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 enviroment by making it more unit test friendly. Instead of writing a new system browser we chose to realise this in the form of a Nautilus plugin since this makes it easy to set up and builds on an established part of the Pharo enviroment. Our plugin's functions includes making it easier to creating new tests for a given method. Specifically being able to see a method and the test that is currently being written side by side and a fast way to create new tests with if necessary new test classes and packages. The second provided functionality is a search for existing tests corresponding to the currently selected method. The search takes in to account if the methods are similarly named, if the supposed test method is a subclass of TestCase and if it contains the selector of the original method. The user can then add and remove elements from the resulting collection of tests to ensure that only acctual tests are shown. These features not only facilitate unit testing but can also be used to help understanding methods by looking at their tests.

Contents

1 Introduction

1.1 Unit Testing

The TestView plugin was designed with this approach to testing in mind. Unit testing follows the paradigm of isolation the smallest inseparable part of a programm and testing it. In smalltalk these units are often methods since the language heavily relies on sending messages between object. Also implied with this approach is that the programm is already written when the tests are written. The plugin helps during the creation of each test since the user can easily view the the tested method as well as the test that is being written. Further it creates templates that the user can alter to make writting tests faster. I also helps finding existing test and thus finding methods without tests. This functionaly however is at the moment not automated and the user will have to select method after method to find those without tests.

1. Designed with this in mind
2. Method first, test later
3. Looking at both method and test being written
4. Facilitate creating tests

1.2 Test-driven Development

1. Hard to do with this plugin
2. Test first, method later
3. Search mechanism searches tests for methods
4. When creating a test it can not be linked to a class since the class doesn't exist yet
5. Helps adapting tests to methods if requirements change or during development problems with the orignal tests show up

1.3 The Nautilus Plugin Framework

2
Related Work

In which we learn what have other done to address similar problems. For example, the work of Star [**?**]
eclipse
-easy to view multiple methods at once
-no automated testsearch
Nautilus/pharo
-automated testsearch is a bit lacking but exists (doesn't support multiple tests for one method¡-check if true)
-no easy way to view multiple methods
–history droplist can't be used when user klicks on too many classes/methods
–could open another Nautilus window but dublicate package tree takes up a lot of space
–best solution so far: lock method, then navigate to testmethod or write new test (but multipleMethodEditor is kinda broken)

3
The Problem

As seen in the examples many current development environments are not optimized for unit testing. The search for unit tests to a specific method is often lacking meaning not present or simplistic and not able to recognize multiple tests for one method. Important features for unit test friendly environments are automated test searching, the ability to view tests and methods side by side and the ability to easily create new tests. Since not many environments were created with specificaly unit testing in mind there are often some of these features missing. To compensate for this the user is required to manualy execute many tasks like for example 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 writting tests.

Taking a closer look at the tasks involved in different goals will help understanding the problem of those missing functionalities and exactly at which point they are needed. A very important feature for unit testing is the ability to see which methods are currently untested. This has to be as easy to see as possible to be really beneficial. The user should be able to determine with one glance which methods have no tests. If this feature is not implemented the user has to switch back and forth between a class and all its test classes to check. If the test classes are not known to them they even have to find them first. Both of those tasks can take alot of effort.

Similarly the user might want too see all the tests that have been written for a method and not just if it has tests at all. This not only helps the user decide if it is necessary to write a specific test or if a very similar test has already been created but also lets them view how methods work by looking at all their tests.

Another reoccuring goal is the creation of new test 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 it is already clear where the test will be located since other tests corresponding to the same method are already there. If there are no already existing tests or if for any reason the existing test classes do not fit the new test then the user has to create a new test. In this case the test package name and the test class name are often derived from the original package and class. Both of those should be set as default values for the new class and package to save some time. When writting the actual code for the test it is beneficial for

the user to be able to see test and method side by side. With this it the user is able to easily do white-box testing. For black-box testing this feature is less important but if it can be easily turned off it should not interfere. Similarly as above this helps preventing the user switching back and forth between methods.

These examples show where problems can occour that hinder the user while doing unit testing. They increase the time and effort needed and to create unit tests and thus make writing tests a chore. By facilitating this process on the other hand the user is encouraged to write tests and hopefully increase the quality of the software that is being written.

4 The TestView Plugin

In this chapter the TestView plugin is explainded. The first section will show how to install the plugin while the following ones will go over what the visual components do and explain the inner workings of the plugin, namely how test are found and how they can be created using the plugin.

4.1 Installation and activation

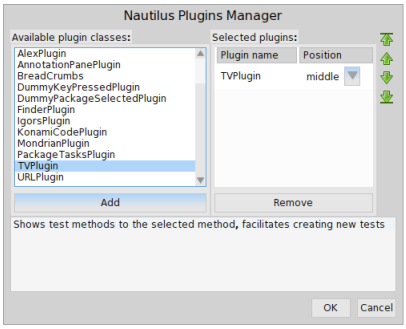
To install and active the TVPlugin follow the steps listed bellow:

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

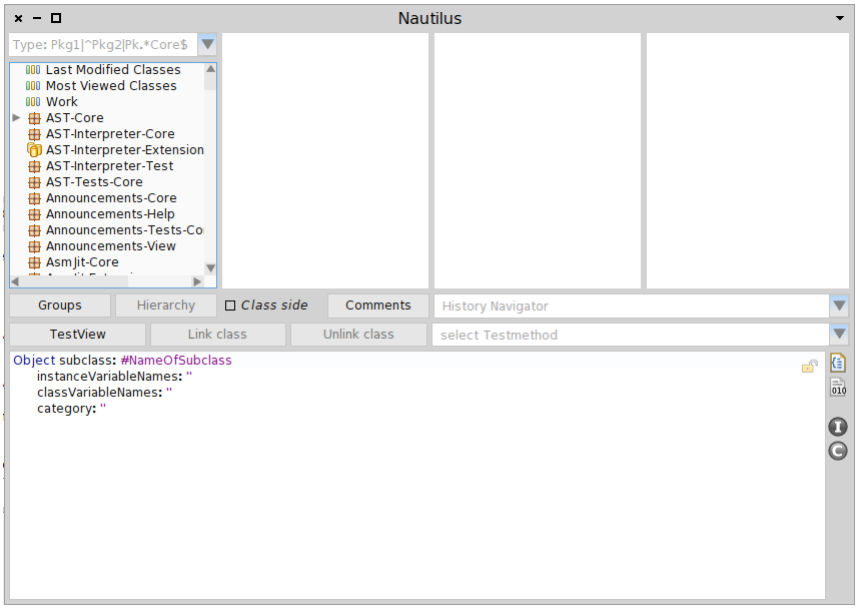
```
url: 'http://smalltalkhub.com/mc/DominicSina/TestView/main';
package: 'ConfigurationOfTestView';
load.
(Smalltalk at: #ConfigurationOfTestView) loadDevelopment.
NautilusPluginManager new openInWorld
```

   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” collumn you can specify where most visual elements will be shown in your Nautilus windows. Click ”Ok” to confirm.



3. When you open 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.



4.2 Basic Components

Here a quick overview of all the visiual elements can be found. 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. This list will change whenever a new method is selected in the Nautilus window. The initial list shown in the droplist is composed of tests found through the search algorithm described under 4.3 The search Algorithm. The additional buttons ”Link Class” and ”Unlink Class” allow to customise this list if the search for any reason did not find some tests.

4.3 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. For this all classes in the your enviroment 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 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. If those two conditions are met then the class in question is 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.

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

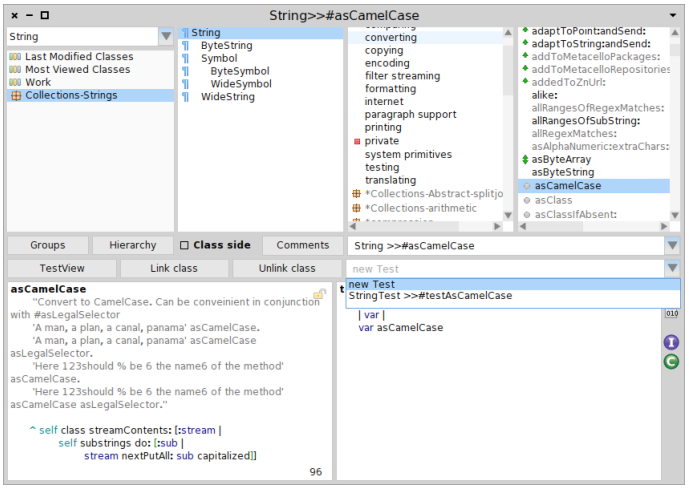
In the second stage all test methods corresponding to the selected method out of all methods of the test classes are determined. Similarly as before the test method name needs to contain the name of the selected method in addition to ”test”. The second criterion is if the method in question uses the selector of the selected method. Unlike stage one this stage 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 satify the naming requirement are shown below these and the last ones are those that only contain the selector of the selected method.

4.4 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. This option is an exception to the order discribed at the end of 4.3 The search Algorithm and is always shown on top of the list. This option also can be found there regardless wheter 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 recognised by the search algorithm even if its name does not confirm to the naming of a test class.


String>>#asCamelCase
String
Last Modified Classes
Most Viewed Classes
Work
Collections-Strings
String
ByteString
Symbol
ByteSymbol
WideSymbol
WideString
converting
copying
encoding
filter streaming
formatting
internet
paragraph support
printing
private
system primitives
testing
translating
*Collections-Abstract-splitjo
*Collections-arithmetic
adaptToPoint:andSend:
adaptToString:andSend:
addToMetacelloPackages:
addToMetacelloRepositories
addedToZnUrl:
alike:
allRangesOfRegexMatches:
allRangesOfSubString:
allRegexMatches:
asAlphaNumericExtraChars:
asByteArray
asByteString
asCamelCase
asClass
asClassIfAbsent:
Groups
Hierarchy
Class side
Comments
String >>#asCamelCase
TestView
Link class
Unlink class
new Test
new Test
StringTest >>#testAsCamelCase
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]]
96
| var |
var asCamelCase

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

This consists of additional documentation, e.g. a tutorial, user guide etc. Required by the Informatik regulation.