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. These functionalities include a way to check if a method is untested, a convenient way to see all tests that have been written for a certain method, a quick way to add new tests to existing methods and the ability to view a method and a corresponding test that is currently being written 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

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.

1.1 Unit Testing

Unit testing follows the paradigm of isolating the smallest inseparable parts of a program and testing them independent 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 a whole implementation. By making automated unit tests one ensures that the tested method behave as expected and also speeds up development since these automated test are very fast to execute.

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 do this an instance of the object providing the method 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 set postconditions are fulfilled. To simplify the setup of those preconditions and to keep the tests independent mock objects are often used to emulate the behavior of other objects besides the object containing the tested method. That way it can be avoided having to fully prepare these other objects and since these mocks don't execute the real code one fault in the actual implementation will not cause a chain reaction of unsuccessful tests. An example of a post condition could be the returned value of the method or the state in which the instance variable of the object are after the method call. In summary it is to note that unit testing often happens at a method level.

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 focused nature of unit testing requires many tests to cover a whole implementation. 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 each path on the decision tree of the method should be covered. In the following we will take a look at some testing paradigms that can be combined with unit testing.

1.2 Test-driven Development

In Test-driven Development the tests for an application are written before the application itself. Afterwards the application is implemented until all these tests are satisfied. It is possible to add, change or remove test during the implementation but the basic idea is to carefully plan out the requirements first. While it might seem tedious this planning is also 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 tests.

As shown in section 1.1 it is fair to look at testing on a method level when discussing unit testing. Since in Test-driven Development the methods have not been written when the test are created no data can be extracted from the methods to help during the creation of the tests. On the other hand it would be possible to gather data from the tests to help while implementing the methods but this will not be discussed further here due to it not facilitating unit testing. 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 differently. Namely since it is not possible to gather data from the implemented methods to facilitate writing tests when doing Test-driven Development.

1.3 Blackbox and Whitebox Testing

Whitebox and blackbox testing determines how closely the tests are made to suit the method. In whitebox testing one is completely aware of the method 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 the method. Especially interesting are worst case scenarios and method invocations that result in as much executed lines as possible. Contrary in black testing the programmer does not know the inner workings of the method being tested. 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 values it is necessary to choose the critical input values where an error might occur.

For the purpose of this work it is important to remember that depending on which one of those paradigms is used it is necessary to have in depth knowledge of the method being tested in order to write a test for it. Namely while doing whitebox testing. On the other hand when doing blackbox testing it is very much discouraged to look at the implemented method, provided it has already been written.

1.4 The Nautilus Plugin Framework

2 The Problem

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 check 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 1. Namely unit testing necessitates to see if a method is covered by at least one tests, 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 original method and the test at the same time but for the sake of blackbox testing this feature has to be optional. Since not many environments were created with specifically unit testing in mind there are often some of these features missing. Concrete examples for environments lacking these specific features will be pointed out in chapter 3. To compensate for this the user is required to manually execute many tasks like navigating back and forth between method and test and creating new classes and packages for new tests. 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 which will help understanding the problem of those missing functionalities and exactly at which point they are needed.

In unit testing it is important that each method is tested. As discussed in **??** a method with no tests is unsafe and has 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 the user has 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 letting the user run test suits and then see which part of the implementation were executed. This also will tell if a method never got called when running a certain test suit and this is untested.

Also resulting from unit testing is the requirement to look up a method and all the tests that have been written for it and not just if a method has tests at all. Just checking if a method is tested at all is not enough, the user wants to see if the method is tested sufficiently. To ensure this multiple test methods are required. Similar to showing if a method is untested this helps the user to decide if it is necessary to write a new test or in this case if a very similar test has already been created. Further a user could study how a method works by locking at the corresponding tests. Since this test search relies strongly on the relation between tests and method code coverage tools provide less help. 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 as mentioned before 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. It is quite easy to see that if a search can return all tests to a method then it can also decide if a method is tested or untested. 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 above. The ability to see if a method is untested will thus no longer be discussed separately here. 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 very tedious and new testers will have trouble getting an overview of the existing test suit.

As discussed previously it should be as simple as possible to get indications if a new test is needed. Almost as a direct consequence the user has to create these new tests when a method seems undertested. 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 less tests. While creating a new test the user first has to decide in which package and class the new test will be added. In many cases both of these containers have already been decided on since other tests corresponding to the same method are already there. If there are no test containers or if the existing test containers do not fit the new test then the user has to create new containers. In this case the test package name and the test class name are often derived from the original package and class names. Both of those derivations should be set as default values for the new class and package to possibly save some time. A slight drawback of these default names is that the method that is being tested has to be defined previously. Especially when using Test-driven Development the programming environment can not easily make such name derivations. To make this approach work for Test-driven Development one would have to do the planning and then create all the needed methods with an empty body. From there on it would be possible to do test driven development while receiving default names. Conclusively it does not matter if the containers already exist or if they need to be added either way it is possible to support the user during the creation of new tests. This can be done through a fast way to specify existing containers and giving default names for new containers. While it helps to have these aids a lack of them seems not as bad as a lack of the test search functionalities described previously in this chapter.

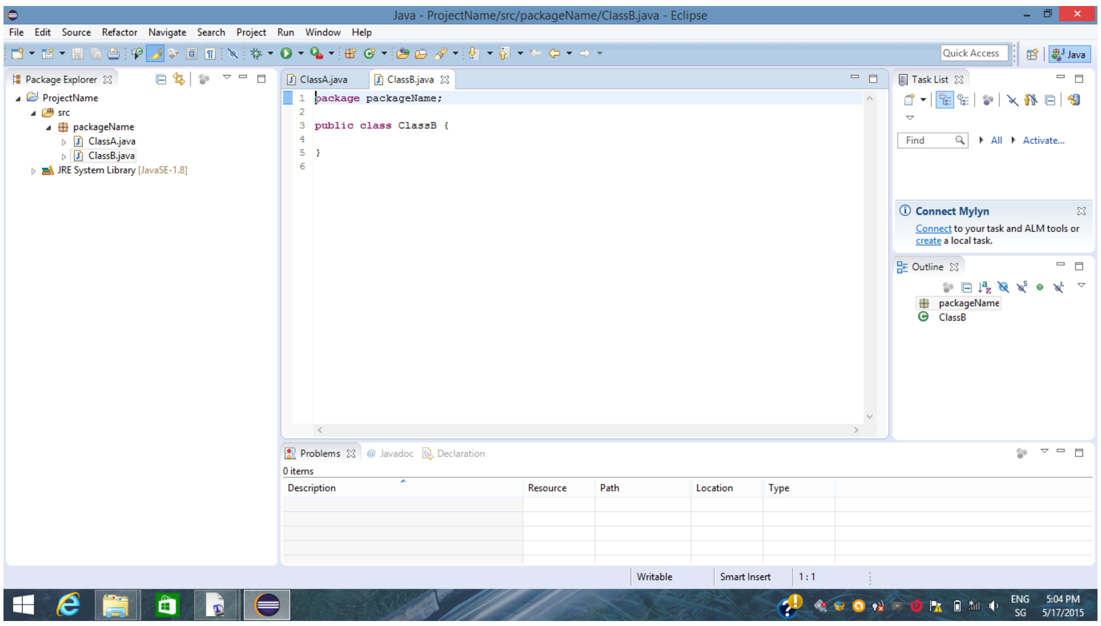
Resulting from whitebox testing is the need know a method's inner workings while testing. It is expected that the programmer who writes the test is trying to test every line of code. Thus it is important to provide the user with information on how the method works. Ways to do this include writing notes or showing selected sections of the tested method but the possibly simplest way is to let the users see the whole method in question at the same time as they are writing the test for it. In Black-box Testing this feature can be counterproductive. If not desired the additional information can spoil blackbox testing and even if completely ignored it still takes up space on the screen. Blackbox testing thus requires that this feature can be turned off. While this feature is not very important in blackbox testing it helps during whitebox testing. Due to the simplicity and effectiveness of the solution we will focus on letting the user see the method under test and the test that is being written at the same time. It is important not only to provide this functionality but also to optimize it. If no information about a method is provided by the programming environment the users have to make notes for themselves, look things up in design documents or resort to additional analysis tools. This can be very tedious and take a lot of time. By making it unnecessary to switch between gathering information about a method, writing tests for it and looking up the same information again a great deal of repetitive manual tasks that the user has to execute becomes obsolete.

These basic features that a unit test friendly environment should have will be used in the next chapter as a starting point to critically analyze existing environments. If some of these are missing then the user often might have to do simple but also quite repetitive tasks to compensate. This increases the time and effort needed to create unit tests and thus make writing tests a chore. By automating those minor tasks on the other hand various use cases involving unit tests can be simplified. In turn the user is encouraged to write tests what will hopefully increase the quality of the software that is being written.

3
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 2. 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 then be compared and possible places for improvement will be discussed.

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 the picture below you see a Java project opened in Eclipse with a package and two classes.

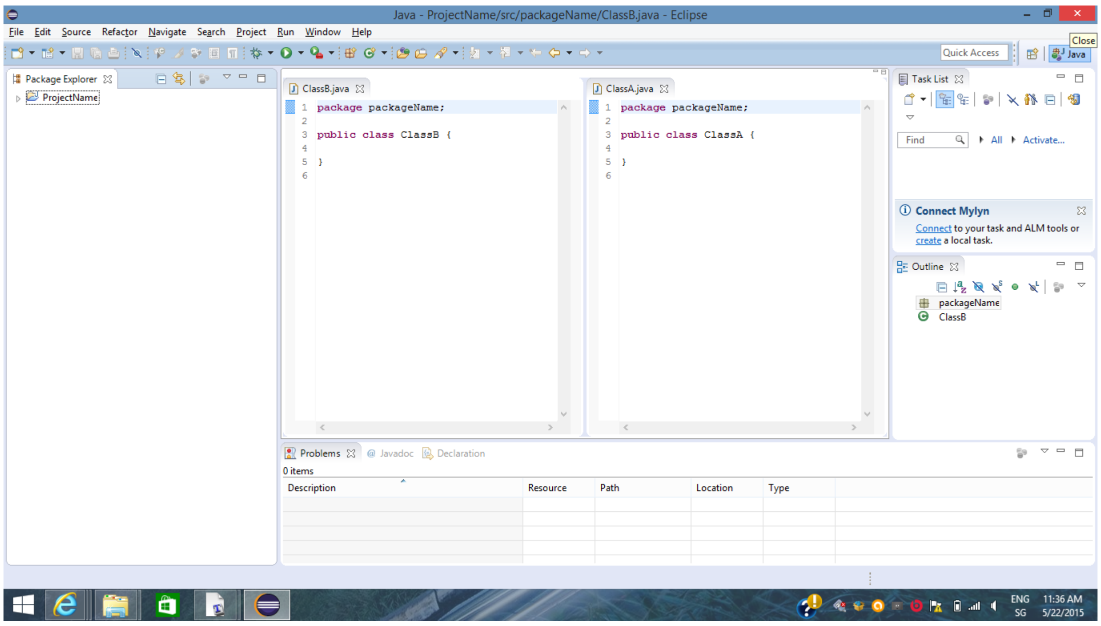

Java - ProjectName/src/packageName/ClassB.java - Eclipse

```
package packageName;

public class ClassB {

}
```

Let us start with the discussion of how the users can see the tests that they are currently writing and the methods that are being tested at the same time in Eclipse. 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. Classes that are opened through the Package Explorer or otherwise will then be shown in the middle of the screen and a new tab on top of the code editor will appear. Unless you close these tabs all the classes you opened last will be quickly available through their tabs. This is similar to how many Internet browsers work and is easily understood by most through this resemblance. 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 when creating 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 on the picture below.



```
package packageName;

public class ClassB {

}
```

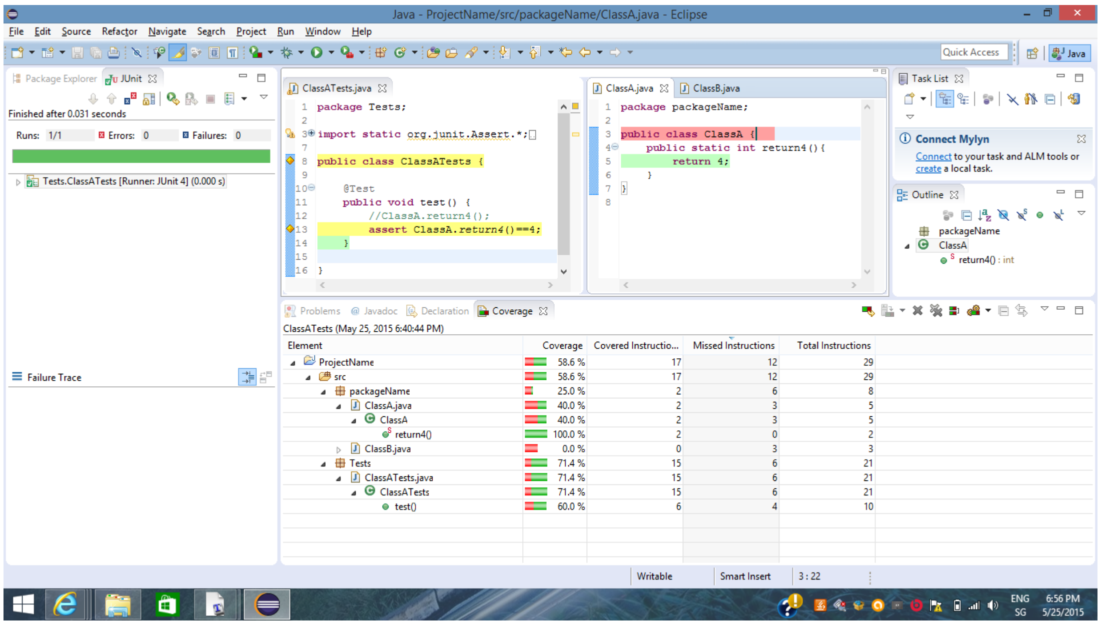
```
package packageName;

public class ClassA {

}
```

As shown while writing the test Eclipse provides multiple ways to make information about the tested method easily accessible.

Another feature we discussed in the previous chapter was the ability to find all tests to a certain method. In Eclipse this functionality is not included but something similar can be added through various test coverage plugins. As mentioned a code coverage tool can provide a similar functionality. An example of a code coverage plugin for Eclipse is EclEmma. When run on a class containing unit tests the coverage of these tests will be computed. With this it is possible to see if a specific method never was executed while the tests were running and thus is untested. Disadvantages of this are that in the test the method under question does not have to be called directly. This can lead to tests covering methods they are not intended to. It also is difficult to tell how many tests for a method have been written depending on how much information the code coverage tool gathers and displays. So while code coverage tools can help managing unit tests its use is limited. In the end a convenient way to find all tests to a given method is missing in Eclipse.



```
package Tests;

import static org.junit.Assert.*;

public class ClassATests {

    @Test
    public void test() {
        //ClassA.return4();
        assert ClassA.return4()==4;
    }

}
```

```
package packageName;

public class ClassA {
    public static int return4(){
        return 4;
    }
}
```

ClassATests (May 25, 2015 6:40:44 PM)

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
ProjectName	58.6 %	17	12	29
src	58.6 %	17	12	29
packageName	25.0 %	2	6	8
ClassA.java	40.0 %	2	3	5
ClassA	40.0 %	2	3	5
return4()	100.0 %	2	0	2
ClassB.java	0.0 %	0	3	3
Tests	71.4 %	15	6	21
ClassATests.java	71.4 %	15	6	21
ClassATests	71.4 %	15	6	21
test()	60.0 %	6	4	10

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 will 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 better support the user. In the newly open wizard a default name is already created and the selected class is put into the ”Class under test” field. As default package the package of the class under test is used but it can be changed. In this wizard are also other options to create method stubs for standard methods used in JUnit tests which might be useful for the new tests. By clicking on the ”Next” button it is possible to select each method of the class under test and appropriate test stubs. On one hand this is very good for creating new test classes but on the other hand Eclipse does not provide a way to add new test method stubs in an already existing test class. Another problem is that it is not possible to create through the wizard more than one method stub per method in the class under test. Eclipse makes it easy for the user to create new test classes but the creation of new tests in existing classes leaves something to be desired.

Although Eclipse is not absolutely optimized for unit testing it is very well sufficient. Eclipse's biggest strengths are the ability to keep recently opened methods as tabs as well as splitting the code editor horizontally and vertically. This is very good for keeping details about a method's implementation close to the user and thus supports whitebox testing. The biggest drawback is the lack of a test search functionality.

Now let us take a look at how Pharo and its system browser Nautilus compare to this. It is worth mentioning that Pharo 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 supported in many ways.

As with Eclipse the first functionality we will look at is the ability to view tests and methods side by side. Pharo 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 on the left hand side of each Nautilus window. There also exists the option to lock a method and then select a different one inside the same Nautilus window without opening a new one. The users can then see both methods at the same time in the code editor or each individually depending on what they choose. At the moment though there exist problems with this locking mechanism which can lead to multiple code editors being displayed over each other. This makes the method lock function hard to use. Apart from this Nautilus also has a History Navigator which allows fast access to all recently viewed methods and classes. This can be used to switch between method and test in 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 Pharo there are various things a user can do to view multiple pieces of code at the same time but each of those features has more or less severe drawbacks. Compared to Eclipse this functionality is definitively lacking. The fact that this feature is missing or only partially working in Nautilus was a major inspiration for this paper. Whitebox testing can become very hard without this feature.

The next functionality discussed is how Pharo finds existing tests to a specified method. Contrary to Eclipse Nautilus has some sort of test search implemented. It is very limited thought and only finds tests that are placed in classes with a very specific name and have a very specific name themselves. 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”. This also leads to the problem that effectively only one test per method will be found since others with a slightly different name will not fulfill these naming criteria. While this is very restrictive it allows Nautilus to add a button to each method where tests have been found which can be pressed to execute the test and shows if this corresponding test was successful. Similar to Eclipse code coverage tools can be used to determine if a method is untested. With a more elaborate test search Nautilus would improve by a lot since the execution of associated test is very easy. Sadly like Eclipse it lacks a sufficient, inbuilt function for this.

The creation of new tests to a method is slightly easier than in Eclipse but has certain drawbacks. Similarly to Eclipse with a right-click on a class the option ”Generate test and jump” is given. An improvement compared to Eclipse is that the new test will automatically be add to a certain test class in a specific package (both of which will be created if needed). Using these predefined locations is a bit faster but robs the user of the ability to specify different containers. An advantage is that tests created in this manner confirm to the naming standards imposed by Nautilus. Similarly as in Eclipse only one test per method can be created in this way. Eclipse's and Nautilus' way of adding new tests are very comparable. A case could be made for both versions of this feature.

4
The TestView Plugin

Having discussed some important features of a unit test friendly environment and how their implementation is often lacking we will now try to present a solution to this problem: the TestView Plugin. Since the lack of the features can not be corrected for all environments at once we chose to extend Natilus' functionality. The reasoning behind this was that the inability to view tests and methods side by side in Pharo in an efficient way was perceived by us as the gravest of these issues and that Eclipse already has a big community which provides many plugins. Also since I have already done a plugin for Eclipse I out of curiosity wanted to see how this is done in another environment. Although it is not necessary it is recommended here that the reader quickly familiarizes themselves with the TestView plugin. Following the user guide provided in chapter 8 might be the easiest way to do this. 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 Pharo.

As stated one of the bigger drawbacks concerning unit testing that Nautilus has is that it is hard to view tests and methods at the same time. Whitebox testing becomes very cumbersome through this. Seeing both method and test together should be quick and not introduce too much redundancy that might clutter the environment. Following this the TestView Plugin allows to split the code editor panel of the Nautilus window vertically into two parts. With this approach the unneeded redundancy of opening two Nautilus windows is eliminated.The file hierarchy for example will not be displayed a second time and less screen space will be occupied. Unlike in Eclipse the user has no absolute control over this additional code editor but on the other hand only methods of interest are shown there and the user does not have to manage what should be displayed there. 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 test method. The user might still choose to open two Nautilus windows due to the need to view methods together that are not method and corresponding test. The TestView 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. Compared to Nautilus' original functionality this is a notable improvement and compared to Eclipse it requires less work to set up.

As a direct consequence of the heavily selected content of the second code editor it became necessary to implement the functionality to search for tests to a corresponding method. Without this it would be impossible to show a relevant selection. The improvements compared to Nautilus' test search are less restrictive criteria to classify as a corresponding test, that one method can correspond to multiple tests and that the results are customizable on a class level. The user has the ability to choose which element of this search result should be shown in the right code editor. This new search yields better results than Nautilus' search and the customization allows to link or unlink test classes whose tests will then be added to the results. This allows the user to use the TestView Plugin even with projects that do not conform to the criteria of the test search described in chapter 8. This is especially important to let the user correct the almost inevitably faulty test search if needed. As stated when provided with a list of all corresponding tests the user can quickly decide if a method is untested. If this is the case then the returned list of found test will be empty. Compared to both Eclipse and Pharo this test search should be a significant improvement. Both are lacking this feature or provide only a very basic version of it.

5
The Validation

In which you show how well the solution works.

6 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.

7 Anleitung zu wissenschaftlichen Arbeiten

7.1 Installation and activation

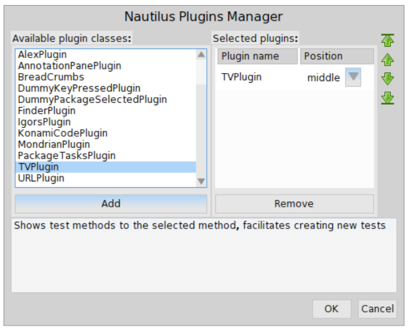
To install and active the TVPlugin***BS:***what's a TVPlugin? follow the steps listed bellow:

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

   ```
   \brs{Something missing here?}
   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. Here you click on "TVPlugin" under "Available plugin classes" and then press on the "Add" button. In the "Selected plugins" column you can specify where most visual elements will be shown in your Nautilus windows. Click "Ok" to confirm.



Nautilus Plugins Manager

Available plugin classes:

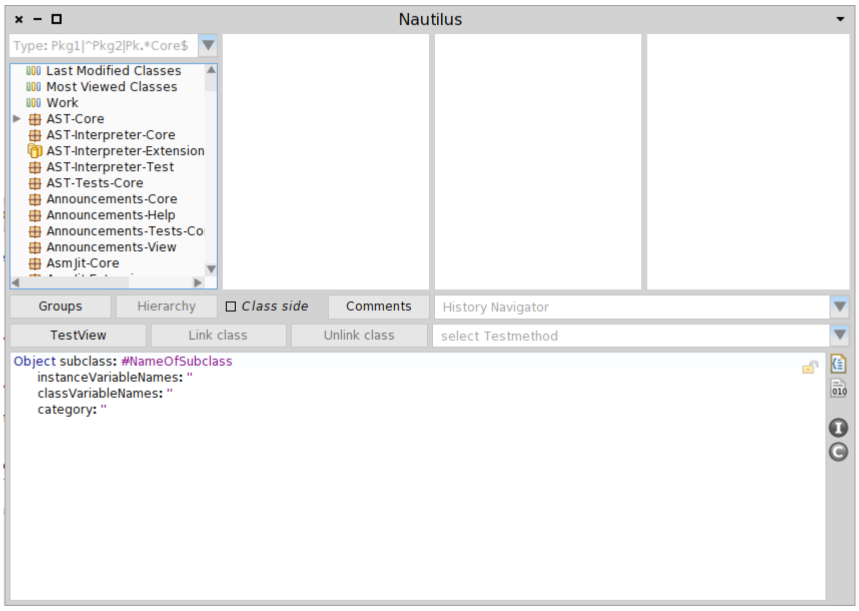
AlexPlugin
AnnotationPanePlugin
BreadCrumbs
DummyKeyPressedPlugin
DummyPackageSelectedPlugin
FinderPlugin
IgorsPlugin
KonamiCodePlugin
MondrianPlugin
PackageTasksPlugin
TVPlugin
URLPlugin

Selected plugins:

Plugin name	Position
TVPlugin	middle

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

3. When you open a Nautilus*BS:*what's a nautilus? window from now on the plugin will be started until you remove it again using the Nautilus Plugins Manager. To verify if the plugin is activated check if this row is displayed in the position you selected.



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

7.2 Basic Components

*BS:*if you are talking about GUI components say so in section title, components can be anythingHere a quick overview of all the visiual elements can be found*BS:*passive again. The TVPlugin adds 3 buttons and one droplist to your Nautilus windows. The "TestView" button toggles the search for test methods to the currently selected method. If it is toggled on a second source code editor will appear at the bottom where the test methods will be displayed.*BS:*pic, or did not happen This list will change whenever a new method is selected in the Nautilus window. The initial list shown in the droplist*BS:*the what? is composed of tests found through the search algorithm described under 7.3 The search Algorithm*BS:*use autoref in stead of ref and nameref. The additional buttons "Link Class" and "Unlink Class" allow to customise this list if the search for any reason did not find some tests.

7.3 The search Algorithm

In this section I will provide a detailed explanation how corresponding tests to a certain method are found. It is a hierarchical search with two stages.

In the first stage all test classes are determined*BS:*what does this even mean?. For this all classes in the your environment are taken into consideration in the beginning. By the end of the first stage only those classes that inherit from "TestCase" as well as pass a substring search *BS:*the what now? in the class name are then passed over to the next step. The substring search requires the name of the class in question to contain both "test" and the name of the selected class.*BS:*oh... well, say this up front, don't make me guess what you mean by sub-string search If those two conditions are met then the class in question is *BS:*is it really? how can you be so sure? if you are not 100% sure, you need to say "We consider it" or something similar. a test class of the selected class. The substring search is not case sensitive and the matches for "test" as well as the selected method can't overlap, meaning one letter can only be used to match partially either "test" or the selected class name.***BS:***split the two conditions in two separate subsections or paragraphs and explain them in detail with either code, block diagram algorithms, math or something to help the reader understand

Original class name	Possible class names	Not a test class name
String	StringTest, stringtest, TestString	String, Test, StrinTgEST
Protest	ProtestTest	Protest

BS:All tables, figures, listings, etc. MUST be numbered, named, and mentioned + refereed to in the text.

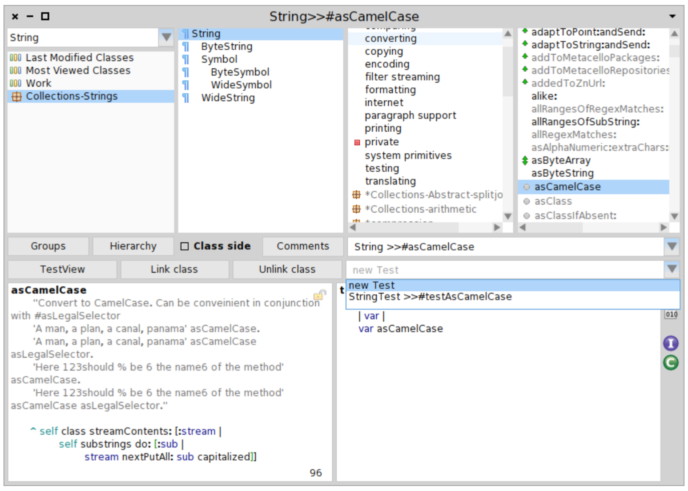
In the second stage all test methods corresponding to the selected method out of all methods of the test classes are determined ***BS:***determining is not the right term for this, at least not in this phrasing. . Similarly as before the test method name needs to contain the name of the selected method in addition to "test". The second criterion is if the method in question uses the selector of the selected method. ***BS:***you have like 4 or 5 different steps in this process and they are all divided in to pairs for some reason and all of them are 'in the second part, the second criteria etc. very confusing Unlike stage one this stage***BS:***name the stages, this usually helps is fairly inclusive in that if only one of those criteria is satisfied it still qualifies as a test to the selected method.

The two criteria of the second stage also serve to order the found test methods. The ones displayed on top satisfy both criteria. The ones that only satisfy the naming requirement are shown below these and the last ones are those that only contain the selector of the selected method.

7.4 Creating new Tests

The other main functionality of the plugin besides finding tests is to facilitate creating them. When you have a method open in the first code editor to which you would like to add a test select the "new Test" option from the droplist***BS:***run on sentence. This option is an exception to the order described at the end of 7.3 The search Algorithm and is always shown on top of the list. This option also can be found there regardless whether there were any tests found for the selected method. A template will appear in the second editor where the user can write the new test. When the test is accepted the user will be asked to name the class in which this test will be added. If the specified class doesn't exist it will be created and the user will be asked for a package to place it in.

A faster way to create a new test is to select an existing test from the droplist and alter it. By selecting a new name the old test will not be overwritten and the new test will be added to the class of the old test. Any test that was written using the second editor will result in its class being linked to the class of the selected method. This makes sure that the class in which the test was added is later recognized by the search algorithm even if its name does not confirm to the naming of a test class.



```
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]]
```

```
t
	| var |
	var asCamelCase
```

8
User Guide