the Pharo environment. This plug-in includes various functionalities that help with finding and writing unit tests. Further it provides ways to check if a method is untested, to see all tests that have been written for a certain method, to add new tests to existing methods and the ability to view a method and a corresponding test side by side. Before explaining the plug-in and its functionalities in more detail we will take a look at various terms surrounding unit testing and analyze the develop environments Eclipse and Pharo. They will be compared to each other in how unit test friendly they are and opportunities to improve them will be discussed. The aim of this thesis is to take a closer look at unit testing and in form of a Nautilus plug-in provide an example of how unit testing can be facilitated.

Contents

1 Introduction

Testing has become a very important part of software development. The reason for this is that testing is useful to validate a project. While it is true that “Program testing can be used to show the presence of bugs, but never to show their absence!” [2] testing still helps to validate at least parts of the functionality of a program. Tests are for example useful to determine if newly written code behaves as expected and can also be used to detect if recent changes break previously working code (regression errors).

User studies indicate that about 10-15% of the development time is spend waiting for tests to finish executing and fixing regression errors, and simply increasing the frequency with which the tests are performed reduces this wasted time by 31-81% [7]. This also means that the additional time it takes to execute tests more frequently is less than the time saved by fixing regression errors early.

Since testing provides these benefits it has become a key feature of many current software engineering paradigms like for example extreme programing (XP) [6]. Together with the other features of these development methods testing helps to speed up the development and increase the quality of the software being developed. Additionally for example in XP the time that is spent writing a test will be regained. As Wells[8] puts it: “...during the life of a project an automated test can save you a hundred times the cost to create it... ”. This is done by guarding against bugs and regression errors. Since this guard is in place ”refactoring” and ”frequent integration” of new code become possible.

This brings us the problem that, while testing clearly has its benefits, it is still seen as optional or too time consuming. Often tests they are not executed as often as they should be which indicates that the use of unit tests is underestimated. In their case studies Pressman and Ernst found that a 31-82% reduction of waisted time can be achieved by simply running the same tests two to five times as often[6]. Especially under pressure unit testing is often deemed to time consuming.

Not testing properly will in the long run slow down the development process since all the previously discussed benefits of testing are missing. So there might be no time for testing in the future as well.

Breaking out of this circle can require an outside influence [1]. While certainly some developers have no problem writing tests no matter the situation a little encouragement might be needed for others. In this bachelor thesis we attempt to provide another one of those outside influences by making testing easier and faster.

2
A Closer Look at Testing

In this chapter we will take a look at different testing paradigms in order to specify how these terms are used throughout this thesis. During the next chapters these terms will then be applied to better describe how current programming environments support these paradigms and where precisely this support is lacking. It is important to note that these paradigms are not mutually exclusive.

2.1 Unit Testing

Unit testing follows the paradigm of isolating the smallest inseparable parts of a program and testing them independently of each other. Each of these tests is done as low level as possible. System wide tests are generally not a part of unit testing. This means that the scope of each test is very small and consequently many tests are needed to cover the whole implementation. By making unit tests one ensures that the tested parts behave as expected. In Smalltalk these smallest units are often methods since the language heavily relies on sending messages (what roughly translates to a method invocation in other languages) between objects.

To start testing a method an instance of the class under test is created and brought into a state where a set of preconditions is met. An example for preconditions would be the requirement for the objects instance variables to have certain values. From this initial state in a deterministic environment a specific outcome can be expected after the method is executed. The outcome is a success if all of the previously defined postconditions are fulfilled.

To illustrate this let us examine how unit tests for a simple stack might work. The stack has a maximum capacity that is set once the stack is created. Further the stack provides a method named ”push(x)” which adds an element ”x” on top of the stack and a method named ”pop()” which removes and returns the element that was added last to the stack. Now let us examine how to make a unit test for this stack that checks if the last element that was added is returned by ”pop()” and not some other object.

The preconditions need to make sure that the stack is initialized with a capacity bigger than zero. If no stack is created or if its size is smaller than one then the stack can return nothing even if everything is correctly implemented. After this precondition the test will now call ”push(x)” and then ”pop()”.

Now we need to make sure that operations work as expected or in other word we need to check our postconditions. In this case if the object returned by ”pop()” is the one that we pushed to the stack and that the stack is empty again after the element was popped. Below is an example in SmallTalk of how such a test could look like. To check if the stack is empty again after calling pop() a separate test could be written.

```
testPopReturnedElement
| stack |

stack:=Stack withCapacity: 2
stack push: 5

self assert: (stack pop=5)
```

For the purpose of this paper two features of unit testing are important to keep in mind. The first one is that unit testing takes place at a method level and the second one is that the narrow scope of unit tests requires many tests to cover multiple methods. Further, due to how it is recommended to keep each test as small as possible even a single method is likely to require multiple corresponding tests since every path in the decision tree of the method should be covered.

2.2 Test-driven Development

In Test-driven Development the tests for an implementation are written before the implementation itself. Afterwards the implementation is written and improved until those tests are satisfied. After this more tests are added to test additional features. This process is repeated until the implementation is sufficiently working. Test-driven development thus requires the developers to regularly determine the requirements for a feature. Below are some basic rules of Test-Driven Development as laid out by Martin[5].

> *First Law: You may not write production code until you have written a failing unit test.*
>
> *Second Law: You many not write a more of a unit test than is sufficient to fail, and not compiling is failing.*
>
> *Third Law: You may not write more production code than is sufficient to pass the currently failing test.*

While it might seem tedious this repeated careful planning is the biggest advantage of this approach. Applications developed with Test-driven Development have to be very thought-out and every requirement has to be clear since those are required to begin writing the production code.

As shown in section 2.1 it is fair to look at testing on a method level when discussing unit testing. Since in Test-driven Development the methods may not have been written when the tests are created the method declarations and the method locations in the class hierarchy can not always be used to make certain assumptions that could be used to facilitate creating new tests. For example before a method in the production code is written it needs to have a failing tests. Only for subsequent tests to the same method can the method declaration and the method location in the class hierarchy be found in the production code.

Test-driven Development bears mentioning because it is one of the major paradigms concerned with testing and because in the context of this paper it has to be treated a bit differently. Namely since it is not possible to always get the method declaration and location in the class hierarchy to help generating tests when doing Test-driven Development.

2.3 Blackbox and Whitebox Testing

Whitebox and blackbox testing refer to how closely tests are written to suit a method. In whitebox testing the tester is completely aware of the method and its inner workings when writing a test for it. This means that an adequate amount of unit tests done with a whitebox approach will cover each important path in the decision tree of a method. Especially interesting are worst case scenarios that can be found by examining the code and method invocations that result in as much executed lines as possible.

Contrary in blackbox testing the test writer does not know the inner workings of the method that is being tested. Ghezzi et al. describe that during blackbox testing ”[t]est sets are developed and their results evaluated solely on the basis of the specifications” [3]. This means that the tests have to be made so that they return the right value for all tested input values. Since it might be unfeasible to test all values it is necessary to choose the critical input values where an error might occur.

3
The Problem

As shown in chapter 1 some current software development paradigms include unit testing but as important as unit testing is not all current development environments are optimized for it. For example, the search for unit tests to a specific method is often lacking or not present at all.

In this work we assume that important features for unit test friendly environments are a quick way to see if a method is tested, an automated test search, the ability to view tests and methods side by side and an option to easily create new tests. These criteria are based on the definitions of the various testing paradigms previously described in chapter 2.

Namely unit testing can be supported by letting the developer see if a method is covered by at least one test, the search for all tests corresponding to a specific method and the ability to create new tests as fast as possible. In whitebox testing it is convenient to see the method under test and the test at the same time without switching between the two. With this it becomes easier to write tests that together cover each path in the decision tree of a method.

Since not many environments were created with specifically unit testing in mind there are often some of these features missing. Concrete examples of environments lacking these specific features will be pointed out in chapter 4. To compensate for these missing features the user is required to manually execute many tasks like navigating back and forth between method and test and creating new test classes and packages. This repeatedly breaks the programmers flow and in effect discourages them from writing tests. In the following we will take a closer look at the tasks involved in different goals arising from unit testing which will help to understand the problem of those missing functionalities and exactly at which point they are needed.

3.1 Is a method tested?

In unit testing it is important that each method is tested. As discussed in section 2.1 a method with no tests is unsafe and might even have to be tested manually which is quite slow. Thus a very important feature in unit testing is the ability to check if a method has at least one test. This has to be as easy to see and as uninterruptive as possible.

If this feature is not implemented users have to switch back and forth between a class and all its test classes to check manually. If the test classes are not known to them they have to find them first. Both of those tasks can take a lot of effort. Luckily code coverage tools also provide this functionality. They allow the user to run test suits and then see which part of the implementation were executed. This will show if a method never got called and thus is untested.

3.2 Showing all tests to a method

Unit testing requires to look up a method and all the tests that have been written for it. Just checking if a method is tested at all is not enough, the developers might want to see if the method is tested sufficiently. To ensure this multiple test methods are required. Also if at least one test for a method could be found then the method is not untested. This functionality can thus easily substitute the one described in section 3.1. The previous ability to see if a method is untested will thus no longer be discussed separately in this paper.

Similar to showing if a method is tested this helps the user to decide if it is necessary to write a new test. Further this functionality helps the user to see if a certain test case has already been created and thus is not need again. A user could also study how a method works by looking at the corresponding tests.

A lack of this test search function will require great effort on part of the user to keep track of test classes and test methods. If the user decides to add a new test then it is necessary to take a short look at every test to determine if a similar test has not been written. In this case the tester needs to have quick access to all tests of this method. A lack of this feature makes unit testing in anything but the smallest applications and test suits very tedious and new testers will have trouble getting an overview of the existing test suit.

Code coverage tools can not fully substitute an automated test search since it relies strongly on the relation between tests and method. To be useful the code coverage tool would have to save for each method from which test it was called during the execution of a test suit. Also coverage is not necessary a good metric to determine if a tests actually tests a method. Nested method calls might yield a large number of false positives.

3.3 Creating new tests

As discussed previously it should be as simple as possible to check if a method is not sufficiently tested. Almost as a direct consequence the user has to create these new tests if this is the case. Especially at the start of a project many new tests will have to be created. So it is safe to say that in unit testing another reoccurring user goal is the creation of new tests for methods with no or too few tests.

While creating a new test the user first has to decide in which package and class the new test will be added and how the test will be called. In many cases the location of this new test has already been decided on since other tests corresponding to the same method are already there. In this case a quick option to specify an existing test class should be offered.

If there are no test classes and packages or if the existing ones do not fit the new test then the user has to create a new test class. In this case the test package, test class and test names are often derived from the original package, class and method names. A slight drawback of these default name derivations is that the method that is being tested has to be defined previously. Especially during Test-driven Development the programming environment can not always make such name derivations.

As shown in the last two paragraphs it does not matter if the test class and package already exist or if they need to be added. Creating and placing new tests can be facilitated either by letting the user quickly specify existing test classes or by making creating new test classes easier. Another helpful feature is the ability to let the user create these packages, classes and tests together instead of individually.

If the default names are missing the developer has to enter similarly structured names for packages, classes and test over and over. In case that test packages, classes and methods can not be created together the user will have to click around in the environment multiple times to start the creation of each of these.

While it helps to have these aids a lack of them seems not as bad as a lack of the test search functionality described previously in section 3.2.

3.4 Methods and Tests Side by Side

In whitebox testing it is required to know a method's inner workings. It is expected that the programmer who writes the test is trying to test every line of code. Thus it is important to provide the user with information on how the method works. The possibly simplest way is to let the users see the whole method in question at the same time as they are writing a test for it.

In Black-box Testing this feature can be counterproductive by distracting the users from the method specifications. The additional information can thus spoil blackbox testing and even if completely ignored still takes up space on the screen. Blackbox testing thus requires that this feature can be turned off.

If during whitebox testing the environment does not provide information about a method the users have to gather it themselves. This can be done by taking notes, switching back and forth between method and tests or opening an additional window to show both at the same time with every new test.

This can be very tedious and take a lot of time. Making it unnecessary to switch between gathering information about a method, writing tests for it and looking up the same information again can save valuable time and effort.

4 Related Work

In this section current programming environments, namely Eclipse and Pharo, are analyzed in how unit test friendly they are. Special focus will be placed on the features described in chapter 3. These features are: viewing test and method side by side, search all tests to a specified method and facilitating the creation of new tests. The two environments will be compared and possible places for improvement will be discussed.

4.1 Unit Testing in Eclipse

We will start by taking a closer look at Eclipse since its use is very widespread and it has many parallels to other programming environments like for example Visual Studio. In Figure 4.1 you see a Java project opened in Eclipse with a package and two classes.

4.1.1 Methods and Tests Side by Side

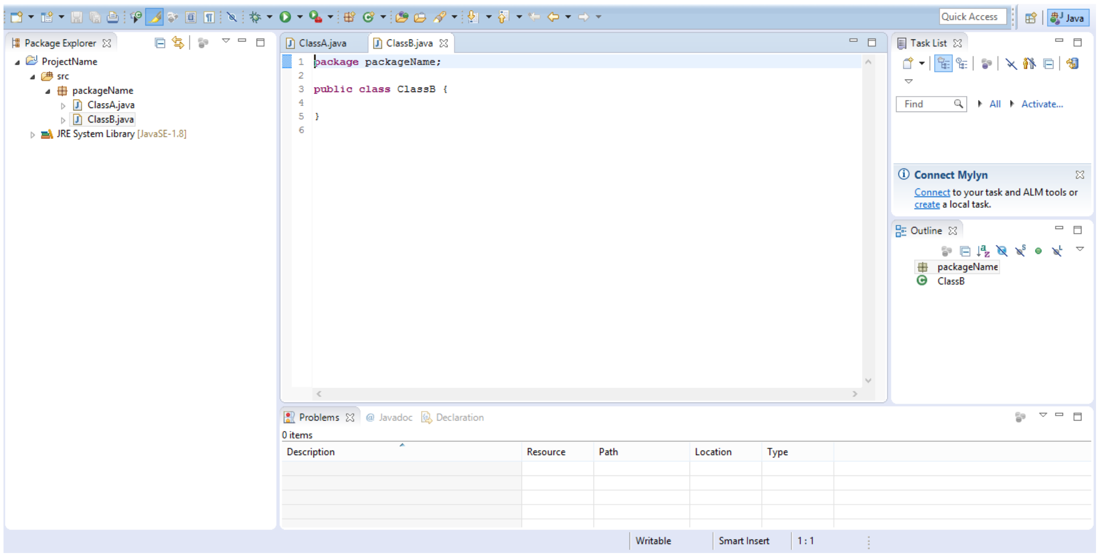
Let us start with the discussion of how in Eclipse the users can see the tests that they are currently writing and the methods that are being tested at the same time. In Figure 4.1 on the left hand side of the window you can see the Package Explorer. In this file hierarchy you can see every project, package and class in the current workspace.

Classes that are opened through the Package Explorer will be shown in the middle of the screen and a new tab on top of the code editor will appear. In Figure 4.1 there are two tabs ”ClassA” and ”ClassB”. The ”ClassB” tab is currently selected and thus the content of ”ClassB” is displayed in the code editor. Unless you close these tabs all the classes you opened will be quickly available through their tabs.

A big advantage of this system is that users can create favorites by not closing important tabs. Through this they can switch back and forth between methods and tests. A slight drawback is that the user has to close unneeded tabs from time to time. In most cases the use of these tabs will be smooth enough to allow the programmer to switch uninterrupted between tests and methods via their classes.

If for some reason this does not suffice then it is also possible to split the code editor multiple times either horizontally or vertically like in Figure 4.2. Since each code panel is smaller than the original one this splitting can only be done a few times. After this the panels get too small to be useful.

As shown Eclipse provides multiple ways to make the tested method easily accessible during testing.



```
package packageName;

public class ClassB {

}
```

Figure 4.1: Eclipse's user interface

4.1.2 Test Search

Another feature we discussed in the previous chapter was the ability to find all tests to a certain method. Eclipse provides a good option to search for tests. By pressing "Search" from the top menu bar followed by "Referring Tests..." it is possible to find all tests that reference the currently selected method.

Sadly this feature is very badly documented. The requirements for a test to qualify as a test to a certain method seem to include that the test class extends "TestCase" and that the test name starts with "test". "@Test" annotations are ignored by this function. It seems like this test search does not yet support JUnit 4 standards.

4.1.3 Creating new Tests

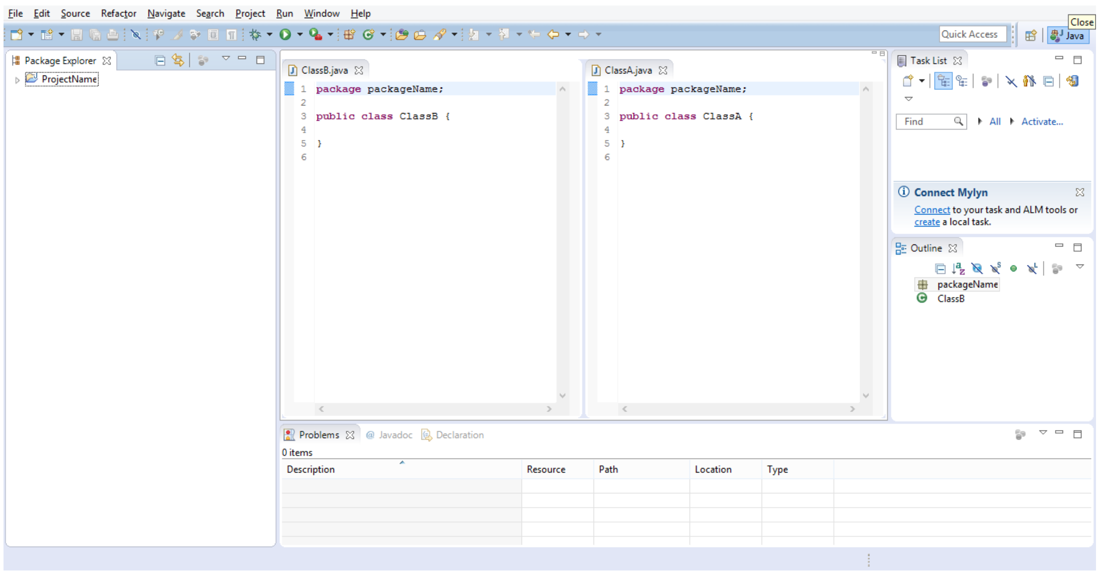
The next feature in Eclipse we now discuss is how the creation of new tests is facilitated. Eclipse has the option to add a new test class by right clicking on the class that should be tested, selecting "New" from the list that is shown and then clicking "JUnit Test Case". There are other ways to do this but by making sure that the right class is selected in the Package Explorer Eclipse is able to support the user better.

In the newly open wizard a default name is already created and the selected class is put into the "Class under test" field, see Figure 4.3. As default package the package of the class under test is used but it can be changed. By clicking on the "Next" button it is possible to select methods of the class under test and create test stubs for every selected method, see Figure 4.4.

On one hand this wizard is very good for creating new test classes but on the other hand Eclipse helps neither to add new test methods in an already existing test class nor to create more than one test per method.

4.2 Unit Testing in Pharo

Now let us take a look how the Pharo IDE and its system browser Nautilus compare to Eclipse. It is worth mentioning that the Pharo IDE is quite different from Eclipse in that the placement of its visual elements is less rigid. The users are able to customize the appearance of the environment very quickly and adapt it to



ClassB.java:

```
package packageName;

public class ClassB {

}
```

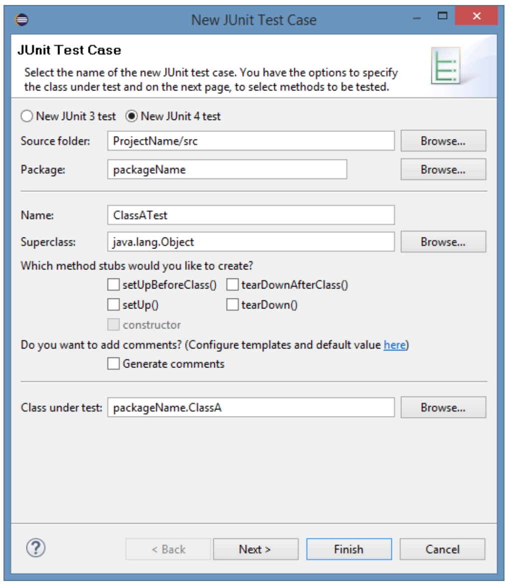
ClassA.java:

```
package packageName;

public class ClassA {

}
```

Figure 4.2: Split code panel

their wishes. On the other hand Pharo is less wide spread and not as well maintained in many ways as Eclipse.

Like in section 4.1 the discussed features are viewing test and method side by side, searching all tests to a specified method and facilitating the creation of new tests.


New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

New JUnit 3 test [✓ New JUnit 4 test]

Source folder:	ProjectName/src	Browse...
Package:	packageName	Browse...
Name:	ClassATest	
Superclass:	java.lang.Object	Browse...

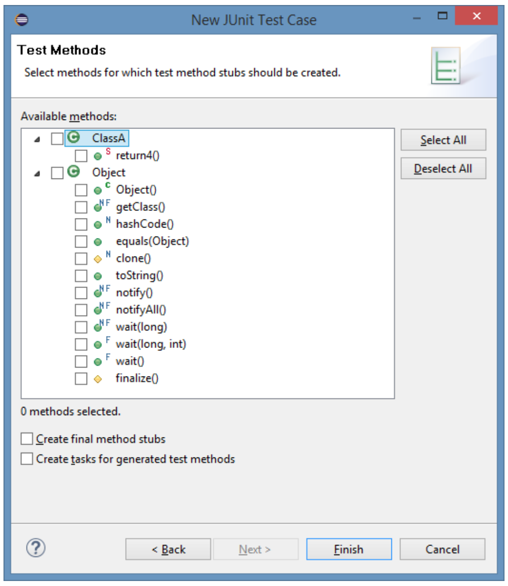
Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value here)
☐ Generate comments

Class under test: packageName.ClassA Browse...

Figure 4.3: Test class wizard


New JUnit Test Case

Test Methods

Select methods for which test method stubs should be created.

Available methods:

- ☐ ClassA
  - ☐ return4()
- ☐ Object
  - ☐ Object()
  - ☐ getClass()
  - ☐ hashCode()
  - ☐ equals(Object)
  - ☐ clone()
  - ☐ toString()
  - ☐ notify()
  - ☐ notifyAll()
  - ☐ wait(long)
  - ☐ wait(long, int)
  - ☐ wait()
  - ☐ finalize()

0 methods selected.

☐ Create final method stubs
☐ Create tasks for generated test methods

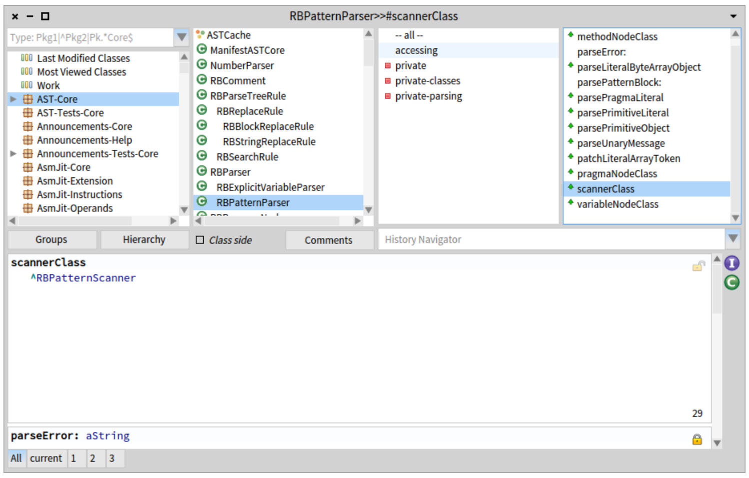
Figure 4.4: Select methods to test

4.2.1 Methods and Tests Side by Side

As with Eclipse the first functionality we will look at is the ability to view tests and methods side by side. The Pharo IDE provides a very customizable environment and it is almost necessary for a user to arrange all windows that are created in a way that is comfortable. It is possible to create multiple Nautilus windows and arrange them so that in one the method and in the other the test for this method is shown. Through this a very similar effect to Eclipse's split code editor is achieved.

The free placement of those Nautilus windows gives the user more freedom to customize the environment but several visual elements will be duplicated which reduces the available space on the screen to arrange these. An example for such a duplicated element is the file hierarchy that takes up the top half of each Nautilus window like for example in Figure 4.5.

A possibility that does not have this drawback is to lock a method or class and then select others inside the same Nautilus window. It is possible to select which of the locked elements is shown in the code panel by using the "All", "Current" and number buttons shown at the bottom of Figure 4.5. Through these buttons the users can switch between seeing all locked methods and classes at the same time or each individually. This is very similar to the tabs of opened classes that Eclipse has. Although with many locked classes and methods it is hard to remember which number belongs to what.



```
scannerClass
    ^RBPatternScanner
```

```
parseError: aString
```

Figure 4.5: Nautilus window with active locks

Apart from this Nautilus also has the History Navigator which allows fast access to all recently viewed methods and classes. The History Navigator can be seen in the middle right in Figure 4.5. It can be used to switch between method and test in a way that is comparable to Eclipse's tabs of recently opened classes. The drawback here is that only a certain amount of those recently accessed classes and methods is stored and if the user looks at different methods and classes it is quickly necessary to reopen the previous tests and methods to put them back in the History Navigator.

To conclude in the Pharo IDE are various things a user can do to view multiple pieces of code at the same time. Compared to Eclipse there are more ways to see things side by side or to switch between them but each of these ways has a slight drawback.

4.2.2 Test Search

The next functionality that will be discussed is how Pharo finds existing tests to a specified method. Like Eclipse Nautilus has some sort of test search implemented. Like using Eclipse's test search only tests with a very specific name are found. Additionally they have to be placed in classes with an equally specific name. Examples of tests corresponding to certain methods can be found in Table 4.1.

Class name	*Method name*	*Test class name*	*Test name*
AClassName	aMethodName	AClassNameTest	testAMethodName
Nautilus	selectedClass	NautilusTest	testSelectedClass
RxMatcher	matches:	RxMatcherTest	testMatches
Stack	push:	StackTest	testPush
ProtoObject	ifNil:ifNotNil:	ProtoObjectTest	testIfNilIfNotNil

Table 4.1: Methods and corresponding tests from Nautilus

Namely test classes and test method have to contain the full name of the original class or respectively method. Additionally the name of the test class has to have the suffix ”Test” and the name of the test method has to have the prefix ”test”. Also tests have no parameters and thus lack all colons in their name. If anything more is part of the test name then it will not be found. The test classes also have to inherit from TestCase in order to be found by the test search.

Both the restrictions on class names and method names are quite restrictive. From a test class name the class under test can be quite reliably inferred[4]. On the other hand Marschall also states that the method name and the corresponding test name for it are less directly related. These restrictive name criteria also lead to the problem that at most only one test per method will be found since others with a slightly different name will not fulfill these naming criteria.

This test search function is not directly available to the developers. It is used by Nautilus in the file hierarchy to add a button to the left of each method where a test has been found. This button can be pressed to execute the test and show if this corresponding test was successful or not.

The test search is better in Eclipse since it is not only less restrictive with its name-based criterion but also checks if the test calls the method under test. Eclipse's test search is also able to find more than one test to a method. On the other hand in Nautilus the test search is better integrated in the environment. The additional button to execute a corresponding test conveniently allows running the test without navigating to it.

4.2.3 Creating new tests

In Nautilus with a right click on a method the option ”Generate test” is given. The new test will automatically be added to a certain test class in a specific package (both of which will be created if needed). More precisely the test package is named like the original package with the suffix ”-Tests”. The test class and method will be named like shown in Table 4.1. All the tests created through this will thus be found by Nautilus's test search function. ”Generate test and jump” works very similar but also selects and shows the newly created test so that the user can start implementing.

Using these predefined package and class names is faster since the user does not need to enter them. Another advantage is that tests created in this manner confirm to the naming standards imposed by Nautilus. Sadly this robs the user of the ability to specify different package and class names. Also only one test per method can be created in this manner. All additional tests for the same method will simply overwrite the previous test.

Eclipse's and Nautilus's way of adding new tests are very comparable. A case could be made for both versions of this feature. Eclipse's version is more customizable but only supports the user when creating a new test class while Nautilus's version is faster and supports the user with each new test but is less flexible. An easy way to create multiple tests for one method is not provided by neither Nautilus nor Eclipse.

5
The TestView Plugin

Having discussed some important features of a unit test friendly environment in chapter 3 and where their implementation is lacking in chapter 4 we will now try to present a solution to this problem: the TestView Plugin.

Since the lack of unit test supporting features can not be corrected for all environments at once we chose to extend Nautilus's functionality. The reasoning behind this was that the inability to view tests and methods side by side in a single Nautilus window was perceived by us as the gravest of these issues and that Eclipse already has a big community which provides many plug-ins.

In this section we will comment on the same three functionalities discussed in the last chapters: viewing test and method side by side, finding tests to certain methods and creating new tests. Special focus will be put on improvements compared to Eclipse and the Pharo IDE.

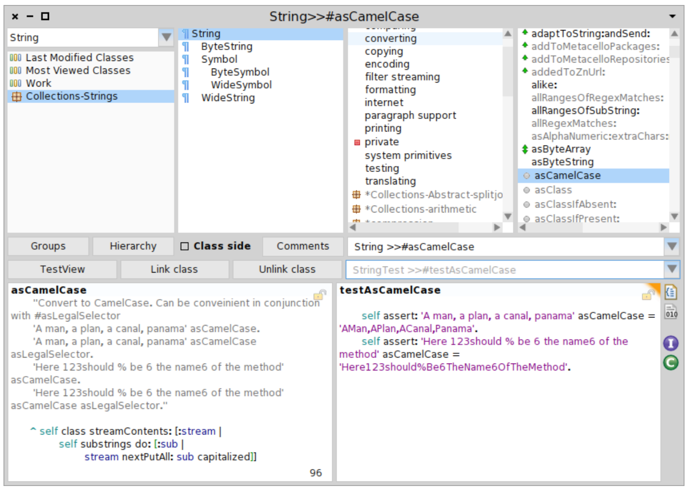
5.1 Methods and Tests Side by Side

As stated one of the drawbacks concerning unit testing with Nautilus is that it is hard to view tests and methods at the same time. Whitebox testing becomes cumbersome through this. Seeing both method and test together should be quick, convenient and not introduce too much redundant user interface elements that might clutter the environment.

Following this the TestView Plugin allows to split the code editor panel of a Nautilus window vertically into two parts, the result can be seen in Figure 5.1. With this approach the need to open two Nautilus windows is eliminated. The file hierarchy for example will not be displayed a second time and less screen space is occupied.

It also becomes unnecessary to use Nautilus's locking feature or History Navigator to be able to switch fast between methods and tests. Thus it also is not required anymore to manage the locked or recently viewed classes and methods in order to quickly navigate back and forth.

Unlike in Eclipse the user has no absolute control over this additional code editor. Only methods of interest are shown there and the user does not have to manage what should be displayed. This is possible due to the fact that some assumptions can be made about what code the user wants to see if this code editor is only used during unit testing. The left code editor will be displaying the in the file hierarchy selected method and the right code editor will show the corresponding tests.



```
asCamelCase
	"Convert to CamelCase. Can be conveinient in conjunction
with #asLegalSelector
	'A man, a plan, a canal, panama' asCamelCase.
	'A man, a plan, a canal, panama' asCamelCase
asLegalSelector.
	'Here 123should % be 6 the name6 of the method'
asCamelCase.
	'Here 123should % be 6 the name6 of the method'
asCamelCase asLegalSelector."

	^ self class streamContents: [:stream |
		self substrings do: [:sub |
			stream nextPutAll: sub capitalized]]
```

```
testAsCamelCase

	self assert: 'A man, a plan, a canal, panama' asCamelCase =
'AMan,APlan,ACanal,Panama'.
	self assert: 'Here 123should % be 6 the name6 of the
method' asCamelCase =
'Here123should%Be6TheName6OfTheMethod'.
```

Figure 5.1: Nautilus window with two code editors

Which test is currently shown can be changed through the drop list that can be seen in Figure 5.1 just below of the Nautilus History Navigator.

This approach combines the convenience of Eclipse's ability to show code side by side with limited but more focused contend in the right code editor. This results in an easy to use support for whitebox testing. Since it is not possible to view multiple methods in one Nautilus window at the same time this is a notable improvement.

5.2 Test Search

As a direct consequence of the heavily selected content of the second code editor it became necessary to use a test search. Without this it would be impossible to show a relevant selection. As discussed in subsection 4.2.2 the test search that Nautilus provides is very restrictive and only finds at most one test per method. In order to find better results the TestView Plugin uses its own test search.

In this section I will provide a detailed explanation how corresponding tests to a certain method are found. The plugin uses a hierarchical search with two stages. First the TestView Plugin searches all test classes to the currently selected class and then inside of these test classes all test methods to the currently selected method.

This search is performed every time a different method is selected. This makes it possible that the additional code panel introduced by the plugin always shows a relevant selection.

5.2.1 Corresponding Test Classes

A purely name-based search works quite well to find test classes corresponding to a certain class[4]. This is already used in the Nautilus test search and a similar name based search is also part of the test search in the TestView Plugin.

In Table 5.1 examples of which corresponding test classes that the TestView Plugin finds are shown. All classes with names like in the column of possible class names are considered corresponding test classes if they also inherit from TestCase.

The name-based criterion requires the name of the class in question to contain both ”test” and the name of the selected class. The substring search is not case sensitive and the matches for ”test” and for the name

Original Class Name	*Possible Class Names*	*Not Test Classes*
String	StringTest, stringtest, TestString	String, Test, StrinTgEST
Contest	ContestTest, contesttest	Contest

Table 5.1: TestView name criterion for test classes

of the selected class can not overlap. This is why ”Contest” is not considered to be a test class to a class that is also named ”Contest” even though it contains ”test” as well as the full name of the class under test.

Since the results of this search for corresponding classes are used as base for the search for corresponding tests it is important that it works as well as possible. If this search for corresponding test classes is inadequate the developers can use the ”Link Class” and ”Unlink Class” buttons provided by the TestView Plugin to add and remove test classes from the results.

5.2.2 Corresponding Test Methods

After some possible test classes have been identified using the criteria described in subsection 5.2.1 the results will be searched for corresponding test methods to the currently selected method.

Similar as the name criterion for test classes described in subsection 5.2.1 the tests have to contain the full name of the selected method(without colons) as well as ”test”. Examples can be found in Table 5.2.

Original method name	*Test names*	*Not test names*
doStuff	doStuffTest, testDoStuff, TestdosTuff	doStuff, doTestStuff, testStuff
do:on:	testDoOn, testDoOn2, doOnTest	testDo:On:, do:testOn:
attest	attestTest, testAttest	attest

Table 5.2: Test naming criterion

The second criterion is that the test contains a method invocation with the same selector as the selected method. If either of those criteria is fulfilled the method is considered a test to the selected method. If just one or both of those criteria is met is used to order the found tests in the drop list that can be seen in Figure 5.1 right below of the Nautilus History Navigator.

This new search should yield better results than Nautilus' search since it allows multiple test classes and methods to a particular method. It works very similar to the test search provided by Eclipse and like Nautilus's search is used directly by the IDE and does not have to be called manually.

5.3 Creating new tests

The purpose of this functionality is to prevent the users from having to repetitively enter similarly structured class and package names while still letting them retain the freedom to choose each name if needed.

Every time developers start to write a new unit test they have to determine to which test class the new test belongs. We can split the creation of new tests into two basic use cases: adding the new test in a new test class or adding the new test in a test class where there already are tests corresponding to the selected method.

5.3.1 Adding a new Test Class

First let us talk about what can be done to facilitate the creation of a new test that does not yet have a fitting test class. The easiest way to do this is to let the users write the new test and later determine where this test will be put. With this the user is encouraged to immediately start writing a test as soon as a method is created.

When as the test is saved the plugin will ask the user how the new test class should be named and in which package this class should be put. Here default values based on the class name and the package name of the method under test are given which in many cases might already be sufficient since test class names and test package names can often be derived from the original class and package names [4].

For example if the method under test is contained in a class called “Queue” within a package “Collections” then the proposed names for the test class and the test package will be “QueueTest” and “Collections-Tests”. The new test class will also automatically subclass TestCase.

These names and the inheritance from TestCase is also what Nautilus’s test search expects and thus this does not break existing conventions. Even though default names are provided the user can still ignore them and specify different ones.

5.3.2 Adding a test to an existing Test Class

The second use case is that a developer wants to add a test to a method that already has tests. In this case the corresponding test classes should have already been found by the plugin. The users just have to select any test that is in the test class where the new test should be added. No test package has to be specified.

One potential problem is how this choice between making new test classes and test packages and using existing ones is communicated to the user. The users have to make their intentions clear to the plugin by selecting a specific item from the list of found tests.

When the user saves a test while having selected the first item in the list the plugin will always ask the user to give names for the test class and the test package. When the user has any other item selected then the test is saved in the test class that was selected. This might create some confusion but on the other hand with only two clicks(open the list and selecting target class) the user can specify where the new test should be saved.

Now let us take a look at how this functionality compares to Eclipse and Nautilus. Improvements compared to Eclipse are that the user can add a single new test in an already existing test class. Eclipse has an advantage when creating a new test class and filling it with multiple new tests. Eclipse supports the user only at the start of writing a new test class while the TVPlugin keeps supporting the user during the addition of any new test.

In Nautilus the user can extremely quickly create new tests in fixed test packages and test classes but only one test per method can be created that way. If the “Generate test” or the “Generate test and jump” option is pressed multiple times then the previous test is overwritten. The TVPlugin helps the programmers to add multiple tests to the same method.

6
The Validation

In which you show how well the solution works.

7 Conclusion and Future Work

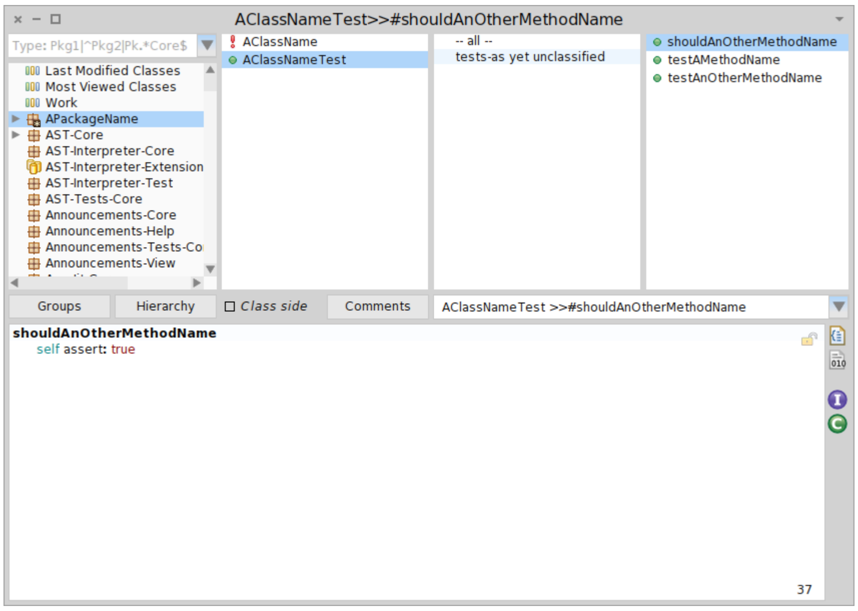
In which we step back, have a critical look at the entire work, then conclude, and learn what lays beyond this thesis.

8
Anleitung zu wissenschaftlichen Arbeiten

8.1 User Guide

8.1.1 What's the TestView Plugin?

Nautilus is the default system browser in the current Pharo version. Figure 8.1 shows how a Nautilus window normally looks like.

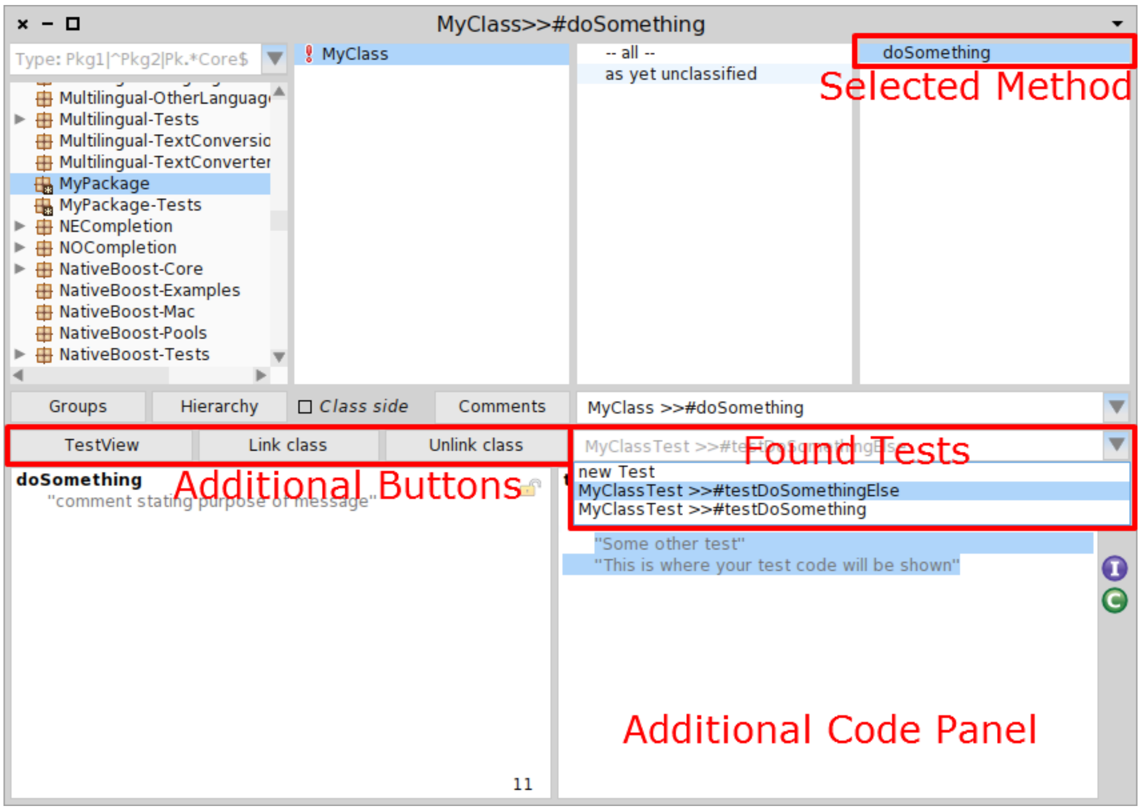


```
shouldAnOtherMethodName
    self assert: true
```

Figure 8.1: A Nautilus Window

The TestView Plugin (or TVPlugin) is a Nautilus plugin to facilitate unit testing. It provides quick ways to add new test methods and classes, find existing tests and view tests and methods at the same time in a single Nautilus window.

Let us take a look at the TVPlugin in Figure 8.2 and discuss the features that it provides.



```
doSomething
    "comment stating purpose of message"
```

```
    "Some other test"
    "This is where your test code will be shown"
```

Figure 8.2: Nautilus with the TVPlugin

The **“TestView”** button toggles the additional code panel to the right of the original code panel on and off. All features described here require that the plugin is turned on.

The **selected method** is a central part of the plugin. All funtions the plugin provides are relative to what you have selected in the Nautilus window. Whenever you select a Method in the Nautilus class hierarchy the plugin will automatically search for corresponding tests.

In the **found tests droplist** every test corresponding to the selected method is shown. The first element in this list is special and will always be there independently from which method you have selected in the Nautilus window. Click this element to create a new test in a possibly new test class. How to do this is explained in detail in subsection 8.1.3. The remaining items in the list are all existing tests that are corresponding to the method you selected in the Nautilus window. By clicking on one of these the right code panel will display the selected test.

The **additional code panel** to the right of the original code panel will always show the test that has been selected in the found tests droplist. With this it becomes possible to look at the implemented method and at the tests for it inside of the same Nautilus window.

You can use the **“Link Class”** and **“Unlink Class” buttons** if the found tests for the method you selected in the Nautilus window are incomplete or show methods that are not tests to what you selected. You can use both these buttons to influence the automated test search that gets performed whenever you select a different method in the Nautilus class hierarchy.

With the “Link Class” button you can specify a test class that is not found by the automated test search. The plugin will then redo the test search and include the newly linked class as a possible source for tests on every search to the class of the selected method.

With the “Unlink Class” button you can exclude classes from being searched for tests. Like with the “Link Class” button this exclusion will only count for the class of the method that you have currently selected in the Nautilus class hierarchy. Detailed instructions on how to use these functionalities are found in subsection 8.1.5 and subsection 8.1.6.

8.1.2 Installation and activation

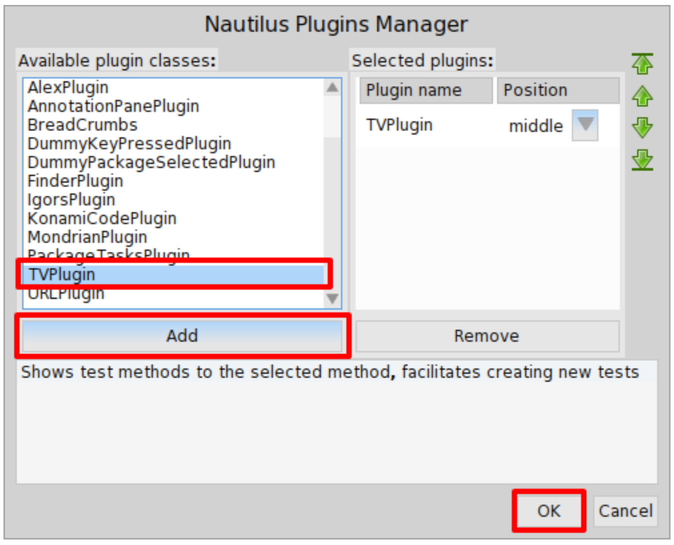
To install and activate the TVPlugin follow the steps listed bellow:

1. To download the necessary packages simply execute the following lines in a Pharo workspace

   ```
   Gofer new
   url: 'http://smalltalkhub.com/mc/DominicSina/TestView/main';
   package: 'ConfigurationOfTestView';
   load.
   (Smalltalk at: #ConfigurationOfTestView) loadDevelopment.
   NautilusPluginManager new openInWorld
   ```

   Once this is finished the Nautilus Plugins Manager will open.

2. In the window shown in Figure 8.3 you click on ”TVPlugin” under ”Available plugin classes” and then press on the ”Add” button. Click ”Ok” to confirm.

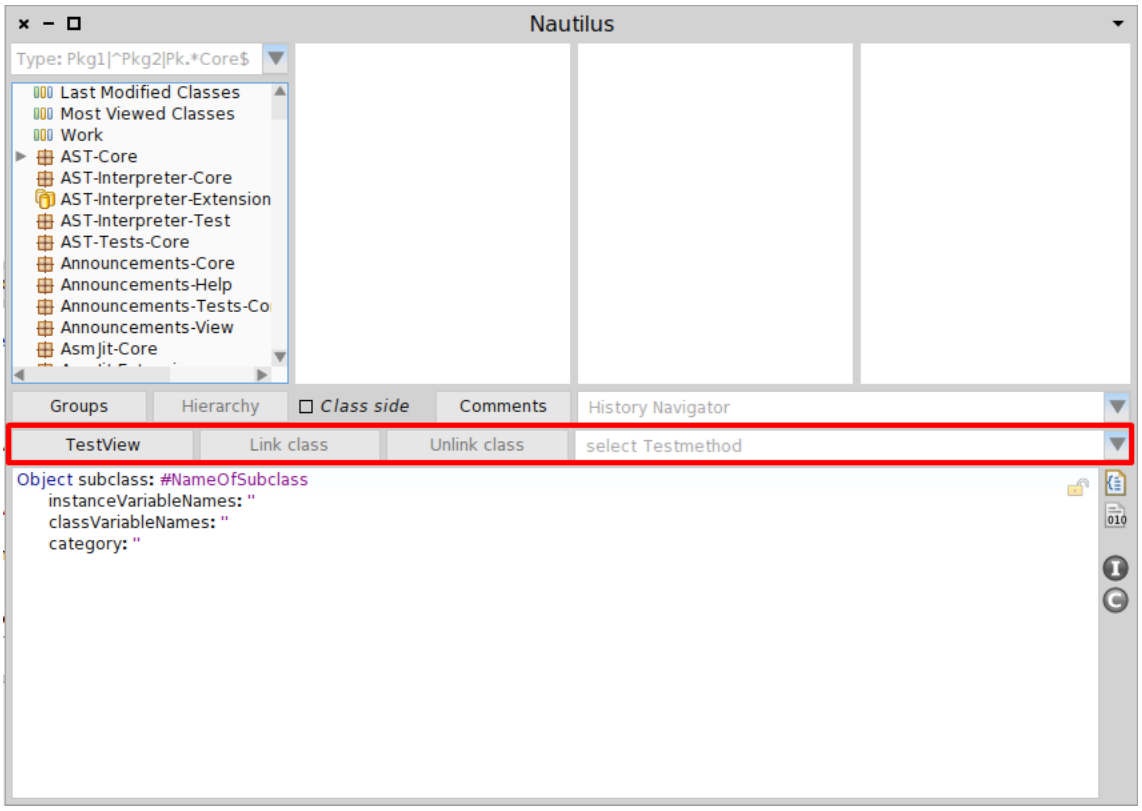


Plugin name	Position
TVPlugin	middle

Shows test methods to the selected method, facilitates creating new tests

Figure 8.3: The Nautilus Plugin Manager

3. When you open a Nautilus window from now it should look like in Figure 8.4. To verify if the plugin is activated check if the highlighted row is displayed in the position that you selected. The plugin will be shown there until you remove it again using the Nautilus Plugins Manager.



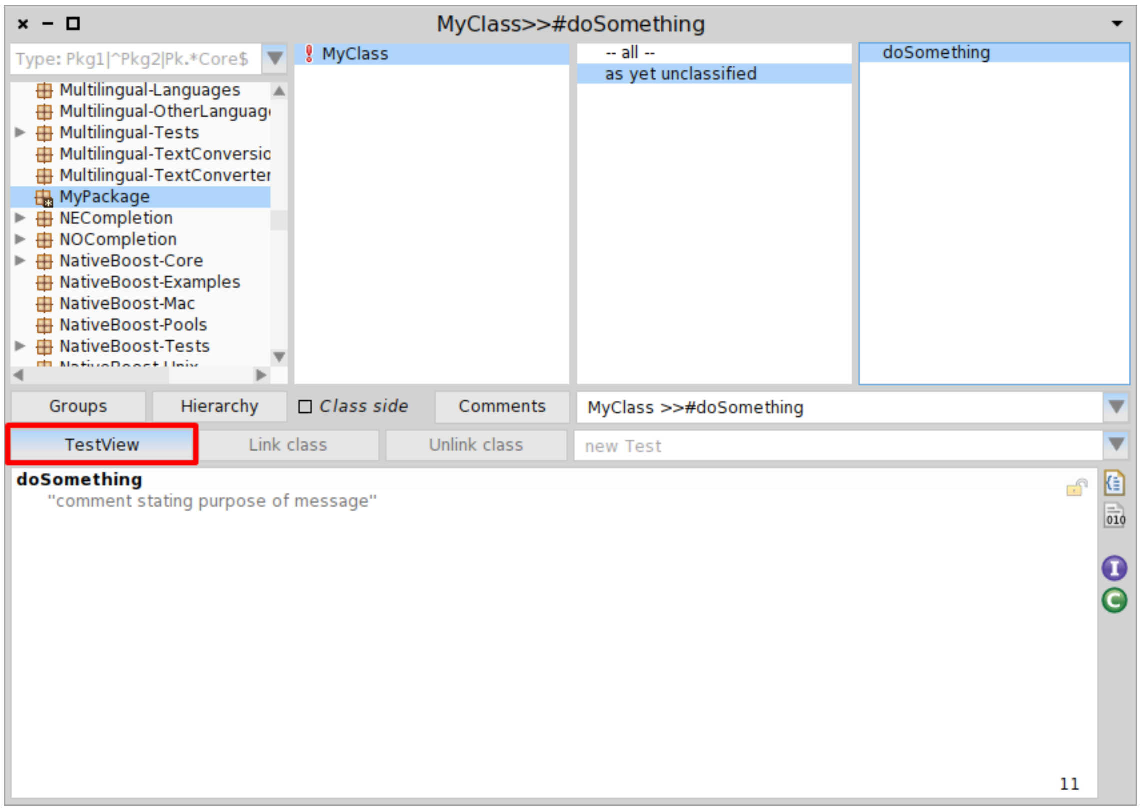
```
Object subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
    category: ''
```

Figure 8.4: TestView Plugin once it is activated

8.1.3 Adding a new test inside a new test class

So now that the plugin is set up after following the steps in subsection 8.1.2 let us add a new test to a method. In this example it is assumed that you have a method to test named “doSomething” inside of a class named “MyClass” and a package called “MyPackage”.

1. Turn the TVPlugin on by clicking on the “TestView” button highlighted in Figure 8.5. Once this is done a second code panel will appear besides the original code panel that shows a new test to your selected method.

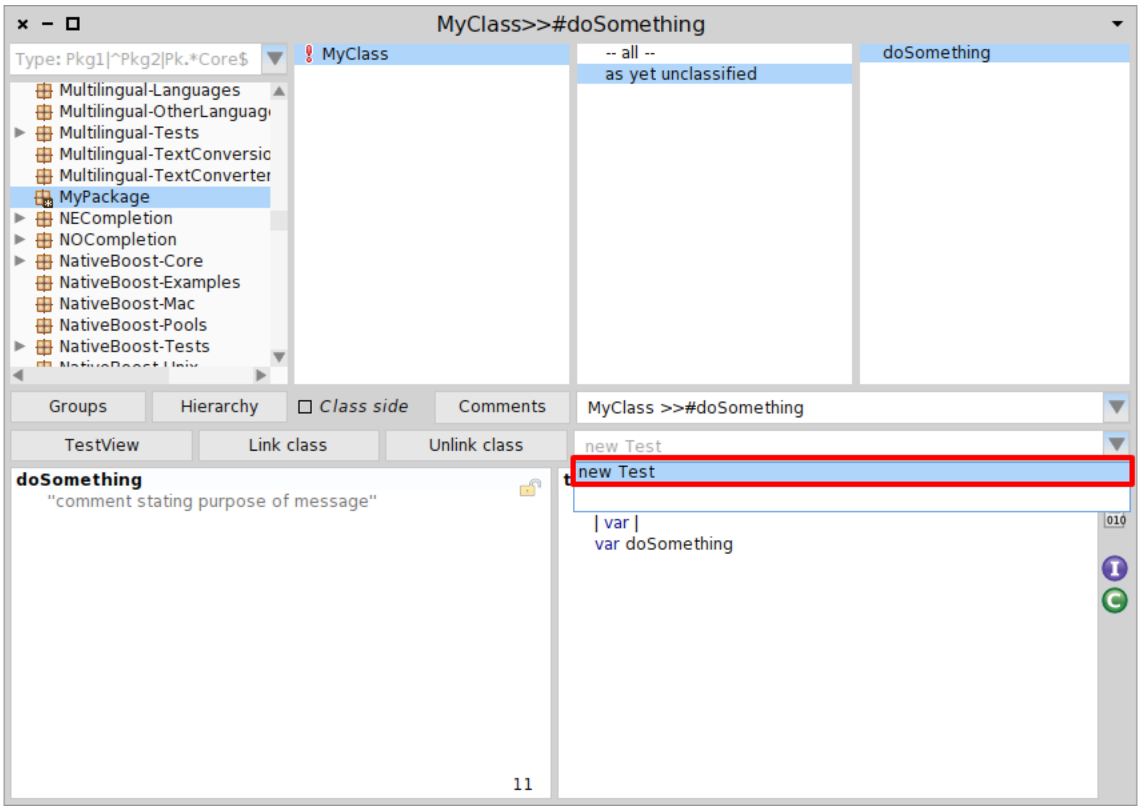


```
doSomething
    "comment stating purpose of message"
```

Figure 8.5: Toggle the TVPlugin on

2. Make sure that in the Nautilus window you have selected the method for which you want to add a test.

3. Now open the droplist showing all the tests that have been found for your method. Click on “new Test” as shown in Figure 8.6. By doing this you signal to the plugin that you want to add a test in a new test class.



```
doSomething
    "comment stating purpose of message"
```

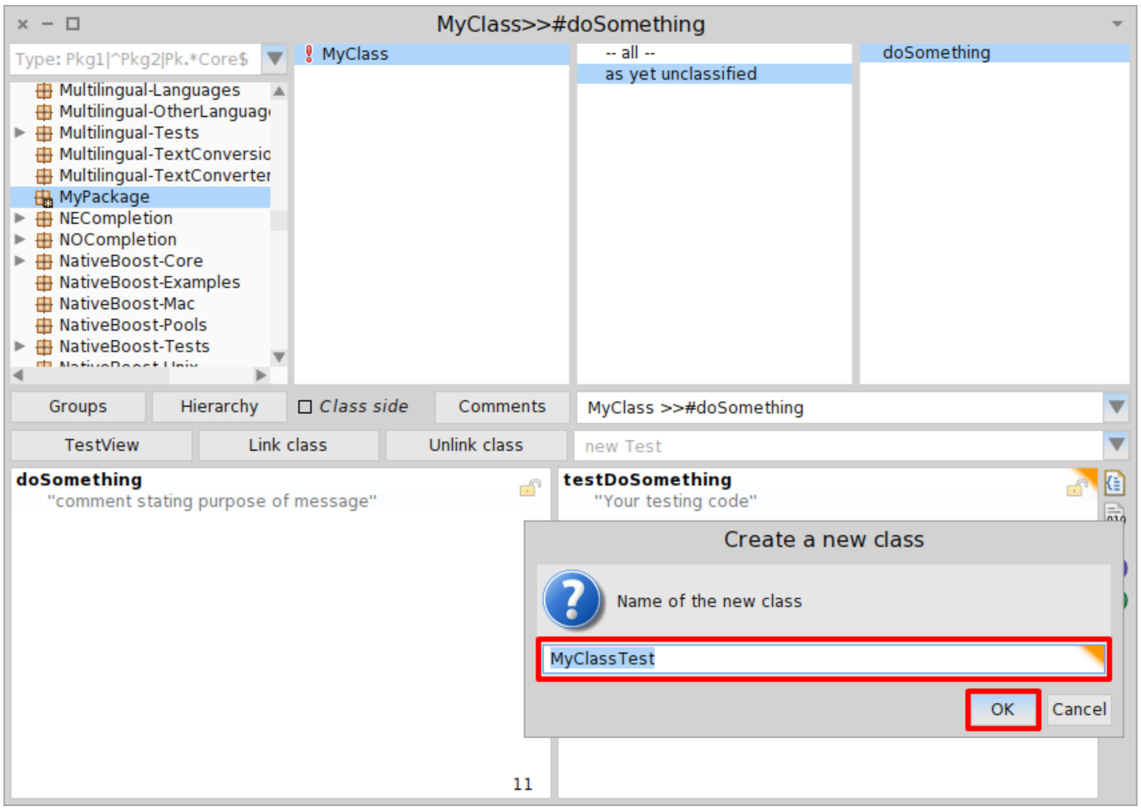
```
| var |
var doSomething
```

Figure 8.6: Signal that you want to save the test in a new test class

4. Write your test in the right code panel. A template to start writing is already provided there by the TVPlugin.

5. Make sure the right code panel is still selected and accept your new test by pressing ctrl+s.

6. Since you previously selected “new Test” from the droplist the plugin is not sure in which test class this new test should be saved and will ask for clarification. As shown in Figure 8.7 a default name will already be provided but you can write your own test class name. Click “OK” when you have entered a name.

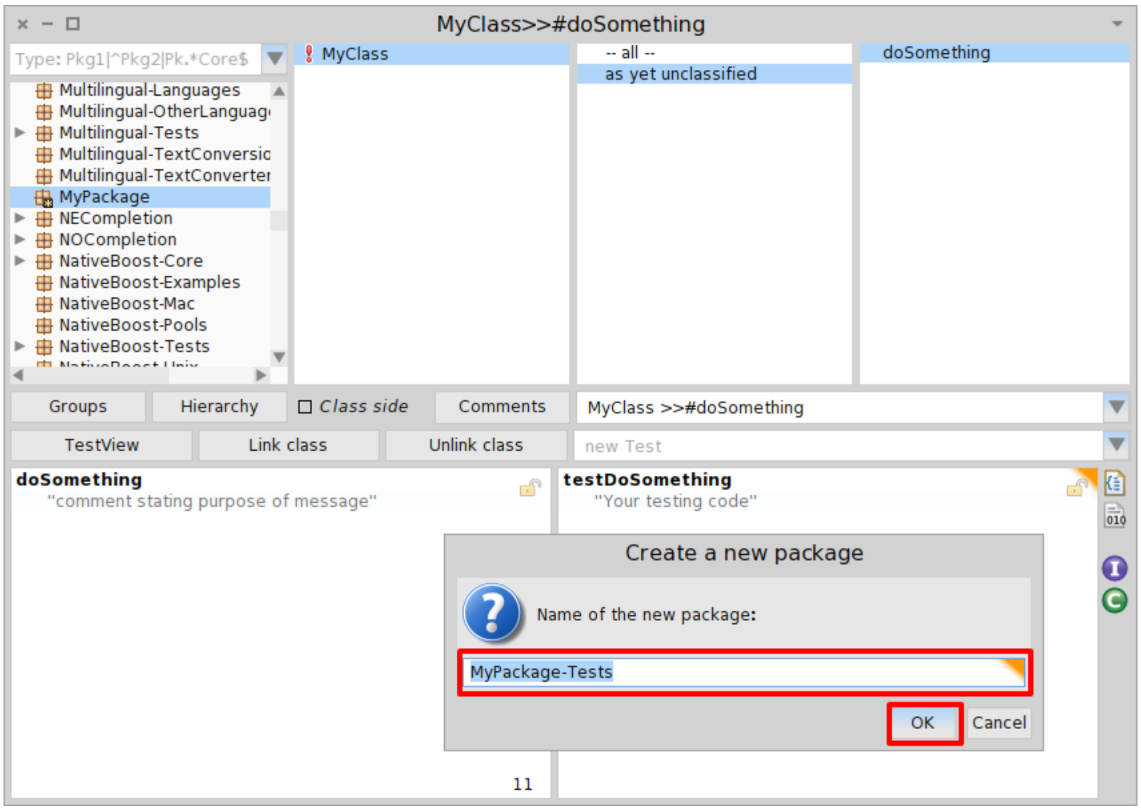


```
doSomething
    "comment stating purpose of message"
```

```
testDoSomething
    "Your testing code"
```

Figure 8.7: Name the new test class

7. Now you need to name in which package this new test class will be added. Similarly as before a default name will be provided but you can enter your own. Click the “OK” button highlighted in Figure 8.8 to confirm the package name. This test package will now be created if it did not exist previously.

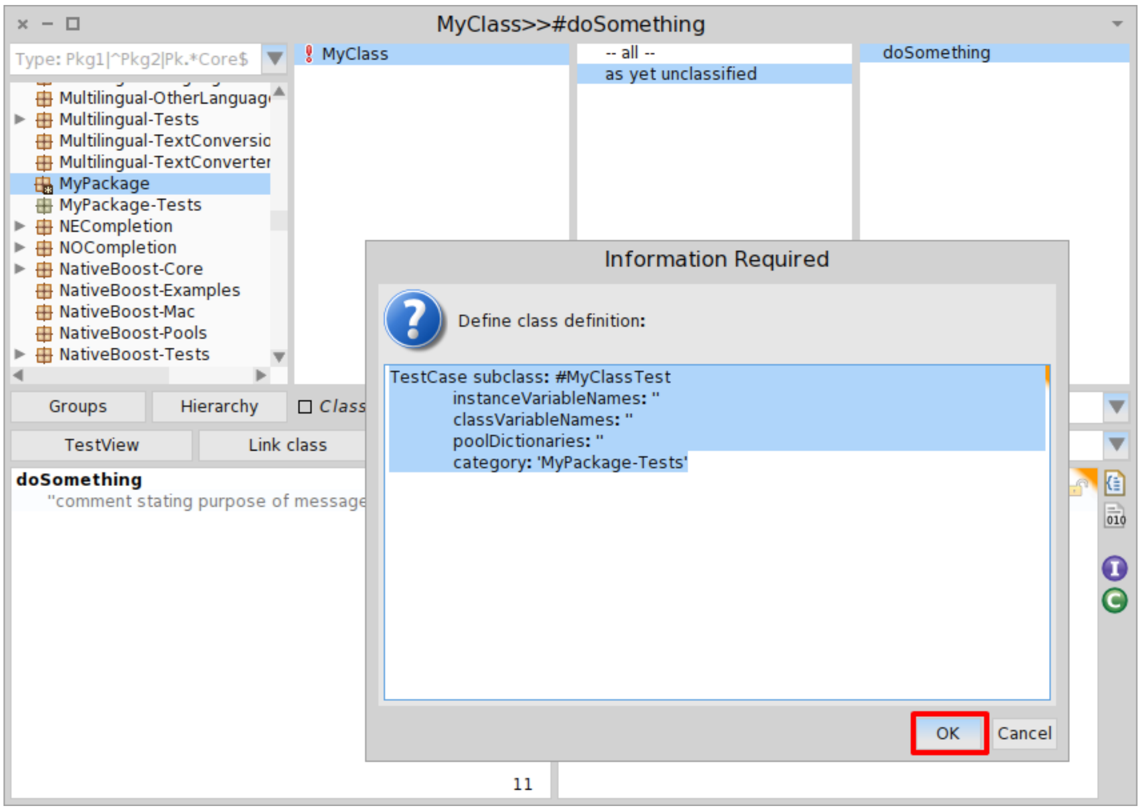


```
doSomething
    "comment stating purpose of message"
```

```
testDoSomething
    "Your testing code"
```

Figure 8.8: Name the new test Package

8. You will now see a pop-up similar to Figure 8.9 with a class definition according to what you entered in the previous steps. You can one last time change your mind and cancel the creation of the new class or enter new class and package names. Once you have finished checking and if necessary have made additional changes click on “OK”. The new test will now be added to the newly created test class and test package.



```
doSomething
    "comment stating purpose of message
```

```
TestCase subclass: #MyClassTest
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MyPackage-Tests'
```

Figure 8.9: Confirm the new test class

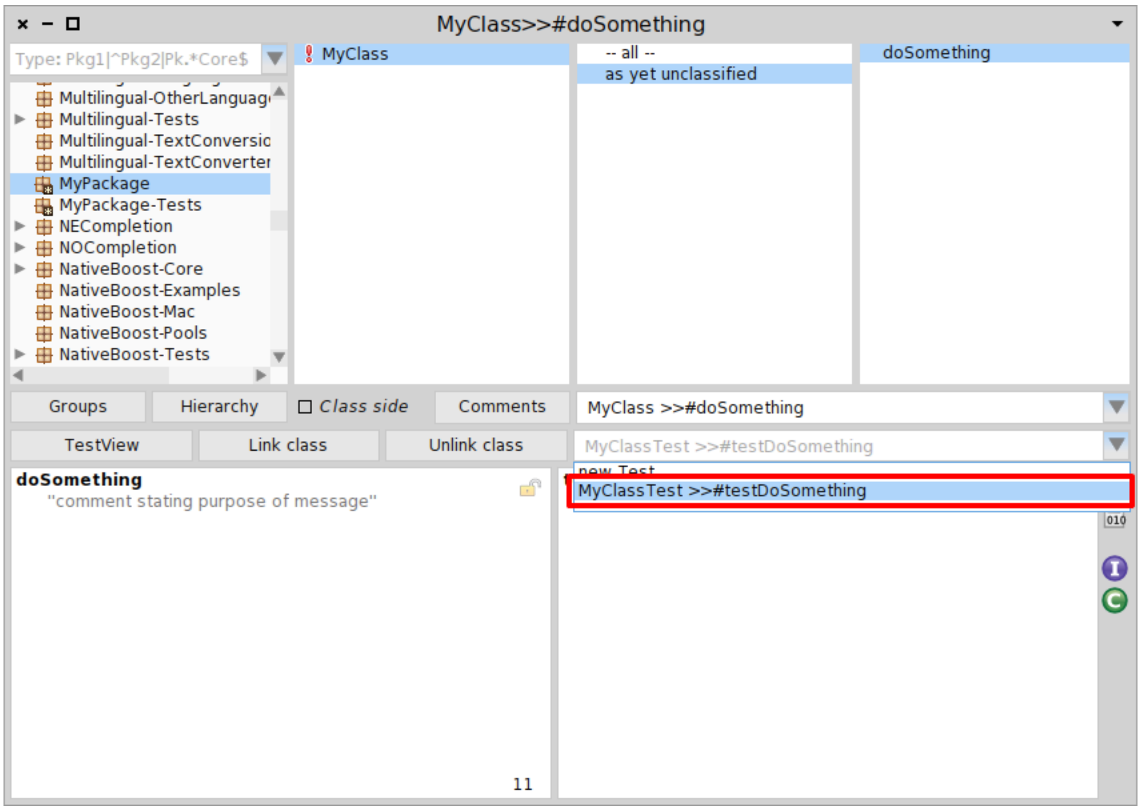
8.1.4 Adding a new test to an existing test class

In this section we will take a look at how to add a new test inside of an existing test class. It is assumed that the plugin is installed and activated. If not follow the steps outlined in subsection 8.1.2.

1. First make sure that the plugin is toggled on. If it is the Nautilus window has two code panels in the bottom. If it is not turn it on by clicking the “TestView” button. See Figure 8.5 if you can not find the button.

2. Make sure that in the Nautilus window you have selected the method for which you want to add a test.

3. Now expand the droplist with the results and select any test that is contained in the class where you want your new test to be. In this example this will be “MyClassTest” so we select “MyClassTest >>#testDoSomething” as can be seen in Figure 8.10.



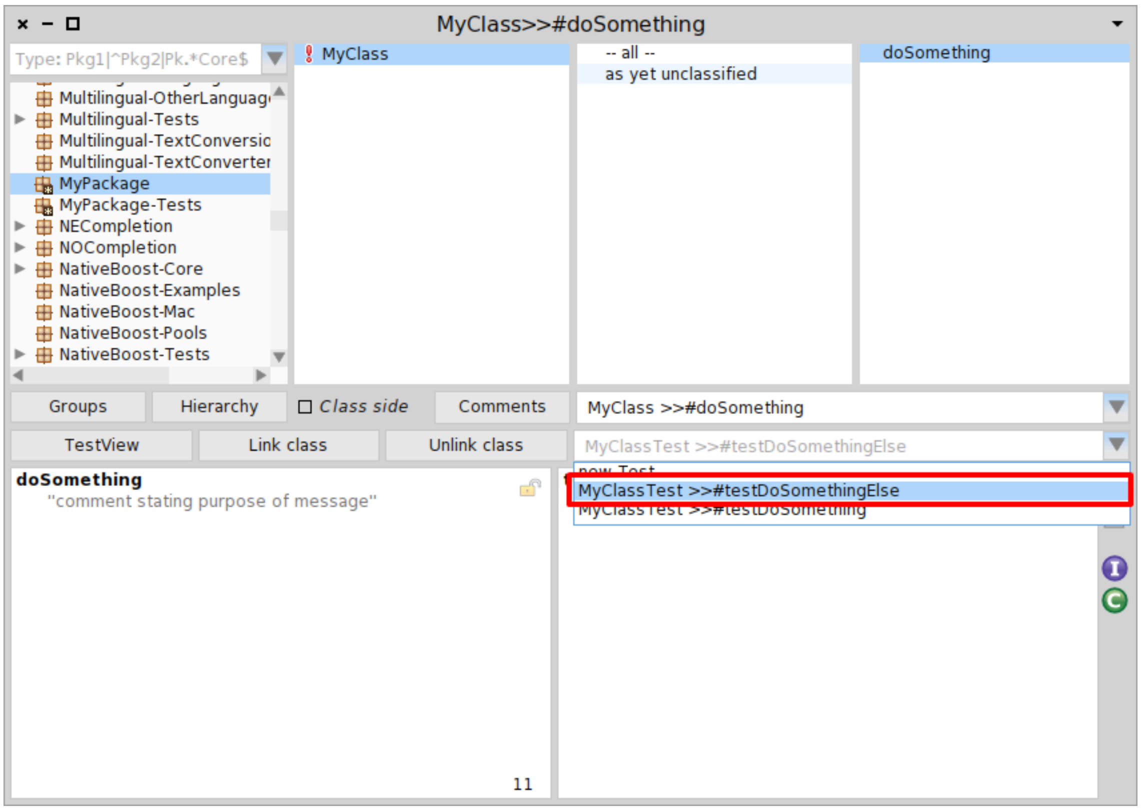
```
doSomething
	"comment stating purpose of message"
```

Figure 8.10: Select any test in the desired test class

4. The test you selected will now appear in the right code panel. Just write your test over it but make sure to give it a different name or else the test you selected will be overwritten.

5. Make sure the right code panel is still selected and accept your new test by pressing ctrl+s.

6. Now your new test will be added to the test class you selected previously. You can verify this by opening the found tests droplist again and checking if your new test is there as can be seen in Figure 8.11.



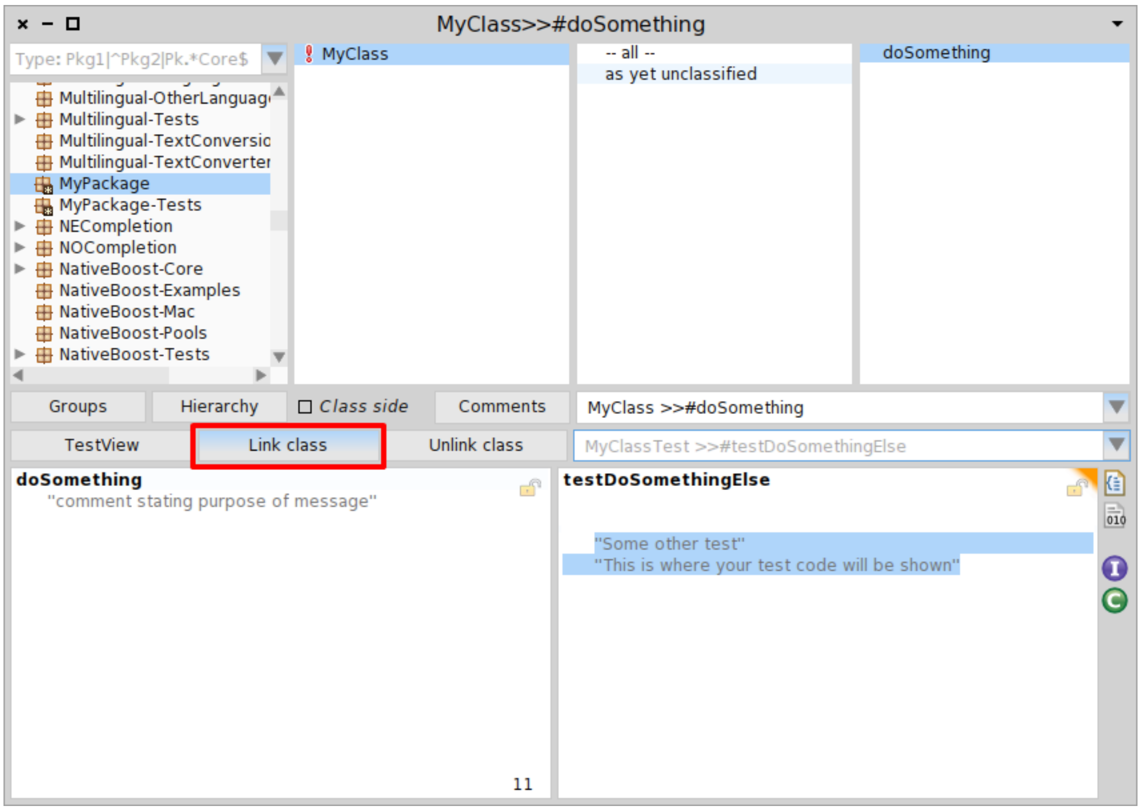
```
doSomething
    "comment stating purpose of message"
```

Figure 8.11: Check if your new test is shown here

8.1.5 Linking an existing test class

When the TVPlugin does not find tests that you have created then this might be because it does not recognize the your test class as a possible source for tests to the class of the method that you currently have selected in the Nautilus class hierarchy. Here you find a step by step list of how to add your test class to the considered test classes.

1. First in the Nautilus class hierarchy select the method that is missing tests in the found tests droplist.
2. Click the “Link class” button highlighted in Figure 8.12.

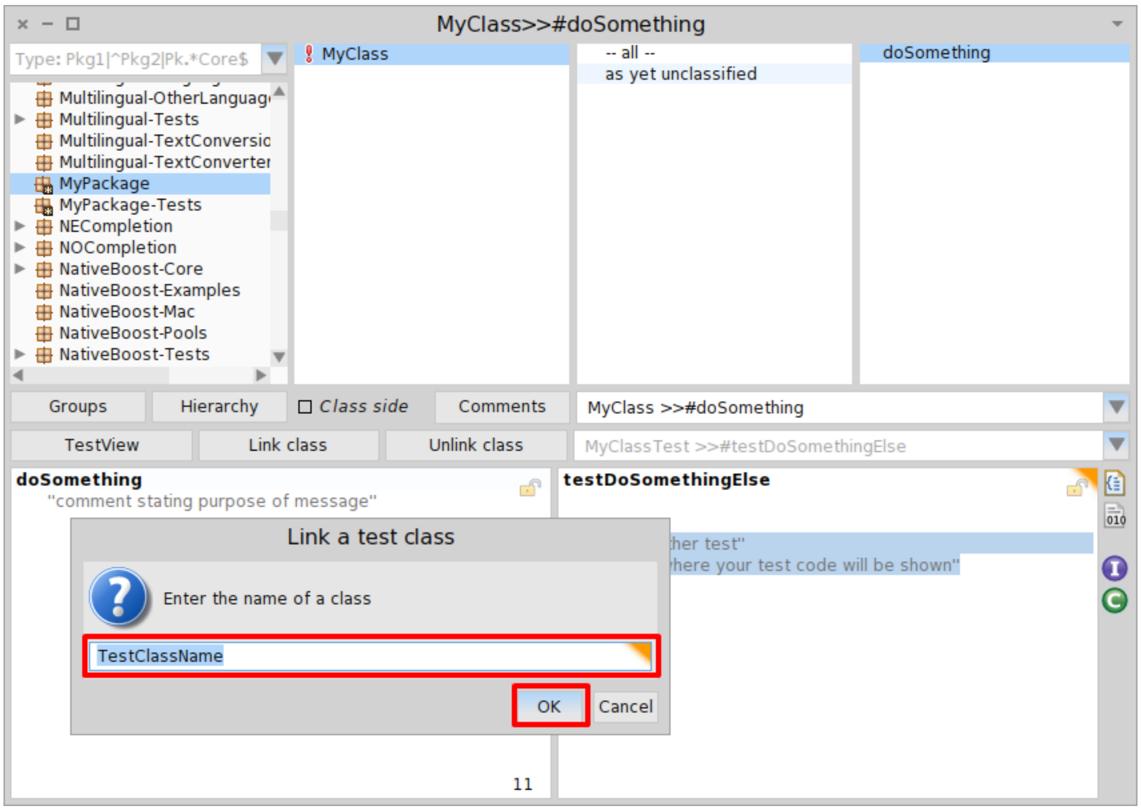


```
doSomething
    "comment stating purpose of message"
```

```
testDoSomethingElse


    "Some other test"
    "This is where your test code will be shown"
```

Figure 8.12: The “Link class” button

3. The TVPlugin will now ask which class you want to link as a test class to the class that is currently selected. Enter the name of the desired test class and click “OK” as shown in Figure 8.13.



```
doSomething
    "comment stating purpose of message"
```

```
testDoSomethingElse
her test"
here your test code will be shown"
```

Figure 8.13: Pop-up asking for the test class name

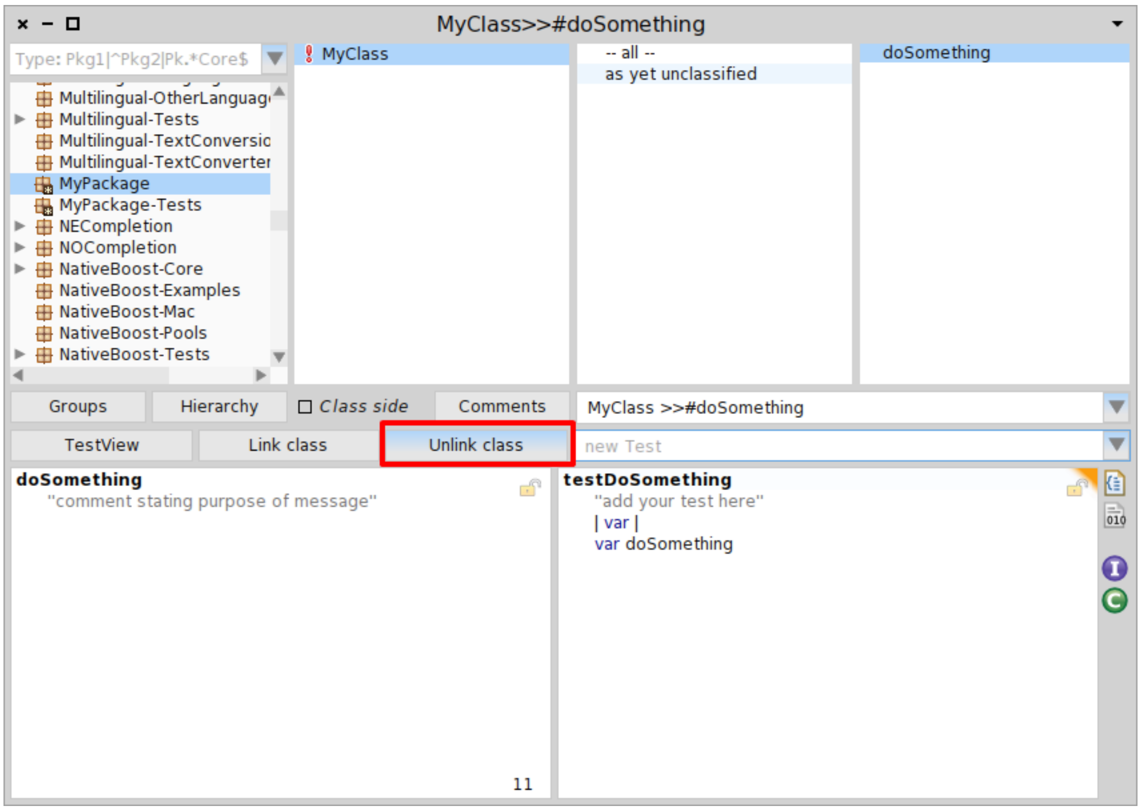
4. The specified test class is now added to the test classes that will be considered when searching for tests for the currently selected class. You can verify if it now works by opening the found tests dropview.

8.1.6 Unlinking an existing test class

In case the TVPlugin shows you tests from a test class that you do not recognize as a test class for the currently selected class you can remove this class from consideration by using the “Unlink class” button.

1. First in the Nautilus class hierarchy select the method that does show too many tests in the found tests droplist.

2. Click the “Unlink class’ ’ button highlighted in Figure 8.15.

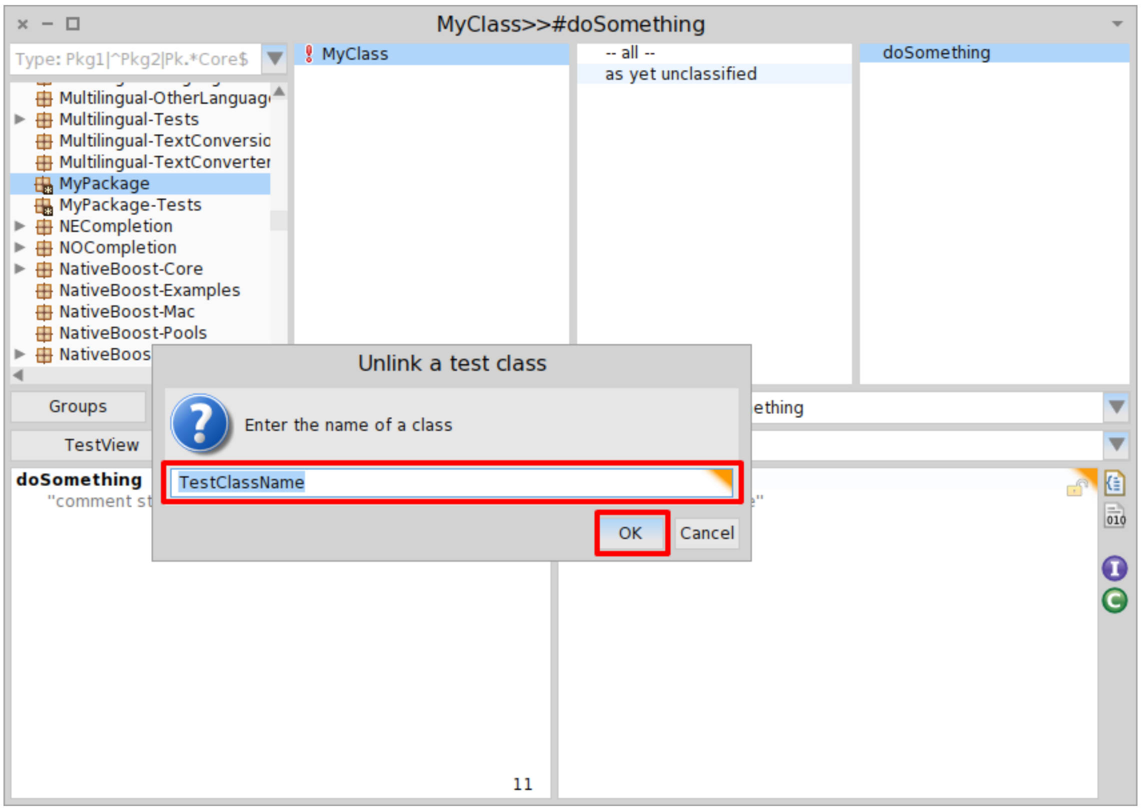


```
doSomething
	"comment stating purpose of message"
```

```
testDoSomething
	"add your test here"
	| var |
	var doSomething
```

Figure 8.14: The “Unlink class” button

3. The TVPlugin will now ask which class you want to unlink as a test class to the class that is currently selected. Enter the name of the desired test class and click “OK” as shown in Figure 8.15.



Unlink a test class

Enter the name of a class

TestClassName

OK Cancel

Figure 8.15: Pop-up asking for the test class name

4. The specified test class is now removed from consideration. All tests contained in this class will not be considered to be tests to the currently selected class. You can verify if this now works by opening the found tests dropview. No test contained in the unlinked class should be shown there.

8.1.7 Troubleshooting

In this section possible problems with the TVPlugin are listed and possible solutions to them outlined.

8.1.7.1 Tests are not found by the TVPlugin

In case all tests from one or multiple tests classes are missing in the found test droplist you can add those test classes manually by following subsection 8.1.5. If you want to add individual tests from a test class where already some tests are found then continue with subsubsection 8.1.7.3.

First make sure that your test classes are subclasses of ”TestCase”. If they are not the plugin will not consider them to be test classes. Simply go to the definition of your test classes and change the first line to ”[YourClassName] subclass: #TestCase”. Check again if the tests from your test classes are found now.

If this does not help you can force the plugin to recognize your classes as test classes by linking the missing test classes to the currently selected class in the Nautilus class hierarchy. To do this follow the steps in subsection 8.1.5. Recheck if your tests are found. If this does not work either then your tests themselves are written in a way that the TVPlugin does not recognize. You will have to change them so that they fulfill certain criteria. In this case also continue with subsubsection 8.1.7.3.

8.1.7.2 Tests that do not test the currently selected method are shown

If the classes that contains these wrongly indentified tests do not contain any tests for the currently selected class then you can simply unlink these classes. Follow the steps outlined in subsection 8.1.6. All the tests from these classes should now be gone from the found tests droplist.

If you want to remove only some tests from the found tests droplist then continue with subsubsection 8.1.7.3.

8.1.7.3 Adding and removing individual tests to the tests found by the TVPlugin

Both adding and removing an indiviual test from the tests that the TVPlugin finds are features that are directly supported. It is still possible to make these adjustments although likely not for all projects and not without changing your tests. The way the TVPlugin identifies individual tests as corresponding to the selected method has two components. The first one is soley based on the name of your test and the name of the method that it is supposed to test. The second one is if the test calls a method with the same name as the method that it supposedly tests.

The name based criterion is a check if your test contains the full name of your method under test with at least one additonal time the substring ”test”. Examples of what is recognized as a test and what not based on this criterion are shown in Table 8.1. This check is not case sensitive. ”:” from methods names with parameters will be ignored when looking for a test.

Original method name	*Test names*	*Not test names*
doStuff	doStuffTest, testDoStuff, TestdosTuff	doStuff, doTestStuff, testStuff
do:on:	testDoOn, testDoOn2, doOnTest	testDo:On:, do:testOn:

Table 8.1: Test naming criterion

The second criterion is if a method call is done inside of the possible test method to a method with the same name as the in the Nautils class hierachy selected method. This time the ”:” are not removed from the check. If the supposed test method to a method named ”do:on:” does call ”doOn” then it will still not count as a test.

A test is recognized as such if either or both of these criteria are fulfilled. Conversely if neither is then the test is not considered a test to the selected method. Adding a specific method to the tests found for a certain method can thus be done in two ways. Either make the name of the test contain the full name of the method in addition to ”test” or call the method you want to test directly in the test. To remove a method from the found tests you have to make sure that the test name does not contain ”test” and the full method name, and that inside of the test never a method with the same name as the supposed method under test is called.

Bibliography

[1] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[2] E. W. Dijkstra. Notes on structured programming. In E. Dijkstra, O-J. Dahl, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82. Academic Press, Inc., New York, NY, 1972.

[3] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[4] Philippe Marschall. Detecting the methods under test in Java. Informatikprojekt, University of Bern, April 2005.

[5] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2009.

[6] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2014.

[7] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Fourteenth International Symposium on Software Reliability Engineering ISSRE 2003*. IEEE, November 2003.

[8] Don Wells. Unit tests. `http://www.extremeprogramming.org/rules/unittests.html`. Accessed: 2015-11-18.