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 plugin since this makes it easy to set up and builds on established parts of the Pharo environment. This plugin 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 for existing methods and the ability to view a method and a corresponding test side by side. Before explaining the plugin 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. We also discuss opportunities for improvement. The aim of this thesis is to take a closer look at unit testing and in form of a Nautilus plugin provide an example of how unit testing can be facilitated.

Contents

1 Introduction

Testing has become an important part of software development. While it is true that “Program testing can be used to show the presence of bugs, but never to show their absence!” [2] testing helps to at least partially validate a program. Tests for example aids verifying newly written code and detecting if changes broke previously working code (regression errors).

Not only does testing make code more reliable but it also helps to speed up development by recognizing and preventing regression errors. User studies indicate that about 10-15% of the development time is spent waiting for tests to finish, executing and fixing regression errors and simply increasing the frequency with which the tests are executed reduces this wasted time by 31-81% [8]. This 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) [7]. 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[9] 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.

While testing clearly has its benefits, especially under stress unit testing is still seen as optional and too time consuming. Tests 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[7].

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]. Many environments are missing convenient features to facilitate unit testing and thus repetitively break the developers flow during unit testing. This only makes breaking out of this circle harder.

In this thesis we attempt to provide such an outside influence for the Pharo IDE. Our solution is realized as a plugin for Nautilus, the default system browser in Pharo. The TestView plugin provides additional functionality such as an easy way to view tests and methods side by side, an automated test search and a quick way to create new tests.

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 at as low a 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[1] 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. These postconditions are checked by using assertions.

To illustrate this let us examine how a unit test for an imaginary "*Stack*" class in Pharo might work. Each stack has a maximum capacity that is set once the stack is created. The stack provides a method named "*push: anObject*" which adds an element 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 the pop operation and not some other object. An example of how such a test could be implemented can be found in Listing 1.

The preconditions need to make sure that the stack is initialized with a capacity bigger than zero. This is done in line 4 of our example. Then an element is pushed and popped again in line 5 and line 7. The

[1] Roughly translates to a method invocation in other languages

postcondition is checked with the *assert* that can be found in line 7. The popped element should be the same as the one that was pushed to the stack earlier.

```
1 testPopReturnedElement
2     | stack |
3
4     stack:=Stack withCapacity: 2.
5     stack push: 5.
6
7     self assert: (stack pop=5).
```

Listing 1: Test for a stack's pop method

For the purpose of this thesis 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, since 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 sufficient. Below are some basic rules of Test-Driven Development as laid out by Robert C. Martin[5].

> ***First Law*** *You may not write production code until you have written a failing unit test.*
>
> ***Second Law*** *You may not write 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, repeating this 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.

Test-driven Development bears mentioning because it is one of the major paradigms concerned with testing and because in the context of this thesis it has to be treated a bit differently. Namely when trying to facilitate testing Test-Driven Development can be difficult to properly support. Some useful information to support the user is only available after a method has been implemented. This includes the method declaration and the location of this method. Without these no test name and test location can be derived for new tests.

2.3 Blackbox and Whitebox Testing

Whitebox and blackbox testing refer to whether or not the implementation of a method is taken into account while writing tests for it.

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 the most extreme cases. These are the method invocations that result in as many 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 "test 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 impossible to test all possible input 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. Still, as important as unit testing is, not all current development environments are optimized for it. For example, the search for unit tests of 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:

1. A quick way to see if a method is tested
2. An automated test search
3. The ability to view tests and methods side by side
4. 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, some of these features are often 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 sections we will take a closer look at the tasks involved in different goals arising from unit testing. This will help to understand the problem of those missing functionalities.

3.1 Is a method tested?

In unit testing it is important that each method is tested (baring trivial things like getters and setters). 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 non intrusive 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 like the Test Runner for the Pharo IDE or EclEmma for Eclipse also provide this functionality. They allow the user to run test suites and then see which parts of the implementation were executed. This will show if a method never got called and thus is untested.

Coverage tools were not designed to check if a method is tested or not. Methods that are called indirectly by a test suite will still show up as covered. Developers thus might believe that a method is tested even if it never was called directly by a test.

3.2 Showing all tests of 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 needed again. A developer 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 might make unit testing in larger applications and test suites very tedious and new test writers will have trouble getting an overview of the existing test suite.

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 suite. Also coverage is not necessarily 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. If this is the case the user has to create these new tests. Especially at the start of a project many new tests will have to be created. So it is safe to say that another reoccurring user goal in unit testing is the creation of new tests for insufficiently tested methods.

While creating a new test the developer first has to decide which package and class the new test will be added to 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. The programming environment can not always make such name derivations especially during Test-driven Development.

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 interact with 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. Possibly the simplest way is to let the users see the whole source code of the 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 and writing tests for it can save valuable time and effort.

4
Related Work

In this section we describe current programming environments, namely Eclipse and Pharo with respect to 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 for a specified method and facilitating the creation of new tests. The two environments are compared and possible places for improvement are discussed.

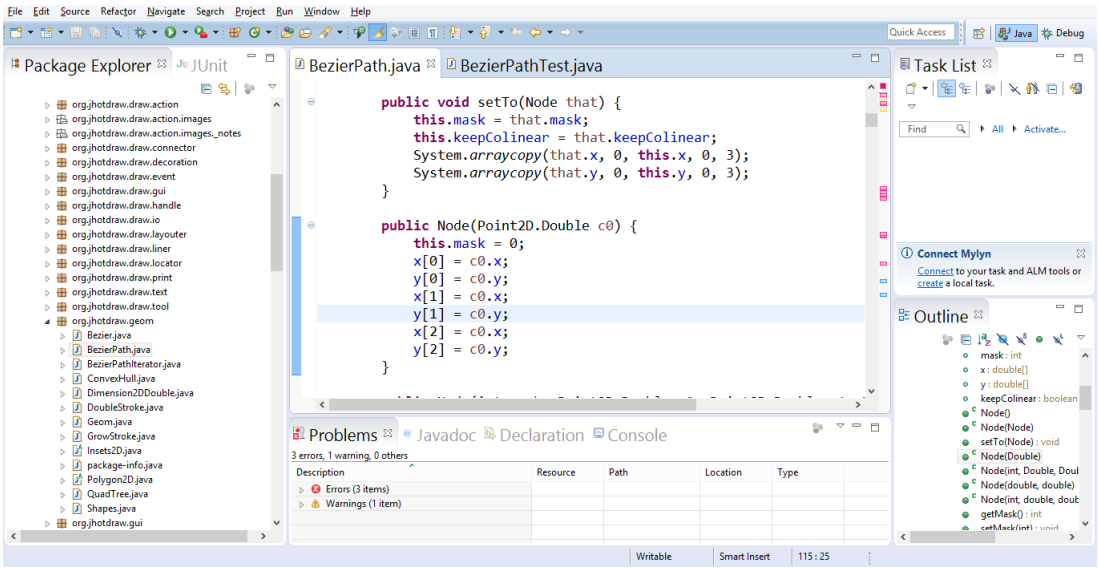

Figure 4.1: Eclipse's user interface

4.1 Unit Testing in Eclipse

We will start by taking a closer look at Eclipse since it is widely used and many parallels to other programming environments (*e.g.,* Visual Studio) can be made.

In Figure 4.1 you see the Java project JHotDraw 7.6[1] opened in Eclipse 4.5.1 with multiple packages and classes. The packages and classes can be seen on the left side in the "Package Explorer" pane.

4.1.1 Methods and Tests Side by Side

We start with a discussion on how Eclipse enables users to see a method and corresponding tests at the same time.

One option is to open both classes through the Package Explorer. The opened classes 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 "*BezierPath.java*" and "*BezierPathTest.java*". The "*BezierPath.java*" tab is currently selected and thus the content of "*BezierPath.java*" 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 to quicker find the currently important ones. In most cases the use of these tabs will be smooth enough to allow the programmer to switch uninterrupted between classes and their tests.

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 limited number of 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.

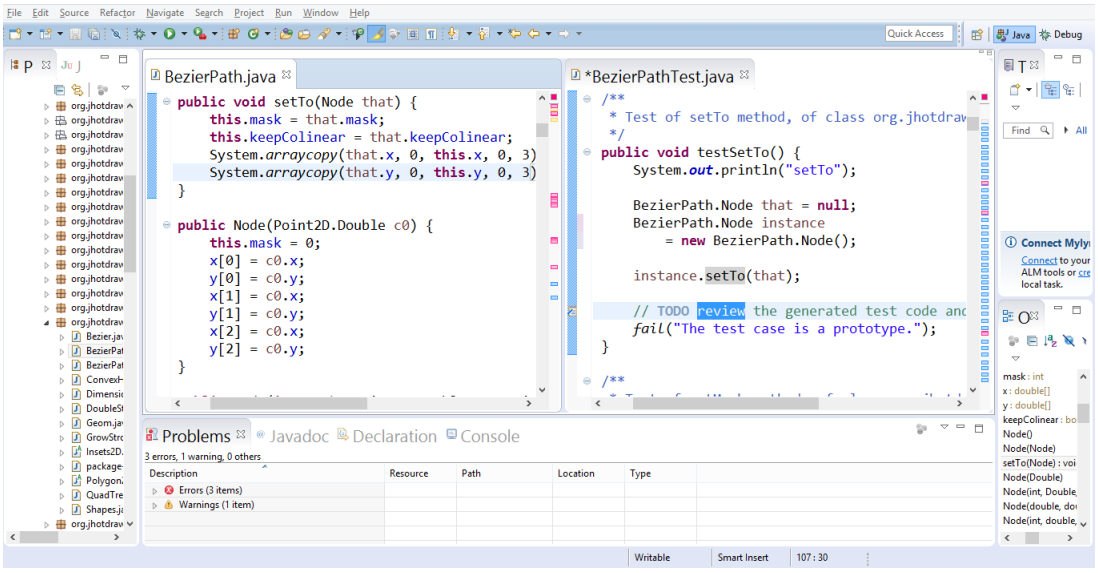

Figure 4.2: Split code panel

[1] http://www.randelshofer.ch/oop/jhotdraw/

4.1.2 Test Search

Another feature we discussed in the previous chapter was the ability to find all tests of a certain method. Eclipse provides a good option to search for tests. By selecting the “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 be associated with a certain method seem to include that the test class extends “TestCase” and that the test name starts with “test”. The “@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 that we now discuss is how the creation of new tests is facilitated. Eclipse provides an option for adding 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”.

In the newly open wizard a default name is already created and the selected class is put into the “Class under test” field, as shown in 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. During Test-Driven Development its use is also limited, since it can only add tests to existing methods.

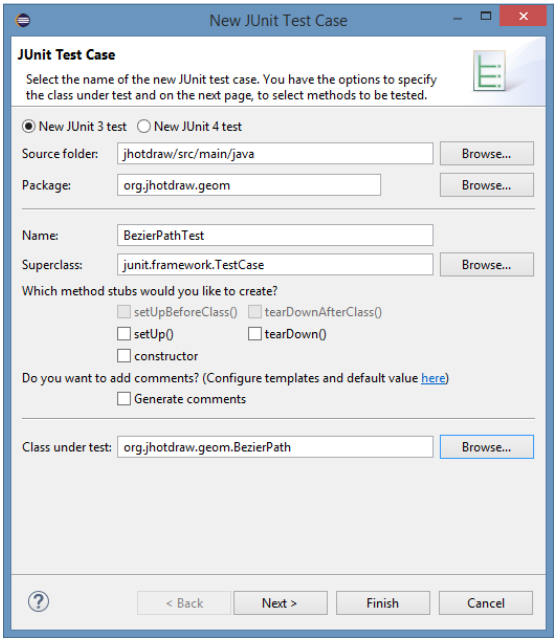

Figure 4.3: Test class wizard

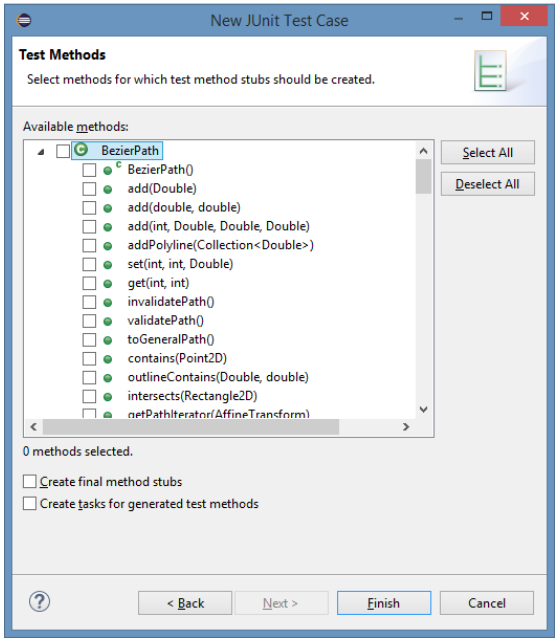

Figure 4.4: Select methods to test

4.2 Unit Testing in Pharo

Now let us take a look how the Pharo IDE and its system browser Nautilus support unit testing and how they compare to Eclipse. In this context it seems 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 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 for a specified method and facilitating the creation of new tests.

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. It invites users to arrange all windows that are created in a way that seems comfortable. It is possible to create multiple Nautilus windows and arrange them so that one shows the method and the other the test for this method. 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 list of packages and classes 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.

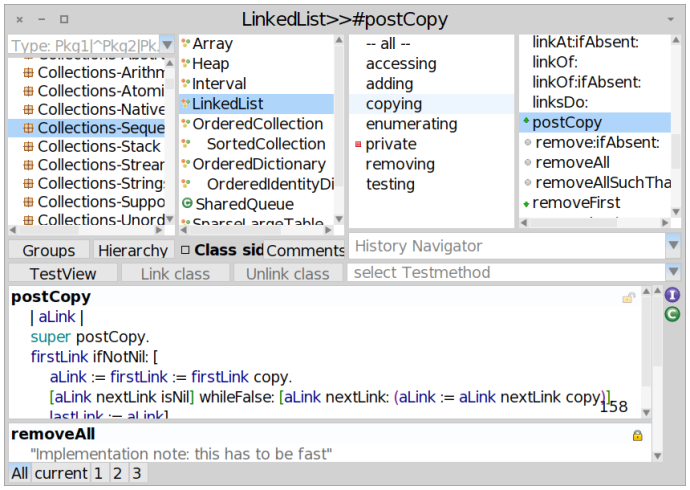

Figure 4.5: Nautilus window with active locks

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 for a specified method. Like Eclipse Nautilus has a test search implemented. Like with Eclipse's test search only tests with very specific names are found. Additionally they have to be placed in classes with an equally specific name. Methods and classes with a different name than what follows are not considered corresponding test classes and methods.

Namely, test classes and test methods have to contain the full name of the class or method under test. 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 by Nautilus's test search. The test classes also have to inherit from *TestCase* in order to be found by the test search. 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 found by Nautilus

Both the limitations on class names and method names are quite restrictive. From a test class name the class under test can be quite reliably inferred. On the other hand the method name and the corresponding test name for it are less directly related[4]. These restrictive name criteria also lead to the problem that at most only one test class and method 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 for 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

Right clicking on a method in Nautilus gives the "Generate test" option. 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 package under test with the suffix "-Tests". The test class and method

will be named like Table 4.1 shows. 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 since they will have exactly the same name.

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 present an attempt to address 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 it is much easier to implement a plugin for Nautilus than for Eclipse. 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.

In this chapter we will comment on how the TestView Plugin performs regarding viewing test and method side by side, finding tests for certain methods and creating new tests. Comparisons to Eclipse and the Pharo IDE will be drawn.

5.1 Methods and Tests Side by Side

As stated in subsection 4.2.1 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 and convenient. It should also not introduce too much redundant user interface elements that might clutter up the environment.

Following this, the TestView Plugin allows splitting the code editor panel of a Nautilus window vertically into two parts. The result of this can be seen in Figure 5.1. With this approach the need to open two Nautilus windows is eliminated. The list of classes and packages 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.

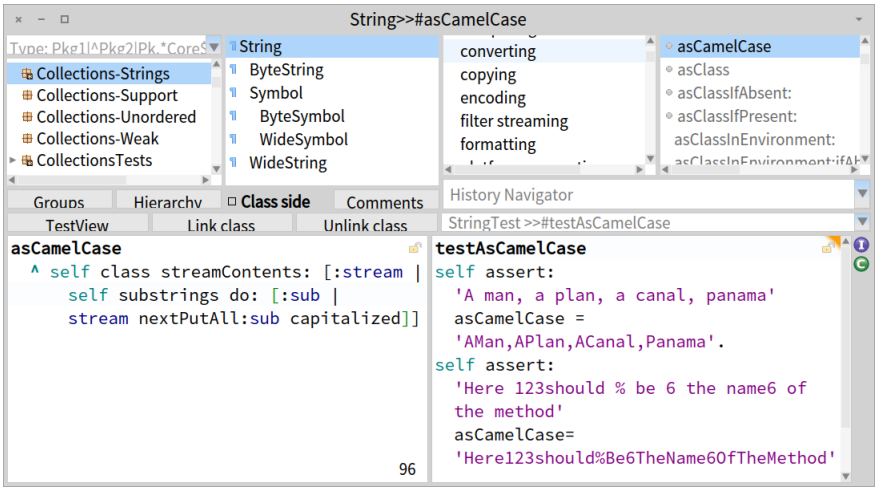

Figure 5.1: Nautilus window with installed TestView Plugin

Which test is currently shown can be changed through the drop down 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 implementation.

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

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

5.2.1 Corresponding Test Classes

To qualify as a test class a class has to inherit from "*TestCase*". This requirement has to be fulfilled in Nautilus' test search as well.

After this a name-based search is performed on the remaining classes. A name-based search works quite well to find test classes corresponding to a certain class[4]. A name-based criterion is used in Nautilus' test search and something similar is also part of the test class search in the TestView Plugin.

As discussed in subsection 4.2.2 Nautilus' naming criteria result in at most one corresponding test class being found. The naming criterion that the TVPlugin uses is less restrictive.

The name-based criterion of the TVPlugin requires the name of the class in question to contain the full name of the selected class in addition to "test". Unlike Nautilus' naming criterion this substring search is not case sensitive and it does not matter which of those substrings comes first or if the name contains additional characters.

In Table 5.1 examples of test class names that fulfill and that do not fulfill the naming criterion of the TestView Plugin are shown.

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

More precisely the substring search works as follows. First all class names are gathered in a **class name collection**. In a **substring collection** the name of the class under test and the string "test" is put. The substring collection is sorted so that the longest string is first.

Starting with the longest substring out of the substring collection each element in the class name collection is now checked if it contains this substring. If it does the first occurrence of this substring is removed and it is checked if the remaining string contains the second substring from the substring collection. A class name qualifies as test class name if both substrings from the substring collection were found in it.

To illustrate this substring search let us examine why a class named "Contest" is not considered a test class for a class named "Contest". To confirm to the naming criterion used by the TVPlugin the test class name would have to contain the substrings "Contest" and "test". The substring search first checks if the substring "Contest" is included in "Contest". Since it is the first occurrence of "Contest" is removed, which leaves an empty string. Now it is checked if this empty string contains "test" as well. Since it does not "Contest" is not a test class for "Contest".

Since the test method search builds on the results of the test class search it became important to make the test class search works works as good as possible.

Our solution to this is to let the users customize the search results. If the 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.

Classes that have been unlinked will no longer be searched for test methods while linked classes will be searched even if they do not inherit from "*TestCase*" or do not fulfill the naming criterion.

The plugin keeps track of the linked and unlinked classes in two *Dictionaries*. When a class is linked or unlinked the currently selected class is used as a key to store the specified class. Since the currently selected class is used as a key each linking or unlinking of a class only counts for the currently selected class.

5.2.2 Corresponding Test Methods

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

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

The substring search works like described in subsection 5.2.1. First a collection is created that contains all method names of all found corresponding test classes. Then the substring collection is filled with the name of the selected method and "test". The only difference here is that all colons are removed from the

Original method name	*Test names*	*Not test names*
addDays:	testAddDays	addDays, testAddDay
asFloat	testFractionAsFloat, testFractionAsFloat2	testFloatAsFraction
print24:on:	testPrint24On, testPrint24OnWithPM	testPrint24withNanos
attest	attestTest, testAttest	attest

Table 5.2: Test naming criterion

name of the selected method before it is put in the substring collection. Like before the substring collection is sorted according to length, both substring checks are made and matches removed. If both substring checks are successful the method is a test for the selected method-

The second criterion besides the name 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 for the selected method.

How many of those criteria are met is used to order the found tests in the drop down list that can be seen in Figure 5.1 right below of the Nautilus History Navigator. Specifically each found test method has an associated rating. The rating is increased by one if the method uses the selected method and by two if the naming criterion is fulfilled. Test methods with a higher rating will be displayed first in the results.

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

5.3 Creating new tests

Another purpose of the TVPlugin is to make the creation of new tests as quick as possible. This is done by providing the user with default names as much as possible and making the creation of new test packages, classes and methods as easy as possible.

Every time a developer starts 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 tests corresponding to the selected method already exist.

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 placed. With this the user is encouraged to immediately start writing a test as soon as a method is created. The new test class option is activated by having the topmost element "new Test" from the drop down list of found tests as show in Figure 5.2.

When the test method is saved the plugin will ask the user for the name of the test class and in which package this class should be placed. Default values, based on the class name and the package name of the method under test, are provided. This might already be sufficient since test class names and test package names can often be derived from the original class and package names [4].

To create the default name for a new test package "-Test" is added to the name of the package under test. Similarly the default test class name is derived from the name of the class under test with the suffix "Test".

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

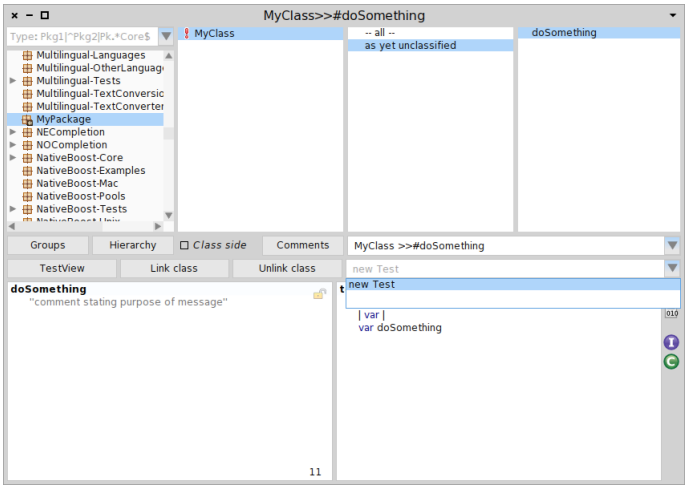

Figure 5.2: Expanded drop down list with new test option

"Collections-Tests" respectively. 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 for a method that already has tests. In this case the corresponding test classes should have already been found by the test search described in section 5.2. The users just have to select any found test that is in the test class where the new test should be added. No test package has to be specified since only when a new test class is made the package might also be a new one.

When the user saves a test while having selected the "new Test" option described in subsection 5.3.1, 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 in this drop down list then the test is saved in the test class that was selected.

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 new tests 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

To see if the TestView plugin fulfills our initial aim to encourage unit testing we conducted a small scale controlled experiment coupled with a questionnaire. Our most important research question was to see if the TestView plugin increases test coverage by reminding the users to write tests. Other research questions were if the plugin reduces the time, clicks and keystrokes necessary to implement some simple classes and methods.

6.1 The Setup

We evaluated the plugin with four PhD students from the Software Composition Group of the University of Bern. The only requirements for the participants were that they know both the Pharo IDE and programming language.

The evaluation was printed on multiple sheets in order to prevent later questions and tasks from influencing the participants. The evaluation started with two simple multiple choice questions surrounding test-driven development and unit testing. This was done to be able to group the participants according to how comfortable they were with testing. These questions were:

1. *Do you know what Test-driven Development is?*
   Yes No
2. *How often do you write unit tests for your code?*
   Never Rarely Often Always

The next part of the evaluation was the controlled experiment. For this we selected two problems in advance. The participants were asked to write in the Pharo IDE a solution to these problems. One problem was the "Even Fibonacci numbers" problem[1] and the other problem was the "Sum Square Difference" problem[2].

We paraphrased these problems taken from Project Euler to ensure that both tasks would require the same number of methods to complete:

[1] https://projecteuler.net/problem=2
[2] https://projecteuler.net/problem=6

Task1

Implement the "Even Fibonacci numbers" problem.

(Hint the Fibonacci numbers are: 1,2,3,5,8,13,21,34,55,89,...)

a) Write a class with a method that generates all Fibonacci numbers below a certain number "collectFibBelow: aNumber".

b) Additionally the class should have a method to sum up all even numbers in a collection.

c) Write a method that combines a) and b) and sums up the even Fibonacci numbers below a certain number.

Task2

Implement the "Sum Square Difference" Problem.

a) Write a method "sumSquaresOf: aCollection" that receives a collection of numbers, squares each number and sums up the squares.

b) Write a second method "squareSumOf: aCollection" that receives a collection of numbers, sums them up and returns the square of the sum.

c) Write a third method "sumSquareDiff: aCollection" that calculates the difference between "squareSumOf: aCollection" and "sumSquaresOf: aCollection".

The participants were required to use a prepared laptop with two Pharo images. One image had the TV Plugin installed and the other did not. In each image one of the tasks had to be implemented.

To keep track of the time it took the participants to complete the tasks and the user interactions with the IDE we used DFlow[3]. DFlow is a tool written for the Pharo IDE that tracks user interactions like mouse movement, key presses and various windows opened inside of Pharo[6]. We modified DFlow slightly to correctly keep track of mouseclicks and made it log the gathered data localy instead of sending it to a server.

In order for the sequence of the images and the difficulty of the problems not to influence the evaluation we used counter balancing on both of these factors. This means that the first two people started both with the "Even Fibonacci numbers" Problem while the last two people ended with this problem. Every participant with an odd number started without the TVPlugin while those with an even number started the first task with it and then completed their second task without it.

The evaluation ended with an open question to gather opinions about the usefulness of the plugin:

3. *Did the TestView plugin help in your opinion?*

To measure if the plugin encourages the users to write test we used the test coverage percentage. We neither told the participants explicitly to write tests nor that the plugin was there. With this we wanted to see if simply by being there the plugin would remind users to write tests. Only if the participants decided to write tests we told them to use the plugin if they did not notice it.

To quantify the participants' performance we took additional measurements. With these the participants that used the plugin and those that did not should be compared. This should give us an indication if the plugin facilitates testing or not.

[3]http://dflow.inf.usi.ch/experiment.html

6.2 The Results

As expected everyone answered question 1 with yes. For question 2 two participants stated that they write unit tests rarely and two stated that they write unit tests often.

The results of the experiments can be seen in Table 6.1 and Table 6.2. The results seem pretty unexpected which might be due to the small number of participants. All measurements were taken with DFlow. The tracked values are:

1. time to complete the task in seconds
2. number of pressed keyboard keys
3. number of mouse clicks
4. number of opened windows
5. test coverage percentage

Participant	Problem	First Task	Time	Keys pressed	Clicks	Windows	Coverage
1	SumDiff	No	581	540	65	12	0
2	Fibonacci	Yes	1945	1550	542	24	100
3	Fibonacci	No	1089	810	287	17	0
4	SumDiff	Yes	516	486	91	5	0
Average:			1033	847	246	15	25

Table 6.1: Results with the TV Plugin

Participant	Problem	First Task	Time	Keys pressed	Clicks	Windows	Coverage
1	Fibonacci	Yes	702	687	89	13	0
2	SumDiff	No	919	739	170	9	100
3	SumDiff	Yes	433	261	107	4	0
4	Fibonacci	No	1023	1178	108	8	0
Average			769	716	119	8.5	25

Table 6.2: Results without the TV Plugin

The averages of the experiments with the TV Plugin seem to be slightly worse than those without. This is very surprising since except participant number two nobody used or noticed the plugin. Most likely the small number of participants made the differences between them the decisive factor.

The participants, with the exception of participant number, two did not write unit tests or make use of the TestView plugin. The one participant that used the plugin has a test coverage of 100% with or without the plugin. Also Person two was faster, pressed less keys or mouse buttons and opened less windows without the plugin. This difference is most likely due to the varying difficult of the problems or with the participant not being familiar with the plugin. Everybody seemed to have more difficulties with the Fibonacci problem.

Is is interesting to note that while only one person wrote unit tests everyone was testing if parts of their code worked as expected in the Pharo Playground. Testing definitively is a necessity but sometimes developers seem to prefer to do it manually. By making unit testing easier a lot of those manual test could hopefully become unit tests. That everybody answered positive to question 3 underlines the need for additional unit testing tools and an environment that properly supports unit testing.

The participants also gave general feedback to the plugin. The two most mentioned points were that the plugin should stand out more and that there should be a quick option to run all the found tests.

If making the plugin stand out more is really necessary is debatable since this was only wished by people who did not see the plugin. While this certainly would help users to discover the plugin, other users that already know about it could possibly be annoyed by it standing out to much.

A quick option to run all the found tests would certainly be a useful addition to the plugin. Luckily there is already the Nautilus TestRunner, a tool to execute tests and keep track of the test coverage of those test.

Other possible points for improvement that some participants mentioned were to include the *setUp* method in the drop down list of found tests and to enhance the template for new tests so that more *assert* variations would be used.

Nautilus itself allows the execution of tests by pressing a button next to each method or class that has corresponding tests. Sadly it is quite possible that not all tests get executed through this since as described in subsection 4.2.2 only one corresponding test per method or class is found by Nautilus. So while implementing a functionality to run tests through the TestView Plugin might be an option, enhancing the existing tools would to keep the number of required tools low.

7 Conclusion and Future Work

Unit testing is an important part of current software development paradigms like Extreme Programming. Sadly in many software development environments unit testing is not as well supported as it could be. This lack of support can break the developers' flow and discourage them from testing.

In chapter 3 we examined Eclipse and the Pharo IDE with its system browser Nautilus to see where exactly unit testing could be facilitated. Features that are lacking include: the ability to conveniently view methods and their tests side by side, an automated test search and an option to easily create new tests.

In an attempt to fix these problems concerning unit testing for the Pharo IDE we made a Nautilus plugin called TestView Plugin or TVPlugin.

With the TVPlugin it becomes possible to horizontally split the code editor of a Nautilus window. This second code editor will only be used for tests so certain optimizations can be made. Namely the environment can derive which tests the user wants to see in this second editor from whats displayed in the first code editor.

To be able to always show relevant tests the TVPlugin makes use of its own test search. Improvements compared to Nautilus' test search include less restrictive criteria for a method to qualify as a corresponding test. Additionally one method can now have multiple corresponding tests. See section 5.2 for a more detailed explanation of how the search works and what exactly the new criteria are.

To facilitate the creation of new tests the plugin provides default names and a template for each new test. Both adding a test in a new test class or an existing one is supported.

To evaluate if the TVPlugin encourages and facilitates testing we conducted a controlled experiment with a short questionnaire. The results although were not very decisive. While all participants agreed that the TVPlugin could help writing test the recorded measurements showed now clear improvement. This might be due to the low number of participants and the differences in their programming styles. Another user study would be needed to show the effectiveness of the plugin.

The participants of the user study suggested to implement an option to run all tests found by the plugin as well as to enhance the test search so that the *setUp* methods also would be found. More debatable but wished for by many participants is to make the plugin stand our more visually.

Another possible improvement would be to make the search results more customizable. At the moment the user can only add and remove corresponding test classes. A similar option to link and unlink individual test methods could be useful.

Partially integrating some of the features of the TVPlugin directly into Nautilus could also be beneficial. For example the TVPlugin's ability to split the code editor in half could be used to show other things than methods and corresponding tests side by side.

Another feature of the TVPlugin that could be integrated into Nautilus is its test search. Nautilus' own test search is used to display a button besides each method with which the corresponding test can be executed. With an enhanced test search it would become possible to run all found tests instead of just one through this button.

The test method templates that the TVPlugin creates to make adding new tests easier could also be used by Nautilus to give each new test some default content.

8

Anleitung zu wissenschaftlichen Arbeiten

8.1 User Guide

8.1.1 What's the TestView Plugin?

Nautilus is the default system browser in Pharo 4.0. Figure 8.1 shows how a Nautilus window normally looks like.

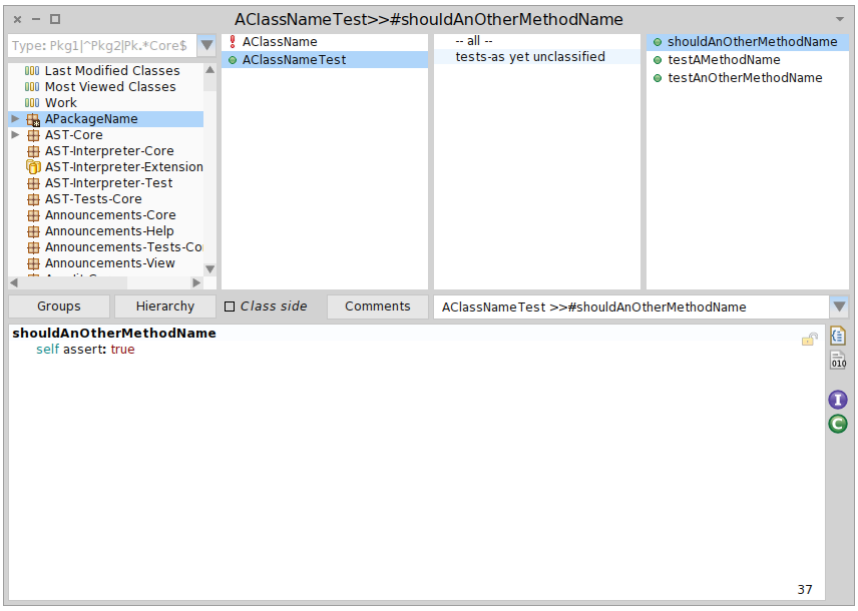

Figure 8.1: A Nautilus Window

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.

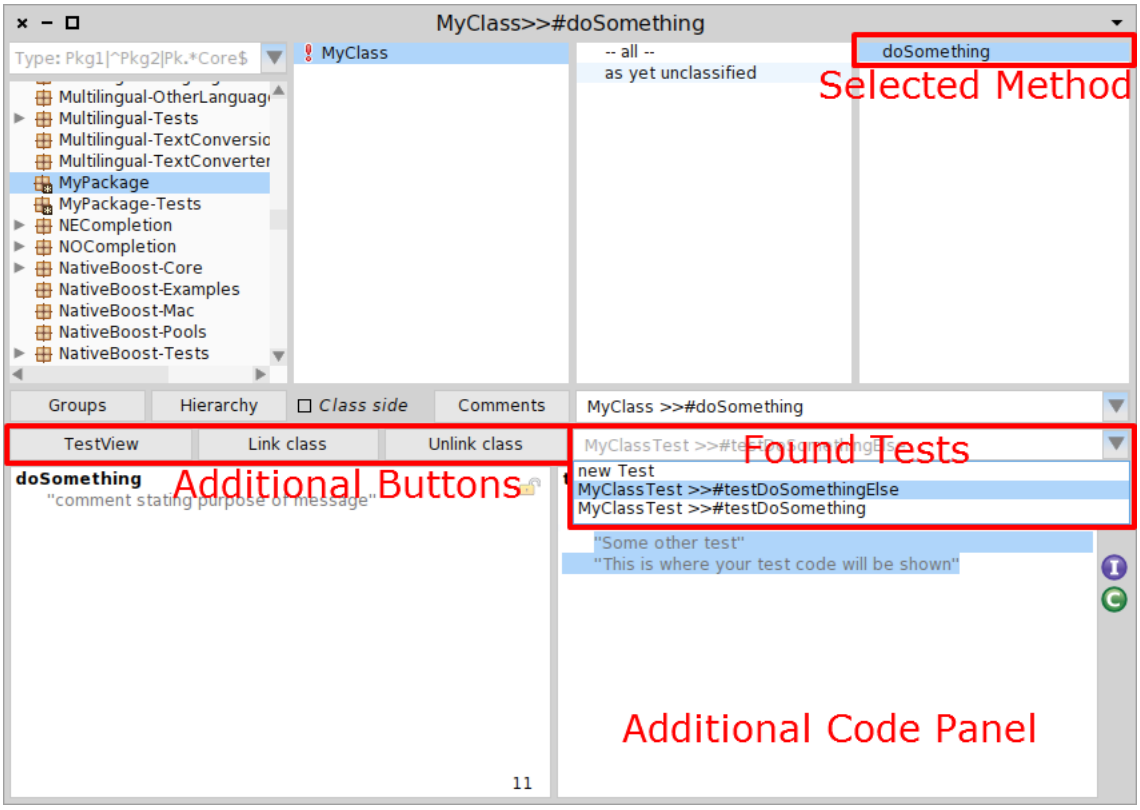

Figure 8.2: Nautilus with the TVPlugin

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 TestView Plugin heavily relies on knowing which method is currently selected in a Nautilus window. All functions 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 drop down list** 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 drop down list. 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 for 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.

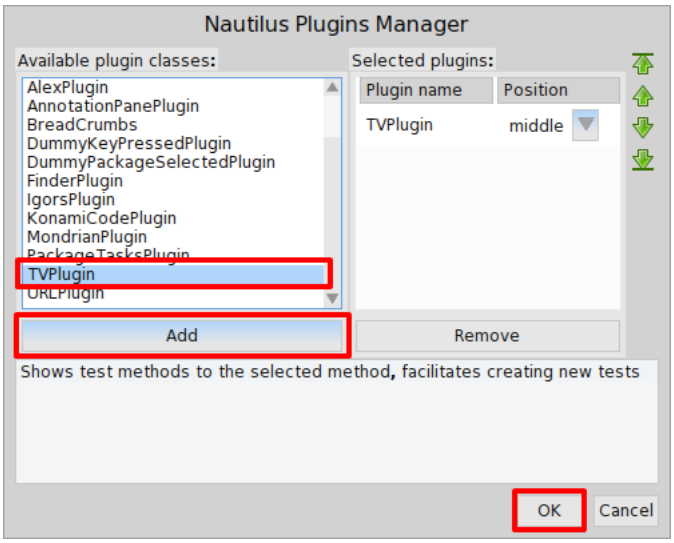

Figure 8.3: The Nautilus Plugin Manager

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.

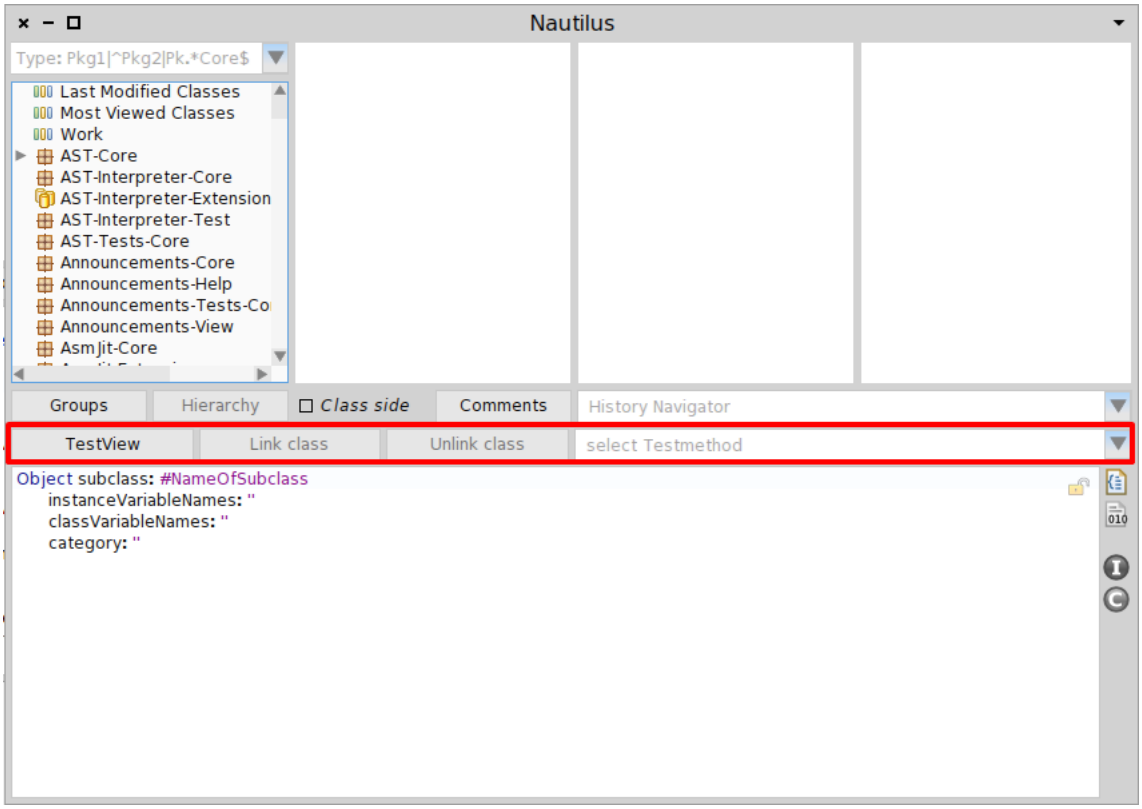

Figure 8.4: TestView Plugin once it is activated

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 we can add a new test for 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 for your selected method.

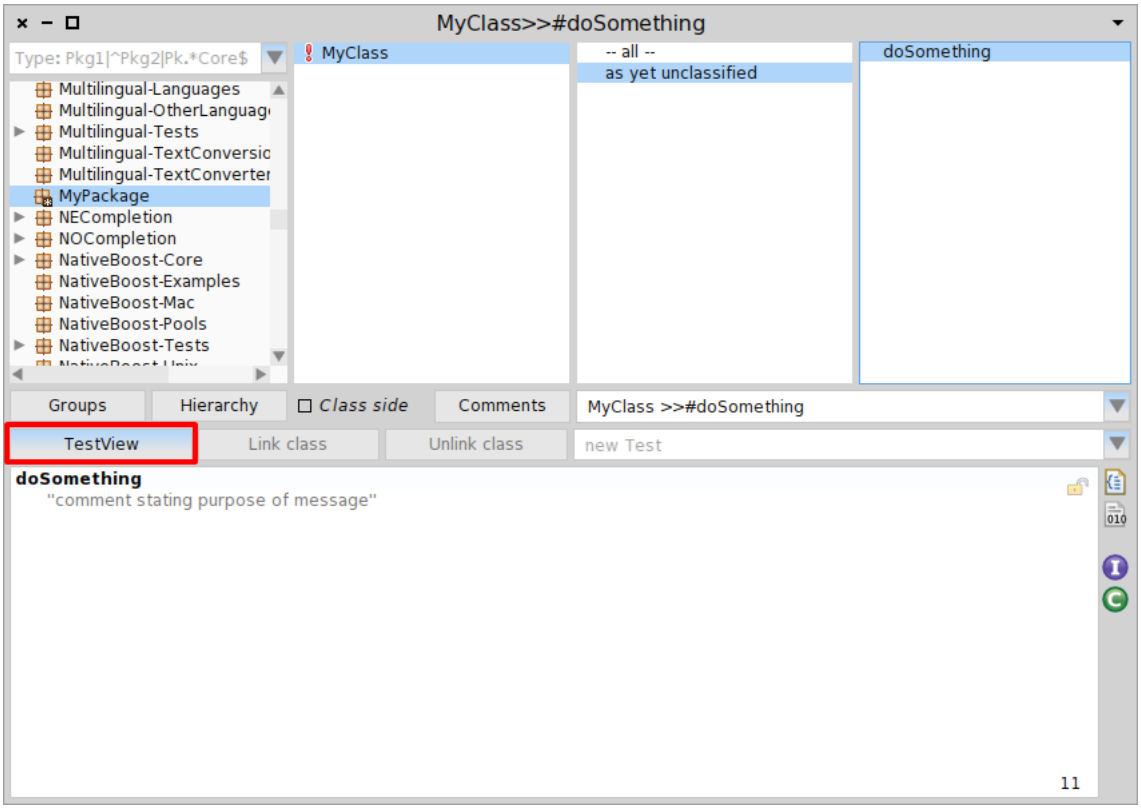

Figure 8.5: Toggle the TVPlugin on

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 drop down list 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.

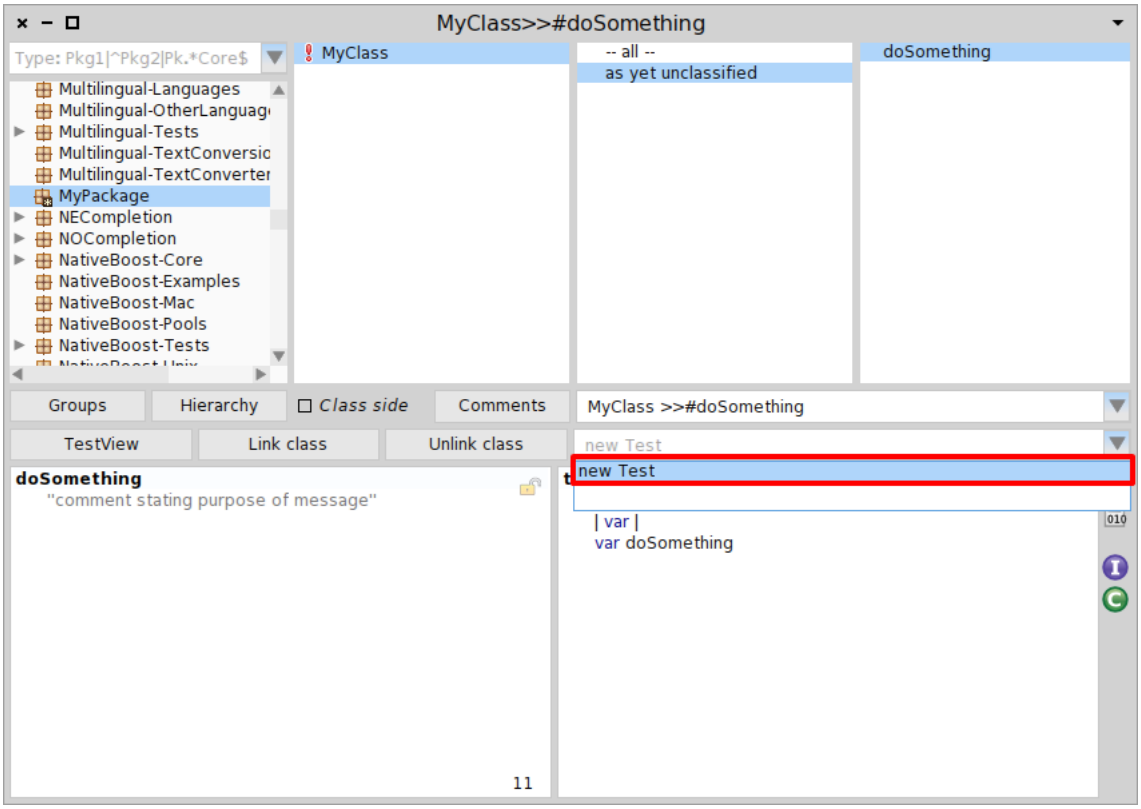

Figure 8.6: Signal that you want to save the test in a new test class

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 drop down list 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.

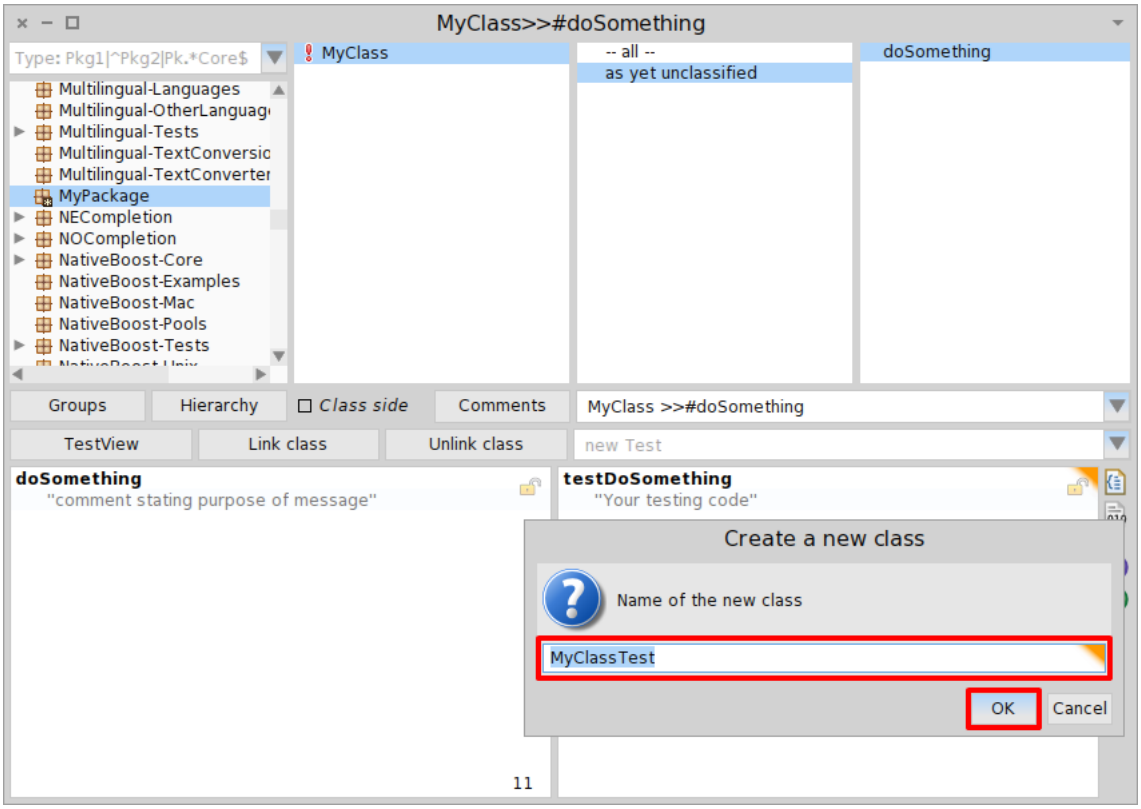

Figure 8.7: Name the new test class

7. Now you need to specify to 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.

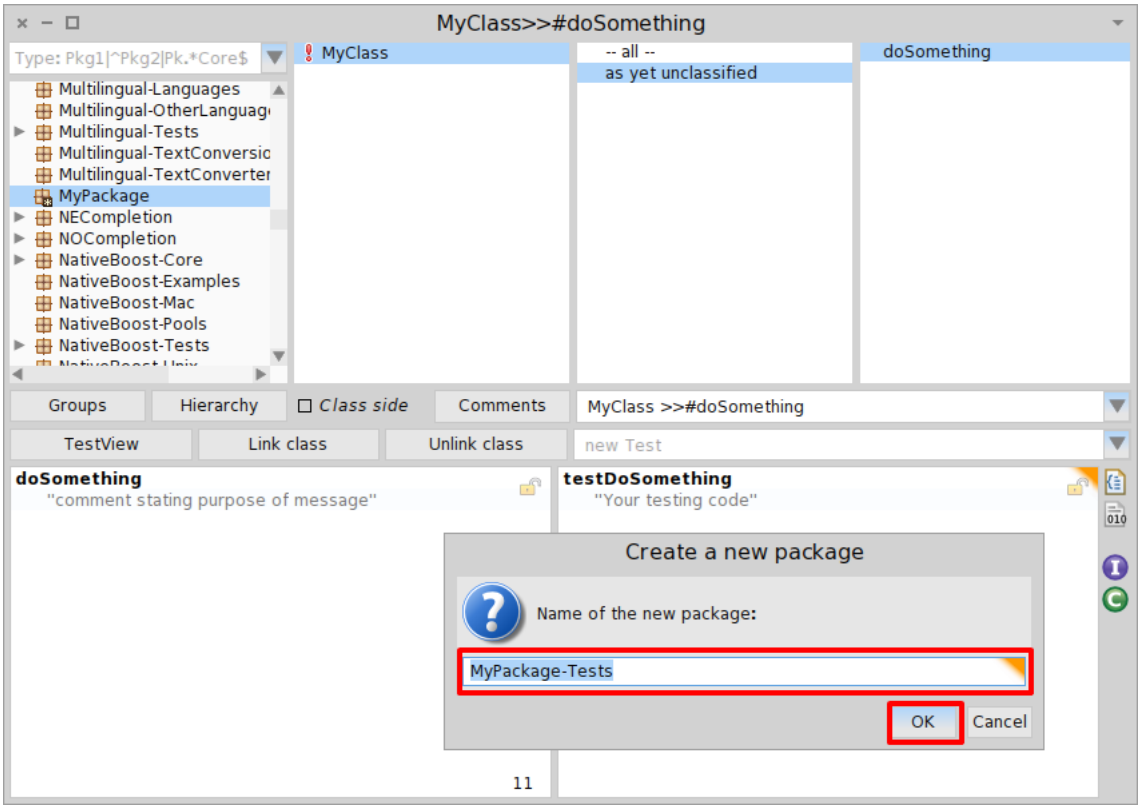

Figure 8.8: Name the new test Package

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.

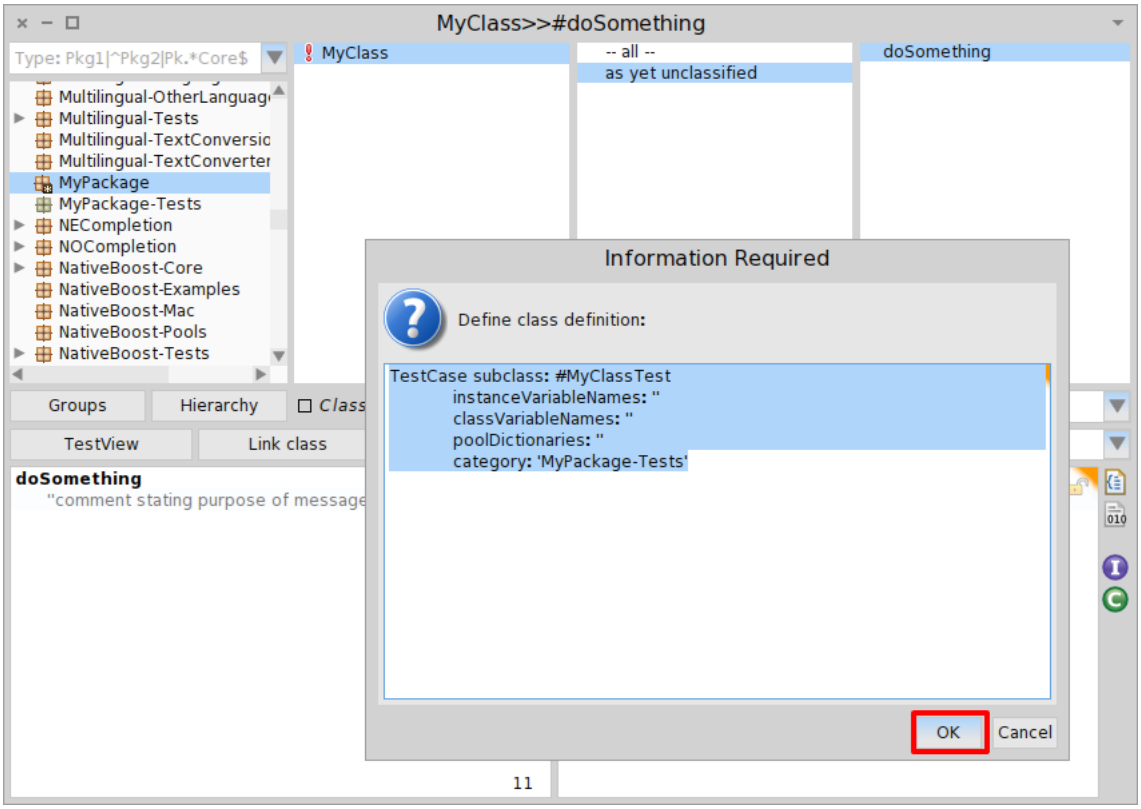

Figure 8.9: Confirm the new test class

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 drop down list 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.

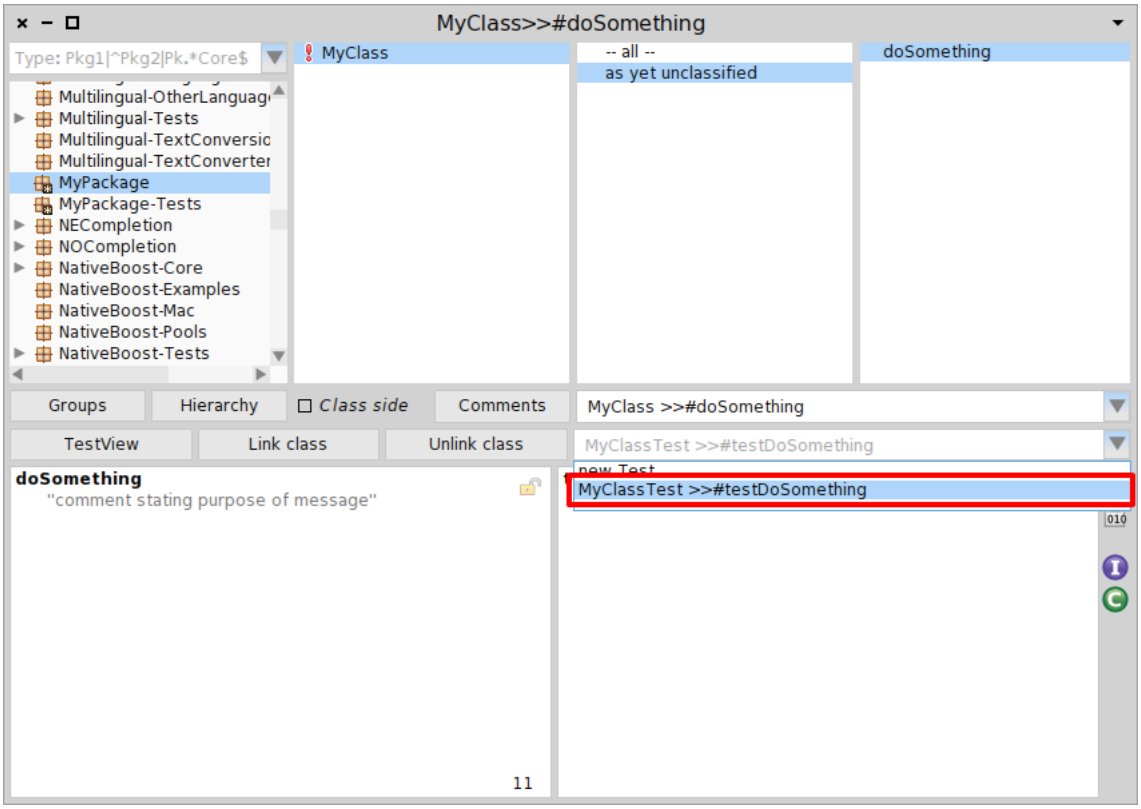

Figure 8.10: Select any test in the desired test class

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 drop down list again and checking if your new test is there as can be seen in Figure 8.11.

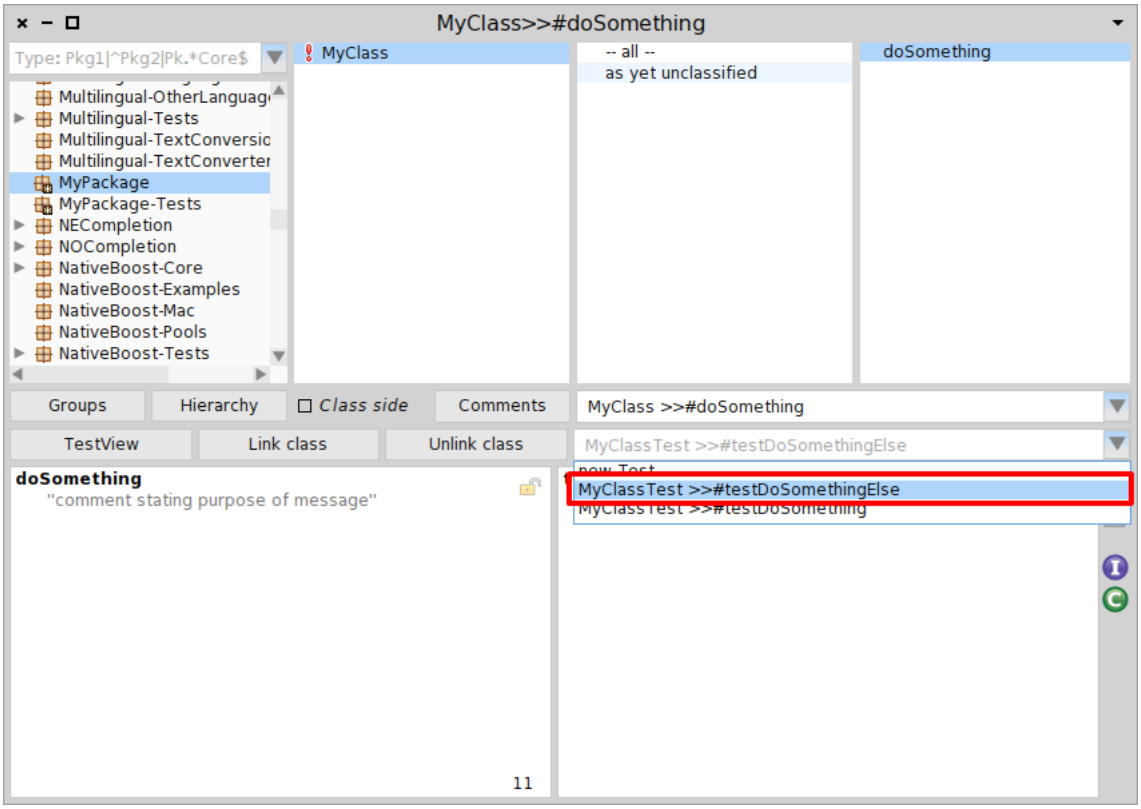

Figure 8.11: Check if your new test is shown here

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 for 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 drop down list.

2. Click the “Link class” button highlighted in Figure 8.12.

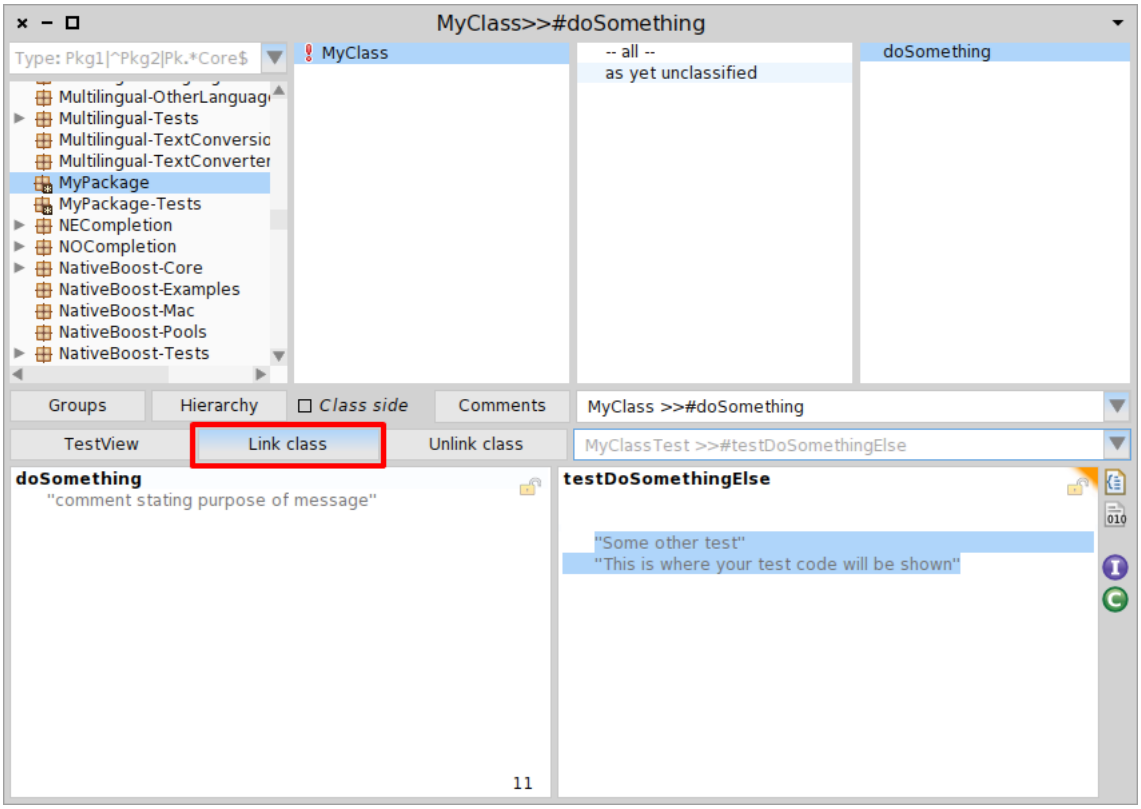

Figure 8.12: The “Link class” button

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.

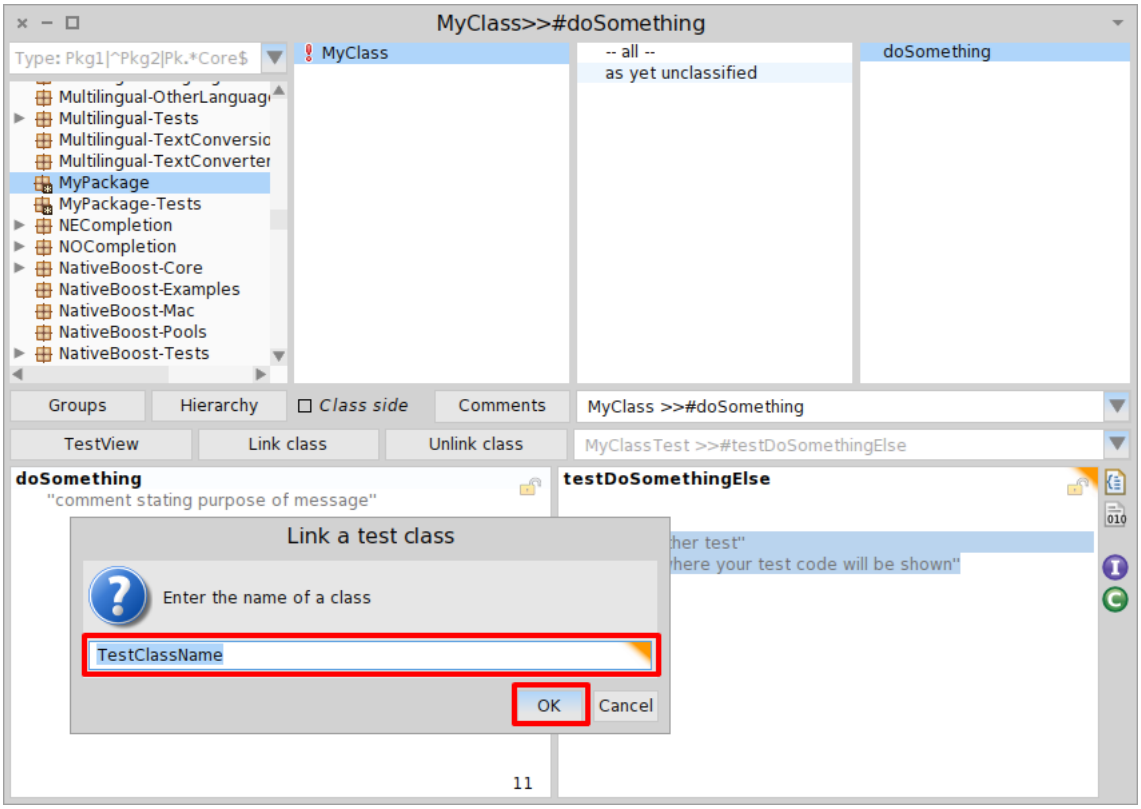

Figure 8.13: Pop-up asking for the test class name

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 drop down list.

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

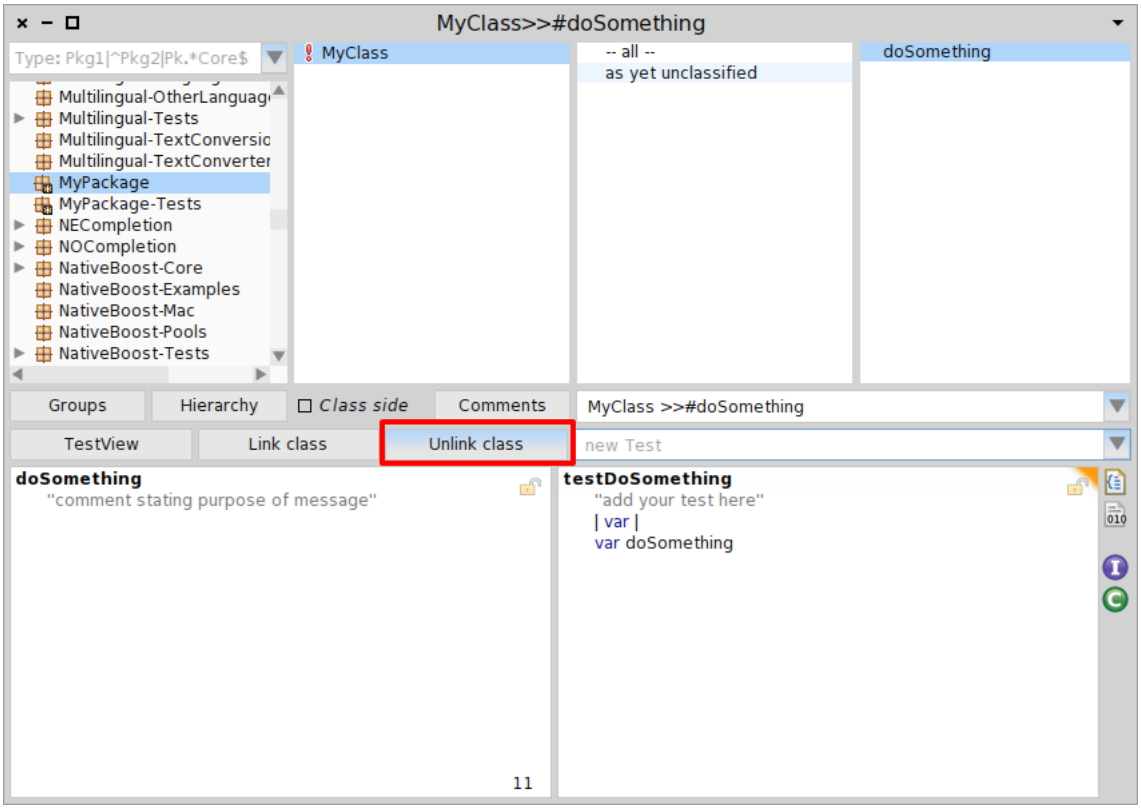

Figure 8.14: The “Unlink class” button

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.

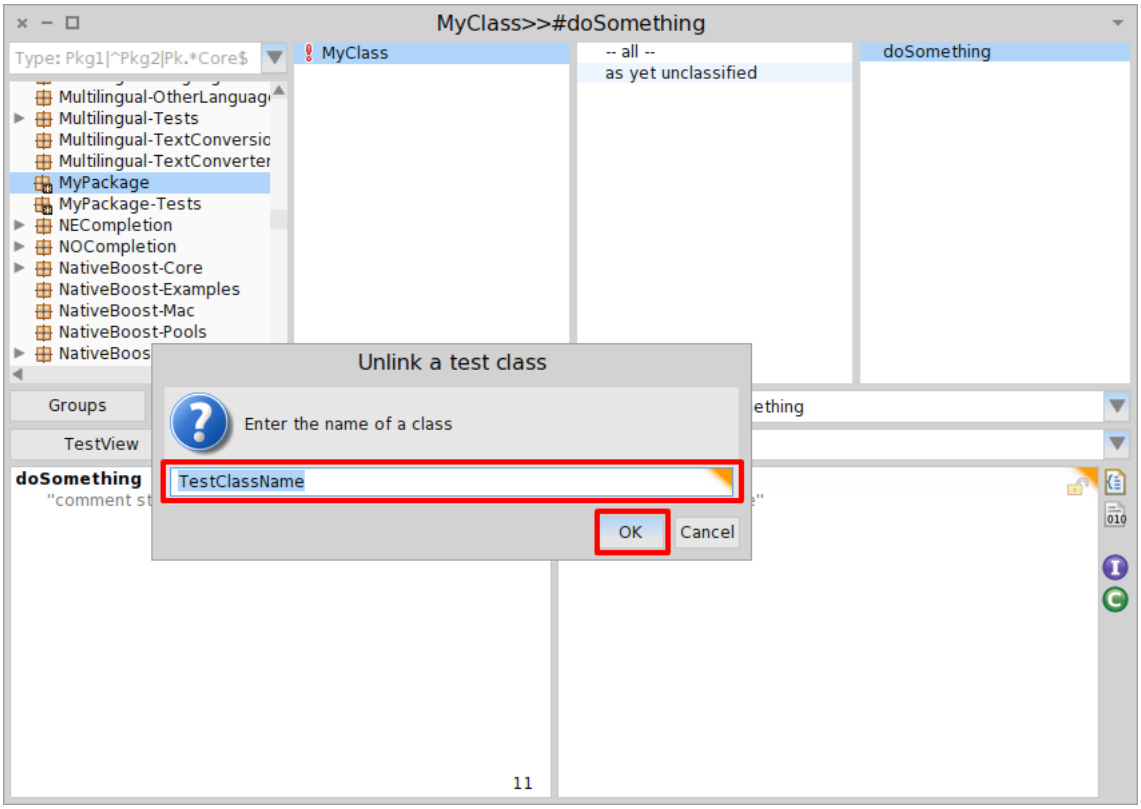

Figure 8.15: Pop-up asking for the test class name

4. The specified test class is now removed from consideration. All tests contained in this class will not be considered to be tests for the currently selected class. You can verify if this now works by opening the found tests drop down list. 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 drop down list 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 if necessary change the first line to “[YourClassName] subclass: #TestCase”. Check again if the tests from your test classes are found now by selecting the method for which some tests were not found.

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 identified 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 drop down list.

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

8.1.7.3 Adding and removing specific found tests

Both adding and removing an individual test from the tests that the TVPlugin finds are features that are directly supported. It is still possible to make these adjustments although 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 solely 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 additional 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. The “:” character from methods names with parameters will be ignored when looking for a test.

Original method name	*Test names*	*Not test names*
addDays:	testAddDays	addDays, testAddDay
asFloat	testFractionAsFloat, testFractionAsFloat2	testFloatAsFraction
print24:on:	testPrint24On, testPrint24OnWithPM	testPrint24withNanos
attest	attestTest, testAttest	attest

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 Nautilus class hierarchy selected method. This time the “:” characters are not removed from the check. If the supposed test method of a method named “do:on:” does call “doOn” then it would 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 for 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] Roberto Minelli, Andrea Mocci, and Michele Lanza. Measuring navigation efficiency in the ide. In *Proceedings of IWESEP 2016 (7^{th} IEEE International Workshop on Empirical Software Engineering in Practice)*, page to appear, 2016.

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

[8] 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.

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