VAASAN AMMATTIKORKEAKOULU
VASA YRKESHÖGSKOLA
UNIVERSITY OF APPLIED SCIENCES

Milosz Mazurkiewicz

# GUI TEST AUTOMATION WITH SWTBOT

Technology and Communication
2010

VAASAN AMMATTIKORKEAKOULU

UNIVERSITY OF APPLIED SCIENCES

Degree Programme of Information Technology

**ABSTRACT**

| | |
|---|---|
| Author | Milosz Mazurkiewicz |
| Title | GUI Test Automation with SWTBot. |
| Year | 2010 |
| Language | English |
| Pages | 50 + 3 Appendices |
| Name of Supervisor | Dr Smail Menani (VAMK), Jakub Drzazga (NSN) |

In this thesis the author presents theoretical background of GUI test automation as well as technologies, tools and methodologies required to fully understand the test program written in SWTBot. Practical part of the thesis was to implement a program testing File Menu options of Pegasus RCP application developed in Nokia Siemens Networks.

Concluding this dissertation, in the author's opinion test programs written using SWTBot are relatively easy to read and intuitive for people familiar with basic concepts of Eclipse RCP. Without any doubt SWTBot's API is large enough to test effectively such complex application as Pegasus RCP. The biggest problem could be lack of good documentation and tutorials which makes it difficult to start with. The best and most up to date is Javadoc that comes with SWTBot's code. Therefore a developer which wants to start writing SWTBot tests has to spend some time with it to get familiar with the API experimenting with the code. Although SWTBot is not a commercial tool, the growing community of testers and developers using it seems promising considering the need of support. Despite of what is published on SWTBot's home page, SWTBot lacks a recorder which implicates that the tester has to be a programmer. As the analysis has shown the automated tests execute over twenty times faster comparing with manual tests.

Pieces of code presented in this document, instructions how to achieve tests independence, error recovery, and test synchronization, together with big test program implementation (APPENDIX 2) make up a great tutorial which will help developers to get familiar with SWTBot's capabilities. FileMenuTest.java consists of 30 test cases and has almost 1500 lines of code. The author has made those tests compliant with all the objectives for well-designed test cases. The program was integrated with the nightly builds of CruiseControl, which means the author's work is utilised on a daily basis.

Keywords SWTBot, software testing, GUI testing, test automation

**CONTENTS**

## APPENDICES

## MARKINGS AND ABBREVIATIONS

| | |
|---|---|
| SWTBot | SWTBot is an open-source Java based GUI testing tool for SWT and Eclipse based applications. |
| SWT | Standard Widget Toolkit is an open-source graphical widget toolkit for Java platform (alternative for AWT and Swing). |
| Eclipse RCP | Eclipse Rich Client Platform is a framework to build Java applications that have look-and-feel of native applications. It utilises Equinox OSGI, plug-in architecture, SWT and JFace. |
| Pegasus RCP | Pegasus test automation framework is NSN's product for all kinds of tests: regression, reliability, stability, protocol tests, performance measurements. Pegasus is used mostly in telecommunication branch. It is the AUT in this thesis work. |
| JUnit | JUnit is a unit testing framework for the Java programming language. |
| Agile | Agile software development refers to a group of software development methodologies which follow Agile Manifesto. |
| Scrum | Scrum is a popular Agile methodology. It is an iterative incremental framework for managing software projects. |
| record/playback | A mechanism that allows recording user actions and replaying them. Can also be referred to as capture/replay. |
| bug | A defect that can make a program to crash, give unexpected output or just causes inconvenience to the user. |
| AUT | Application Under Test. The application, product, subsystem, or component that is being tested. |
| test case | The smallest unit of a test. A subroutine that tests a single aspect (or function) of the AUT. |
| GUI | Graphical User Interface |
| NSN | Nokia Siemens Networks |

# 1 INTRODUCTION

## 1.1 Rationale and Background of the Project

The topic of this dissertation: **GUI Test Automation with SWTBot** was a natural choice for the author since he had to become SWTBot specialist for Pegasus project held in Nokia Siemens Networks Software Development Centre in Wroclaw. The purpose of automating GUI tests was to increase the code coverage for parts of the code which could not be tested with JUnit (user interface), and therefore increasing the quality of the developed software. SWTBot as an open source, Java based functional testing tool targeted for SWT and Eclipse based applications seemed the best choice for this task.

## 1.2 Aims and Restrictions

Testing is an invincible part of software development process, especially when using Agile methodologies. In this thesis the author introduces theoretical background of test automation as well as technologies, tools and methodologies used in the project. Practical part of the thesis was to implement a program testing File Menu options of Pegasus application. The program was later integrated with the nightly builds of CruiseControl, which means the author’s work is utilised on a daily basis. The thesis document itself was already used as a guide for Pegasus team members and other teams in NSN considering using SWTBot in their projects. SWTBot is still an incubation project and this itself makes it challenging and innovative to work with. Deeper analysis and calculating return on investment for the test automation project was not possible because the author was not allowed to disclose any financial related data about NSN.

## 1.3 Materials for Thesis Writing

Literature for thesis writing consists of latest publications available in the field of GUI test automation. Materials used range from books and articles written by greatest authorities in the field of test automation, through professional periodicals, to projects’ official home pages. Writing test program required mostly studying SWTBot’s Javadoc and a lot of hands-on experience.

# 2 THEORY OF GUI TEST AUTOMATION

## 2.1 Introduction and Background

Modern software applications become more and more complex. This applies also to graphical user interfaces (GUIs) being implemented, offering enhanced ease of use, and of course higher profit to the vendor. It seems to be obvious that the GUI code should be tested. First commercial GUI test tools started to appear in the late 1980s. They provided possibility to write test scripts manipulating the graphical user interface identically as a human would use it.

Since then many software companies tried to apply different GUI test tools in their projects. The rationale behind it was usually the same: because manual testing is costly, therefore we should focus our effort on automating the tests so that they could be run repeatedly, which should give more time to test new features instead of retesting the old code. But without proper tools and - even more importantly – proper methodology for designing and building GUI tests suites, the investments usually did not pay off. A systematic approach to planning, designing, building flexible and maintainable automated test suites needs to be employed. An approach that would bring the best out of the testing efforts. (Fewster, Graham 1994, 517; Hutcheson 2003, 4-20.)

## 2.2 Why Do We Need a Methodology for GUI Testing?

A question can be asked: why do we need a methodology? What makes GUI testing unlike any other form of automated testing? The answer lies in the nature of graphical user interfaces and the way they are developed and maintained.

1. The **GUI changes more often than the business logic** that it invokes. Introducing new functionality to the application often requires GUI reorganization to be able to present the new data in a coherent way. This implicates that more effort needs to be put into maintenance as it becomes very probable that new versions will break the existing GUI tests. Business logic tests, like unit tests, seem much more static.

2. **AUT's GUI complexity**. There are usually several ways to perform a given operation, which translates into more things to test.
3. **GUI test tools complexity**. Majority of GUI test tools available on the market have their own robust scripting language. GUI test tools have to run as a separate process from the AUT which requires synchronizing test execution speed with the speed of actions performed on the application. Large number of GUI controls to be manipulated and queried entails the large API that has to be available to the script writer.
4. **Custom controls handling is difficult**. Modern GUI test tools can handle the common controls in an abstract way, utilising semantics corresponding with the function of the control, e.g. *button("OK").click()*. Dealing with custom controls usually has to be handled in a very primitive way, like using *x* and *y* screen coordinates which makes the test scripts inflexible. (Fewster, Graham 1994, 519; Patton, Ron 2005, 324)

## 2.3 The Benefits of Test Automation

Test automation enables some testing tasks to be executed far more efficiently in comparison with manual testing. It has many benefits, including those listed below (Fewster, Graham 1994, 9-10; Graham, Veenendaal, Evans, Black 2007, 185):

1. **Run existing tests on a new version of a program**. This is probably the most obvious objective for the tests. Test automation should minimise the effort used to perform regression testing. The tests that worked with the previous version of the program should also pass on the latest version.

2. **Run more tests more often**. A visible benefit of test automation is the capability to run many tests in less time, therefore making it possible to run them more frequently. This leads to increased confidence in the system.

3. **Execute tests difficult or impossible to do manually**. Trying to perform a live test of an online system with say 500 users would be impossible to do but the input from 500 users can easily be simulated using automated tests. Automated tests can be run by technical person who do not have to know all the complexities of the application, or even by another tool like *CruiseControl*.

4. **Tests reusability**. The efforts and costs put into designing, building and making tests highly reliable can be distributed over many runs of those tests. Writing modular and reliable code improves the quality and speeds up building of new tests.

5. **Better use of resources**. By automating time consuming and boring tasks, like entering long list of test inputs, one can improve the staff morale and free the skilled testers to be creative and put more energy into designing better test cases to be executed. Also computers can be used to run the tests instead of being idle overnight or at the weekend.

6. **Coherency and repeatability of tests**. Tests which are repeated automatically will be repeated exactly each time. Of course this assumes that tests are synchronised well with the AUT. Although automation offers a level of consistency that is very difficult to achieve in manual tests. The same tests can be run on different hardware and operating systems. This can confirm cross-platform quality and consistency which would be costly and almost impossible to accomplish with manual testing.

7. **Increased confidence**. If an application passes extensive set of well designed automated tests, there can be higher confidence that no unpleasant surprise will show up in the released version. The tester can make more profound testing with less effort, resulting with higher quality and productivity.

8. **Earlier time to market**. After you automate a set of tests, they can be executed much faster than they would be manually. Therefore the time

needed to test application decreases and developers can start fixing the bugs earlier.

## 2.4 The Risks of Test Automation

While trying to automate testing a number of problems can be encountered. Realising the possible problems is needed to be able to avoid them or overcome them. The most common risks are listed below (Fewster, Graham 1994, 9-10; Graham, Veenendaal, Evans, Black 2007, 186-187):

1. **Poor testing practices**. When the tests are poor quality, badly organised and designed, with none or ambiguous documentation, with little chance to find any bug, than there is no sense to automate them. The focus should be to better the testing effectiveness first rather than improving efficiency. After automating chaos, all you get is a faster chaos.

2. **Unrealistic expectations**. Testing tools vendors promise that their tool will solve all your problems. Ecstatic enthusiasm of marketing people and salesmen is contagious. Vendors obviously emphasize the benefits that can be achieved, examples of (real or not) implementation victories, in the same time playing down the amount of effort needed to get lasting benefits. If this wild optimism will translate into management's unrealistic expectations, than no matter how close to perfection the tool is, it will not meet the expectations.

3. **Anticipation that automated tests should find many new bugs**. The biggest probability to find a defect is during the first time the test is run. Rerunning the same test gives much less probability to find a new bug, unless application's code was changed. Those changes could brake application's functionalities directly or indirectly. Other possibility to rise chances to find a defect is to rerun the tests in a different hardware or software environment. Tests which do not find defects are not useless. Although a well designed test should aim at finding defects, however

passing test suits can give the confidence that the changes made in the code did not break previously implemented functionalities.

4. **Tests maintenance**. After some changes have been made in the application under test it is often inevitable to update some, or in the worst case scenario all test so that they could be re-executed successfully. The effort required for tests maintenance was often the last nail in the coffin for many test automation initiatives. If it is less time consuming to rewrite the test than just update them, then the test automation is very likely to be ceased. This dissertation aims to help the reader not to become a victim of excessive maintenance costs.

5. **False confidence**. Only because a test suite executed successfully, it does not mean that the software is faultless. The tests themselves can contain defects or they can be just incomplete. There is always a chance that the tester has unconsciously implemented tests which can preserve the incorrect results for indefinite time.

6. **Technical problems and lack of customer support**. Test execution tools are third-party software products which themselves are not error proof. This is not a good showcase if testing tool is not properly tested itself, although it does happen. When choosing a testing tool one should also take into consideration if there is technical support for the product or at least active community of product’s users. Commercial test execution tools are usually big and complex products and extensive technical know-how is needed to gain the best out of them. Therefore it would be good if tool vendor would offer training or at least extensive documentation for the future test automation engineers. Additionally to technical problems with test automation tool you can encounter technical problems with the application under test itself. Application to be tested should be designed and implemented with testability in mind, so that it would not be difficult to test it automatically or manually.

7. **Organizational issues**. Test automation needs support and understanding from the management and it requires to be fit into the culture of the organization. Time is needed for choosing the right tool, for training, experimenting and getting hands on experience, and for promoting use of the tool within the organisation. Having a kind of tool champion would increase radically the probability of success. A charismatic person enthusiastic about tests automation could help a lot to promote the tool within the organisation. Start-up costs are always relatively high when introducing test automation, therefore a long term approach needs to be used. Not introducing standards from the very beginning can cause that incoherent approaches to test automation will be used, which can make it difficult to transfer and share automated tests and testers between teams. Also tool's licensing has to be carefully thought over. Having too few licenses for people who want to use the tool could affect the success and cost of the test automation effort. Perception of work effort needs to be reviewed. Even if tests run automatically overnight, their results need to reviewed and analysed by the tester. Test analysis becomes a separate activity comparing with manual tests, where it was embedded in test execution activity.

## 2.5 Test Planning and Design

Relevant test planning is needed to be successful in applying automated testing. Testing effort has to be planned as any other aspect of software development process. But especially in case of GUI testing, it is hard to fight the temptation to trifle and neglect the planning stage, as: testing is supposed to be simple (especially with record/playback), the testing code will not be shipped to the customer anyway, and the deadlines are tight so "let's better get to write the tests immediately". Neglecting the need of proper planning and design is false economy. The quality of planning and design done "one the fly" while implementing the tests would be poor. Therefore it is fundamental to treat test planning as a separate intellectual effort from implementing the tests. As analysed in the previous chapter, GUI test automation is difficult:

- Test planning and design is hard to do properly
- Learning to use testing tool properly is time consuming
- Writing and debugging the test scripts is difficult

Trying to do all those tasks in the same time would make them even harder. (Pettichord 2001; Fewster, Graham 1994, 9-10)

### 2.5.1 The Purpose of Test Documentation

Test documentation has two main functions: it should be a **communications vehicle** and a **blueprint for development**. Contents of the documents will be of more significance than their form. The test specification is a kind of functional specification for testing. Its level of detail depends on the needs of the specific project. It usually covers topics like: testing environment, testing scope, risks, staffing needs, etc.  Test specification can be treated as a kind of checklist to insure that all activities follow the plan and schedule accepted by management. If the company does not yet have a software specification document to follow, a good template to start with is contained in IEEE Standard 829 – IEEE Standard for Test Documentation.

The test design acts as a blueprint, a detailed description of what will be tested (and what not), and how each test case should be implemented. In the simplest form a test plan can be just a list of tests. A test case design should consist of at least:

- Test name or ID which should uniquely identify the test.
- Test purpose: short description of what the test is supposed to do.
- Test method: clear steps which should allow the user to perform a given test manually.
- Pass/Fail criteria: how to tell if the test if the test works.

Putting the effort to create test documentation as described above has several advantages:

1. It allows reviewing the test plan and gives a complete overview of tasks to be done. During test reviews the team should make mindful trade-offs to improve the tests.

2. It gives a basis for deciding which test cases to automate. In case of GUI tests not everything can be automated. And even more importantly, not everything is possible to automate cost effectively. Good example of operation which should be left for manual tests is printing. From the reviewed and accepted list of tests the ones that can not or should not be automated should be segregated. These tests should be added to manual testing checklist.

3. It provides the basic test case documentation which can give a general view of the tests. It improves maintainability of tests. More detailed documentation should be included in the testing code comments.

4. GUI test design process discovers bugs. This is the strongest argument for doing test design. The surprising fact is that while composing test cases and clicking through application a lot of bugs can be found before even starting to write automated scripts. This is an interesting issue worth of discussing in a separate chapter. (Fewster, Graham 1994, 522-525; Institute of Electrical and Electronics Engineers, Inc. 1998)

### 2.5.2 Finding Bugs During Test Design Process

As mentioned in previous chapter the surprising fact is that during writing detailed GUI test designs many bugs in application can be found even before starting to write test code. To begin with making detailed test plans usually the tester has to wait till the GUI stabilises. Luckily modern GUI builders allow engineers to build and modify complex user interfaces in very short time. While writing test design document the tester performs manual testing. Writing this document is often a first time anyone has systematically tested the user interface. Taking into account the complexity of modern GUIs, it is not surprising that plenty of defects are discovered. This denies the widely accepted opinion that “if you commit to building an automated GUI Test Suite, don’t expect it to pay off during the current release”. Usually approximately half of the defects are discovered during test design phase as a result of creating a test case to specifically test for that certain condition. Other bugs will be discovered incidentally, like misspelled labels in dialog windows or missing icons. The bugs reported range from trivial (e.g. inconsistent labels in dialog: *File name:*, *File Name*) to serious which crash the application under test.

The fact that till the test automation document is ready many defects will be discovered should be used to gain a kind of confidence credit inside the company. Even if the schedule is so tight that building automated tests will not be possible, documenting the manual testing actions in a way described in previous chapters is still an excellent way of finding bugs and in the same time making a big step towards test automation. (Fewster, Graham 1994, 525-526; Pettichord 2001)

## 2.6 Well-designed Test Cases

After test specification and design have been completed and reviewed, it is time to begin writing the tests. The effort put into creating well-designed Test Suites is to satisfy fundamental objectives.

- They must be maintainable
- They must be modular
- They must be robust
- They must be well documented
- They should be built of reusable components

The ideal tests should possess those attributes. (Fewster, Graham 1994, 527; Myers 2004, 43-44.)

### 2.6.1 A Test Case Is Independent

Every test case has to take care of its own set-up, verification, and clean-up. In the set-up phase application should be brought to a state where the actual test can be executed. In verification phase the actual testing is performed, results are evaluated to a pass/fail status. Clean-up phase should bring the application back to a state from before setup – so called base state – to make it ready for the next test. If a test case would rely on the results of the previous test case, then if the first test case would fail, it would most likely cause failure of the preceding test case. Such cascading errors would make it very difficult to find what the root cause of these failures was. It also enforces implicit ordering of the test cases which in practice is rarely documented. Unconscious reordering of test case execution (e.g. by a tester or testing framework) could cause a chain of failures in a Test Suite which executed faultlessly the previous day. Test cases should be executable in any sequence. This allows the maintainer to choose a subset of test cases to run without having to concern the interdependencies between test cases. This is sometimes hard to apply in practice. For tests that modify a complex global state (e.g. creating or modifying a database) beginning all tests from zero for each test case would be far too expensive. In such situations test cases relying on specific state can be grouped together but the interdependencies between them should be well documented to help future maintainers to analyse such a Test Suite (Fewster, Graham 1994, 527.)

### 2.6.2 A Test Case Has a Single Purpose

An ideal test case should have a single purpose. This should help keeping the code relatively short and simple to make it easy to understand, debug, and maintain. Moreover it also means that the outcome of the test case should always be one of the two: pass or fail. This makes it much easier to interpret the results. In case of failure of a single purpose test case it is trivial to locate the application function at fault. This implicates that a Test Suite should consist of many smaller test cases instead of few large ones (Fewster, Graham 1994, 527-528.)

### 2.6.3 Unsuccessful Test Case Should Not Cause Others to Fail

A test case that fails due to an unexpected error leaves the application in an unknown state. The AUT is out of synch with what the test case is expecting. A well-behaved test should log the failure, abort and reset the application to a known base state. It is a task of a test tool to isolate test case failures so that an unexpected error in one test case does not cause a whole script to abort. All of the most popular GUI test tools provide this functionality (Fewster, Graham 1994, 528.)

### 2.6.4 A Test Case Is Well Documented

As already mentioned, one of the advantages of writing a good test design is that you can take the test case description from this document and use them as header comments in the test code. Of course also in-line comments should be used to describe the logic when necessary.

All attributes discussed in chapter 2.6 make up requirements for well-designed test cases (Fewster, Graham 1994, 528.)

## 2.7 Standardised error recovery

There are many things besides a bug in the AUT which can cause a test case to fail. It could be a bug in the testing code, environmental error (e.g. network connection down, no disk space), intentional change in the application, excessive machine load causing timing errors, etc. All the modern GUI tools have built-in ability to detect an error, log it (with traceback), and move on to the next test case.

The problem is that when the test crashes, it leaves the application in an unknown state and unless some actions are taken to reset the AUT to a known state, subsequent tests are likely to fail. A base state is usually main window open, active, and not minimized (just as if application was just started). A good practice is to implement the capability of recovering to a state which would allow continuing with next test cases. A recovery routine could be for instance:

1. Log the error
2. Abort the test case (because it is in an unknown state)
3. Make attempt to come back to a known state (e.g. close all opened windows till the main window will be active).
4. Resume execution with the next test case

Implementing those steps should allow execution of the consequent test cases (Fewster, Graham 1994, 528-533.)

## 2.8 Testing in Agile Development Environment

### 2.8.1 Agile Methodology

Agile methodologies started to become widely used at the beginning of this decade. To name just a few: Scrum, Extreme Programming, Crystal, FDD, and DSDM are probably most popular. What they all have in common was gathered in so called Agile Manifesto published in 2001 (Beck, et al. 2001):

***Manifesto for Agile Software Development***

*We are uncovering better ways of developing*
*software by doing it and helping others do it.*
*Through this work we have come to value:*

***Individuals and interactions** over processes and tools*

***Working software** over comprehensive documentation*

***Customer collaboration** over contract negotiation*

***Responding to change** over following the plan*

*That is, while there is value in the items on*

*the right, we value the items on the left more.*

Using the values from the Manifesto it is possible to deliver small chunks of business value in very short release cycles (2-4 weeks).

### 2.8.2 Agile Testing

Everyone in Agile team is a tester. Anyone can perform testing tasks. If that is true, than what is the role of an agile tester? An Agile tester is a professional tester who embraces change, collaborates well with both business and technical people, understands the concept of using tests to drive development and document requirements. Agile testers are willing to learn what customers do to understand better what users' and customers' software requirements are.

**Ten principles for Agile testers** listed below were derived from the Agile Manifesto and so called Twelve Principles of Agile Software (Beck, et al. 2001):

1. **Provide continuous feedback**. If tests drive Agile projects, than it's no surprise that feedback is very important in Agile team. Tester plays a role of information provider within a team. He helps the customer to articulate requirements for each story in the form of examples and tests. After that he cooperates with team members to turn those requirements into executable tests which will give meaningful feedback.

2. **Deliver value to the customer**. The tester has influence on quality and the priority of the pieces of functionality that should be delivered to the customer.

3. **Enable face to face communication**. Agile tester which knows the application from customer's point of view and understands the technical aspects and limitations related to implementing features can help customers and developers achieve a common language.

4. **Have courage**. Tester should have courage to ask questions, challenge the ways how things are done, join meetings and conversations he wasn't invited for.

5. **Keep it simple**. Test “just enough” with the lightest-weight tools and techniques that can be found which will do the job.

6. **Practice continuous improvement**. Searching for ways to do a better job should be part of an agile tester’s mind-set.

7. **Respond to change**. Agile testers need to respond to frequently changing requirements, priorities and they have to accommodate changes.

8. **Self-organize**. When the Agile team faces a major problem, like a broken build, it is everyone’s problem. Team members discuss the issue right away and decide how to fix the problem and who will do it.

9. **Focus on people**. Every team member should have opportunity to grow and develop his skills. Agile testers are not treated as second-class citizens in software development world. They contribute unique value to their teams.

10. **Enjoy**. Working in environment where all team members are responsible for quality and testing, where everyone collaborates, where you are engaged in the project from start to finish - it seems like a tester’s Utopia, therefore enjoy it. (Crispin, Gregory 2009, 19-34.)

## 2.9 Best Practices Summarised

This chapter will summarise briefly the theory of GUI test automation by pointing out the most important steps which should lead to successful implementation of test automation (Fewster, Graham 1994, 534-535; Pettichord 2001.)

### 2.9.1 Test Project Planning

- Create a test plan containing high-level aspects of the project;

- Create a test design document that should be a base for test cases creation;

- Get official approval of the design before you start coding tests;

### 2.9.2 Writing the Tests

- Keep the test programs simple and easy to understand;
- Build independent test cases. Do not rely on the results of a previous test case as a basis for another test case;
- Take header comments for each test case from the test plan. Make the code easy to maintain. Comment and document well, especially any workarounds and interdependencies in the test code;
- Make sure each test case has clear result: pass or fail;

# 3 SWTBOT IN GUI TEST AUTOMATION

## 3.1 Introduction to SWTBot

SWTBot is an open-source Java based functional/UI testing tool for testing SWT and Eclipse based applications. It is a kind of click-robot. SWTBot provides simple to read and write Application Programming Interfaces. APIs hide the complexities of SWT and Eclipse, making the tests more intuitive to write. SWTBot integrates well with Eclipse, and supports Ant tasks so that one can run his builds from within CruiseControl or any other Continuous Integration tool. SWTBot can run on any platform that SWT runs on, therefore also with Eclipse RCP applications. Very few testing tools provide support for such a variety of platforms. SWTBot also supplies its own set of assertions that are useful for SWT. After installing SWTBot into Eclipse using the SWTBot update site, a new "Run As…" choice appears (SWTBot Test). SWTBot's test case classes, SWTBotTestCase (for standalone SWT app testing), and SWTBotEclipseTestCase (for Eclipse SWT plug-in testing) subclass JUnit's TestCase class, so all of the JUnit facilities are available, plus some extra magic for accessing SWT widgets:

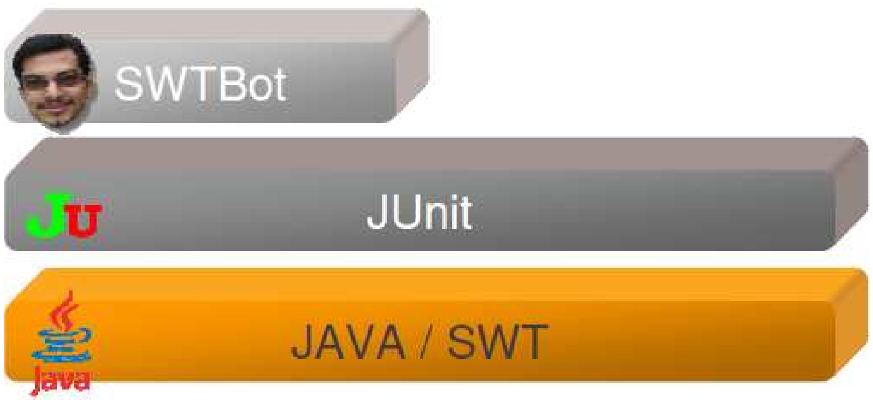


**Figure 1 SWTBot is an API that depends on JUnit and SWT**

Many events that SWTBot sends to the UI are blocking. SWT dialogs are a good example of one of them. What it means in practice is that functions opening dialogs will block until the dialog closes. Since we do not want our tests to be blocked when a dialog opens up, SWTBot runs in a non-UI thread, and posts events to the UI thread:

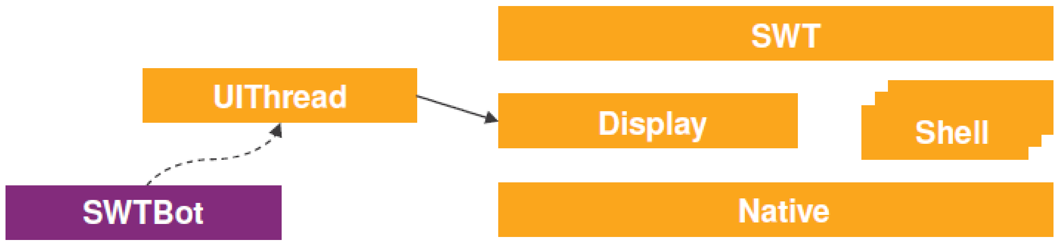


**Figure 2 SWTBot and threading**

In SWT, access to native resources is controlled by a single Display object. This object is created by the single UI thread. Shell objects represent windows and of course there can be many of them in your application. (Pedegaonkar, Rodrigues 2009)

In December 2008 SWTBot project moved from SourceForge to Eclipse. It is currently in Incubation Phase of the Eclipse development process. This means that its developers are currently implementing the requirements that a full Eclipse project must meet in terms of its processes, community and technology. The code base of the project is already stable and mature. Nevertheless, one have to be prepared for one or the other change in the API. (Ebert 2009)

## 3.2 Installation and Configuration of SWTBot

### 3.2.1 Install Eclipse IDE

SWTBot tests require Eclipse IDE. In case of the project the author was involved in - **Eclipse for RCP/Plug-in Developers** should be installed. It can be found on the official Eclipse web site: http://www.eclipse.org/downloads/packages/eclipse-rcpplug-developers/galileor [referenced 15.09.2009].

### 3.2.2 Install SWTBot Plug-in

Next step is installation of SWTBot plug-in. The easiest way is to use the update site: http://download.eclipse.org/technology/swtbot/galileo/dev-build/update-site/ [referenced 15.09.2009]. Using the menu in Eclipse with **Help** > **Software Updates** > **Installed Software** the following entries should be displayed:

- SWTBot Eclipse Feature
- SWTBot Feature

Both of them have to be installed and Eclipse should be restarted.

### 3.2.3 Create Test Plug-in

After completion of SWTBot plug-in installation, a test plug-in can be created. The author advises to use a naming convention in which the name of the plug-in is suffixed by **.swtbottests**. Example:

- Plugin to be tested: pegasus.core
- Test plug-in: Pegasus.core.swtbottests

A test plug-in is a normal Eclipse Plug-in. One can create it in the normal way by unchecking some options:

- Uncheck <This plug-in will make contributions to the UI>
- Uncheck <Generate an activator, …>
- Uncheck <Create a plug-in using one of the templates>

Add the following required plug-in dependencies to your test plug-in manifest file:

- org.junit4
- org.eclipse.swtbot.eclipse.finder
- org.eclipse.swtbot.swt.finder
- org.hamcrest
- org.apache.log4j
- org.eclipse.swt

- org.eclipse.ui

And last but not the least, the plug-in to be tested:

- pegasus.core

### 3.2.4 Create a Simple Test

Code example in **APPENDIX 1: SimpleExampleTest.java** can be added to the plug-in: pegasus.core.swtbottests.SimpleExampleTest.java. The code can be used as a skeleton for writing other SWTBot tests for Pegasus. The test opens Run perspective from the menu bar, makes some assertions, than switches back to Test perspective. What it does exactly is not important at the moment. Its main purpose is to check if any SWTBot test can be run.

### 3.2.5 Run a Test

In order to verify the correctness of steps described in chapters 3.2.1 to 3.2.4, running a test is needed:

- Perform **Run as SWTBot test** on the test class **SimpleExampleTest.java**
- Even if the first run leads to errors – it creates a default run configuration
- Perform **Run As / Run Configurations…**
- Open the **Main** tab
- Define **Run a product** as pegasus.branding.PegasusRCP
- **Run** the test

After the test run a JUnit view opens. If the tests execute, then everything is ok. The environment is prepared to write SWTBot tests.

## 3.3 Eclipse RCP

RCP provides a generic Eclipse workbench that developers can extend in order to construct their own applications. Among many advantages of RCP applications the most important are: easy to extend plug-in architecture, responsiveness, native-looking user interface, easy to write help system. Every Eclipse RCP application comprises of at least one custom plug-in and uses the same UI elements as eclipse 3.5 IDE. It is important to get familiar with the basic elements of Eclipse user interface to understand SWTBot tests which will be covered later. Figure 3 shows those elements in Pegasus RCP – application to be tested later:

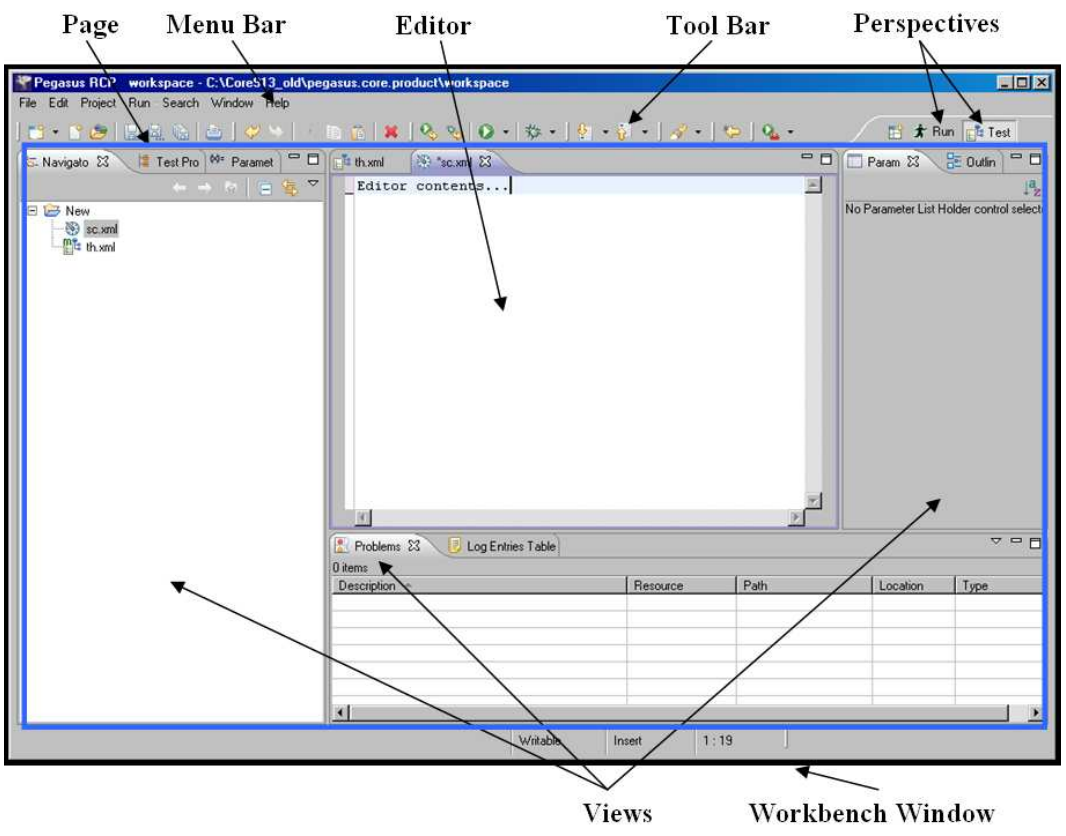


**Figure 3 Basic UI elements of Pegasus RCP**

Pegasus RCP – application which the author is going to test with SWTBot was written using Eclipse RCP framework. It consists of elements like views, editors, perspectives, etc. This chapter helps to get familiar with basic terminology regarding Eclipse RCP applications which will benefit in better understanding of SWTBot's API covered in consecutive chapters.

The **workbench** provides a robust set of classes and interfaces for building complex user interfaces. Workbench window (IWorkbenchWindow) is the top-level window in a workbench. It is the frame that holds the **menu bar**, **tool bar**, pages, views, editors, etc. The term workbench can also be used loosely to refer to “the window that opens when you start the platform”. Next paragraphs describe the main visual components that make up the workbench.

Inside the workbench window there is one **page** (IWorkbenchPage) that in turn contains parts. Pages are used for grouping parts.

**Perspectives** are an additional layer of organization inside the workbench page. A perspective defines an appropriate collection of views, their layout, and available actions for a given user task. Users can switch between perspectives as they move across tasks.

When the plug-in programmer adds a visual component to the workbench, he must decide whether he wants to implement a view or an editor. How does he decide this?

A **view** is typically used to create a file navigator, to open an editor, or to display properties for the active editor. For example, in Pegasus there is a Navigator view which allows browsing the contents of the workspace. Properties and outline views are used to show information about an object in the active editor. Any modifications made in a view (like changing value of a property) are saved immediately.

An **editor** is mostly used to edit or browse a document or input object. Modifications made in an editor follow an open-save-close model, similarly to any external file system editor. (Vogel 2009)

# 4 DESCRIBING SWTBOT TEST CODE

This chapter uncovers SWTBot API and explains the nuances of SWTBot test program written as a practical part of the thesis: **FileMenuTest.java** (see **APPENDIX 2**). FileMenuTest was designed to test File Menu options. It consists of 30 test cases and almost 1500 lines of code. Movie presenting the tests running can be found in **APPENDIX 3**.

## 4.1 The Skeleton of the Code

This chapter familiarises the reader with the basic elements of every SWTBot test, and other issues that the test automation engineer should be familiar with.

**SWTWorkbenchBot** offers API for testing Eclipse workbench items like views, editors and perspectives. What is also interesting here is that all non-constant member variables in the code start with the underscore. This coding convention comes from Java Sun™ Coding Standard. The rationale behind it is that it facilitates auto-completion (typing '_' shows class members only). The drawback is that it reduces readability for programmers not familiar with this coding convention. **Message if assertion fails** _assertionFailedMessage is also a member variable. It is utilized to store the messages displayed when assertion fails.

**@BeforeClass** annotation precedes a setUpBeforeClass() method. When several tests need to share a computationally expensive setup, the setup code can be put inside its body. While this can compromise the independence of tests, sometimes it is a necessary optimization. This method will be run once before any of the test methods in the class. In the discussed test code it is used to instantiate the _bot variable (of type SWTWorkbenchBot).

**@AfterClass** annotation precedes teardownAfterClass() method. If expensive external resources were allocated in BeforeClass method, they have to be released after all the tests in the class have run. All @AfterClass methods are guaranteed to run even if BeforeClass method throws an exception. Therefore here the _bot resource is released by setting it to null.

**@Before** annotation causes the method setUp() to run before every @Test method in the current class.

**@After** annotation precedes tearDown() method which is usually used to release the resources (allocated in @Before method) after the test runs. This method will execute even if the @Test method throws exception. In the discussed code a cleanup() method was put to assure the tests independence. The cleanup() method will be covered in more detail later.

**@Test** annotation tells JUnit that the public void method to which it is attached can be run as a test case. To run the method, JUnit first constructs a fresh instance of the class than invokes the annotated method. Any exceptions thrown by the test will be reported by JUnit as a failure. If no exceptions are thrown, the test is assumed to have succeeded.

It is worth to mention that **each test class has its own workspace**. If there is a need for the test to have a fresh workspace, than a separate test class have to be written. Otherwise the test methods inside the same class will share a common workspace.

**Javadoc** comments are used throughout all the code although there are not many comments inside the test methods. The reason for it is that the author was taught at work in NSN that the code itself should be clear enough not to need comments. If there is a need for the comment – than writing another method or thinking of naming variables and methods in more descriptive way should be considered. The other reason is that the comments themselves become additional thing to maintain. The author agrees with those rules and tries to comply with them the best he can.

## 4.2 SWTBot API in Examples

The pieces of code gathered in this chapter form a kind of tutorial needed to get familiar with the basics of SWTBot's API. Its main goal is to equip the reader in the knowledge essential to be able to understand the code of **FileMenuTest** from **APPENDIX 2**.

### 4.2.1 Menu

```
//CLICK ON A MENU ITEM
_bot.menu("File").menu("New").menu("Test Hierarchy").click();


//CHECK IF MENU ITEM IS ENABLED
assertTrue(_bot.menu("File").menu("Move...").isEnabled());
```

### 4.2.2 Views

```
//SET FOCUS ON PARAMETER VIEW
_bot.viewByTitle("Navigator").setFocus();


//SHOW VIEW
_bot.viewByTitle("Navigator").show();


//PRINT LIST OF ALL VISIBLE VIEWS TO CONSOLE
ArrayList<SWTBotView> viewsList;
viewsList = new ArrayList<SWTBotView>(_bot.views());

if (viewsList.isEmpty())
    System.out.println("\n\n>There is no view<\n\n");
else {
    System.out.println("There are " + viewsList.size() + " views
    visible:");
for (int i = 0; i < viewsList.size(); i++) {
    System.out.println(viewsList.get(i).getTitle());
}

//EXAMPLE OUTPUT:
//   There are 6 views available:
//   Navigator
//   Test Procedures
//   Parameter Types
//   Problems
//   Parameter
//   Outline
```

### 4.2.3 Editors

```
//GET THE TITLE FROM EDITOR TAB
String th1 = _bot.editorByTitle("TestHierarchy1.xml").getTitle();


//GET TEXT FROM EDITOR WITH GIVEN TITLE
String editorContents =
_bot.editorByTitle("TestHierarchy2.xml").toTextEditor().getText();


//SET TEXT INSIDE THE EDITOR
_bot.editorByTitle("TestHierarchy3.xml").toTextEditor()
                              .setText("My text here");
```

```
//CLOSE EDITOR TAB WITH A GIVEN TITLE
_bot.editorByTitle("TestHierarchy4.txt").close();


//SAVE EDITOR TAB WITH A GIVEN TITLE
_bot.editorByTitle("TestHierarchy5.xml").save();


//SET FOCUS ON THE EDITOR WITH A GIVEN TITLE
_bot.editorByTitle("TestHierarchy6.xml").setFocus();


//RETURN THE NUMBER OF EDITOR TABS
_bot.editors().size();
```

### 4.2.4 Windows

```
//CHECK IF WINDOW IS ACTIVE
assertTrue(_bot.shell("New Project").isActive());


//CLOSE WINDOW
_bot.shell("New Project").close();


//GET ACTIVE WINDOW'S TITLE
String activeWindow = _bot.activeShell().getText();
```

### 4.2.5 Tree

```
//SELECT A TREE ITEM (2 EQUIVALENT WAYS SHOWN)
_bot.tree().expandNode("General").select("Project");
_bot.tree().expandNode("General").expandNode("Project").select();


//SELECT A TREE NODE (INSIDE A VIEW)
_bot.tree(1).expandNode("Project1").getNode("th1.xml").select();


//CHECK WHETHER A TREE ITEM IS AVAILABLE
assertTrue(_bot.tree().getTreeItem("slave1") != null);
assertTrue(_bot.tree().expandNode("slave1").select("Properties")!=
null);


//CHECK WHETHER A TREE LEAF IS AVAILABLE
assertTrue(_bot.tree().expandNode("slave1")
 .expandNode("Properties").getNode("slaveName = slave1") != null);


//CHECK IF TREE Project1 (INSIDE A VIEW) CONTAINS NODE th1.xml
assertTrue((_bot.tree(1).expandNode("Project1").getNodes())
                        .contains("th1.xml"));
```

```
//GET THE NUMBER OF TOP LEVEL NODES (INSIDE A VIEW)
int topLevelNodes = _bot.tree(1).getAllItems().length;
```

```
//DELETE THE TOPMOST ITEM (INSIDE A VIEW)
_bot.tree(1).getAllItems()[0].contextMenu("Delete").click();
```

### 4.2.6 Perspectives

```
//ACTIVATE A PERSPECTIVE WITH A GIVEN LABEL
_bot.perspectiveByLabel("Run").activate();
```

```
//CHECK IF PERSPECTIVE WITH A GIVEN LABEL IS CURRENTLY ACTIVE
_bot.perspectiveByLabel("Test").isActive();
```

### 4.2.7 Buttons

```
//CLICK BUTTON
_bot.button("Next >").click();
```

```
//CHECK WHETHER A BUTTON IS DISABLED
assertFalse(_bot.button("Finish").isEnabled());
```

### 4.2.8 Tool Bar Buttons

```
//CLICK TOOL BAR BUTTON
_bot.toolbarButtonWithTooltip("New").click();
```

```
//CHECK WHETHER TOOL BAR BUTTON IS ENABLED
assertTrue(_bot.toolbarButtonWithTooltip("New").isEnabled());
```

### 4.2.9 Check Boxes

```
//SELECT CHECK BOX
_bot.checkBox().select();
```

### 4.2.10 Text Fields

```
//WRITE TEXT INTO TEXT FIELD
_bot.textWithLabel("Project name:").setText("MyTestProject3_1");
```

```
//READ TEXT FROM TEXT FIELD
String text = _bot.textWithLabel("Project name:").getText();
```

#### 4.2.11 Time and Speed Control

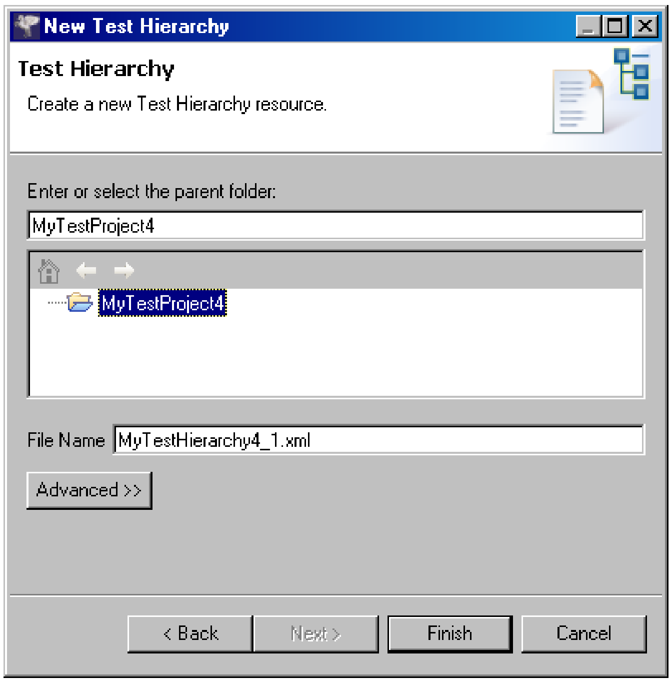
```
//SLEEP (VALUE IN MILLISECONDS)
_bot.sleep(1000);

//SLOW DOWN THE EXECUTION OF TESTS (VALUE IN MILLISECONDS)
SWTBotPreferences.PLAYBACK_DELAY = 100;
```

### 4.3 Error Recovery Implementation

In chapter 2.7 a standardised error recovery approach was introduced to the reader theoretically. In this chapter it will be explained in practice. To understand what error recovery does, a real situation will be uncovered.

#### 4.3.1 Test without Error Recovery

One of the developers accidentally checked-in a code that caused creating new Test Hierarchy files to stop working. More precisely, clicking Finish button does not create a Test Hierarchy file and does not close the 'New Test Hierarchy' window:


New Test Hierarchy

Test Hierarchy

Create a new Test Hierarchy resource.

Enter or select the parent folder:

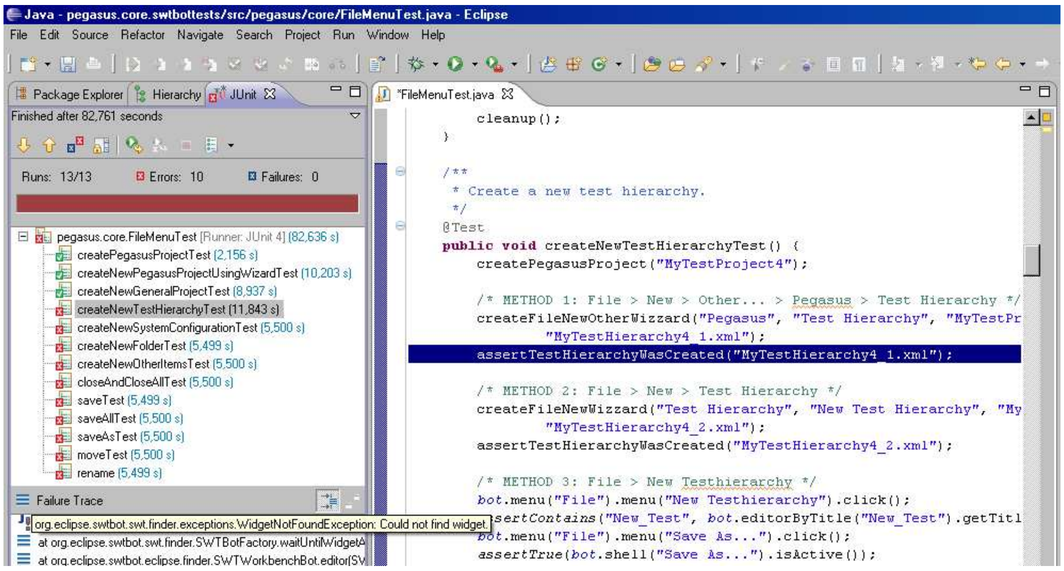
MyTestProject4

MyTestProject4

File Name MyTestHierarchy4_1.xml

**Figure 4 New Test Hierarchy window - Finish button stopped working**

As a result all the subsequent tests after the failed test will also fail (see Figure 5) because the focus stays on ‘New Test Hierarchy’ window, not the default ‘Pegasus RCP’ that every test expects. After double-clicking on the first failed test one can see exactly in which line the test failed and Failure Trace gives a hint about the reason of the failure. But all the consequent tests fail. Even the ones that have nothing to do with creating Test Hierarchies will fail in the first line of their execution. This can be very confusing for the maintainer of the test who needs to analyze the reason of the failure fast. In this situation it is difficult to isolate the real reason of the failure and even worse – it is easy to overlook some other bug that might have appeared in the same time.


Java - pegasus.core.swtbottests/src/pegasus/core/FileMenuTest.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
JUnit
*FileMenuTest.java
Failure Trace

```
Finished after 82,761 seconds

Runs: 13/13     Errors: 10     Failures: 0

pegasus.core.FileMenuTest [Runner: JUnit 4] (82,636 s)
    createPegasusProjectTest (2,156 s)
    createNewPegasusProjectUsingWizardTest (10,203 s)
    createNewGeneralProjectTest (8,937 s)
    createNewTestHierarchyTest (11,843 s)
    createNewSystemConfigurationTest (5,500 s)
    createNewFolderTest (5,499 s)
    createNewOtherItemsTest (5,500 s)
    closeAndCloseAllTest (5,500 s)
    saveTest (5,499 s)
    saveAllTest (5,500 s)
    saveAsTest (5,500 s)
    moveTest (5,500 s)
    rename (5,499 s)

org.eclipse.swtbot.swt.finder.exceptions.WidgetNotFoundException: Could not find widget.
at org.eclipse.swtbot.swt.finder.SWTBotFactory.waitUntilWidgetA
at org.eclipse.swtbot.eclipse.finder.SWTWorkbenchBot.editor(SV
```

```
        cleanup();
    }

    /**
     * Create a new test hierarchy.
     */
    @Test
    public void createNewTestHierarchyTest() {
        createPegasusProject("MyTestProject4");

        /* METHOD 1: File > New > Other... > Pegasus > Test Hierarchy */
        createFileNewOtherWizzard("Pegasus", "Test Hierarchy", "MyTestPr
                "MyTestHierarchy4_1.xml");
        assertTestHierarchyWasCreated("MyTestHierarchy4_1.xml");

        /* METHOD 2: File > New > Test Hierarchy */
        createFileNewWizzard("Test Hierarchy", "New Test Hierarchy", "My
                "MyTestHierarchy4_2.xml");
        assertTestHierarchyWasCreated("MyTestHierarchy4_2.xml");

        /* METHOD 3: File > New Testhierarchy */
        bot.menu("File").menu("New Testhierarchy").click();
        assertContains("New_Test", bot.editorByTitle("New_Test").getTitl
        bot.menu("File").menu("Save As...").click();
        assertTrue(bot.shell("Save As...").isActive());
```

**Figure 5 Execution of SWTBot tests without error recovery**

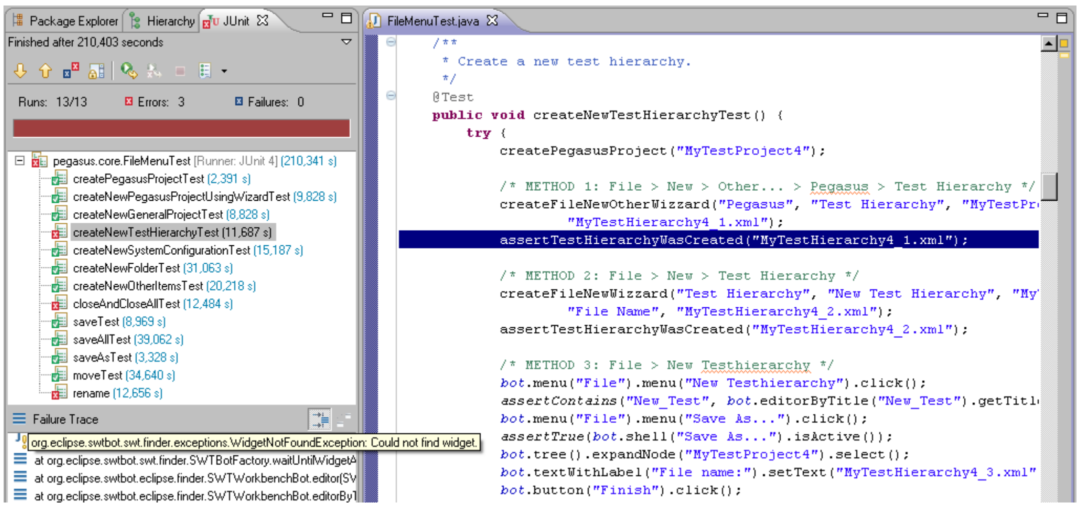
### 4.3.2 Test with Error Recovery

The error recovery routine code is presented below:

```
try {
    //METHOD BODY
}
catch (WidgetNotFoundException e) {
    closeDialogWindow();
    throw e;
}

/**
 * Closes dialog window if there is any (when 'Pegasus RCP'
 * is not currently active shell).
 */
private void closeDialogWindow() {
    String activeShellName = _bot.activeShell().getText().trim();
    String labelText = "Pegasus RCP";
```

```
        if (!activeShellName.equals(labelText)) {
            _bot.shell(activeShellName).close();
        }
    }
```

The error recovery relies on taking the method body into try-catch block. When some unexpected situation occurs, usually WidgetNotFoundException will be thrown. In the catch block first information is logged that the closeDialogWindow() method was entered. Then the algorithm checks if there is any dialog window opened so that it could make an attempt to close dialog window (if any) to bring the application back to its base state and allow other tests to continue. And at the end it rethrows the exception to be able to see its contents for each failed test. A good proof of error recovery algorithm correctness will be running the tests again with this algorithm implemented for every test:



```
Finished after 210,403 seconds
Runs: 13/13    Errors: 3    Failures: 0

pegasus.core.FileMenuTest [Runner: JUnit 4] (210,341 s)
    createPegasusProjectTest (2,391 s)
    createNewPegasusProjectUsingWizardTest (9,828 s)
    createNewGeneralProjectTest (8,828 s)
    createNewTestHierarchyTest (11,687 s)
    createNewSystemConfigurationTest (15,187 s)
    createNewFolderTest (31,063 s)
    createNewOtherItemsTest (20,218 s)
    closeAndCloseAllTest (12,484 s)
    saveTest (8,969 s)
    saveAllTest (39,062 s)
    saveAsTest (3,328 s)
    moveTest (34,640 s)
    rename (12,656 s)

org.eclipse.swtbot.swt.finder.exceptions.WidgetNotFoundException: Could not find widget.
at org.eclipse.swtbot.swt.finder.SWTBotFactory.waitUntilWidgetA
at org.eclipse.swtbot.eclipse.finder.SWTWorkbenchBot.editor(SV
at org.eclipse.swtbot.eclipse.finder.SWTWorkbenchBot.editorBy1
```

```
    /**
     * Create a new test hierarchy.
     */
    @Test
    public void createNewTestHierarchyTest() {
        try {
            createPegasusProject("MyTestProject4");

            /* METHOD 1: File > New > Other... > Pegasus > Test Hierarchy */
            createFileNewOtherWizzard("Pegasus", "Test Hierarchy", "MyTestPr
                    "MyTestHierarchy4_1.xml");
            assertTestHierarchyWasCreated("MyTestHierarchy4_1.xml");

            /* METHOD 2: File > New > Test Hierarchy */
            createFileNewWizzard("Test Hierarchy", "New Test Hierarchy", "My
                    "File Name", "MyTestHierarchy4_2.xml");
            assertTestHierarchyWasCreated("MyTestHierarchy4_2.xml");

            /* METHOD 3: File > New Testhierarchy */
            bot.menu("File").menu("New Testhierarchy").click();
            assertContains("New_Test", bot.editorByTitle("New_Test").getTitl
            bot.menu("File").menu("Save As...").click();
            assertTrue(bot.shell("Save As...").isActive());
            bot.tree().expandNode("MyTestProject4").select();
            bot.textWithLabel("File name:").setText("MyTestHierarchy4_3.xml"
            bot.button("Finish").click();
```

**Figure 6 Execution of SWTBot tests with error recovery implemented**

Now the behaviour of the test is correct. Only the tests that use faulty feature of creating new Test Hierarchies fail. For every failed test it is possible to see the exact line in which the test has failed with correct error messages.

## 4.4 Tests Independence Implementation

In compliance with the rules of well designed test cases - the tests in FileMenuTest are independent of one another. This means each test method can be executed separately of any other. It makes the tests more modular, simplifies their maintenance and analysis after failure.

### 4.4.1 Cleanup Method

Tests independence is achieved mostly thanks to the cleanup() method which is placed in tearDown() method, therefore it is executed after each @Test method even if it throws an exception. The cleanup() method has the following contents:

```
/**
 * Performs cleanup needed after tests: close all opened editors,
 * reset perspective, delete all projects from Navigator view.
 */
private void cleanup() {
    if (_bot.menu("File").menu("Close All").isEnabled()) {
        _bot.menu("File").menu("Close All").click();
    }
    _bot.menu("Window").menu("Reset Perspective...").click();

    if (_bot.shell("Reset Perspective").isActive()) {
        _bot.button("OK").click();
    }

    deleteAllFromNavigator();
    }
}
```

### 4.4.2 Test Synchronisation

Consider the following code:

```
/**
 * Wait until shell 'shellTitle' closes. Do nothing if it does
 * not exist anymore.
 *
 * @param shellTilte
 */
private void waitUntilShellCloses(String shellTilte) {
try {
_bot.waitUntil(Conditions.shellCloses(_bot.shell(shellTilte)));
}
catch (WidgetNotFoundException e) {
    /*
     * This have to be in empty catch block because waitUntil
     * throws exception when window 'shellTitle' is not found.
     * This window could be automatically closed after
     * operation has finished.
     */
     }
}
```

Sometimes there is need to use the waitUntil(…) method. After confirming execution of some costly operation (e.g. creating a new file), this operation can take some time. If the next test would start to execute, it would crash because the focus would be still set to some dialog window or progress bar from previous test.

First option to deal with such situation is using the sleep() method. But how to know how long will it take for the program to finish the operation? Setting a fixed number of seconds seems very inflexible.

Much better option is using a SWTBot method designed specially to deal with such cases:

```
_bot.waitUntil(Conditions.shellCloses(_bot.shell(shellTilte)));
```

But once in few executions an unpleasant situation occurs – the tests will crash. The reason is that if the operation will finish quicker and the window will close sooner than expected, then `_bot.shell(shellTilte)` will throw WidgetNotFoundException. The only way to deal with it is to catch it, write a description why it was caught and log it. And exactly this is done in the code.

# 5 ANALYSIS

## 5.1 Analysis of Capture/Playback Mechanism

Many GUI test tool vendors focus most of their attention on so called *capture/playback* (also known as *capture/replay* or *record/playback*) methodology. Those terms are in common use although they are somewhat confusing. Anyway, to know what exactly to do, test execution tool needs a test script, which is a program written in a programming language.

### 5.1.1 Why Capture/Playback Is a False Economy?

A captured test is a linear script and it is far from good solution for a number of reasons, including:

1. The test script only stores inputs that have been recorded, not test cases. So it does not know what the expected results are until you program it.

2. Small change introduced in the AUT can break most of your scripts.

3. The captured script can only cope with precisely the same conditions as when it was captured. Any unexpected event (e.g. dialog window shows up because file already exists) will be interpreted as a bug.

4. In addition to performing operation on AUT, the tool users are endlessly interrupted to insert verification points, test data and other check points. This is labour intensive and tedious task. (Li, Wu 2004, 20-21.)

There are situations when recorded test inputs can be useful in short term. Captured tests can be acceptable for some tests where the effort to update them when the software changes is not very substantial. But they definitely will not scale to hundreds or thousands of tests. (Graham, Veenendaal, Evans, Black 2007, 187-188.)

The record feature can generate a lot of code but this code usually has to be reprogrammed by the tester in order to integrate it into the test. Therefore capturing tests does have a place but it is not significant in terms of automating test execution. Decreasing test creation time is a good idea but only if it does not increase the cost of maintenance. Schemes that optimise the test creation at the expense of test maintenance will in fact increase the life cycle costs instead of reducing them. (Fewster, Graham 1994, 520.)

### 5.1.2 Recorder in SWTBot – a Marketing Stunt

On the home page of SWTBot project (Pedegaonkar 2009) you can read: “SWTBot can record and playback tests…”. But the truth is that **SWTBot lacks a Recorder!** The SWTBot developers were working under it intensively till version 1.3 of the software. Since then it is not actively maintained, and even Ketan Pedegaonkar himself discourages using it. In the opinion of the author of this thesis not having a Recorder is not a big disadvantage. Recorder would help only initially but once you need to start writing reusable test modules, a Recorder looses much of its value.

## 5.2 What Cannot Be Tested with SWTBot

While writing SWTBot tests, a lot of time can be wasted while trying to do things that cannot be done. Things that will not be supported, or are not implemented yet. Therefore it seems a good idea to mention them in this thesis work, especially that the author did not found any similar list on any Internet source:

- There is **no support for native widgets** (Open/Save File Dialogs, Color Dialogs, etc.).
- When trying to test AUT’s **restart - it restarts also all the tests** and goes into infinite loop.
- **Cannot access status bar text;**
- Only **partial support for close button x** in the top right corner of every window. One can use close method for shells but it fails if there will

be a confirmation dialog (which is a standard Eclipse behavior for the main window).

- **Cannot access context menu's submenu items** (e.g. in Navigator view). There is a workaround - treat submenus as elements of main context menu - but it fails when in different submenus there are elements with repeating names (e.g. "Other…").

## 5.3 Measuring Benefit of Using Automated Tests

The table below presents the comparison between automatic and manual tests. It regards the test code from APPENDIX 2. It shows execution time in seconds of the consecutive tests which could also be observed on the movie attached as APPENDIX 3. The author has also measured the time of manual tests for all the tests. The last column presents how many times faster the automated test was.

**Table 1** Comparison of Automated and Manual Tests Execution

| Test No | Automated Test [s] | Manual Test [s] | Manual Test [s] / Automated Test [s] |
|---|---|---|---|
| 1 | 2,454 | 61 | 24,9 |
| 2 | 3,471 | 134 | 38,6 |
| 3 | 2,141 | 56 | 26,2 |
| 4 | 1,732 | 48 | 27,7 |
| 5 | 1,844 | 72 | 39,0 |
| 6 | 1,207 | 64 | 53,0 |
| 7 | 8,002 | 231 | 28,9 |
| 8 | 8,157 | 57 | 7,0 |
| 9 | 7,828 | 249 | 31,8 |
| 10 | 3,561 | 144 | 40,4 |
| 11 | 8,203 | 61 | 7,4 |
| 12 | 14,032 | 487 | 34,7 |
| 13 | 14,000 | 398 | 28,4 |
| 14 | 8,156 | 228 | 28,0 |
| 15 | 2,157 | 61 | 28,3 |
| 16 | 63,735 | 419 | 6,6 |
| 17 | 8,282 | 211 | 25,5 |
| 18 | 8,125 | 274 | 33,7 |

| | | | |
|---|---|---|---|
| 19 | 32,938 | 279 | 8,5 |
| 20 | 32,735 | 926 | 28,3 |
| 21 | 8,156 | 238 | 29,2 |
| 22 | 8,141 | 135 | 16,6 |
| 23 | 15,735 | 451 | 28,7 |
| 24 | 30,235 | 978 | 32,3 |
| 25 | 14,656 | 411 | 28,0 |
| 26 | 14,797 | 211 | 14,3 |
| 27 | 19,766 | 578 | 29,2 |
| 28 | 26,357 | 709 | 26,9 |
| 29 | 15,516 | 438 | 28,2 |
| 30 | 21,013 | 422 | 20,1 |
| | SUM = 407,132 | SUM = 9031 | AVG = 22,2 |

The last row summarizes results for all the tests. Execution of automated tests takes less than 7 minutes (407.132 seconds) while the same manual tests would take more than 2.5 hours (9031 seconds). This means that manual tests were about 22 times more time consuming than the automated tests.

This doesn't concern that the automated tests are integrated with Cruise Control's nightly builds and the tests are executed automatically without the need of programmer's intervention. Considering that the tester would have time to perform those tests once a month rather than once a day, this difference is actually much bigger.

Therefore how to measure the increase in quality? One could calculate Return on Investment (ROI)  which should show the real value that the automation gives. The ROI value is not the value of automation versus the cost of executing tests manually. It could be defined rather as the benefit of this type of testing plus the benefit of whatever the manual tester is doing while the automated tests are executing. While calculating ROI one should take into account:

- the cost of executing automated tests considering infrastructure required
- The cost of maintaining the automation as the product evolves
- The cost of additional tasks required by automation (reviewing results, maintaining documentation, training for staff, etc.)

- ROI varies depending on the business environment, application under test, and even the type of tests which were automated.

Calculating ROI for test automation project could be a topic for another bachelor thesis and it is beyond the scope of this thesis, which has more development and implementation nature. The other thing is that the author is not allowed to disclose any financial related data about NSN, therefore with these constraints it is impossible to cover calculating ROI.

# 6 CONCLUSION

As the reader of this dissertation had chance to discover, test programs written using SWTBot are relatively **easy to read and intuitive** for people familiar with basic concepts of Eclipse/Eclipse RCP. Without any doubt SWTBot's **API is large** enough to test effectively such complex application as Pegasus RCP. The biggest problem could be **lack of good documentation** and tutorials which makes it difficult to start with. The best and most up to date is Javadoc that comes with SWTBot's code. Therefore a developer which wants to start writing SWTBot tests has to spend some time with it to get familiar with the API experimenting with the code. Although SWTBot is not a commercial tool, the **growing community** of testers and developers using it seems promising considering the need of support.

Despite of what is published on SWTBot's home page, **SWTBot lacks a recorder**. Apparently there is one, but it is not actively maintained and its use is discouraged even by the project's lead developer, Ketan Pedegaonkar. Lack of recorder is not a big disadvantage since the recorder generates inflexible, non-modular, non-reusable code which makes it rather a nice to have toy than serious tool for test automation. Not having a recorder implicates also that the **tester has to be a programmer**.

Automating tests has many benefits. The tests can be run overnight allowing to utilize hardware resources more effectively. **Bugs can be detected and corrected much faster**. Developers could even execute automated GUI tests before checking-in production code, without the need to wait for nightly build tests' results. Release tester can remove already automated tests from manual tests list having **more time to test new features** rather than retesting the old ones. As the analysis has shown, the automated tests execute over **twenty times faster** comparing with manual tests. Considering that the tester would have time to perform those tests once a month rather than once a day, this difference is actually much bigger.

Pieces of code presented in this document, instructions how to achieve tests independence, error recovery, test synchronization, together with big test program implementation which has almost 1500 lines of code (APPENDIX 2) make up a great tutorial which will help developers to get familiar with SWTBot's capabilities. FileMenuTest.java consists of 30 test cases. The author has made those tests compliant with all the objectives for well-designed test cases listed in chapter 2.6. The tests proved to be reliable. They are written in a modular way allowing for code reuse. The author will continue his work as test automation engineer. Goals for nearest future include creating utilities abstract class with most useful methods, preparing training about SWTBot for developers and tracking the latest trends in GUI test automation.

# LITERATURE FOR THESIS WRITING

Bach, James. Test Automation Snake Oil. 1999.

Beck, Kent, et al. 2001. Manifesto for Agile Software Development. [referenced 23.08.2009]. Available in www-form: <URL:http://agilemanifesto.org/>

Crispin, Lisa and Gregory, Janet. 2009. Agile Testing. A Practical Guide for Testers and Agile Teams. Crawfordsville, USA. Addison-Weseley.

Ebert, Ralf. 2009. German Java Magazin article about testing Eclipse RCP applications with SWTBot. Available in www-form: <http://www.ralfebert.de/articles/swtbot/>

Fewster, Mark and Graham, Dorothy. 1994. Software Test Automation. Effective use of test automation tools. New York, USA. ACM Press.

Graham, D., Van Veenendaal, E., Evans, I., Black, R. Foundations of Software Testing. ISTQB Certification. Cengage, USA. Thomson.

Hutcheson, Marnie. 2003.Software Testing Fundamentals: Methods and Metrics. New York, USA. Wiley Publishing Inc.

Institute of Electrical and Electronics Engineers, Inc., IEEE Standard for Software Test Documentation. IEEE Std 829-1998. 1998, USA.

Li, Kanglin and Wu, Mengqui. Effective GUI Test Automation: Developing an Automated GUI Testing Tool. Alameda, USA. Sybex Inc.

Patton, Ron. 2005. Software Testing ($2^{nd}$ Edition). Indianapolis, USA. Sams Publishing.

Myers, Glenford J. 2004. The Art of Software Testing. Second Edition. New Jersey, USA. John Wiley Publishing Inc.

Pedegaonkar, Ketan. 2009. SWTBot Home Page. [referenced 19.09.2009]. Available in www-form: <URL: http://www.eclipse.org/swtbot/>

Pedegaonkar, Ketan and Rodrigues, Vitor. 2009. SWTBot FAQ. [referenced 15.09.2009]. Available in www-form: <URL: http://wiki.eclipse.org/SWTBot/FAQ>

Pettichord, Bret 2001. Seven Steps to Test Automation. Revised version of a paper originally presented at STARWEST conference, San Jose.

Vogel, Lars. 2009. Eclipse RCP – Tutorial (Eclipse 3.5). [referenced: 17.09.2009]. Available in www-form:
< http://www.vogella.de/articles/RichClientPlatform/article.html>

```java
/* SimpleExampleTest.java */

package pegasus.core;
import static org.eclipse.swtbot.swt.finder.SWTBotTestCase.assertEnabled;
import static org.eclipse.swtbot.swt.finder.SWTBotTestCase
              .assertNotEnabled;
import static org.eclipse.swtbot.swt.finder.SWTBotTestCase.assertVisible;
import org.eclipse.swtbot.eclipse.finder.SWTWorkbenchBot;
import org.junit.AfterClass;
import org.junit.BeforeClass;
import org.junit.Test;

/**
 * Simple test to prove SWTBot works
 * @author mmazurki
 */
public class SimpleTest {

  /**
   * @see SWTWorkbenchBot
   */
  private static SWTWorkbenchBot bot;

  /**
   * @see BeforeClass
   * @throws Exception
   */
  @BeforeClass
  public static void setUpBeforeClass() throws Exception {
     bot = new SWTWorkbenchBot();
  }

  /**
   * @see AfterClass
   * @throws Exception
   */
  @AfterClass
  public static void tearDownAfterClass() throws Exception {
    bot = null;
  }

  /**
   * Switch to Run perspective via the Open Perspective dialog
   */
  @Test
  public void openOtherRunPerspectiveTest() {
    bot.menu("Window").menu("Open Perspective").menu("Other...").click();
    bot.table().select("Run");
    bot.button("OK").click();

    assertVisible(bot.toolbarButtonWithTooltip("New"));
    assertNotEnabled(bot.toolbarButtonWithTooltip("Save As"));
    assertEnabled(bot.toolbarButtonWithTooltip("Search"));
  }

  /**
   * Quick switch to Test perspective
   */
  @Test
  public void openTestPerspectiveTest() {
    bot.menu("Window").menu("Open Perspective").menu("Test").click();
    bot.sleep(2000);
  }
}
```