Selenium Documentation

Release 1.0

Selenium Project

July 03, 2011

CONTENTS

Contents:

CHAPTER

ONE

NOTE TO THE READER–*DOCS BEING REVISED FOR SELENIUM 2.0*!

Hello, and welcome! The Documentation Team would like to welcome you, and to thank you for being interested in Selenium.

We are currently updating this document for the Selenium 2.0 release. This means we are currently writing and editing new material, and revising old material. While reading, you may experience typos or other minor errors. If so, please be patient with us. Rather than withholding information until it's finally complete, we are frequently checking-in new writing and revisions as we go. Still, we do check our facts first and are confident the info we've submitted is accurate and useful. Still, if you find an error, particularly in one of our code examples, please let us know. You can send an email directly to the Selenium Developers forum ("selenium-developers" <selenium-developers@googlegroups.com>) with "Docs Error" in the subject line.

We have worked very, very hard on this document. And, as just mentioned, we are once again working hard, on the new revision. Why? We absolutely believe this is the best tool for web-application testing. We feel its extensibility and flexibility, along with its tight integration with the browser, is unmatched by available proprietary tools. We are very excited to promote Selenium and, hopefully, to expand its user community. In short, we really want to "get the word out" about Selenium.

We believe you will be similarly excited once you understand how Selenium approaches test automation. It's quite different from other automation tools. Whether you are brand-new to Selenium, or have been using it for awhile, we believe this documentation will truly help to spread the knowledge around. We have aimed our writing so that those completely new to test automation can use this document as a stepping stone. However, at the same time we have included a number of advanced, test design topics that should be interesting to the experienced software engineer. In both cases we have written the "Sel-Docs" to help test engineers of all abilities to quickly become productive writing your own Selenium tests. Experienced users and "newbies" alike will benefit from our Selenium User's Guide.

Thanks very much for reading.

– the Selenium Documentation Team

CHAPTER

TWO

INTRODUCTION

2.1 Test Automation for Web Applications

Many, perhaps most, software applications today are written as web-based applications to be run in an Internet browser. The effectiveness of testing these applications varies widely among companies and organizations. In an era of highly interactive and responsive software processes where many organizations are using some form of Agile methodology, test automation is frequently becoming a requirement for software projects. Test automation is often the answer. Test automation means using a software tool to run repeatable tests against the application to be tested. For regression testing this provides that responsiveness.

There are many advantages to test automation. Most are related to the repeatability of the tests and the speed at which the tests can be executed. There are a number of commercial and open source tools available for assisting with the development of test automation. Selenium is possibly the most widely-used open source solution. This user's guide will assist both new and experienced Selenium users in learning effective techniques in building test automation for web applications.

This user's guide introduces Selenium, teaches its features, and presents commonly used best practices accumulated from the Selenium community. Many examples are provided. Also, technical information on the internal structure of Selenium and recommended uses of Selenium are provided.

Test automation has specific advantages for improving the long-term efficiency of a software team's testing processes. Test automation supports:

- Frequent regression testing
- Rapid feedback to developers
- Virtually unlimited iterations of test case execution
- Support for Agile and extreme development methodologies
- Disciplined documentation of test cases
- Customized defect reporting
- Finding defects missed by manual testing

2.2 To Automate or Not to Automate?

Is automation always advantageous? When should one decide to automate test cases?

It is **not** always advantageous to automate test cases. There are times when manual testing may be more appropriate. For instance, if the application's user interface will change considerably in the near future, then any automation might need to be rewritten anyway. Also, sometimes there simply is not enough time to build test automation. For the short term, manual testing may be more effective. If an application has a very tight deadline, there is currently no test automation available, and it's imperative that the testing get done within that time frame, then manual testing is the best solution.

2.3 Introducing Selenium

Selenium is set of different software tools each with a different approach to supporting test automation. Most Selenium QA Engineers focus on the one or two tools that most meet the needs of their project, however learning all the tools will give you many different options for approaching different test automation problems. The entire suite of tools results in a rich set of testing functions specifically geared to the needs of testing of web applications of all types. These operations are highly flexible, allowing many options for locating UI elements and comparing expected test results against actual application behavior. One of Selenium's key features is the support for executing one's tests on multiple browser platforms.

2.4 Brief History of The Selenium Project

Selenium first came to life in 2004 when Jason Huggins was testing an internal application at ThoughtWorks. Being a smart guy, he realized there were better uses of his time than manually stepping through the same tests with every change he made. He developed a Javascript library that could drive interactions with the page, allowing him to automatically rerun tests against multiple browsers. That library eventually became Selenium Core, which underlies all the functionality of Selenium Remote Control (RC) and Selenium IDE. Selenium RC was ground-breaking because no other product allowed you to control a browser from a language of your choosing.

While Selenium was a tremendous tool, it wasn't without it's drawbacks. Because of its Javascript based automation engine and the security limitations browsers apply to Javascript, different things became impossible to do. To make things "worst", webapps became more and more powerful over time, using all sorts of special features new browsers provide and making this restrictions more and more painful.

In 2006 a plucky engineer at Google named Simon Stewart started work on a project he called WebDriver. Google had long been a heavy user of Selenium, but testers had to work around the limitations of the product. Simon wanted a testing tool that spoke directly to the browser using the 'native' method for the browser and operating system, thus avoiding the restrictions of a sandboxed Javascript environment. The WebDriver project began with the aim to solve the Selenium' pain-points.

Jump to 2008. The Beijing Olympics mark China's arrival as a global power, massive mortgage default in the United States triggers the worst international recession since the Great Depression, The Dark Knight is viewed by every human (twice), still reeling from the untimely loss of Heath Ledger. But the most important story of that year was the merging of Selenium and WebDriver. Selenium had massive community and commercial support, but WebDriver was clearly the tool of the future. The joining of the two tools provided a common set of features for all users and brought some of the brightest minds in test automation under one roof. Perhaps the best explanation for why WebDriver and Selenium are merging was detailed by Simon Stewart, the creator of WebDriver, in a joint email to the WebDriver and Selenium community on August 6, 2009.

> "Why are the projects merging? Partly because webdriver addresses some shortcomings in selenium (by being able to bypass the JS sandbox, for example. And we've got a gorgeous

> API), partly because selenium addresses some shortcomings in webdriver (such as supporting a broader range of browsers) and partly because the main selenium contributors and I felt that it was the best way to offer users the best possible framework."

2.5 Selenium's Tool Suite

Selenium is composed of multiple software tools. Each has a specific role.

2.5.1 Selenium 2 (aka. Selenium Webdriver)

Selenium 2 is the future direction of the project and the newest addition to the Selenium toolkit. This brand new automation tool provides all sorts of awesome features, including a more cohesive and object oriented API as well as an answer to the limitations of the old implementation.

As you can read in Brief History of The Selenium Project, both the Selenium and WebDriver developers agreed that both tools have advantages and that merging the two projects would make a much more robust automation tool.

Selenium 2.0 is the product of that effort. It supports the WebDriver API and underlying technology, along with the Selenium 1 technology underneath the WebDriver API for maximum flexibility in porting your tests. In addition, Selenium 2 still runs Selenium 1's Selenium RC interface for backwards compatibility.

2.5.2 Selenium 1 (aka. Selenium RC or Remote Control)

As you can read in Brief History of The Selenium Project, Selenium RC was the main Selenium project for a long time, before the WebDriver/Selenium merge brought up Selenium 2, the newest and more powerful tool.

Selenium 1 is still actively supported (mostly in maintenance mode) and provides some features that may not be available in Selenium 2 for a while, including support for several languages (Java, Javascript, PRuby, HP, Python, Perl and C#) and support for almost every browser out there.

2.5.3 Selenium IDE

Selenium IDE (Integrated Development Environment) is a prototyping tool for building test scripts. It is a Firefox plugin and provides an easy-to-use interface for developing automated tests. Selenium IDE has a recording feature, which records user actions as they are performed and then exports them as a reusable script in one of many programming languages that can be later executed.

Note: Even though Selenium IDE has a "Save" feature that allows users to keep the tests in a table-based format for later import and execution, it *is not designed to run your test passes nor is it designed to build all the automated tests you will need*. Specifically, Selenium IDE doesn't provide iteration or conditional statements for test scripts. At the time of writing there is no plan to add such thing. The reasons are partly technical and partly based on the Selenium developers encouraging best practices in test automation which always requires some amount of programming. **Selenium IDE is simply intended as a rapid prototyping tool**. The Selenium developers recommend for serious, robust test automation either Selenium 2 or Selenium 1 to be used with one of the many supported programming languages.

2.5.4 Selenium-Grid

Selenium-Grid allows the Selenium RC solution to scale for large test suites and for test suites that must be run in multiple environments. Selenium Grid allows you to run your tests in parallel, that is, different tests can be run at the same time on different remote machines. This has two advantages. First, if you have a large test suite, or a slow-running test suite, you can boost its performance substantially by using Selenium Grid to divide your test suite to run different tests at the same time using those different machines. Also, if you must run your test suite on multiple environments you can have different remote machines supporting and running your tests in them at the same time. In each case Selenium Grid greatly improves the time it takes to run your suite by making use of parallel processing.

2.6 Choosing Your Selenium Tool

Many people get started with Selenium IDE. If you are not already experienced with a programming or scripting language you can use Selenium IDE to get familiar with Selenium commands. Using the IDE you can create simple tests quickly, sometimes within seconds.

We don't, however, recommend you do all your test automation using Selenium IDE. To effectively use Selenium you will need to build and run your tests using either Selenium 2 or Selenium 1 in conjunction with one of the supported programming languages. Which one you choose depends on you.

At the time of writing the Selenium developers are planning on the Selenium-WebDriver API being the future direction for Selenium. Selenium 1 is provided for backwards compatibility. Still, both have strengths and weaknesses which are discussed in the corresponding chapters of this document.

We recommend those who are completely new to Selenium to read through these sections. However, for those who are adopting Selenium for the first time, and therefore building a new test suite from scratch, you will probably want to go with Selenium 2 since this is the portion of Selenium that will continue to be supported in the future.

2.7 Supported Browsers

IMPORTANT: This list was for Sel 1.0, It requires updating for Sel2.0–we will do that very soon.

Browser	Selenium IDE	Selenium 1 (RC)	Operating Systems
Firefox 3.x	Record and playback tests	Start browser, run tests	Windows, Linux, Mac
Firefox 3	Record and playback tests	Start browser, run tests	Windows, Linux, Mac
Firefox 2	Record and playback tests	Start browser, run tests	Windows, Linux, Mac
IE 8	Test execution only via Selenium RC*	Start browser, run tests	Windows
IE 7	Test execution only via Selenium RC*	Start browser, run tests	Windows
IE 6	Test execution only via Selenium RC*	Start browser, run tests	Windows
Safari 4	Test execution only via Selenium RC	Start browser, run tests	Windows, Mac
Safari 3	Test execution only via Selenium RC	Start browser, run tests	Windows, Mac
Safari 2	Test execution only via Selenium RC	Start browser, run tests	Windows, Mac
Opera 10	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Opera 9	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Opera 8	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Google Chrome	Test execution only via Selenium RC	Start browser, run tests	Windows, Linux, Mac
Others	Test execution only via Selenium RC	Partial support possible**	As applicable

* Tests developed on Firefox via Selenium IDE can be executed on any other supported browser via a simple Selenium RC command line.

** Selenium RC server can start any executable, but depending on browser security settings there may be technical limitations that would limit certain features.

2.8 Flexibility and Extensibility

You'll find that Selenium is highly flexible. There are many ways you can add functionality to both Selenium test scripts and Selenium's framework to customize your test automation. This is perhaps Selenium's greatest strength when compared with other automation tools. These customizations are described in various places throughout this document. In addition, since Selenium is Open Source, the sourcecode can always be downloaded and modified.

2.9 What's in this Book?

This user's guide targets both new users and those who have already used Selenium but are seeking additional knowledge. We introduce Selenium to new users and we do not assume prior Selenium experience. We do assume, however, that the user has at least a basic understanding of test automation. For the more experienced user, this guide can act as a reference. For the more experienced, we recommend browsing the chapter and subheadings. We've provided information on the Selenium architecture, examples of common usage, and a chapter on test design techniques.

The remaining chapters of the reference present:

Selenium IDE Introduces Selenium IDE and describes how to use it to build test scripts. using the Selenium Integrated Development Environment. If you are not experienced in programming, but still hoping to learn test automation this is where you should start and you'll find you can create quite a few automated tests with Selenium IDE. Also, if you are experienced in programming, this chapter may still interest you in that you can use Selenium IDE to do rapid prototyping of your tests. This section also demonstrates how your test script can be "exported" to a programming language for adding more advanced capabilities not supported by Selenium IDE.

Selenium 2 Explains how to develop an automated test program using Selenium 2.

Selenium 1 Explains how to develop an automated test program using the Selenium RC API. Many examples are presented in both programming languages and scripting languages. Also, the installation and setup of Selenium RC is covered here. The various modes, or configurations, that Selenium RC supports are described, along with their trade-offs and limitations. An architecture diagram is provided to help illustrate these points. Solutions to common problems frequently difficult for new Sel-R users are described here, for instance, handling Security Certificates, https requests, pop-ups, and the opening of new windows.

Test Design Considerations This chapter presents programming techniques for use with Selenium-WebDriver and Selenium RC. We also demonstrate techniques commonly asked about in the user forum such as how to design setup and teardown functions, how to implement data-driven tests (tests where one can varies the data between test passes) and other methods of programming common test automation tasks.

Selenium-Grid *This chapter is not yet developed.*

User extensions Describes ways that Selenium can be modified, extended and customized.

2.10 The Documentation Team–Authors Past and Present

In alphabetical order, the following people have made significant contributions to the authoring of this user's guide or with out publishing infrastructure or both.

- Dave Hunt
- Mary Ann May-Pumphrey
- Noah Sussman
- Paul Grandjean
- Peter Newhook
- Santiago Suarez Ordonez
- Tarun Kumar

2.10.1 Acknowledgements

A huge special thanks goes to Patrick Lightbody. As an administrator of the SeleniumHQ website, creator of Selenium RC, and long term involvement in the Selenium community, his support was invaluable when writing the original user's guide. Patrick helped us understand our audience. He also set us up with everything we needed on the seleniumhq.org website for publishing the documents. Also thanks goes to Andras Hatvani for his advice on publishing solutions, and to Amit Kumar for participating in our discussions and for assisting with reviewing the document.

And of course, we must *recognize the Selenium Developers*. They have truly designed an amazing tool. Without the vision of the original designers, and the continued efforts of the current developers, we would not have such a great tool to pass on to you.

CHAPTER

THREE

SELENIUM-IDE

3.1 Introduction

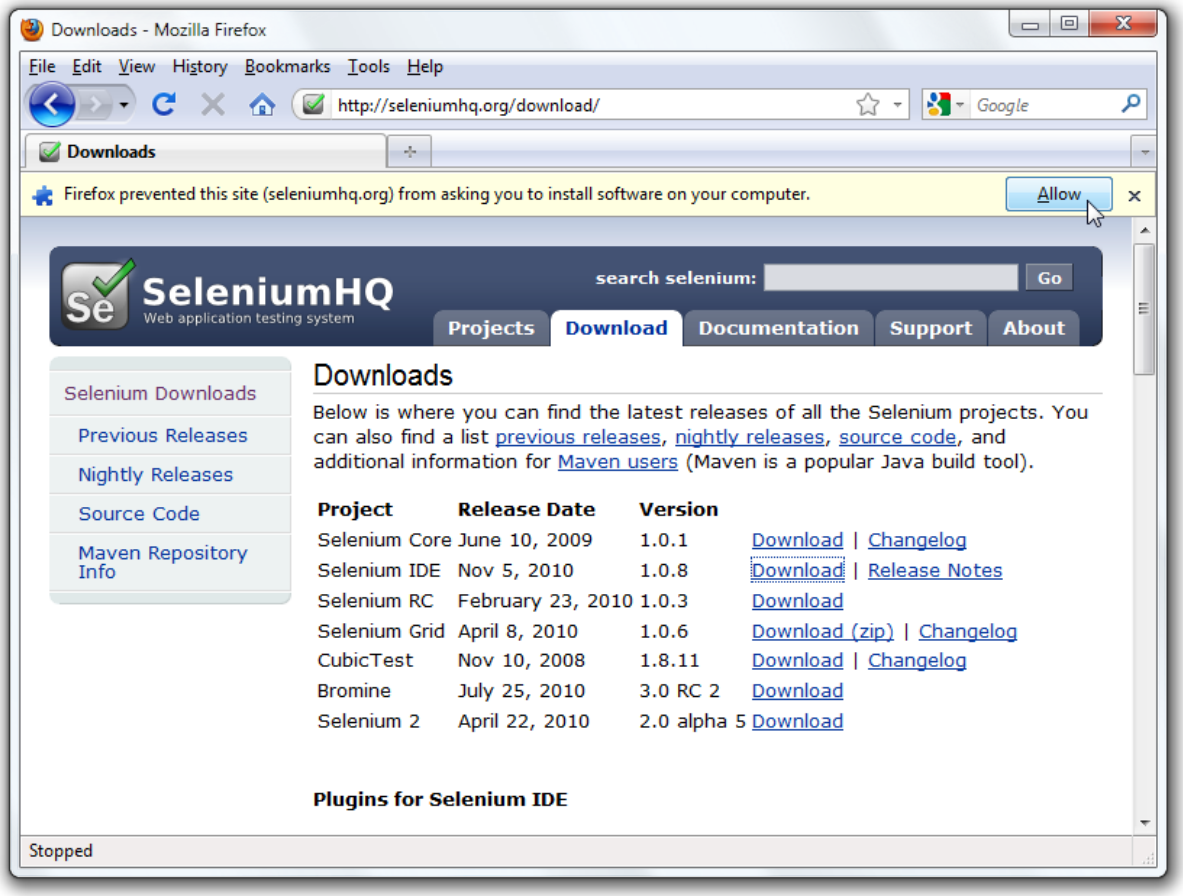
The Selenium-IDE (Integrated Development Environment) is the tool you use to develop your Selenium test cases. It's an easy-to-use Firefox plug-in and is generally the most efficient way to develop test cases. It also contains a context menu that allows you to first select a UI element from the browser's currently displayed page and then select from a list of Selenium commands with parameters pre-defined according to the context of the selected UI element. This is not only a time-saver, but also an excellent way of learning Selenium script syntax.

This chapter is all about the Selenium IDE and how to use it effectively.

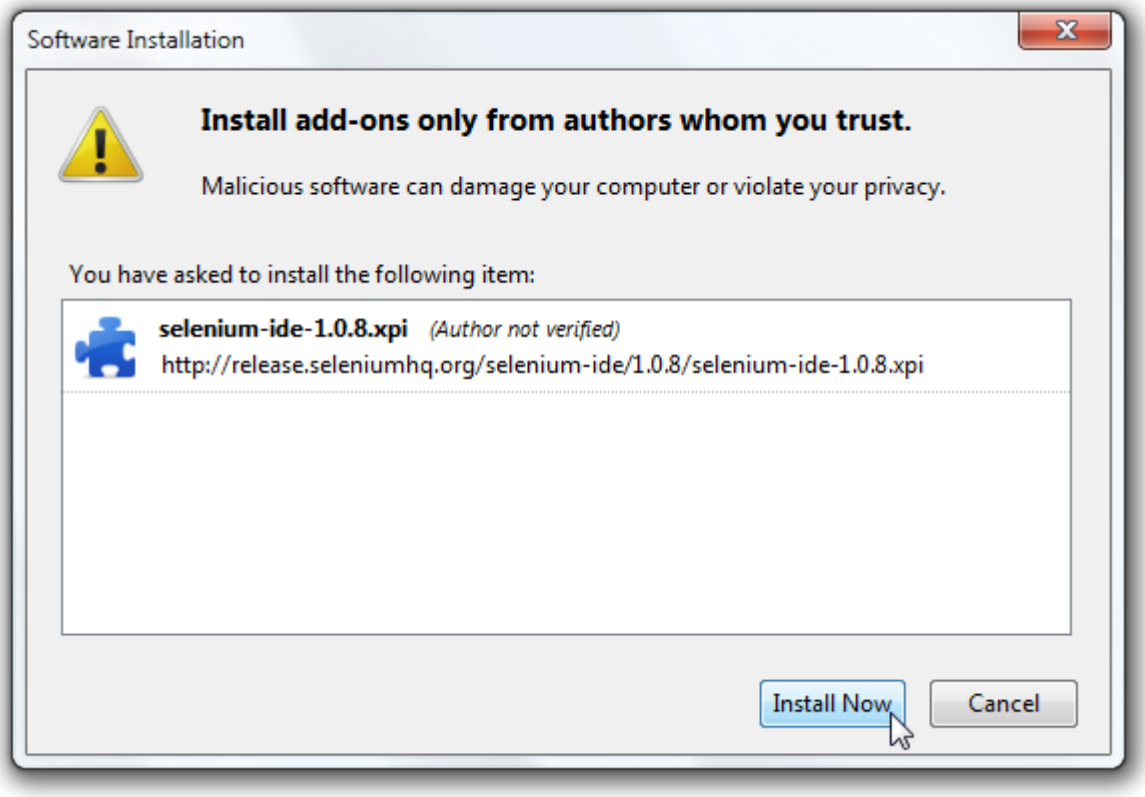
3.2 Installing the IDE

Using Firefox, first, download the IDE from the SeleniumHQ downloads page

Firefox will protect you from installing addons from unfamiliar locations, so you will need to click 'Allow' to proceed with the installation, as shown in the following screenshot.

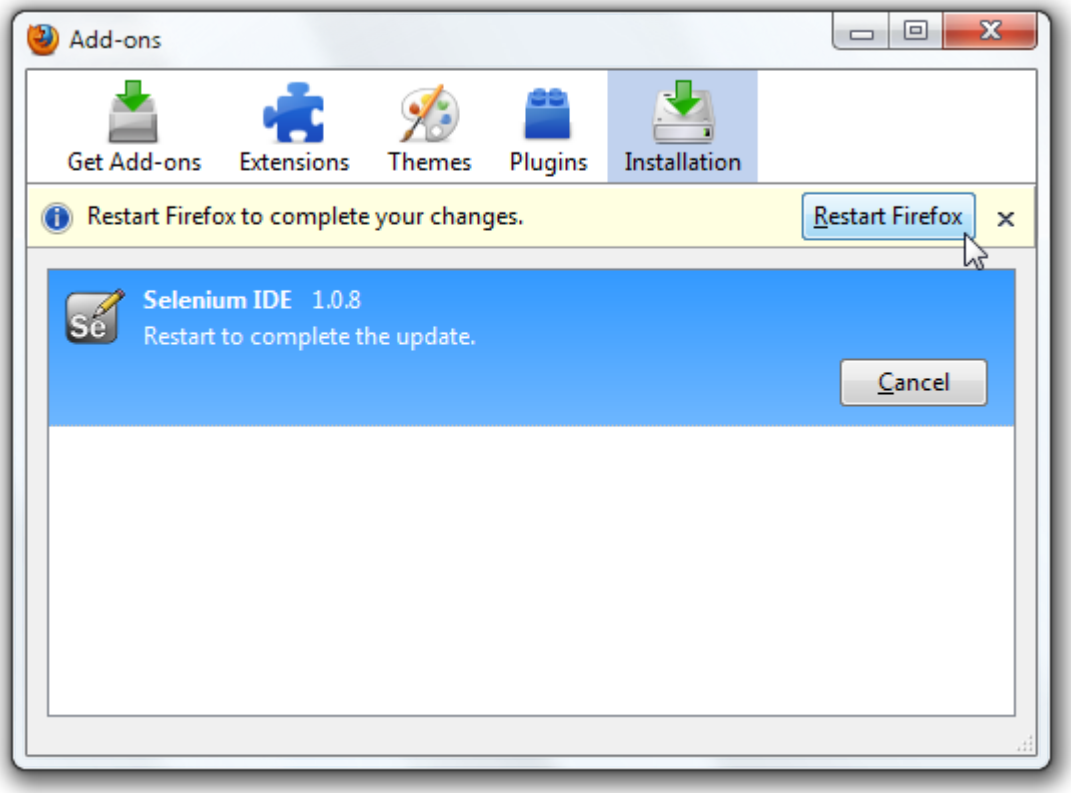


When downloading from Firefox, you'll be presented with the following window.

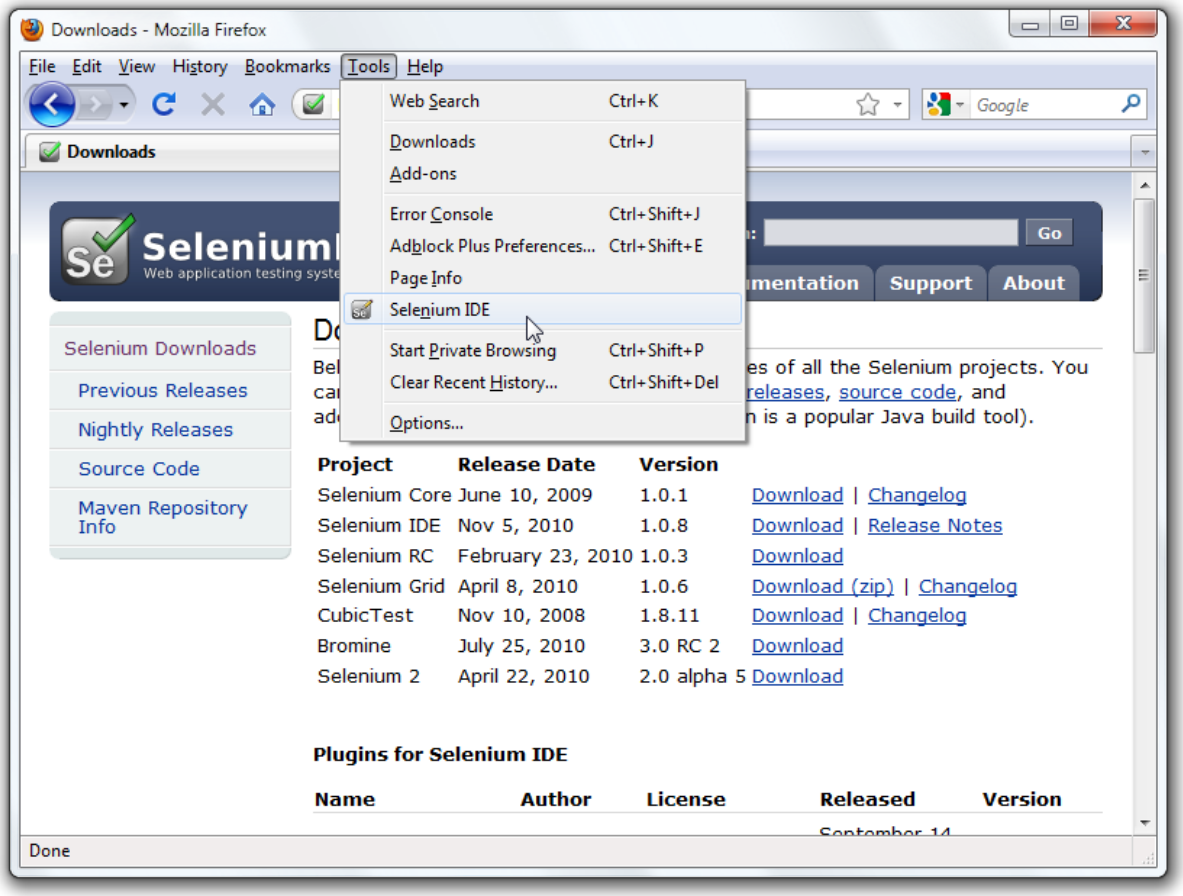


Select Install Now. The Firefox Add-ons window pops up, first showing a progress bar, and when the

download is complete, displays the following.

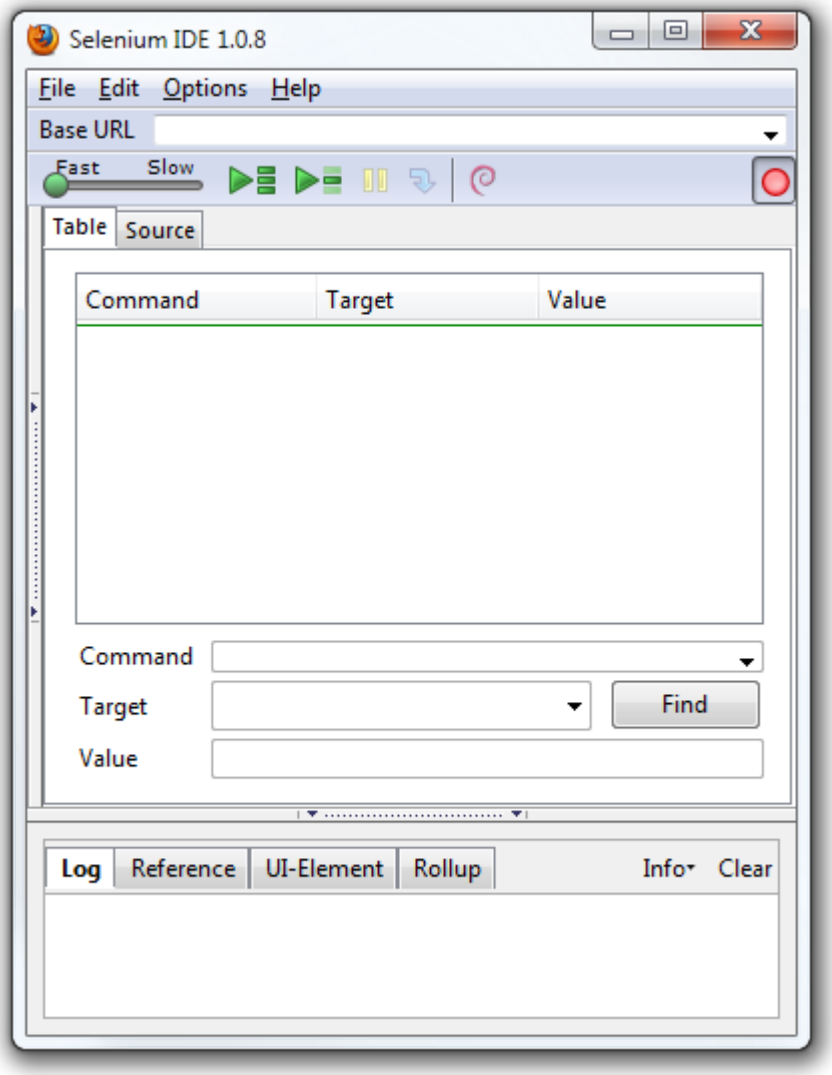


Restart Firefox. After Firefox reboots you will find the Selenium-IDE listed under the Firefox Tools menu.



3.3 Opening the IDE

To run the Selenium-IDE, simply select it from the Firefox Tools menu. It opens as follows with an empty script-editing window and a menu for loading, or creating new test cases.



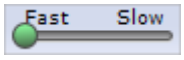
3.4 IDE Features

3.4.1 Menu Bar

The File menu allows you to create, open, and save test case and test suite files. The Edit menu allows copy, paste, delete, undo, and select all operations for editing the commands in your test case. The Options menu allows the changing of settings. You can set the timeout value for certain commands, add user-defined user extensions to the base set of Selenium commands, and specify the format (language) used when saving your test cases. The Help menu is the standard Firefox Help menu; only one item on this menu–UI-Element Documentation–pertains to Selenium-IDE.

3.4.2 Toolbar

The toolbar contains buttons for controlling the execution of your test cases, including a step feature for debugging your test cases. The right-most button, the one with the red-dot, is the record button.



Speed Control: controls how fast your test case runs.


Run All: Runs the entire test suite when a test suite with multiple test cases is loaded.


Run: Runs the currently selected test. When only a single test is loaded this button and the Run All button have the same effect.


Pause/Resume: Allows stopping and re-starting of a running test case.


Step: Allows you to "step" through a test case by running it one command at a time. Use for debugging test cases.


TestRunner Mode: Allows you to run the test case in a browser loaded with the Selenium-Core TestRunner. The TestRunner is not commonly used now and is likely to be deprecated. This button is for evaluating test cases for backwards compatibility with the TestRunner. Most users will probably not need this button.


Apply Rollup Rules: This advanced feature allows repetitive sequences of Selenium commands to be grouped into a single action. Detailed documentation on rollup rules can be found in the UI-Element Documentation on the Help menu.


Record: Records the user's browser actions.

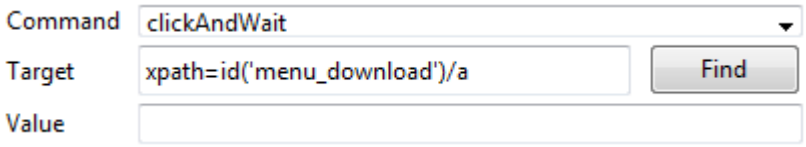
3.4.3 Test Case Pane

Your script is displayed in the test case pane. It has two tabs, one for displaying the command and their parameters in a readable "table" format.

Command	Target	Value
open	/	
waitForPageToLoad		
clickAndWait	xpath=id('menu_download')/a	
assertTitle	Downloads	
verifyText	xpath=id('mainContent')/h2	Downloads

The other tab - Source displays the test case in the native format in which the file will be stored. By default, this is HTML although it can be changed to a programming language such as Java or C#, or a scripting language like Python. See the Options menu for details. The Source view also allows one to edit the test case in its raw form, including copy, cut and paste operations.

The Command, Target, and Value entry fields display the currently selected command along with its parameters. These are entry fields where you can modify the currently selected command. The first parameter specified for a command in the Reference tab of the bottom pane always goes in the Target field. If a second parameter is specified by the Reference tab, it always goes in the Value field.



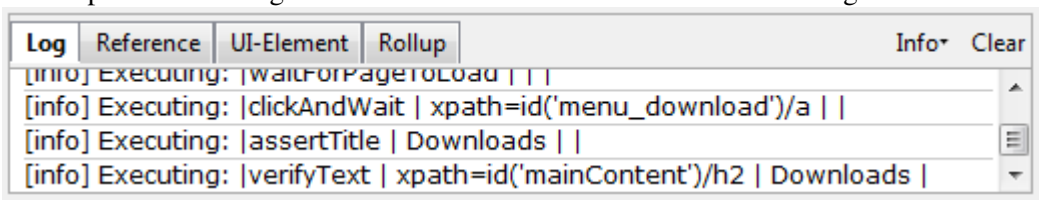
If you start typing in the Command field, a drop-down list will be populated based on the first characters you type; you can then select your desired command from the drop-down.

3.4.4 Log/Reference/UI-Element/Rollup Pane

The bottom pane is used for four different functions–Log, Reference, UI-Element, and Rollup–depending on which tab is selected.

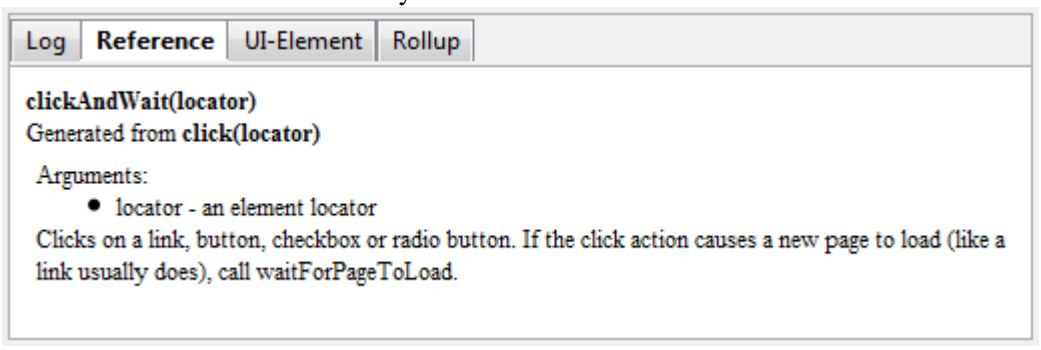
Log

When you run your test case, error messages and information messages showing the progress are displayed in this pane automatically, even if you do not first select the Log tab. These messages are often useful for test case debugging. Notice the Clear button for clearing the Log. Also notice the Info button is a drop-down allowing selection of different levels of information to log.



Reference

The Reference tab is the default selection whenever you are entering or modifying Selenese commands and parameters in Table mode. In Table mode, the Reference pane will display documentation on the current command. When entering or modifying commands, whether from Table or Source mode, it is critically important to ensure that the parameters specified in the Target and Value fields match those specified in the parameter list in the Reference pane. The number of parameters provided must match the number specified, the order of parameters provided must match the order specified, and the type of parameters provided must match the type specified. If there is a mismatch in any of these three areas, the command will not run correctly.



While the Reference tab is invaluable as a quick reference, it is still often necessary to consult the Selenium Reference document.

UI-Element and Rollup

Detailed information on these two panes (which cover advanced features) can be found in the UI-Element Documentation on the Help menu of Selenium-IDE.

3.5 Building Test Cases

There are three primary methods for developing test cases. Frequently, a test developer will require all three techniques.

3.5.1 Recording

Many first-time users begin by recording a test case from their interactions with a website. When Selenium-IDE is first opened, the record button is ON by default. If you do not want Selenium-IDE to begin recording automatically you can turn this off by going under Options > Options... and deselecting "Start recording immediately on open."

During recording, Selenium-IDE will automatically insert commands into your test case based on your actions. Typically, this will include:

- clicking a link - *click* or *clickAndWait* commands
- entering values - *type* command
- selecting options from a drop-down listbox - *select* command
- clicking checkboxes or radio buttons - *click* command

Here are some "gotchas" to be aware of:

- The *type* command may require clicking on some other area of the web page for it to record.
- Following a link usually records a *click* command. You will often need to change this to *clickAndWait* to ensure your test case pauses until the new page is completely loaded. Otherwise, your test case will continue running commands before the page has loaded all its UI elements. This will cause unexpected test case failures.

3.5.2 Adding Verifications and Asserts With the Context Menu

Your test cases will also need to check the properties of a web-page. This requires *assert* and *verify* commands. We won't describe the specifics of these commands here; that is in the chapter on *"Selenese" Selenium Commands*. Here we'll simply describe how to add them to your test case.

With Selenium-IDE recording, go to the browser displaying your test application and right click anywhere on the page. You will see a context menu showing *verify* and/or *assert* commands.

The first time you use Selenium, there may only be one Selenium command listed. As you use the IDE however, you will find additional commands will quickly be added to this menu. Selenium-IDE will attempt to predict what command, along with the parameters, you will need for a selected UI element on the current web-page.

Let's see how this works. Open a web-page of your choosing and select a block of text on the page. A paragraph or a heading will work fine. Now, right-click the selected text. The context menu should give you a *verifyTextPresent* command and the suggested parameter should be the text itself.

Also, notice the Show All Available Commands menu option. This shows many, many more commands, again, along with suggested parameters, for testing your currently selected UI element.

Try a few more UI elements. Try right-clicking an image, or a user control like a button or a checkbox. You may need to use Show All Available Commands to see options other than *verifyTextPresent*. Once you select these other options, the more commonly used ones will show up on the primary context menu. For example, selecting *verifyElementPresent* for an image should later cause that command to be available on the primary context menu the next time you select an image and right-click.

Again, these commands will be explained in detail in the chapter on Selenium commands. For now though, feel free to use the IDE to record and select commands into a test case and then run it. You can learn a lot about the Selenium commands simply by experimenting with the IDE.

3.5.3 Editing

Insert Command

Table View

Select the point in your test case where you want to insert the command. To do this, in the Test Case Pane, left-click on the line where you want to insert a new command. Right-click and select Insert Command; the IDE will add a blank line just ahead of the line you selected. Now use the command editing text fields to enter your new command and its parameters.

Source View

Select the point in your test case where you want to insert the command. To do this, in the Test Case Pane, left-click between the commands where you want to insert a new command, and enter the HTML tags needed to create a 3-column row containing the Command, first parameter (if one is required by the Command), and second parameter (again, if one is required). Be sure to save your test before switching back to Table view.

Insert Comment

Comments may be added to make your test case more readable. These comments are ignored when the test case is run.

Comments may also be used to add vertical white space (one or more blank lines) in your tests; just create empty comments. An empty command will cause an error during execution; an empty comment won't.

Table View

Select the line in your test case where you want to insert the comment. Right-click and select Insert Comment. Now use the Command field to enter the comment. Your comment will appear in purple font.

Source View

Select the point in your test case where you want to insert the comment. Add an HTML-style comment, i.e., `<!-- your comment here -->`.

Edit a Command or Comment

Table View

Simply select the line to be changed and edit it using the Command, Target, and Value fields.

Source View

Since Source view provides the equivalent of a WYSIWYG editor, simply modify which line you wish–command, parameter, or comment.

3.5.4 Opening and Saving a Test Case

Like most programs, there are Save and Open commands under the File menu. However, Selenium distinguishes between test cases and test suites. To save your Selenium-IDE tests for later use you can either save the individual test cases, or save the test suite. If the test cases of your test suite have not been saved, you'll be prompted to save them before saving the test suite.

When you open an existing test case or suite, Selenium-IDE displays its Selenium commands in the Test Case Pane.

3.6 Running Test Cases

The IDE allows many options for running your test case. You can run a test case all at once, stop and start it, run it one line at a time, run a single command you are currently developing, and you can do a batch run of an entire test suite. Execution of test cases is very flexible in the IDE.

Run a Test Case Click the Run button to run the currently displayed test case.

Run a Test Suite Click the Run All button to run all the test cases in the currently loaded test suite.

Stop and Start The Pause button can be used to stop the test case while it is running. The icon of this button then changes to indicate the Resume button. To continue click Resume.

Stop in the Middle You can set a breakpoint in the test case to cause it to stop on a particular command. This is useful for debugging your test case. To set a breakpoint, select a command, right-click, and from the context menu select Toggle Breakpoint.

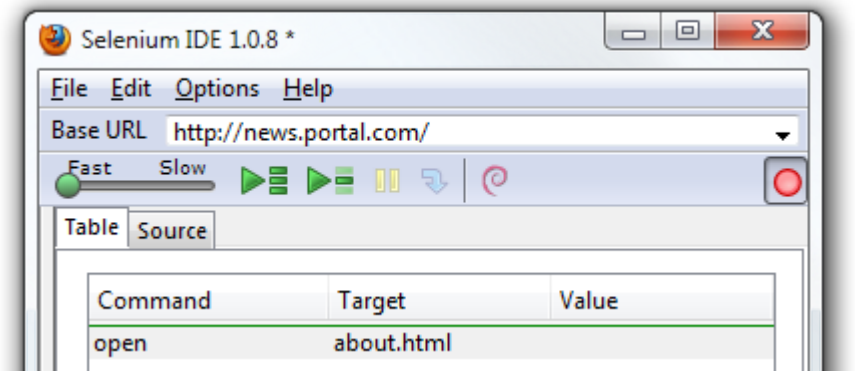
Start from the Middle You can tell the IDE to begin running from a specific command in the middle of the test case. This also is used for debugging. To set a startpoint, select a command, right-click, and from the context menu select Set/Clear Start Point.

Run Any Single Command Double-click any single command to run it by itself. This is useful when writing a single command. It lets you immediately test a command you are constructing, when you are not sure if it is correct. You can double-click it to see if it runs correctly. This is also available from the context menu.

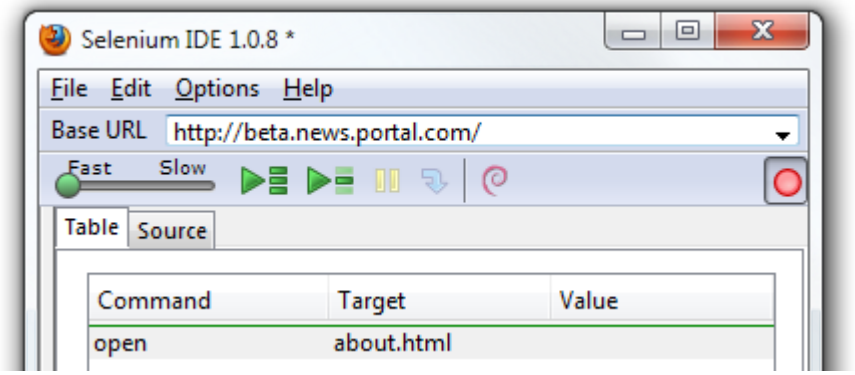
3.7 Using Base URL to Run Test Cases in Different Domains

The *Base URL* field at the top of the Selenium-IDE window is very useful for allowing test cases to be run across different domains. Suppose that a site named http://news.portal.com had an in-house beta

site named http://beta.news.portal.com. Any test cases for these sites that begin with an *open* statement should specify a *relative URL* as the argument to *open* rather than an *absolute URL* (one starting with a protocol such as http: or https:). Selenium-IDE will then create an absolute URL by appending the *open* command's argument onto the end of the value of Base URL. For example, the test case below would be run against http://news.portal.com/about.html:



This same test case with a modified Base URL setting would be run against http://beta.news.portal.com/about.html:



3.8 Selenium Commands – "Selenese"

Selenium commands, often called *selenese*, are the set of commands that run your tests. A sequence of these commands is a *test script*. Here we explain those commands in detail, and we present the many choices you have in testing your web application when using Selenium.

Selenium provides a rich set of commands for fully testing your web-app in virtually any way you can imagine. The command set is often called *selenese*. These commands essentially create a testing language.

In selenese, one can test the existence of UI elements based on their HTML tags, test for specific content, test for broken links, input fields, selection list options, submitting forms, and table data among other things. In addition Selenium commands support testing of window size, mouse position, alerts, Ajax functionality, pop up windows, event handling, and many other web-application features. The Command Reference lists all the available commands.

A *command* is what tells Selenium what to do. Selenium commands come in three "flavors": **Actions**, **Accessors** and **Assertions**.

- **Actions** are commands that generally manipulate the state of the application. They do things like "click this link" and "select that option". If an Action fails, or has an error, the execution of the current test is stopped.

Many Actions can be called with the "AndWait" suffix, e.g. "clickAndWait". This suffix tells Selenium that the action will cause the browser to make a call to the server, and that Selenium should wait for a new page to load.

- **Accessors** examine the state of the application and store the results in variables, e.g. "storeTitle". They are also used to automatically generate Assertions.
- **Assertions** are like Accessors, but they verify that the state of the application conforms to what is expected. Examples include "make sure the page title is X" and "verify that this checkbox is checked".

  All Selenium Assertions can be used in 3 modes: "assert", "verify", and " waitFor". For example, you can "assertText", "verifyText" and "waitForText". When an "assert" fails, the test is aborted. When a "verify" fails, the test will continue execution, logging the failure. This allows a single "assert" to ensure that the application is on the correct page, followed by a bunch of "verify" assertions to test form field values, labels, etc.

  "waitFor" commands wait for some condition to become true (which can be useful for testing Ajax applications). They will succeed immediately if the condition is already true. However, they will fail and halt the test if the condition does not become true within the current timeout setting (see the setTimeout action below).

3.9 Script Syntax

Selenium commands are simple, they consist of the command and two parameters. For example:

verifyText	//div//a[2]	Login

The parameters are not always required; it depends on the command. In some cases both are required, in others one parameter is required, and in still others the command may take no parameters at all. Here are a couple more examples:

goBackAndWait		
verifyTextPresent		Welcome to My Home Page
type	id=phone	(555) 666-7066
type	id=address1	${myVariableAddress}

The command reference describes the parameter requirements for each command.

Parameters vary, however they are typically:

- a *locator* for identifying a UI element within a page.
- a *text pattern* for verifying or asserting expected page content
- a *text pattern* or a selenium variable for entering text in an input field or for selecting an option from an option list.

Locators, text patterns, selenium variables, and the commands themselves are described in considerable detail in the section on Selenium Commands.

Selenium scripts that will be run from Selenium-IDE will be be stored in an HTML text file format. This consists of an HTML table with three columns. The first column identifies the Selenium command, the second is a target, and the final column contains a value. The second and third columns may not require values depending on the chosen Selenium command, but they should be present. Each table row represents a new Selenium command. Here is an example of a test that opens a page, asserts the page title and then verifies some content on the page:

```
<table>
    <tr><td>open</td><td></td><td>/download/</td></tr>
    <tr><td>assertTitle</td><td></td><td>Downloads</td></tr>
    <tr><td>verifyText</td><td>//h2</td><td>Downloads</td></tr>
</table>
```

Rendered as a table in a browser this would look like the following:

open		/download/
assertTitle		Downloads
verifyText	//h2	Downloads

The Selenese HTML syntax can be used to write and run tests without requiring knowledge of a programming language. With a basic knowledge of selenese and Selenium-IDE you can quickly produce and run testcases.

3.10 Test Suites

A test suite is a collection of tests. Often one will run all the tests in a test suite as one continuous batch-job.

When using Selenium-IDE, test suites also can be defined using a simple HTML file. The syntax again is simple. An HTML table defines a list of tests where each row defines the filesystem path to each test. An example tells it all.

```
<html>
<head>
<title>Test Suite Function Tests - Priority 1</title>
</head>
<body>
<table>
  <tr><td><b>Suite Of Tests</b></td></tr>
  <tr><td><a href="./Login.html">Login</a></td></tr>
  <tr><td><a href="./SearchValues.html">Test Searching for Values</a></td></tr>
  <tr><td><a href="./SaveValues.html">Test Save</a></td></tr>
</table>
</body>
</html>
```

A file similar to this would allow running the tests all at once, one after another, from the Selenium-IDE.

Test suites can also be maintained when using Selenium-RC. This is done via programming and can be done a number of ways. Commonly Junit is used to maintain a test suite if one is using Selenium-RC with Java. Additionally, if C# is the chosen language, Nunit could be employed. If using an interpreted language like Python with Selenium-RC than some simple programming would be involved in setting up a test suite. Since the whole reason for using Sel-RC is to make use of programming logic for your testing this usually isn't a problem.

3.11 Commonly Used Selenium Commands

To conclude our introduction of Selenium, we'll show you a few typical Selenium commands. These are probably the most commonly used commands for building tests.

open opens a page using a URL.

click/clickAndWait performs a click operation, and optionally waits for a new page to load.

verifyTitle/assertTitle verifies an expected page title.

verifyTextPresent verifies expected text is somewhere on the page.

verifyElementPresent verifies an expected UI element, as defined by its HTML tag, is present on the page.

verifyText verifies expected text and it's corresponding HTML tag are present on the page.

verifyTable verifies a table's expected contents.

waitForPageToLoad pauses execution until an expected new page loads. Called automatically when clickAndWait is used.

waitForElementPresent pauses execution until an expected UI element, as defined by its HTML tag, is present on the page.

3.12 Verifying Page Elements

Verifying UI elements on a web page is probably the most common feature of your automated tests. Selenese allows multiple ways of checking for UI elements. It is important that you understand these different methods because these methods define what you are actually testing.

For example, will you test that...

1. an element is present somewhere on the page?
2. specific text is somewhere on the page?
3. specific text is at a specific location on the page?

For example, if you are testing a text heading, the text and its position at the top of the page are probably relevant for your test. If, however, you are testing for the existence of an image on the home page, and the web designers frequently change the specific image file along with its position on the page, then you only want to test that *an image* (as opposed to the specific image file) exists *somewhere on the page*.

3.13 Assertion or Verification?

Choosing between "assert" and "verify" comes down to convenience and management of failures. There's very little point checking that the first paragraph on the page is the correct one if your test has already failed when checking that the browser is displaying the expected page. If you're not on the correct page, you'll probably want to abort your test case so that you can investigate the cause and fix the issue(s) promptly. On the other hand, you may want to check many attributes of a page without aborting the test case on the first failure as this will allow you to review all failures on the page and take the appropriate action. Effectively an "assert" will fail the test and abort the current test case, whereas a "verify" will fail the test and continue to run the test case.

The best use of this feature is to logically group your test commands, and start each group with an "assert" followed by one or more "verify" test commands. An example follows:

Command	Target	Value
open	/download/	
assertTitle	Downloads	
verifyText	//h2	Downloads
assertTable	1.2.1	Selenium IDE
verifyTable	1.2.2	June 3, 2008
verifyTable	1.2.3	1.0 beta 2

The above example first opens a page and then "asserts" that the correct page is loaded by comparing the title with the expected value. Only if this passes will the following command run and "verify" that the text is present in the expected location. The test case then "asserts" the first column in the second row of the first table contains the expected value, and only if this passed will the remaining cells in that row be "verified".

3.13.1 verifyTextPresent

The command `verifyTextPresent` is used to verify *specific text exists somewhere on the page*. It takes a single argument–the text pattern to be verified. For example:

Command	Target	Value
verifyTextPresent	Marketing Analysis	

This would cause Selenium to search for, and verify, that the text string "Marketing Analysis" appears somewhere on the page currently being tested. Use `verifyTextPresent` when you are interested in only the text itself being present on the page. Do not use this when you also need to test where the text occurs on the page.

3.13.2 verifyElementPresent

Use this command when you must test for the presence of a specific UI element, rather then its content. This verification does not check the text, only the HTML tag. One common use is to check for the presence of an image.

Command	Target	Value
verifyElementPresent	//div/p/img	

This command verifies that an image, specified by the existence of an <img> HTML tag, is present on the page, and that it follows a tag and a <p> tag. The first (and only) parameter is a *locator* for telling the Selenese command how to find the element. Locators are explained in the next section.

`verifyElementPresent` can be used to check the existence of any HTML tag within the page. You can check the existence of links, paragraphs, divisions , etc. Here are a few more examples.

Command	Target	Value
verifyElementPresent	//div/p	
verifyElementPresent	//div/a	
verifyElementPresent	id=Login	
verifyElementPresent	link=Go to Marketing Research	
verifyElementPresent	//a[2]	
verifyElementPresent	//head/title	

These examples illustrate the variety of ways a UI element may be tested. Again, locators are explained in the next section.

3.13.3 verifyText

Use `verifyText` when both the text and its UI element must be tested. `verifyText` must use a locator. If you choose an *XPath* or *DOM* locator, you can verify that specific text appears at a specific location on the page relative to other UI components on the page.

Command	Target	Value
verifyText	//table/tr/td/div/p	This is my text and it occurs right after the div inside the table.

3.14 Locating Elements

For many Selenium commands, a target is required. This target identifies an element in the content of the web application, and consists of the location strategy followed by the location in the format `locatorType=location`. The locator type can be omitted in many cases. The various locator types are explained below with examples for each.

3.14.1 Locating by Identifier

This is probably the most common method of locating elements and is the catch-all default when no recognized locator type is used. With this strategy, the first element with the id attribute value matching the location will be used. If no element has a matching id attribute, then the first element with a name attribute matching the location will be used.

For instance, your page source could have id and name attributes as follows:

```
1 <html>
2  <body>
3   <form id="loginForm">
4    <input name="username" type="text" />
5    <input name="password" type="password" />
6    <input name="continue" type="submit" value="Login" />
7   </form>
8  </body>
9 <html>
```

The following locator strategies would return the elements from the HTML snippet above indicated by line number:

- `identifier=loginForm` (3)
- `identifier=password` (4)
- `identifier=continue` (5)
- `continue` (5)

Since the `identifier` type of locator is the default, the `identifier=` in the first three examples above is not necessary.

Locating by Id

This type of locator is more limited than the identifier locator type, but also more explicit. Use this when you know an element's id attribute.

```
<html>
 <body>
  <form id="loginForm">
   <input name="username" type="text" />
   <input name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
   <input name="continue" type="button" value="Clear" />
  </form>
 </body>
<html>
```

- `id=loginForm` (3)

Locating by Name

The name locator type will locate the first element with a matching name attribute. If multiple elements have the same value for a name attribute, then you can use filters to further refine your location strategy. The default filter type is value (matching the value attribute).

```
<html>
 <body>
  <form id="loginForm">
   <input name="username" type="text" />
   <input name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
   <input name="continue" type="button" value="Clear" />
  </form>
</body>
<html>
```

- `name=username` (4)
- `name=continue value=Clear` (7)
- `name=continue Clear` (7)
- `name=continue type=button` (7)

Note: Unlike some types of XPath and DOM locators, the three types of locators above allow Selenium to test a UI element independent of its location on the page. So if the page structure and organization is altered, the test will still pass. You may or may not want to also test whether the page structure changes. In the case where web designers frequently alter the page, but its functionality must be regression tested, testing via id and name attributes, or really via any HTML property, becomes very important.

Locating by XPath

XPath is the language used for locating nodes in an XML document. As HTML can be an implementation of XML (XHTML), Selenium users can leverage this powerful language to target elements in their web applications. XPath extends beyond (as well as supporting) the simple methods of locating by id

or name attributes, and opens up all sorts of new possibilities such as locating the third checkbox on the page.

One of the main reasons for using XPath is when you don't have a suitable id or name attribute for the element you wish to locate. You can use XPath to either locate the element in absolute terms (not advised), or relative to an element that does have an id or name attribute. XPath locators can also be used to specify elements via attributes other than id and name.

Absolute XPaths contain the location of all elements from the root (html) and as a result are likely to fail with only the slightest adjustment to the application. By finding a nearby element with an id or name attribute (ideally a parent element) you can locate your target element based on the relationship. This is much less likely to change and can make your tests more robust.

Since only `xpath` locators start with "//", it is not necessary to include the `xpath=` label when specifying an XPath locator.

```
<html>
 <body>
  <form id="loginForm">
   <input name="username" type="text" />
   <input name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
   <input name="continue" type="button" value="Clear" />
  </form>
 </body>
 <html>
```

- `xpath=/html/body/form[1]` (3) - *Absolute path (would break if the HTML was changed only slightly)*
- `//form[1]` (3) - *First form element in the HTML*
- `xpath=//form[@id='loginForm']` (3) - *The form element with attribute named 'id' and the value 'loginForm'*
- `xpath=//form[input/\@name='username']` (4) - *First form element with an input child element with attribute named 'name' and the value 'username'*
- `//input[@name='username']` (4) - *First input element with attribute named 'name' and the value 'username'*
- `//form[@id='loginForm']/input[1]` (4) - *First input child element of the form element with attribute named 'id' and the value 'loginForm'*
- `//input[@name='continue'][@type='button']` (7) - *Input with attribute named 'name' and the value 'continue' and attribute named 'type' and the value 'button'*
- `//form[@id='loginForm']/input[4]` (7) - *Fourth input child element of the form element with attribute named 'id' and value 'loginForm'*

These examples cover some basics, but in order to learn more, the following references are recommended:

- W3Schools XPath Tutorial
- W3C XPath Recommendation

- XPath Tutorial - with interactive examples.

There are also a couple of very useful Firefox Add-ons that can assist in discovering the XPath of an element:

- XPath Checker - suggests XPath and can be used to test XPath results.
- Firebug - XPath suggestions are just one of the many powerful features of this very useful add-on.

Locating Hyperlinks by Link Text

This is a simple method of locating a hyperlink in your web page by using the text of the link. If two links with the same text are present, then the first match will be used.

```
<html>
 <body>
  <p>Are you sure you want to do this?</p>
  <a href="continue.html">Continue</a>
  <a href="cancel.html">Cancel</a>
</body>
<html>
```

- `link=Continue` (4)
- `link=Cancel` (5)

Locating by DOM

The Document Object Model represents an HTML document and can be accessed using JavaScript. This location strategy takes JavaScript that evaluates to an element on the page, which can be simply the element's location using the hierarchical dotted notation.

Since only `dom` locators start with "document", it is not necessary to include the `dom=` label when specifying a DOM locator.

```
 <html>
  <body>
   <form id="loginForm">
    <input name="username" type="text" />
    <input name="password" type="password" />
    <input name="continue" type="submit" value="Login" />
    <input name="continue" type="button" value="Clear" />
   </form>
 </body>
 <html>
```

- `dom=document.getElementById('loginForm')` (3)
- `dom=document.forms['loginForm']` (3)
- `dom=document.forms[0]` (3)

- `document.forms[0].username` (4)
- `document.forms[0].elements['username']` (4)
- `document.forms[0].elements[0]` (4)
- `document.forms[0].elements[3]` (7)

You can use Selenium itself as well as other sites and extensions to explore the DOM of your web application. A good reference exists on W3Schools.

Locating by CSS

CSS (Cascading Style Sheets) is a language for describing the rendering of HTML and XML documents. CSS uses Selectors for binding style properties to elements in the document. These Selectors can be used by Selenium as another locating strategy.

```
<html>
 <body>
  <form id="loginForm">
   <input class="required" name="username" type="text" />
   <input class="required passfield" name="password" type="password" />
   <input name="continue" type="submit" value="Login" />
   <input name="continue" type="button" value="Clear" />
  </form>
 </body>
 <html>
```

- `css=form#loginForm` (3)
- `css=input[name="username"]` (4)
- `css=input.required[type="text"]` (4)
- `css=input.passfield` (5)
- `css=#loginForm input[type="button"]` (4)
- `css=#loginForm input:nth-child(2)` (5)

For more information about CSS Selectors, the best place to go is the W3C publication. You'll find additional references there.

Note: Most experienced Selenium users recommend CSS as their locating strategy of choice as it's considerably faster than XPath and can find the most complicated objects in an intrinsic HTML document.

Implicit Locators

You can choose to omit the locator type in the following situations:

- Locators without an explicitly defined locator strategy will default to using the identifier locator strategy. See Locating by Identifier.

- Locators starting with "//" will use the XPath locator strategy. See Locating by XPath.
- Locators starting with "document" will use the DOM locator strategy. See Locating by DOM

3.15 Matching Text Patterns

Like locators, *patterns* are a type of parameter frequently required by Selenese commands. Examples of commands which require patterns are **verifyTextPresent**, **verifyTitle**, **verifyAlert**, **assertConfirmation**, **verifyText**, and **verifyPrompt**. And as has been mentioned above, link locators can utilize a pattern. Patterns allow you to *describe*, via the use of special characters, what text is expected rather than having to specify that text exactly.

There are three types of patterns: *globbing*, *regular expressions*, and *exact*.

3.15.1 Globbing Patterns

Most people are familiar with globbing as it is utilized in filename expansion at a DOS or Unix/Linux command line such as `ls *.c`. In this case, globbing is used to display all the files ending with a `.c` extension that exist in the current directory. Globbing is fairly limited. Only two special characters are supported in the Selenium implementation:

***** which translates to "match anything," i.e., nothing, a single character, or many characters.

[ ] (*character class*) which translates to "match any single character found inside the square brackets." A dash (hyphen) can be used as a shorthand to specify a range of characters (which are contiguous in the ASCII character set). A few examples will make the functionality of a character class clear:

`[aeiou]` matches any lowercase vowel

`[0-9]` matches any digit

`[a-zA-Z0-9]` matches any alphanumeric character

In most other contexts, globbing includes a third special character, the **?**. However, Selenium globbing patterns only support the asterisk and character class.

To specify a globbing pattern parameter for a Selenese command, you can prefix the pattern with a **glob:** label. However, because globbing patterns are the default, you can also omit the label and specify just the pattern itself.

Below is an example of two commands that use globbing patterns. The actual link text on the page being tested was "Film/Television Department"; by using a pattern rather than the exact text, the **click** command will work even if the link text is changed to "Film & Television Department" or "Film and Television Department". The glob pattern's asterisk will match "anything or nothing" between the word "Film" and the word "Television".

Command	Target	Value
click	link=glob:Film*Television Department	
verifyTitle	glob:*Film*Television*	

The actual title of the page reached by clicking on the link was "De Anza Film And Television Department - Menu". By using a pattern rather than the exact text, the `verifyTitle` will pass as long as the two words "Film" and "Television" appear (in that order) anywhere in the page's title. For example, if the page's owner should shorten the title to just "Film & Television Department," the test would still pass. Using a pattern for both a link and a simple test that the link worked (such as the `verifyTitle` above does) can greatly reduce the maintenance for such test cases.

Regular Expression Patterns

Regular expression patterns are the most powerful of the three types of patterns that Selenese supports. Regular expressions are also supported by most high-level programming languages, many text editors, and a host of tools, including the Linux/Unix command-line utilities **grep**, **sed**, and **awk**. In Selenese, regular expression patterns allow a user to perform many tasks that would be very difficult otherwise. For example, suppose your test needed to ensure that a particular table cell contained nothing but a number. `regexp: [0-9]+` is a simple pattern that will match a decimal number of any length.

Whereas Selenese globbing patterns support only the * and **[ ]** (character class) features, Selenese regular expression patterns offer the same wide array of special characters that exist in JavaScript. Below are a subset of those special characters:

PATTERN	MATCH
.	any single character
[ ]	character class: any single character that appears inside the brackets
*	quantifier: 0 or more of the preceding character (or group)
+	quantifier: 1 or more of the preceding character (or group)
?	quantifier: 0 or 1 of the preceding character (or group)
{1,5}	quantifier: 1 through 5 of the preceding character (or group)
|	alternation: the character/group on the left or the character/group on the right
( )	grouping: often used with alternation and/or quantifier

Regular expression patterns in Selenese need to be prefixed with either `regexp:` or `regexpi:`. The former is case-sensitive; the latter is case-insensitive.

A few examples will help clarify how regular expression patterns can be used with Selenese commands. The first one uses what is probably the most commonly used regular expression pattern–**.*** ("dot star"). This two-character sequence can be translated as "0 or more occurrences of any character" or more simply, "anything or nothing." It is the equivalent of the one-character globbing pattern * (a single asterisk).

Command	Target	Value
click	link=regexp:Film.*Television Department	
verifyTitle	regexp:.*Film.*Television.*	

The example above is functionally equivalent to the earlier example that used globbing patterns for this same test. The only differences are the prefix (**regexp:** instead of **glob:**) and the "anything or nothing" pattern (**.*** instead of just *).

The more complex example below tests that the Yahoo! Weather page for Anchorage, Alaska contains info on the sunrise time:

Command	Target	Value
open	http://weather.yahoo.com/forecast/USAK0012.html	
verifyTextPresent	regexp:Sunrise: *[0-9]{1,2}:[0-9]{2} [ap]m	

Let's examine the regular expression above one part at a time:

`Sunrise: *`	The string **Sunrise:** followed by 0 or more spaces
`[0-9]{1,2}`	1 or 2 digits (for the hour of the day)
`:`	The character **:** (no special characters involved)
`[0-9]{2}`	2 digits (for the minutes) followed by a space
`[ap]m`	"a" or "p" followed by "m" (am or pm)

Exact Patterns

The **exact** type of Selenium pattern is of marginal usefulness. It uses no special characters at all. So, if you needed to look for an actual asterisk character (which is special for both globbing and regular expression patterns), the **exact** pattern would be one way to do that. For example, if you wanted to select an item labeled "Real *" from a dropdown, the following code might work or it might not. The asterisk in the `glob:Real *` pattern will match anything or nothing. So, if there was an earlier select option labeled "Real Numbers," it would be the option selected rather than the "Real *" option.

select	//select	glob:Real *

In order to ensure that the "Real *" item would be selected, the `exact:` prefix could be used to create an **exact** pattern as shown below:

select	//select	exact:Real *

But the same effect could be achieved via escaping the asterisk in a regular expression pattern:

select	//select	regexp:Real *

It's rather unlikely that most testers will ever need to look for an asterisk or a set of square brackets with characters inside them (the character class for globbing patterns). Thus, globbing patterns and regular expression patterns are sufficient for the vast majority of us.

3.16 The "AndWait" Commands

The difference between a command and its *AndWait* alternative is that the regular command (e.g. *click*) will do the action and continue with the following command as fast as it can, while the *AndWait* alternative (e.g. *clickAndWait*) tells Selenium to **wait** for the page to load after the action has been done.

The *AndWait* alternative is always used when the action causes the browser to navigate to another page or reload the present one.

Be aware, if you use an *AndWait* command for an action that does not trigger a navigation/refresh, your test will fail. This happens because Selenium will reach the *AndWait*'s timeout without seeing any navigation or refresh being made, causing Selenium to raise a timeout exception.

3.17 The waitFor Commands in AJAX applications

In AJAX driven web applications, data is retrieved from server without refreshing the page. Using *andWait* commands will not work as the page is not actually refreshed. Pausing the test execution for a certain period of time is also not a good approach as web element might appear later or earlier than the stipulated period depending on the system's responsiveness, load or other uncontrolled factors of the moment, leading to test failures. The best approach would be to wait for the needed element in a dynamic period and then continue the execution as soon as the element is found.

This is done using *waitFor* commands, as *waitForElementPresent* or *waitForVisible*, which wait dynamically, checking for the desired condition every second and continuing to the next command in the script as soon as the condition is met.

3.18 Sequence of Evaluation and Flow Control

When a script runs, it simply runs in sequence, one command after another.

Selenese, by itself, does not support condition statements (if-else, etc.) or iteration (for, while, etc.). Many useful tests can be conducted without flow control. However, for a functional test of dynamic content, possibly involving multiple pages, programming logic is often needed.

When flow control is needed, there are three options:

1. Run the script using Selenium-RC and a client library such as Java or PHP to utilize the programming language's flow control features.
2. Run a small JavaScript snippet from within the script using the storeEval command.
3. Install the goto_sel_ide.js extension.

Most testers will export the test script into a programming language file that uses the Selenium-RC API (see the Selenium-IDE chapter). However, some organizations prefer to run their scripts from Selenium-IDE whenever possible (for instance, when they have many junior-level people running tests for them, or when programming skills are lacking). If this is your case, consider a JavaScript snippet or the goto_sel_ide.js extension.

3.19 Store Commands and Selenium Variables

You can use Selenium variables to store constants at the beginning of a script. Also, when combined with a data-driven test design (discussed in a later section), Selenium variables can be used to store values passed to your test program from the command-line, from another program, or from a file.

The plain *store* command is the most basic of the many store commands and can be used to simply store a constant value in a selenium variable. It takes two parameters, the text value to be stored and a selenium variable. Use the standard variable naming conventions of only alphanumeric characters when choosing a name for your variable.

Command	Target	Value
store	paul@mysite.org	userName

Later in your script, you'll want to use the stored value of your variable. To access the value of a variable, enclose the variable in curly brackets ({ }) and precede it with a dollar sign like this.

Command	Target	Value
verifyText	//div/p	${userName}

A common use of variables is for storing input for an input field.

Command	Target	Value
type	id=login	${userName}

Selenium variables can be used in either the first or second parameter and are interpreted by Selenium prior to any other operations performed by the command. A Selenium variable may also be used within a locator expression.

An equivalent store command exists for each verify and assert command. Here are a couple more commonly used store commands.

3.19.1 storeElementPresent

This corresponds to verifyElementPresent. It simply stores a boolean value–"true" or "false"–depending on whether the UI element is found.

3.19.2 storeText

StoreText corresponds to verifyText. It uses a locater to identify specific page text. The text, if found, is stored in the variable. StoreText can be used to extract text from the page being tested.

3.19.3 storeEval

This command takes a script as its first parameter. Embedding JavaScript within Selenese is covered in the next section. StoreEval allows the test to store the result of running the script in a variable.

3.20 JavaScript and Selenese Parameters

JavaScript can be used with two types of Selenese parameters: script and non-script (usually expressions). In most cases, you'll want to access and/or manipulate a test case variable inside the JavaScript snippet used as a Selenese parameter. All variables created in your test case are stored in a JavaScript *associative array*. An associative array has string indexes rather than sequential numeric indexes. The associative array containing your test case's variables is named **storedVars**. Whenever you wish to access or manipulate a variable within a JavaScript snippet, you must refer to it as **storedVars['yourVariableName']**.

3.20.1 JavaScript Usage with Script Parameters

Several Selenese commands specify a **script** parameter including **assertEval**, **verifyEval**, **storeEval**, and **waitForEval**. These parameters require no special syntax. A Selenium-IDE user would simply place a snippet of JavaScript code into the appropriate field, normally the **Target** field (because a **script** parameter is normally the first or only parameter).

The example below illustrates how a JavaScript snippet can be used to perform a simple numerical calculation:

Command	Target	Value
store	10	hits
storeXpathCount	//blockquote	blockquotes
storeEval	storedVars['hits']-storedVars['blockquotes']	paragraphs

This next example illustrates how a JavaScript snippet can include calls to methods, in this case the JavaScript String object's `toUpperCase` method and `toLowerCase` method.

Command	Target	Value
store	Edith Wharton	name
storeEval	storedVars['name'].toUpperCase()	uc
storeEval	storedVars['name'].toLowerCase()	lc

JavaScript Usage with Non-Script Parameters

JavaScript can also be used to help generate values for parameters, even when the parameter is not specified to be of type **script**. However, in this case, special syntax is required–the JavaScript snippet must be enclosed inside curly braces and preceded by the label `javascript`, as in `javascript {*yourCodeHere*}`. Below is an example in which the `type` command's second parameter `value` is generated via JavaScript code using this special syntax:

Command	Target	Value
store	league of nations	searchString
type	q	javascript{storedVars['searchString'].toUpperCase()}

3.21 *echo* - The Selenese Print Command

Selenese has a simple command that allows you to print text to your test's output. This is useful for providing informational progress notes in your test which display on the console as your test is running. These notes also can be used to provide context within your test result reports, which can be useful for finding where a defect exists on a page in the event your test finds a problem. Finally, echo statements can be used to print the contents of Selenium variables.

Command	Target	Value
echo	Testing page footer now.	
echo	Username is ${userName}	

3.22 Alerts, Popups, and Multiple Windows

Suppose that you are testing a page that looks like this.

```
<!DOCTYPE HTML>
<html>
<head>
  <script type="text/javascript">
    function output(resultText){
      document.getElementById('output').childNodes[0].nodeValue=resultText;
    }

    function show_confirm(){
      var confirmation=confirm("Chose an option.");
      if (confirmation==true){
        output("Confirmed.");
      }
      else{
        output("Rejected!");
      }
    }

    function show_alert(){
      alert("I'm blocking!");
      output("Alert is gone.");
    }
    function show_prompt(){
      var response = prompt("What's the best web QA tool?","Selenium");
      output(response);
    }
    function open_window(windowName){
      window.open("newWindow.html",windowName);
    }
    </script>
</head>
<body>

```

```
34    <input type="button" id="btnConfirm" onclick="show_confirm()" value="Show confirm
35    <input type="button" id="btnAlert" onclick="show_alert()" value="Show alert" />
36    <input type="button" id="btnPrompt" onclick="show_prompt()" value="Show prompt" />
37    <a href="newWindow.html" id="lnkNewWindow" target="_blank">New Window Link</a>
38    <input type="button" id="btnNewNamelessWindow" onclick="open_window()" value="Open
39    <input type="button" id="btnNewNamedWindow" onclick="open_window('Mike')" value="O
40
41    <br />
42    <span id="output">
43    </span>
44  </body>
45  </html>
```

The user must respond to alert/confirm boxes, as well as moving focus to newly opened popup windows. Fortunately, Selenium can cover JavaScript pop-ups.

But before we begin covering alerts/confirms/prompts in individual detail, it is helpful to understand the commonality between them. Alerts, confirmation boxes and prompts all have variations of the following

Command	Description
assertFoo(*pattern*)	throws error if *pattern* doesn't match the text of the pop-up
assertFooPresent	throws error if pop-up is not available
assertFooNotPresent	throws error if any pop-up is present
storeFoo(*variable*)	stores the text of the pop-up in a variable
storeFooPresent(*variable*)	stores the text of the pop-up in a variable and returns true or false

When running under Selenium, JavaScript pop-ups will not appear. This is because the function calls are actually being overridden at runtime by Selenium's own JavaScript. However, just because you cannot see the pop-up doesn't mean you don't have do deal with it. To handle a pop-up, you must call it's `assertFoo(pattern)` function. If you fail to assert the presence of a pop-up your next command will be blocked and you will get an error similar to the following `[error] Error: There was an unexpected Confirmation! [Chose an option.]`

3.22.1 Alerts

Let's start with asserts because they are the simplest pop-up to handle. To begin, open the HTML sample above in a browser and click on the "Show alert" button. You'll notice that after you close the alert the text "Alert is gone." is displayed on the page. Now run through the same steps with Selenium IDE recording, and verify the text is added after you close the alert. Your test will look something like this:

Command	Target	Value
open	/	
click	btnAlert	
assertAlert	I'm blocking	
verifyTextPresent	Alert is gone.	

You may be thinking "Thats odd, I never tried to assert that alert." But this is Selenium-IDE handling and closing the alert for you. If you remove that step and replay the test you will get the following error `[error] Error: There was an unexpected Alert! [I'm blocking!]`. You must include an assertion of the alert to acknowledge it's presence.

If you just want to assert that an alert is present but either don't know or don't care what text it contains, you can use `assertAlertPresent`. This will return true or false, with false halting the test.

Confirmations

Confirmations behave in much the same way as alerts, with `assertConfirmation` and `assertConfirmationPresent` offering the same characteristics as their alert counterparts. However, by default Selenium will select OK when a confirmation pops up. Try recording clicking on the "Show confirm box" button in the sample page, but click on the "Cancel" button in the popup, then assert the output text. Your test may look something like this:

Command	Target	Value
open	/	
click	btnConfirm	
chooseCancelOnNextConfirmation		
assertConfirmation	Choose and option.	
verifyTextPresent	Rejected	

The `chooseCancelOnNextConfirmation` function tells Selenium that all following confirmation should return false. It can be reset by calling chooseOkOnNextConfirmation.

You may notice that you cannot replay this test, because Selenium complains that there is an unhandled confirmation. This is because the order of events Selenium-IDE records causes the click and chooseCancelOnNextConfirmation to be put in the wrong order (it makes sense if you think about it, Selenium can't know that you're cancelling before you open a confirmation) Simply switch these two commands and your test will run fine.

3.23 Debugging

Debugging means finding and fixing errors in your test case. This is a normal part of test case development.

We won't teach debugging here as most new users to Selenium will already have some basic experience with debugging. If this is new to you, we recommend you ask one of the developers in your organization.

3.23.1 Breakpoints and Startpoints

The Sel-IDE supports the setting of breakpoints and the ability to start and stop the running of a test case, from any point within the test case. That is, one can run up to a specific command in the middle of the test case and inspect how the test case behaves at that point. To do this, set a breakpoint on the command just before the one to be examined.

To set a breakpoint, select a command, right-click, and from the context menu select *Toggle Breakpoint*. Then click the Run button to run your test case from the beginning up to the breakpoint.

It is also sometimes useful to run a test case from somewhere in the middle to the end of the test case or up to a breakpoint that follows the starting point. For example, suppose your test case first logs into the website and then performs a series of tests and you are trying to debug one of those tests. However, you only need to login once, but you need to keep rerunning your tests as you are developing them. You can login once, then run your test case from a startpoint placed after the login portion of your test case. That will prevent you from having to manually logout each time you rerun your test case.

To set a startpoint, select a command, right-click, and from the context menu select *Set/Clear Start Point*. Then click the Run button to execute the test case beginning at that startpoint.

3.23.2 Stepping Through a Testcase

To execute a test case one command at a time ("step through" it), follow these steps:

1. Start the test case running with the Run button from the toolbar.

1. Immediately pause the executing test case with the Pause button.

1. Repeatedly select the Step button.

3.23.3 Find Button

The Find button is used to see which UI element on the currently displayed webpage (in the browser) is used in the currently selected Selenium command. This is useful when building a locator for a command's first parameter (see the section on *locators* in the Selenium Commands chapter). It can be used with any command that identifies a UI element on a webpage, i.e. *click*, *clickAndWait*, *type*, and certain *assert* and *verify* commands, among others.

From Table view, select any command that has a locator parameter. Click the Find button. Now look on the webpage: There should be a bright green rectangle enclosing the element specified by the locator parameter.

3.23.4 Page Source for Debugging

Often, when debugging a test case, you simply must look at the page source (the HTML for the webpage you're trying to test) to determine a problem. Firefox makes this easy. Simply right-click the webpage and select 'View->Page Source. The HTML opens in a separate window. Use its Search feature (Edit=>Find) to search for a keyword to find the HTML for the UI element you're trying to test.

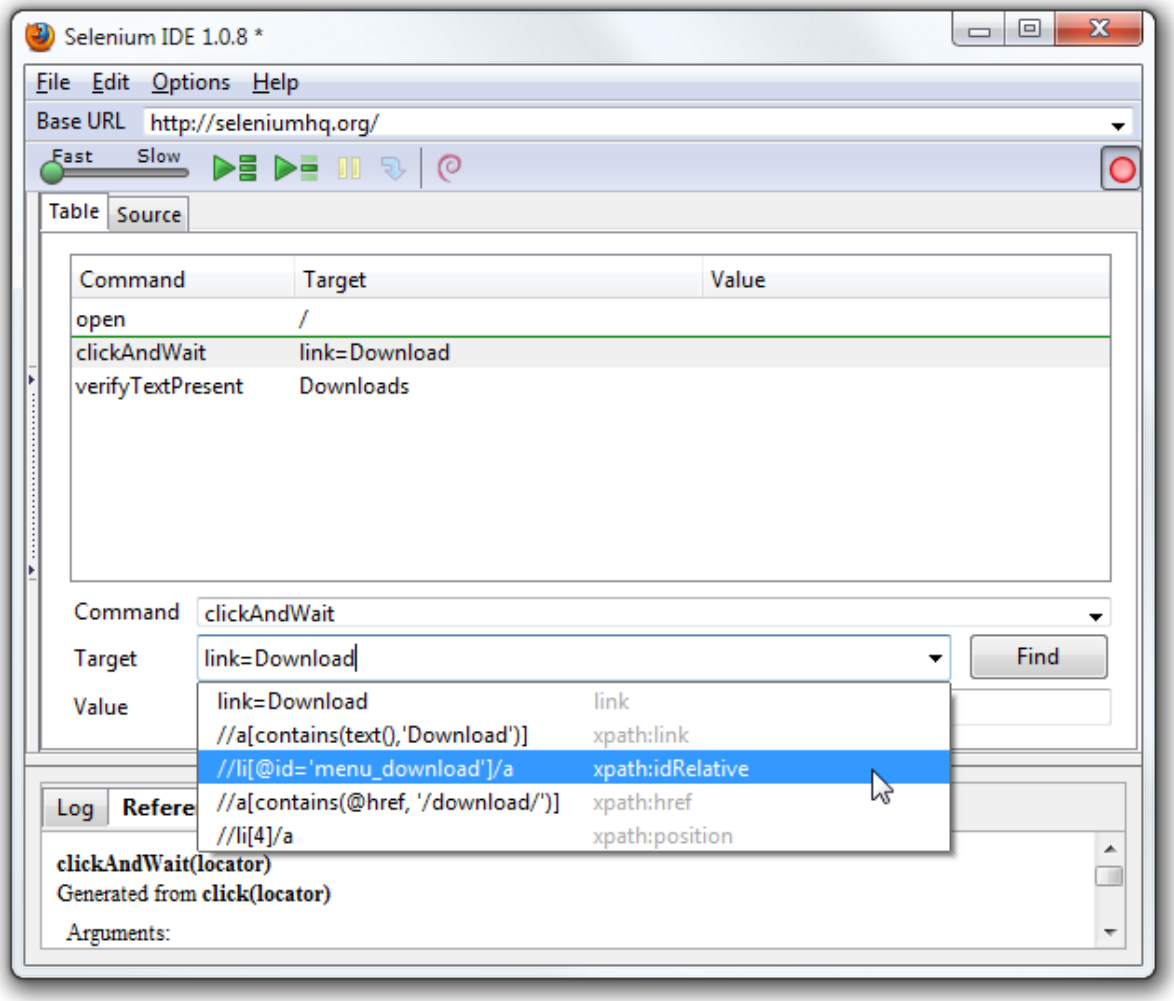
Alternatively, select just that portion of the webpage for which you want to see the source. Then right-click the webpage and select View Selection Source. In this case, the separate HTML window will contain just a small amount of source, with highlighting on the portion representing your selection.

3.23.5 Locator Assistance

Whenever Selenium-IDE records a locator-type argument, it stores additional information which allows the user to view other possible locator-type arguments that could be used instead. This feature can be very useful for learning more about locators, and is often needed to help one build a different type of locator than the type that was recorded.

This locator assistance is presented on the Selenium-IDE window as a drop-down list accessible at the right end of the Target field (only when the Target field contains a recorded locator-type argument). Below is a snapshot showing the contents of this drop-down for one command. Note that the first

column of the drop-down provides alternative locators, whereas the second column indicates the type of each alternative.



3.24 Writing a Test Suite

A test suite is a collection of test cases which is displayed in the leftmost pane in the IDE. The test suite pane can be manually opened or closed via selecting a small dot halfway down the right edge of the pane (which is the left edge of the entire Selenium-IDE window if the pane is closed).

The test suite pane will be automatically opened when an existing test suite is opened *or* when the user selects the New Test Case item from the File menu. In the latter case, the new test case will appear immediately below the previous test case.

Selenium-IDE does not yet support loading pre-existing test cases into a test suite. Users who want to create or modify a test suite by adding pre-existing test cases must manually edit a test suite file.

A test suite file is an HTML file containing a one-column table. Each cell of each row in the <tbody> section contains a link to a test case. The example below is of a test suite containing four test cases:

```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    <title>Sample Selenium Test Suite</title>
</head>
```

```
<body>
    <table cellpadding="1" cellspacing="1" border="1">
        <thead>
            <tr><td>Test Cases for De Anza A-Z Directory Links</td></tr>
        </thead>
    <tbody>
        <tr><td><a href="./a.html">A Links</a></td></tr>
        <tr><td><a href="./b.html">B Links</a></td></tr>
        <tr><td><a href="./c.html">C Links</a></td></tr>
        <tr><td><a href="./d.html">D Links</a></td></tr>
    </tbody>
    </table>
</body>
</html>
```

Note: Test case files should not have to be co-located with the test suite file that invokes them. And on Mac OS and Linux systems, that is indeed the case. However, at the time of this writing, a bug prevents Windows users from being able to place the test cases elsewhere than with the test suite that invokes them.

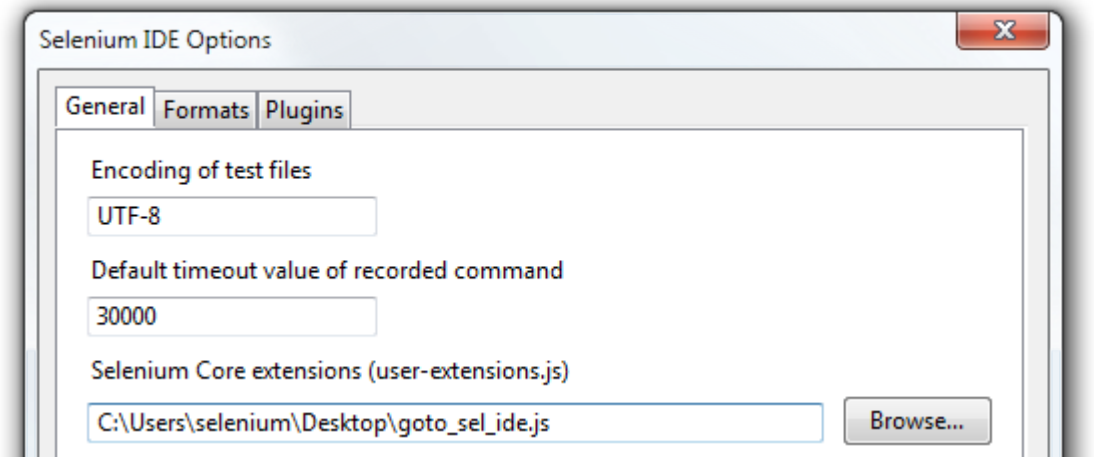
3.25 User Extensions

User extensions are JavaScript files that allow one to create his or her own customizations and features to add additional functionality. Often this is in the form of customized commands although this extensibility is not limited to additional commands.

There are a number of useful extensions created by users.

IMPORTANT: THIS SECTION IS OUT OF DATE–WE WILL BE REVISING THIS SOON. Perhaps the most popular of all Selenium-IDE extensions is one which provides flow control in the form of while loops and primitive conditionals. This extension is the goto_sel_ide.js. For an example of how to use the functionality provided by this extension, look at the page created by its author.

To install this extension, put the pathname to its location on your computer in the **Selenium Core extensions** field of Selenium-IDE's Options=>Options=>General tab.



After selecting the **OK** button, you must close and reopen Selenium-IDE in order for the extensions file to be read. Any change you make to an extension will also require you to close and reopen Selenium-IDE.

Information on writing your own extensions can be found near the bottom of the Selenium Reference document.

3.26 Format

Format, under the Options menu, allows you to select a language for saving and displaying the test case. The default is HTML.

If you will be using Selenium-RC to run your test cases, this feature is used to translate your test case into a programming language. Select the language, i.e. Java, PHP, you will be using with Selenium-RC for developing your test programs. Then simply save the test case using File=>Save. Your test case will be translated into a series of functions in the language you choose. Essentially, program code supporting your test is generated for you by Selenium-IDE.

Also, note that if the generated code does not suit your needs, you can alter it by editing a configuration file which defines the generation process. Each supported language has configuration settings which are editable. This is under the Options=>Options=>Format tab.

Note: At the time of this writing, this feature is not yet supported by the Selenium developers. However the author has altered the C# format in a limited manner and it has worked well.

3.27 Executing Selenium-IDE Tests on Different Browsers

While Selenium-IDE can only run tests against Firefox, tests developed with Selenium-IDE can be run against other browsers, using a simple command-line interface that invokes the Selenium-RC server. This topic is covered in the *Run Selenese tests* section on Selenium-RC chapter. The *-htmlSuite* command-line option is the particular feature of interest.

3.28 Troubleshooting

Below is a list of image/explanation pairs which describe frequent sources of problems with Selenium-IDE:

Table view is not available with this format.

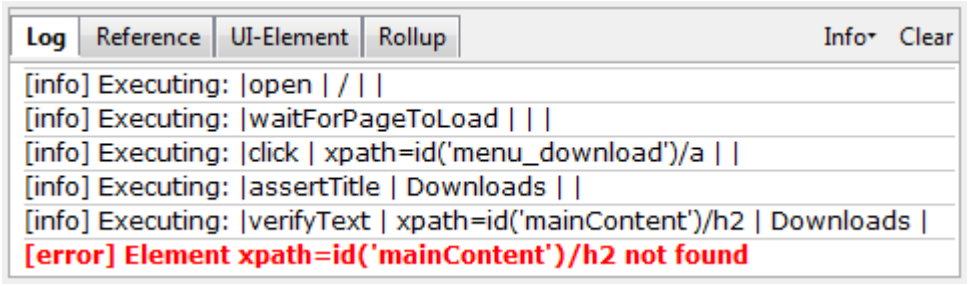
This message can be occasionally displayed in the Table tab when Selenium IDE is launched. The workaround is to close and reopen Selenium IDE. See issue 1008. for more information. If you are able to reproduce this reliably then please provide details so that we can work on a fix.

---

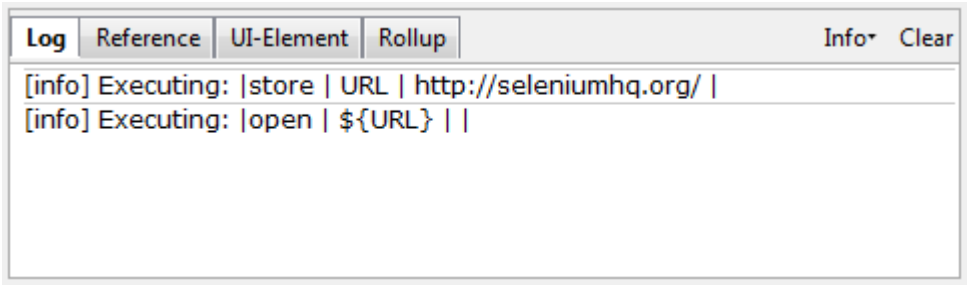
error loading test case: no command found

You've used **File=>Open** to try to open a test suite file. Use **File=>Open Test Suite** instead.

An enhancement request has been raised to improve this error message. See issue 1010.



This type of **error** may indicate a timing problem, i.e., the element specified by a locator in your command wasn't fully loaded when the command was executed. Try putting a **pause 5000** before the command to determine whether the problem is indeed related to timing. If so, investigate using an appropriate **waitFor*** or ***AndWait** command before the failing command.



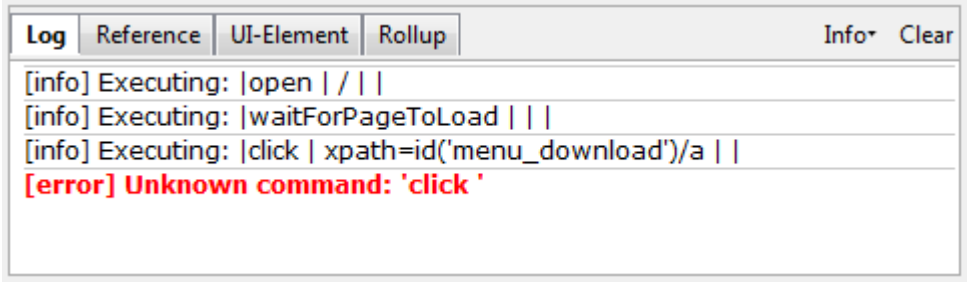
Whenever your attempt to use variable substitution fails as is the case for the **open** command above, it indicates that you haven't actually created the variable whose value you're trying to access. This is sometimes due to putting the variable in the **Value** field when it should be in the **Target** field or vice versa. In the example above, the two parameters for the **store** command have been erroneously placed in the reverse order of what is required. For any Selenese command, the first required parameter must go in the **Target** field, and the second required parameter (if one exists) must go in the **Value** field.

---

error loading test case: [Exception... "Component returned failure code: 0x80520012 (NS_ERROR_FILE_NOT_FOUND) [nsIFileInputStream.init]" nresult: "0x80520012 (NS_ERROR_FILE_NOT_FOUND)" location: "JS frame :: chrome://selenium-ide/content/file-utils.js :: anonymous :: line 48" data: no]

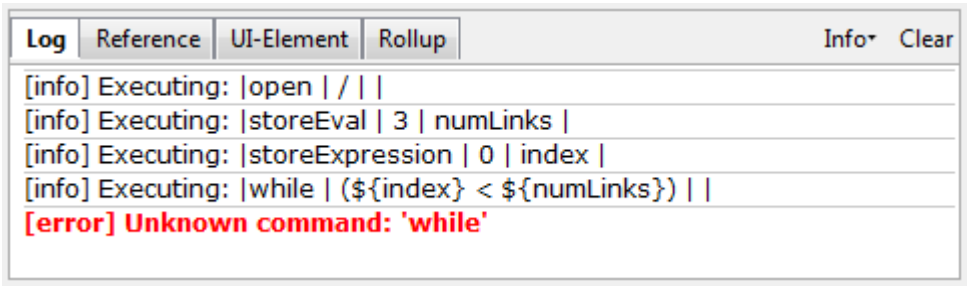
One of the test cases in your test suite cannot be found. Make sure that the test case is indeed located where the test suite indicates it is located. Also, make sure that your actual test case files have the .html extension both in their filenames, and in the test suite file where they are referenced.

An enhancement request has been raised to improve this error message. See issue 1011.

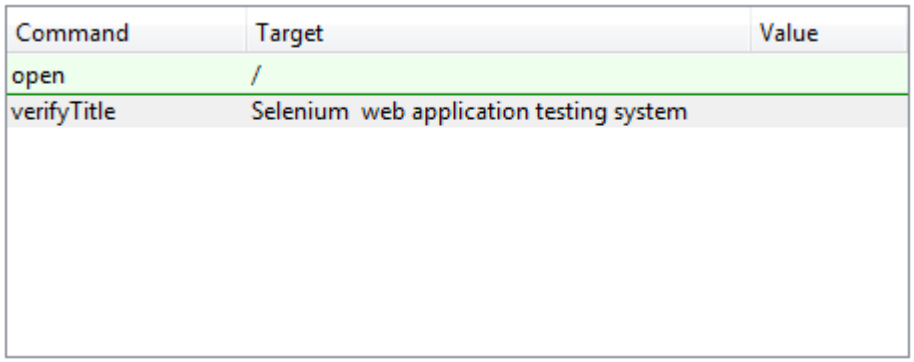


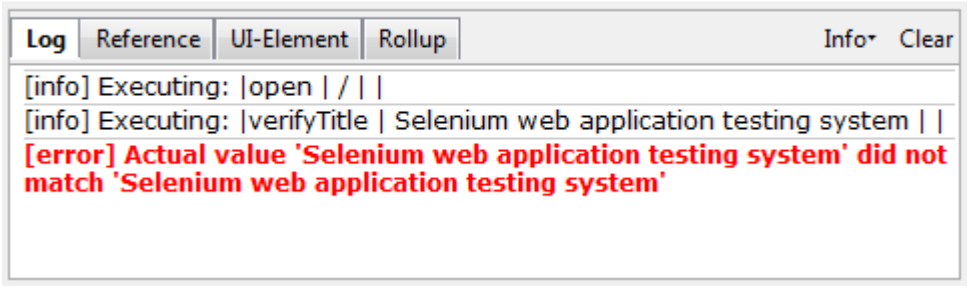
Selenium-IDE is very *space-sensitive*! An extra space before or after a command will cause it to be unrecognizable.

This defect has been raised. See issue 1012.



Your extension file's contents have not been read by Selenium-IDE. Be sure you have specified the proper pathname to the extensions file via **Options=>Options=>General** in the **Selenium Core extensions** field. Also, Selenium-IDE must be restarted after any change to either an extensions file *or* to the contents of the **Selenium Core extensions** field.





This type of error message makes it appear that Selenium-IDE has generated a failure where there is none. However, Selenium-IDE is correct that the actual value does not match the value specified in such test cases. The problem is that the log file error messages collapse a series of two or more spaces into a single space, which is confusing. In the example above, note that the parameter for **verifyTitle** has two spaces between the words "Selenium" and "web" The page's actual title has only one space between these words. Thus, Selenium-IDE is correct to generate an error, but is misleading in the nature of the error.

This defect has been raised. See issue 1013.

CHAPTER

FOUR

SELENIUM 2.0 AND WEBDRIVER

As you can see in the *Brief History of The Selenium Project* The Selenium developers are working towards a Selenium 2.0 release. The primary new feature will be the integration of the WebDriver API into Selenium 1. This will address a number of limitations along with providing an alternative programming interface. The goal is to develop an object-oriented API that provides additional support for a larger number of browsers along with improved support for modern advanced web-app testing problems.

4.1 The 5 Minute Getting Started Guide

WebDriver is a tool for automating testing web applications, and in particular to verify that they work as expected. It aims to provide a friendly API that's easy to explore and understand, which will help make your tests easier to read and maintain. It's not tied to any particular test framework, so it can be used equally well in a unit testing or from a plain old "main" method. This "Getting Started" guide introduces you to WebDriver's API and helps get you started becoming familiar with it.

Start by Downloading the latest binaries and unpack them into a directory. From now on, we'll refer to that as `$WEBDRIVER_HOME`. Now, open your favourite IDE and:

- Start a new project in your favourite IDE/editor
- Add a reference to all the libraries in `$WEBDRIVER_HOME`

You can see that WebDriver acts just as a normal library does: it's entirely self-contained, and you usually don't need to remember to start any additional processes or run any installers before using it, as opposed to the proxy server with Selenium-RC.

You're now ready to write some code. An easy way to get started is this example, which searches for the term "Cheese" on Google and then outputs the result page's title to the console.

4.1.1 Java

```
package org.openqa.selenium.example;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.support.ui.WebDriverWait;
```

```
public class Selenium2Example  {
    public static void main(String[] args) {
        // Create a new instance of the Firefox driver
        // Notice that the remainder of the code relies on the interface,
        // not the implementation.
        WebDriver driver = new FirefoxDriver();

        // And now use this to visit Google
        driver.get( "http://www.google.com" );
        // Alternatively the same thing can be done like this
        // driver.navigate().to("http://www.google.com");

        // Find the text input element by its name
        WebElement element = driver.findElement(By.name( "q" ));

        // Enter something to search for
        element.sendKeys( "Cheese!" );

        // Now submit the form. WebDriver will find the form for us from the element
        element.submit();

        // Check the title of the page
        System.out.println( "Page title is: " + driver.getTitle());

        // Google's search is rendered dynamically with JavaScript.
        // Wait for the page to load, timeout after 10 seconds
        (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.getTitle().startsWith( "cheese!" );
            }
        });

        // Should see: "cheese! - Google Search"
        System.out.println( "Page title is: " + driver.getTitle());

        //Close the browser
        driver.quit();
    }
}
```

4.1.2 C#

```
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium;

class GoogleSuggest
{

    static void Main(string[] args)
    {
        IWebDriver driver = new FirefoxDriver();

        //Notice navigation is slightly different than the Java version
        //This is because 'get' is a keyword in C#
```

```
        driver.Navigate().GoToUrl("http://www.google.com/");
        IWebElement query = driver.FindElement(By.Name("q"));
        query.SendKeys("Cheese");
        System.Console.WriteLine("Page title is: " + driver.Title);
        // TODO add wait
        driver.Quit();
    }

}
```

4.1.3 Python

```
from selenium import webdriver
import time

if __name__=='__main__':
    # Create a new instance of the Firefox driver
    driver = webdriver.Firefox()

    # go to the google home page
    driver.get("http://www.google.com")

    # find the element that's name attribute is q (the google search box)
    inputElement = driver.find_element_by_name("q")

    # type in the search
    inputElement.send_keys("Cheese!")

    # submit the form (although google automatically searches now without submitting)
    inputElement.submit()

    # the page is ajaxy so the title is originally this:
    print driver.title

    # we have to wait for the page to refresh, the last thing that seems to be updated i
    while not driver.title.startswith("cheese!"):
        # this is an infinite loop... should probably put some logic to break after x ti
        # sleep for a second
        time.sleep(1)

    # You should see "cheese! - Google Search"
    print driver.title

    driver.quit()
```

4.1.4 Ruby

TODO

4.1.5 PHP

TODO

4.1.6 Perl

TODO

Hopefully, this will have whet your appetite for more. In the Next Steps section you will learn more about how to use WebDriver for things such as navigating forward and backward in your browser's history, and how to use frames and windows. It also provides a more complete discussion of the examples than The 5 Minute Getting Started Guide. If you're ready, let's take the Next Steps!

4.2 Next Steps For Using WebDriver

4.2.1 Which Implementation of WebDriver Should I Use?

WebDriver is the name of the key interface against which tests should be written, but there are several implementations. These are:

Name of driver	Available on which OS?	Class to instantiate
HtmlUnit Driver	All	org.openqa.selenium.htmlunit.HtmlUnitDriver
Firefox Driver	All	org.openqa.selenium.firefox.FirefoxDriver
Internet Explorer Driver	Windows	org.openqa.selenium.ie.InternetExplorerDriver
Chrome Driver	All	org.openqa.selenium.chrome.ChromeDriver

You can find out more information about each of these by following the links in the table. Which you use depends on what you want to do. For sheer speed, the HtmlUnit Driver is great, but it's not graphical, which means that you can't watch what's happening. As a developer you may be comfortable with this, but sometimes it's good to be able to test using a real browser, especially when you're showing a demo of your application (or running the tests) for an audience. Often, this idea is referred to as "safety", and it falls into two parts. Firstly, there's "actual safety", which refers to whether or not the tests work as they should. This can be measured and quantified. Secondly, there's "perceived safety", which refers to whether or not an observer believes the tests work as they should. This varies from person to person, and will depend on their familiarity with the application under test, WebDriver, and your testing framework.

To support higher "perceived safety", you may wish to choose a driver such as the Firefox Driver. This has the added advantage that this driver actually renders content to a screen, and so can be used to detect information such as the position of an element on a page, or the CSS properties that apply to it. However, this additional flexibility comes at the cost of slower overall speed. By writing your tests against the WebDriver interface, it is possible to pick the most appropriate driver for a given test.

To keep things simple, let's start with the HtmlUnit Driver:

```
WebDriver driver = new HtmlUnitDriver();
```

4.2.2 Navigating

The first thing you'll want to do with WebDriver is navigate to a page. The normal way to do this is by calling "get":

```
driver.get("http://www.google.com");
```

WebDriver will wait until the page has fully loaded (that is, the "onload" event has fired) before returning control to your test or script. It's worth noting that if your page uses a lot of AJAX on load then WebDriver may not know when it has completely loaded. If you need to ensure such pages are fully loaded then you can use "waits".

4.2.3 Interacting With the Page

Just being able to go to places isn't terribly useful. What we'd really like to do is to interact with the pages, or, more specifically, the HTML elements within a page. First of all, we need to find one. WebDriver offers a number of ways of finding elements. For example, given an element defined as:

```
<input type="text" name="passwd" id="passwd-id" />
```

you could find it using any of:

```
WebElement element;
element = driver.findElement(By.id("passwd-id"));
element = driver.findElement(By.name("passwd"));
element = driver.findElement(By.xpath("//input[@id='passwd-id']"));
```

You can also look for a link by its text, but be careful! The text must be an exact match! You should also be careful when using XPATH in WebDriver. If there's more than one element that matches the query, then only the first will be returned. If nothing can be found, a `NoSuchElementException` will be thrown. WebDriver has an "Object-based" API; we represent all types of elements using the same interface: Web Element. This means that although you may see a lot of possible methods you could invoke when you hit your IDE's auto-complete key combination, not all of them will make sense or be valid. Don't worry! WebDriver will attempt to do the Right Thing, and if you call a method that makes no sense ("setSelected()" on a "meta" tag, for example) an exception will be thrown.

So, you've got an element. What can you do with it? First of all, you may want to enter some text into a text field:

```
element.sendKeys("some text");
```

You can simulate pressing the arrow keys by using the "Keys" class:

```
element.sendKeys(" and some", Keys.ARROW_DOWN);
```

It is possible to call sendKeys on any element, which makes it possible to test keyboard shortcuts such as those used on GMail. A side-effect of this is that typing something into a text field won't automatically clear it. Instead, what you type will be appended to what's already there. You can easily clear the contents of a text field or textarea:

```
element.clear();
```

4.2.4 Filling In Forms

We've already seen how to enter text into a textarea or text field, but what about the other elements? You can "toggle" the state of checkboxes, and you can use "setSelected" to set something like an OPTION tag selected. Dealing with SELECT tags isn't too bad:

```
WebElement select = driver.findElement(By.xpath("//select"));
List<WebElement> allOptions = select.findElements(By.tagName("option"));
for (WebElement option : allOptions) {
    System.out.println(String.format("Value is: %s", option.getValue()));
    option.setSelected();
}
```

This will find the first "SELECT" element on the page, and cycle through each of it's OPTIONs in turn, printing out their values, and selecting each in turn. As you can see, this isn't the most efficient way of dealing with SELECT elements. WebDriver's support classes include one called "Select", which provides useful methods for interacting with these.

```
Select select = new Select(driver.findElement(By.xpath("//select")));
select.deselectAll();
select.selectByVisibleText("Edam");
```

This will deselect all OPTIONs from the first SELECT on the page, and then select the OPTION with the displayed text of "Edam".

Once you've finished filling out the form, you probably want to submit it. One way to do this would be to find the "submit" button and click it:

```
driver.findElement(By.id("submit")).click();
// Assume the button has the ID "submit" :)
```

Alternatively, WebDriver has the convenience method "submit" on every element. If you call this on an element within a form, WebDriver will walk up the DOM until it finds the enclosing form and then calls submit on that. If the element isn't in a form, then the `NoSuchElementException` will be thrown:

```
element.submit();
```

4.2.5 Getting Visual Information And Drag And Drop

Here's an example of using the Actions class to perform a drag and drop. As of rc2 this only works on the Windows platform.

```
WebElement element = driver.findElement(By.name("source"));
WebElement target = driver.findElement(By.name("target"));

(new Actions(driver)).dragAndDrop(element, target).perform();
```

4.2.6 Moving Between Windows and Frames

It's rare for a modern web application not to have any frames or to be constrained to a single window. WebDriver supports moving between named windows using the "switchTo" method:

```
driver.switchTo().window("windowName");
```

All calls to `driver` will now be interpreted as being directed to the particular window. But how do you know the window's name? Take a look at the javascript or link that opened it:

```
<a href="somewhere.html" target="windowName">Click here to open a new window</a>
```

Alternatively, you can pass a "window handle" to the "switchTo().window()" method. Knowing this, it's possible to iterate over every open window like so:

```
for (String handle : driver.getWindowHandles()) {
    driver.switchTo().window(handle);
}
```

You can also swing from frame to frame (or into iframes):

```
driver.switchTo().frame("frameName");
```

It's possible to access subframes by separating the path with a dot, and you can specify the frame by its index too. That is:

```
driver.switchTo().frame("frameName.0.child");
```

would go to the frame named "child" of the first subframe of the frame called "frameName". **All frames are evaluated as if from *top*.**

4.2.7 Popup Dialogs

Starting with Selenium 2.0 beta 1, there is built in support for handling popup dialog boxes. After you've triggerd and action that would open a popup, you can access the alert with the following:

```
Alert alert = driver.switchTo().alert();
```

This will return the currently open alert object. With this object you can now accept, dismiss, read its contents or even type into a prompt. This interface works equally well on alerts, confirms, prompts. Refer to the JavaDocs for more information.

4.2.8 Navigation: History and Location

Earlier, we covered navigating to a page using the "get" command (`driver.get("http://www.example.com")`) As you've seen, WebDriver has a number of smaller, task-focused interfaces, and navigation is a useful task. Because loading a page is such a fundamental requirement, the method to do this lives on the main WebDriver interface, but it's simply a synonym to:

```
driver.navigate().to("http://www.example.com");
```

To reiterate: "`navigate().to()`" and "`get()`" do exactly the same thing. One's just a lot easier to type than the other!

The "navigate" interface also exposes the ability to move backwards and forwards in your browser's history:

```
driver.navigate().forward();
driver.navigate().back();
```

Please be aware that this functionality depends entirely on the underlying browser. It's just possible that something unexpected may happen when you call these methods if you're used to the behaviour of one browser over another.

4.2.9 Cookies

Before we leave these next steps, you may be interested in understanding how to use cookies. First of all, you need to be on the domain that the cookie will be valid for:

```
// Go to the correct domain
driver.get("http://www.example.com");

// Now set the cookie. This one's valid for the entire domain
Cookie cookie = new Cookie("key", "value");
driver.manage().addCookie(cookie);

// And now output all the available cookies for the current URL
Set<Cookie> allCookies = driver.manage().getCookies();
for (Cookie loadedCookie : allCookies) {
    System.out.println(String.format("%s -> %s", loadedCookie.getName(), loadedCookie.g
}
```

4.2.10 Next, Next Steps!

This has been a high level walkthrough of WebDriver and some of its key capabilities. You may want to look at the *Test Design Considerations chapter* to get some ideas about how you can reduce the pain of maintaining your tests and how to make your code more modular.

4.3 WebDriver Implementations

4.3.1 HtmlUnit Driver

This is currently the fastest and most lightweight implementation of WebDriver. As the name suggests, this is based on HtmlUnit.

Pros

- Fastest implementation of WebDriver

- A pure Java solution and so it is platform independent.
- Supports JavaScript

Cons

- Emulates other browsers' JavaScript behaviour (see below)

JavaScript in the HtmlUnit Driver

None of the popular browsers uses the JavaScript engine used by HtmlUnit (Rhino). If you test JavaScript using HtmlUnit the results may differ significantly from those browsers.

When we say "JavaScript" we actually mean "JavaScript and the DOM". Although the DOM is defined by the W3C each browser out there has its own quirks and differences in their implementation of the DOM and in how JavaScript interacts with it. HtmlUnit has an impressively complete implementation of the DOM and has good support for using JavaScript, but it is no different from any other browser: it has its own quirks and differences from both the W3C standard and the DOM implementations of the major browsers, despite its ability to mimic other browsers.

With WebDriver, we had to make a choice; do we enable HtmlUnit's JavaScript capabilities and run the risk of teams running into problems that only manifest themselves there, or do we leave JavaScript disabled, knowing that there are more and more sites that rely on JavaScript? We took the conservative approach, and by default have disabled support when we use HtmlUnit. With each release of both WebDriver and HtmlUnit, we reassess this decision: we hope to enable JavaScript by default on the HtmlUnit at some point.

Enabling JavaScript

If you can't wait, enabling JavaScript support is very easy:

```
HtmlUnitDriver driver = new HtmlUnitDriver();
driver.setJavascriptEnabled(true);
```

This will cause the HtmlUnit Driver to emulate Internet Explorer's JavaScript handling by default.

4.3.2 Firefox Driver

Pros

- Runs in a real browser and supports JavaScript
- Faster than the Internet Explorer Driver

Cons

- Slower than the HtmlUnit Driver

Before Going Any Further

The Firefox Driver contains everything it needs in the JAR file. If you're just interested in using this driver, then all you need to do is put the `webdriver-firefox.jar` or `webdriver-all.jar` on your `CLASSPATH`, and WebDriver will do everything else for you.

If you want to dig deeper, though, carry on reading!

Important System Properties

The following system properties (read using `System.getProperty()` and set using `System.setProperty()` in Java code or the `-DpropertyName=value` command line flag) are used by the Firefox Driver:

Property	What it means
webdriver.firefox.bin	The location of the binary used to control Firefox.
webdriver.firefox.profile	The name of the profile to use when starting Firefox. This defaults to WebDriver creating an anonymous profile
webdriver.reap_profile	Should be "true" if temporary files and profiles should not be deleted

Normally the Firefox binary is assumed to be in the default location for your particular operating system:

OS	Expected Location of Firefox
Linux	firefox (found using "which")
Mac	/Applications/Firefox.app/Contents/MacOS/firefox
Windows XP	%PROGRAMFILES%\Mozilla Firefox\firefox.exe
Windows Vista	\Program Files (x86)\Mozilla Firefox\firefox.exe

By default, the Firefox driver creates an anonymous profile

Installing a Downloaded Binary

The "wedriver-all.zip" which may be downloaded from the website, contains all the dependencies (including the common library) required to run the Firefox Driver. In order to use it:

- Copy all the "jar" files on to your `CLASSPATH`.

4.3.3 Internet Explorer Driver

This driver has been tested with Internet Explorer 6, 7 and 8 on XP. It has also been successfully tested on Vista.

Pros

- Runs in a real browser and supports JavaScript

Cons

- Obviously the Internet Explorer Driver will only work on Windows!
- Comparatively slow (though still pretty snappy :)

Installing

Simply add `webdriver-all.jar` to your `CLASSPATH`. You do not need to run an installer before using the Internet Explorer Driver, though some configuration is required.

Required Configuration

Add every site you intend to visit to your "Trusted Sites" If you do not do this, then you will not be able to interact with the page.

4.3.4 Chrome Driver

See below for instructions on how to install the Chrome Driver.

Note that Chrome Driver is one of the newest drivers. Please report any problems through the issue tracker.

Pros

- Runs in a real browser and supports JavaScript
- Because Chrome is a Webkit-based browser, the Chrome Driver may allow you to verify that your site works in Safari. Note that since Chrome uses its own V8 JavaScript engine rather than Safari's Nitro engine, JavaScript execution may differ.

Cons

- Slower than the HtmlUnit Driver

Before Going Any Further

The Chrome Driver contains everything it needs in the JAR file. If you're just interested in using this driver, then all you need to do is put `webdriver-all.jar` on your `CLASSPATH`, and WebDriver will do everything else for you.

The *Chrome Driver_* works with Google Chrome version 4.0 and above.

Important System Properties

The following system properties (read using `System.getProperty()` and set using `System.setProperty()` in Java code or the `-DpropertyName=value` command line flag) are used by the Chrome Driver:

Property	What it means
webdriver.chrome.bin	The location of the binary used to control Chrome.
webdriver.reap_profile	Should be "true" if temporary files and profiles should not be deleted

Normally the Chrome binary is assumed to be in the default location for your particular operating system:

OS	Expected Location of Chrome
Linux	/usr/bin/google-chrome
Mac	/Applications/Google Chrome.app/Contents/MacOS/GoogleChrome or /User/:username/:as_to_the_left
Windows XP	%HOMEPATH%\Local Settings\Application Data\Google\Chrome\Application\chrome.exe
Windows Vista	C:\Users%USERNAME%\AppData\Local\Google\Chrome\Application\chrome.exe

Installing a Downloaded Binary

The "wedriver-all.zip" which may be downloaded from the website, contains all the dependencies required to run the Chrome Driver. In order to use it, copy all the "jar" files on to your `CLASSPATH`.

4.4 Emulating Selenium RC

The Java version of WebDriver provides an implementation of the Selenium RC API. It is used like so:

```
// You may use any WebDriver implementation. Firefox is used here as an example
WebDriver driver = new FirefoxDriver();

// A "base url", used by selenium to resolve relative URLs
 String baseUrl = "http://www.google.com";

// Create the Selenium implementation
Selenium selenium = new WebDriverBackedSelenium(driver, baseUrl);

// Perform actions with selenium
selenium.open("http://www.google.com");
selenium.type("name=q", "cheese");
selenium.click("name=btnG");

// Get the underlying WebDriver implementation back. This will refer to the
// same WebDriver instance as the "driver" variable above.
WebDriver driverInstance = ((WebDriverBackedSelenium) selenium).getUnderlyingWebDriver()

    //Finally, close the browser. Call stop on the WebDriverBackedSelenium instance
    //instead of calling driver.quit(). Otherwise, the JVM will continue running after
    //the browser has been closed.
    selenium.stop();
```

4.4.1 Pros

- Allows for the WebDriver and Selenium APIs to live side-by-side
- Provides a simple mechanism for a managed migration from the Selenium RC API to WebDriver's
- Does not require the standalone Selenium RC server to be run

4.4.2 Cons

- Does not implement every method
- More advanced Selenium usage (using "browserbot" or other built-in JavaScript methods from Selenium Core) may not work
- Some methods may be slower due to underlying implementation differences

4.4.3 Backing WebDriver with Selenium

WebDriver doesn't support as many browsers as Selenium RC does, so in order to provide that support while still using the WebDriver API, you can make use of the `SeleneseCommandExecutor` It is done like this:

```
Capabilities capabilities = new DesiredCapabilities()
capabilities.setBrowserName( "safari" );
CommandExecutor executor = new SeleneseCommandExecutor( "http:localhost:4444/" , "http:/
WebDriver driver = new RemoteWebDriver(executor, capabilities);
```

There are currently some major limitations with this approach, notably that findElements doesn't work as expected. Also, because we're using Selenium Core for the heavy lifting of driving the browser, you are limited by the JavaScript sandbox.

4.5 Tips and Tricks

4.5.1 Changing the user agent

This is easy with the Firefox Driver:

```
FirefoxProfile profile = new FirefoxProfile();
profile.addAdditionalPreference( "general.useragent.override" , "some UA string" );
WebDriver driver = new FirefoxDriver(profile);
```

4.5.2 Tweaking an existing Firefox profile

Suppose that you wanted to modify the user agent string (as above), but you've got a tricked out Firefox profile that contains dozens of useful extensions. There are two ways to obtain this profile. Assuming that the profile has been created using Firefox's profile manager (`firefox -ProfileManager`):

```
ProfileIni allProfiles = new ProfilesIni();
FirefoxProfile profile = allProfiles.getProfile( "WebDriver" );
profile.setPreferences( "foo.bar" , 23);
WebDriver driver = new FirefoxDriver(profile);
```

Alternatively, if the profile isn't already registered with Firefox:

```
File profileDir = new File( "path/to/top/level/of/profile" );
FirefoxProfile profile = new FirefoxProfile(profileDir);
profile.addAdditionalPreferences(extraPrefs);
```

```
WebDriver driver = new FirefoxDriver(profile);
Enabling features that might not be wise to use in Firefox
```

As we develop features in the Firefox Driver, we expose the ability to use them. For example, until we feel native events are stable on Firefox for Linux, they are disabled by default. To enable them:

```
FirefoxProfile profile = new FirefoxProfile();
profile.setEnableNativeEvents(true);
WebDriver driver = new FirefoxDriver(profile);
```

4.6 How XPATH Works in WebDriver

At a high level, WebDriver uses a browser's native XPath capabilities wherever possible. On those browsers that don't have native XPath support, we have provided our own implementation. This can lead to some unexpected behaviour unless you are aware of the differences in the various xpath engines.

Driver	Tag and Attribute Name	Attribute Values	Native XPath Support
HtmlUnit Driver	Lower-cased	As they appear in the HTML	Yes
Internet Explorer Driver	Lower-cased	As they appear in the HTML	No
Firefox Driver	Case insensitive	As they appear in the HTML	Yes

This is a little abstract, so for the following piece of HTML:

```
<input type="text" name="example" />
<INPUT type="text" name="other" />
```

The following number of matches will be found

XPath expression	HtmlUnit Driver	Firefox Driver	Internet Explorer Driver
//input	1 ("example")	2	2
//INPUT	0	2	0

4.6.1 Matching Implicit Attributes

Sometimes HTML elements do not need attributes to be explicitly declared because they will default to known values. For example, the "input" tag does not require the "type" attribute because it defaults to "text". The rule of thumb when using xpath in WebDriver is that you **should not** expect to be able to match against these implicit attributes.

4.7 Getting and Using WebDriver

4.7.1 From a New Download

Unpack the "webdriver-all.zip" you can download from the site, and add all the JARs to your `CLASSPATH`. This will give you the Chrome Driver, Firefox Driver, HtmlUnit Driver, Internet Ex-

plorer Driver, Remote Web Driver client and the support packages. The support packages give you useful helper classes, such as the LiftStyleApi and the PageFactory.

4.7.2 With Maven

If you want to use the HtmlUnit Driver, add the following dependency to your pom.xml:

```
<dependency>
    <groupId>org.seleniumhq.webdriver</groupId>
    <artifactId>webdriver-htmlunit</artifactId>
    <version>0.9.7376</version>
</dependency>
```

If you want to use the Firefox Driver, you need to add the following dependency to your pom.xml:

```
<dependency>
    <groupId>org.seleniumhq.webdriver</groupId>
    <artifactId>webdriver-firefox</artifactId>
    <version>0.9.7376</version>
</dependency>
```

If you want to use the Internet Explorer Driver, you need to add the following dependency to your pom.xml:

```
<dependency>
    <groupId>org.seleniumhq.webdriver</groupId>
    <artifactId>webdriver-ie</artifactId>
    <version>0.9.7376</version>
</dependency>
```

If you want to use the Chrome Driver, you need to add the following dependency to your pom.xml:

```
<dependency>
    <groupId>org.seleniumhq.webdriver</groupId>
    <artifactId>webdriver-chrome</artifactId>
    <version>0.9.7376</version>
</dependency>
```

Finally, if you like to use any of our support classes, you should add the following dependency to your pom.xml:

```
<dependency>
    <groupId>org.seleniumhq.webdriver</groupId>
    <artifactId>webdriver-support</artifactId>
    <version>0.9.7376</version>
</dependency>
```

4.8 Roadmap

The roadmap for WebDriver is available here

4.9 Further Resources

You can find further resources for WebDriver in WebDriver's wiki

CHAPTER

FIVE

WEBDRIVER: ADVANCED USAGE

5.1 Explicit and Implicit Waits

Waiting is having the automated task execution elapse a certain amount of time before continuing with the next step.

5.1.1 Explicit Waits

An explicit waits is code you define to wait for a certain condition to occur before proceeding further in the code. The worst case of this is Thread.sleep(), which sets the condition to an exact time period to wait. There are some convenience methods provided that help you write code that will wait only as long as required. WebDriverWait in combination with ExpectedCondition is one way this can be accomplished.

```
WebDriver driver = new FirefoxDriver();
WebElement myDynamicElement = (new WebDriverWait(driver, 10))
  .until(new ExpectedCondition<WebElement>(){
        @Override
        public WebElement apply(WebDriver d) {
                return d.findElement(By.id("myDynamicElement"));
        }});
```

This waits up to 10 seconds before throwing an Element not found Exception or if it finds the element will return it in 0 - 10 seconds. WebDriverWait by default calls the ExpectedCondition every 500 milliseconds until it returns successfully. A successful return is for ExpectedCondition type is Boolean return true or not null return value for all other ExpectedCondition types.

This example is also functionally equivalent to the first Implicit Waits example.

5.1.2 Implicit Waits

An implicit wait is to tell WebDriver to poll the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0. Once set, the implicit wait is set for the life of the WebDriver object instance.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
WebElement myDynamicElement = driver.findElement(By.id("myDynamicElement"));
```

5.2 RemoteWebDriver

You'll start by using the HtmlUnit Driver. This is a pure Java driver that runs entirely in-memory. Because of this, you won't see a new browser window open.

```
package org.openqa.selenium.example;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;

public class HtmlUnitExample  {
    public static void main(String[] args) {
        // Create a new instance of the html unit driver
        // Notice that the remainder of the code relies on the interface,
        // not the implementation.
        WebDriver driver = new HtmlUnitDriver();

        // And now use this to visit Google
        driver.get("http://www.google.com");

        // Find the text input element by its name
        WebElement element = driver.findElement(By.name("q"));

        // Enter something to search for
        element.sendKeys("Cheese!");

        // Now submit the form. WebDriver will find the form for us from the element
        element.submit();

        // Check the title of the page
        System.out.println("Page title is: " + driver.getTitle());
    }
}
```

HtmlUnit isn't confined to just Java. Selenium makes accessing HtmlUnit easy from any language. Below is the same example in C#. Note that you'll need to run the remote WebDriver server to use HtmlUnit from C#

```
using OpenQA.Selenium;
using OpenQA.Selenium.Remote;

class Example
{
    static void Main(string[] args)
    {
        //to use HtmlUnit from .Net we must access it through the RemoteWebDriver
        //Download and run the selenium-server-standalone-2.0b1.jar locally to run this
        ICapabilities desiredCapabilities = DesiredCapabilities.HtmlUnit();
        IWebDriver driver = new RemoteWebDriver(desiredCapabilities);

        //the .Net Webdriver relies on a slightly different API to navigate to
        //web pages because 'get' is a keyword in .Net
        driver.Navigate().GoToUrl("http://google.ca/");
```

```
        //The rest of the code should look very similar to the Java library
        IWebElement element = driver.FindElement(By.Name("q"));
        element.SendKeys("Cheese!");
        element.Submit();
        System.Console.WriteLine("Page title is: " + driver.Title);
        driver.Quit();
        System.Console.ReadLine();
    }
}
```

Compile and run this. You should see a line with the title of the Google search results as output on the console. Congratulations, you've managed to get started with WebDriver!

5.3 AdvancedUserInteractions

Todo

5.4 HTML5

Todo

5.5 Cookies

Todo

5.6 Browser Startup Manipulation

Todo

Topics to be included:

- restoring cookies
- changing firefox profile
- running browsers with plugins

5.7 Parallelizing Your Test Runs

Todo

CHAPTER

SIX

SELENIUM 1 (SELENIUM RC)

6.1 Introduction

As you can read in *Brief History of The Selenium Project*, Selenium RC was the main Selenium project for a long time, before the WebDriver/Selenium merge brought up Selenium 2, the newest and more powerful tool.

Selenium 1 is still actively supported (mostly in maintenance mode) and provides some features that may not be available in Selenium 2 for a while, including support for several languages (Java, Javascript, PRuby, HP, Python, Perl and C#) and support for almost every browser out there.

6.2 How Selenium RC Works

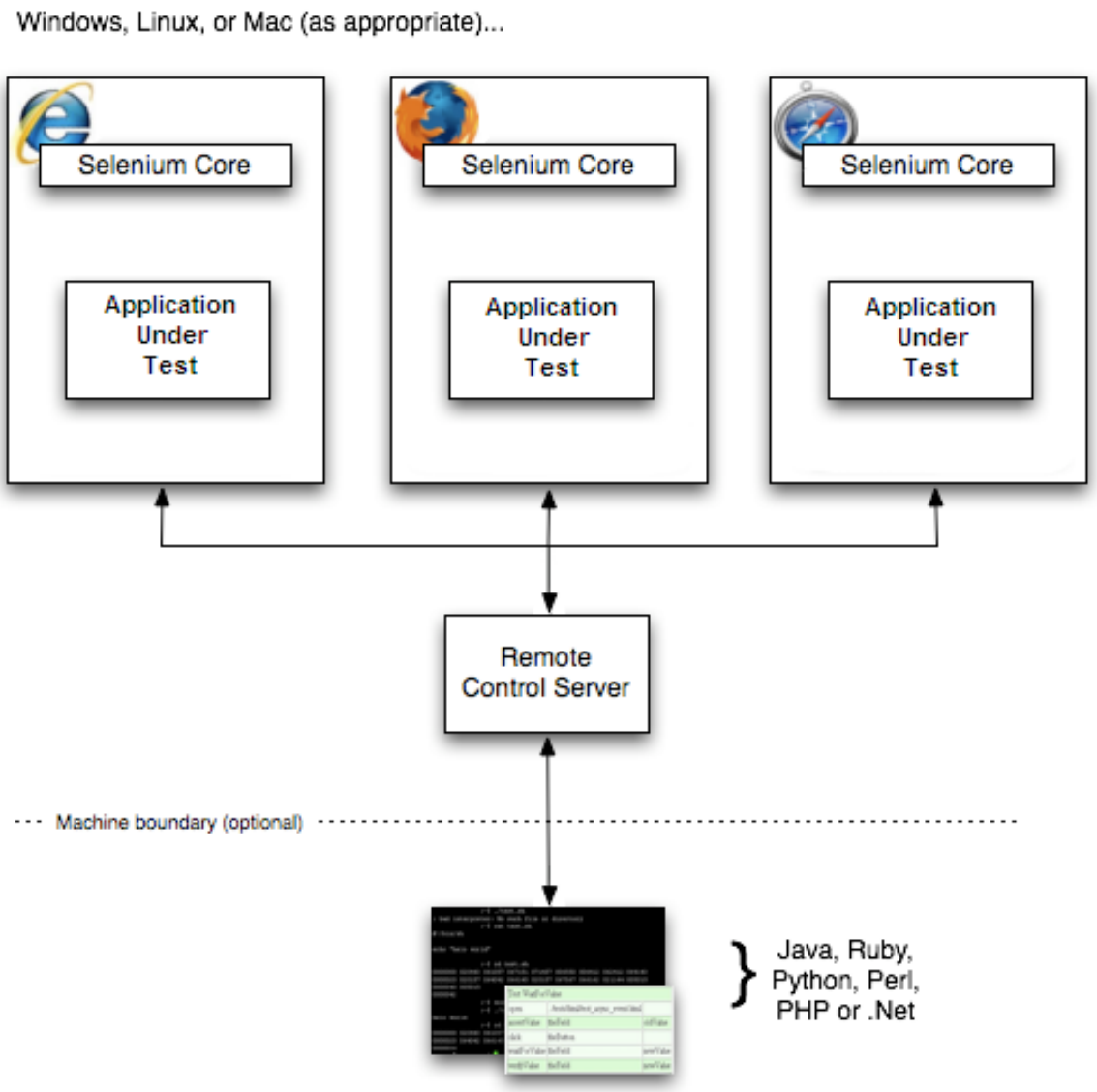
First, we will describe how the components of Selenium RC operate and the role each plays in running your test scripts.

6.2.1 RC Components

Selenium RC components are:

- The Selenium Server which launches and kills browsers, interprets and runs the Selenese commands passed from the test program, and acts as an *HTTP proxy*, intercepting and verifying HTTP messages passed between the browser and the AUT.
- Client libraries which provide the interface between each programming language and the Selenium RC Server.

Here is a simplified architecture diagram....



The diagram shows the client libraries communicate with the Server passing each Selenium command for execution. Then the server passes the Selenium command to the browser using Selenium-Core JavaScript commands. The browser, using its JavaScript interpreter, executes the Selenium command. This runs the Selenese action or verification you specified in your test script.

6.2.2 Selenium Server

Selenium Server receives Selenium commands from your test program, interprets them, and reports back to your program the results of running those tests.

The RC server bundles Selenium Core and automatically injects it into the browser. This occurs when your test program opens the browser (using a client library API function). Selenium-Core is a JavaScript program, actually a set of JavaScript functions which interprets and executes Selenese commands using the browser's built-in JavaScript interpreter.

The Server receives the Selenese commands from your test program using simple HTTP GET/POST requests. This means you can use any programming language that can send HTTP requests to automate Selenium tests on the browser.

6.2.3 Client Libraries

The client libraries provide the programming support that allows you to run Selenium commands from a program of your own design. There is a different client library for each supported language. A Selenium client library provides a programming interface (API), i.e., a set of functions, which run Selenium commands from your own program. Within each interface, there is a programming function that supports each Selenese command.

The client library takes a Selenese command and passes it to the Selenium Server for processing a specific action or test against the application under test (AUT). The client library also receives the result of that command and passes it back to your program. Your program can receive the result and store it into a program variable and report it as a success or failure, or possibly take corrective action if it was an unexpected error.

So to create a test program, you simply write a program that runs a set of Selenium commands using a client library API. And, optionally, if you already have a Selenese test script created in the Selenium-IDE, you can *generate the Selenium RC code*. The Selenium-IDE can translate (using its Export menu item) its Selenium commands into a client-driver's API function calls. See the Selenium-IDE chapter for specifics on exporting RC code from Selenium-IDE.

6.3 Installation

Installation is rather a misnomer for Selenium. Selenium has set of libraries available in the programming language of your choice. You could download them from downloads page

Once you've chosen a language to work with, you simply need to:

- Install the Selenium RC Server.
- Set up a programming project using a language specific client driver.

6.3.1 Installing Selenium Server

The Selenium RC server is simply a Java *jar* file (*selenium-server-standalone-<version-number>.jar*), which doesn't require any special installation. Just downloading the zip file and extracting the server in the desired directory is sufficient.

6.3.2 Running Selenium Server

Before starting any tests you must start the server. Go to the directory where Selenium RC's server is located and run the following from a command-line console.

```
java -jar selenium-server-standalone-<version-number>.jar
```

This can be simplified by creating a batch or shell executable file (.bat on Windows and .sh on Linux) containing the command above. Then make a shortcut to that executable on your desktop and simply double-click the icon to start the server.

For the server to run you'll need Java installed and the PATH environment variable correctly configured to run it from the console. You can check that you have Java correctly installed by running the following on a console:

```
java -version
```

If you get a version number (which needs to be 1.5 or later), you're ready to start using Selenium RC.

6.3.3 Using the Java Client Driver

- Download Selenium RC java client driver from the SeleniumHQ downloads page.
- Extract selenium-java-<version-number>.jar file
- Open your desired Java IDE (Eclipse, NetBeans, IntelliJ, Netweaver, etc.)
- Create a java project.
- Add the selenium-java-<version-number>.jar files to your project as references.
- Add to your project classpath the file selenium-java-<version-number>.jar.
- From Selenium-IDE, export a script to a Java file and include it in your Java project, or write your Selenium test in Java using the selenium-java-client API. The API is presented later in this chapter. You can either use JUnit, or TestNg to run your test, or you can write your own simple main() program. These concepts are explained later in this section.
- Run Selenium server from the console.
- Execute your test from the Java IDE or from the command-line.

For details on Java test project configuration, see the Appendix sections *Configuring Selenium RC With Eclipse* and *Configuring Selenium RC With Intellij*.

6.3.4 Using the Python Client Driver

- Download Selenium RC from the SeleniumHQ downloads page
- Extract the file *selenium.py*
- Either write your Selenium test in Python or export a script from Selenium-IDE to a python file.
- Add to your test's path the file *selenium.py*
- Run Selenium server from the console
- Execute your test from a console or your Python IDE

For details on Python client driver configuration, see the appendix *Python Client Driver Configuration*.

6.3.5 Using the .NET Client Driver

- Download Selenium RC from the SeleniumHQ downloads page
- Extract the folder
- Download and install NUnit (Note: You can use NUnit as your test engine. If you're not familiar yet with NUnit, you can also write a simple main() function to run your tests; however NUnit is very useful as a test engine.)

- Open your desired .Net IDE (Visual Studio, SharpDevelop, MonoDevelop)
- Create a class library (.dll)
- Add references to the following DLLs: nmock.dll, nunit.core.dll, nunit. framework.dll, ThoughtWorks.Selenium.Core.dll, ThoughtWorks.Selenium.IntegrationTests.dll and ThoughtWorks.Selenium.UnitTests.dll
- Write your Selenium test in a .Net language (C#, VB.Net), or export a script from Selenium-IDE to a C# file and copy this code into the class file you just created.
- Write your own simple main() program or you can include NUnit in your project for running your test. These concepts are explained later in this chapter.
- Run Selenium server from console
- Run your test either from the IDE, from the NUnit GUI or from the command line

For specific details on .NET client driver configuration with Visual Studio, see the appendix *.NET client driver configuration*.

6.3.6 Using the Ruby Client Driver

- If you do not already have RubyGems, install it from RubyForge
- Run `gem install selenium-client`
- At the top of your test script, add `require "selenium/client"`
- Write your test script using any Ruby test harness (eg Test::Unit, Mini::Test or RSpec).
- Run Selenium RC server from the console.
- Execute your test in the same way you would run any other Ruby script.

For details on Ruby client driver configuration, see the Selenium-Client documentation

6.4 From Selenese to a Program

The primary task for using Selenium RC is to convert your Selenese into a programming language. In this section, we provide several different language-specific examples.

6.4.1 Sample Test Script

Let's start with an example Selenese test script. Imagine recording the following test with Selenium-IDE.

open	/	
type	q	selenium rc
clickAndWait	btnG	
assertTextPresent	Results * for selenium rc	

Note: This example would work with the Google search page http://www.google.com

6.4.2 Selenese as Programming Code

Here is the test script exported (via Selenium-IDE) to each of the supported programming languages. If you have at least basic knowledge of an object- oriented programming language, you will understand how Selenium runs Selenese commands by reading one of these examples. To see an example in a specific language, select one of these buttons.

In C#:

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
using Selenium;

namespace SeleniumTests
{
    [TestFixture]
    public class NewTest
    {
        private ISelenium selenium;
        private StringBuilder verificationErrors;

        [SetUp]
        public void SetupTest()
        {
            selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://www
            selenium.Start();
            verificationErrors = new StringBuilder();
        }

        [TearDown]
        public void TeardownTest()
        {
            try
            {
                selenium.Stop();
            }
            catch (Exception)
            {
                // Ignore errors if unable to close the browser
            }
            Assert.AreEqual("", verificationErrors.ToString());
        }

        [Test]
        public void TheNewTest()
        {
            selenium.Open("/");
            selenium.Type("q", "selenium rc");
            selenium.Click("btnG");
            selenium.WaitForPageToLoad("30000");
            Assert.AreEqual("selenium rc - Google Search", selenium.GetTitle());
        }
    }
}
```

In Java:

```
    /** Add JUnit framework to your classpath if not already there
     *  for this example to work
    */
package com.example.tests;

import com.thoughtworks.selenium.*;
import java.util.regex.Pattern;

public class NewTest extends SeleneseTestCase {
    public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");
    }
      public void testNew() throws Exception {
          selenium.open("/");
          selenium.type("q", "selenium rc");
          selenium.click("btnG");
          selenium.waitForPageToLoad("30000");
          assertTrue(selenium.isTextPresent("Results * for selenium rc"));
    }
}
```

In Perl:

```
use strict;
use warnings;
use Time::HiRes qw( sleep );
use Test::WWW::Selenium;
use Test::More "no_plan";
use Test::Exception;

my $sel = Test::WWW::Selenium->new( host => "localhost",
                                    port => 4444,
                                    browser => "*firefox",
                                    browser_url => "http://www.google.com/" );

$sel->open_ok("/");
$sel->type_ok("q", "selenium rc");
$sel->click_ok("btnG");
$sel->wait_for_page_to_load_ok("30000");
$sel->is_text_present_ok("Results * for selenium rc");
```

In PHP:

```
<?php

require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class Example extends PHPUnit_Extensions_SeleniumTestCase
{
  function setUp()
  {
```

```
    $this->setBrowser("*firefox");
    $this->setBrowserUrl("http://www.google.com/");
  }

  function testMyTestCase()
  {
    $this->open("/");
    $this->type("q", "selenium rc");
    $this->click("btnG");
    $this->waitForPageToLoad("30000");
    $this->assertTrue($this->isTextPresent("Results * for selenium rc"));
  }
}
?>
```

in Python:

```
from selenium import selenium
import unittest, time, re

class NewTest(unittest.TestCase):
    def setUp(self):
        self.verificationErrors = []
        self.selenium = selenium("localhost", 4444, "*firefox",
                "http://www.google.com/")
        self.selenium.start()

    def test_new(self):
        sel = self.selenium
        sel.open("/")
        sel.type("q", "selenium rc")
        sel.click("btnG")
        sel.wait_for_page_to_load("30000")
        self.failUnless(sel.is_text_present("Results * for selenium rc"))

    def tearDown(self):
        self.selenium.stop()
        self.assertEqual([], self.verificationErrors)
```

in Ruby:

```
require "selenium"
require "test/unit"

class NewTest < Test::Unit::TestCase
  def setup
    @verification_errors = []
    if $selenium
      @selenium = $selenium
    else
      @selenium = Selenium::SeleniumDriver.new("localhost", 4444, "*firefox", "h
      @selenium.start
    end
    @selenium.set_context("test_new")
```

```
  end

  def teardown
    @selenium.stop unless $selenium
    assert_equal [], @verification_errors
  end

  def test_new
    @selenium.open " / "
    @selenium.type " q ", " selenium rc "
    @selenium.click " btnG "
    @selenium.wait_for_page_to_load " 30000 "
    assert @selenium.is_text_present( " Results * for selenium rc " )
  end
end
```

In the next section we'll explain how to build a test program using the generated code.

6.5 Programming Your Test

Now we'll illustrate how to program your own tests using examples in each of the supported programming languages. There are essentially two tasks:

- Generate your script into a programming language from Selenium-IDE, optionally modifying the result.
- Write a very simple main program that executes the generated code.

Optionally, you can adopt a test engine platform like JUnit or TestNG for Java, or NUnit for .NET if you are using one of those languages.

Here, we show language-specific examples. The language-specific APIs tend to differ from one to another, so you'll find a separate explanation for each.

- Java
- C#
- Python
- Ruby
- Perl, PHP

6.5.1 Java

For Java, people use either JUnit or TestNG as the test engine. Some development environments like Eclipse have direct support for these via plug-ins. This makes it even easier. Teaching JUnit or TestNG is beyond the scope of this document however materials may be found online and there are publications available. If you are already a "java-shop" chances are your developers will already have some experience with one of these test frameworks.

You will probably want to rename the test class from "NewTest" to something of your own choosing. Also, you will need to change the browser-open parameters in the statement:

```
selenium = new DefaultSelenium("localhost", 4444, "*iehta", "http://www.google.com/");
```

The Selenium-IDE generated code will look like this. This example has comments added manually for additional clarity.

```
package com.example.tests;
// We specify the package of our tests

import com.thoughtworks.selenium.*;
// This is the driver's import. You'll use this for instantiating a
// browser and making it do what you need.

import java.util.regex.Pattern;
// Selenium-IDE add the Pattern module because it's sometimes used for
// regex validations. You can remove the module if it's not used in your
// script.

public class NewTest extends SeleneseTestCase {
// We create our Selenium test case

      public void setUp() throws Exception {
        setUp("http://www.google.com/", "*firefox");
            // We instantiate and start the browser
      }

      public void testNew() throws Exception {
            selenium.open("/");
            selenium.type("q", "selenium rc");
            selenium.click("btnG");
            selenium.waitForPageToLoad("30000");
            assertTrue(selenium.isTextPresent("Results * for selenium rc"));
            // These are the real test steps
    }
}
```

6.5.2 C#

The .NET Client Driver works with Microsoft.NET. It can be used with any .NET testing framework like NUnit or the Visual Studio 2005 Team System.

Selenium-IDE assumes you will use NUnit as your testing framework. You can see this in the generated code below. It includes the *using* statement for NUnit along with corresponding NUnit attributes identifying the role for each member function of the test class.

You will probably have to rename the test class from "NewTest" to something of your own choosing. Also, you will need to change the browser-open parameters in the statement:

```
selenium = new DefaultSelenium("localhost", 4444, "*iehta", "http://www.google.com/");
```

The generated code will look similar to this.

```
using System;
using System.Text;
```

```
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
using Selenium;

namespace SeleniumTests

{
    [TestFixture]

    public class NewTest

    {
    private ISelenium selenium;

    private StringBuilder verificationErrors;

    [SetUp]

    public void SetupTest()

    {
        selenium = new DefaultSelenium( "localhost", 4444, "*iehta",
        "http://www.google.com/" );

        selenium.Start();

        verificationErrors = new StringBuilder();
    }

    [TearDown]

    public void TeardownTest()
    {
        try
        {
        selenium.Stop();
        }

        catch (Exception)
        {
        // Ignore errors if unable to close the browser
        }

        Assert.AreEqual( "", verificationErrors.ToString());
    }
    [Test]

    public void TheNewTest()
    {
        // Open Google search engine.
        selenium.Open( "http://www.google.com/" );

        // Assert Title of page.
        Assert.AreEqual( "Google", selenium.GetTitle());

        // Provide search term as "Selenium OpenQA"
```

```
        selenium.Type("q", "Selenium OpenQA");

        // Read the keyed search term and assert it.
        Assert.AreEqual("Selenium OpenQA", selenium.GetValue("q"));

        // Click on Search button.
        selenium.Click("btnG");

        // Wait for page to load.
        selenium.WaitForPageToLoad("5000");

        // Assert that "www.openqa.org" is available in search results.
        Assert.IsTrue(selenium.IsTextPresent("www.openqa.org"));

        // Assert that page title is - "Selenium OpenQA - Google Search"
        Assert.AreEqual("Selenium OpenQA - Google Search",
                    selenium.GetTitle());
    }
    }
}
```

You can allow NUnit to manage the execution of your tests. Or alternatively, you can write a simple main() program that instantiates the test object and runs each of the three methods, SetupTest(), TheNewTest(), and TeardownTest() in turn.

6.5.3 Python

Pyunit is the test framework to use for Python. To learn Pyunit refer to its *official documentation* <http://docs.python.org/library/unittest.html>_.

The basic test structure is:

```
from selenium import selenium
# This is the driver's import.  You'll use this class for instantiating a
# browser and making it do what you need.

import unittest, time, re
# This are the basic imports added by Selenium-IDE by default.
# You can remove the modules if they are not used in your script.

class NewTest(unittest.TestCase):
# We create our unittest test case

    def setUp(self):
        self.verificationErrors = []
        # This is an empty array where we will store any verification errors
        # we find in our tests

        self.selenium = selenium("localhost", 4444, "*firefox",
                "http://www.google.com/")
        self.selenium.start()
        # We instantiate and start the browser

    def test_new(self):
        # This is the test code.  Here you should put the actions you need
```

```
        # the browser to do during your test.

        sel = self.selenium
        # We assign the browser to the variable "sel" (just to save us from
        # typing "self.selenium" each time we want to call the browser).

        sel.open("/")
        sel.type("q", "selenium rc")
        sel.click("btnG")
        sel.wait_for_page_to_load("30000")
        self.failUnless(sel.is_text_present("Results * for selenium rc"))
        # These are the real test steps

    def tearDown(self):
        self.selenium.stop()
        # we close the browser (I'd recommend you to comment this line while
        # you are creating and debugging your tests)

        self.assertEqual([], self.verificationErrors)
        # And make the test fail if we found that any verification errors
        # were found
```

6.5.4 Ruby

Selenium-IDE generates reasonable Ruby, but requires the old Selenium gem. This is a problem because the official Ruby driver for Selenium is the Selenium-Client gem, not the old Selenium gem. In fact, the Selenium gem is no longer even under active development.

Therefore, it is advisable to update any Ruby scripts generated by the IDE as follows:

1. On line 1, change `require "selenium"` to `require "selenium/client"`

2. On line 11, change `Selenium::SeleniumDriver.new` to `Selenium::Client::Driver.new`

You probably also want to change the class name to something more informative than "Untitled," and change the test method's name to something other than "test_untitled."

Here is a simple example created by modifying the Ruby code generated by Selenium IDE, as described above.

```
# load the Selenium-Client gem
require "selenium/client"

# Load Test::Unit, Ruby 1.8's default test framework.
# If you prefer RSpec, see the examples in the Selenium-Client
# documentation.
require "test/unit"

class Untitled < Test::Unit::TestCase

  # The setup method is called before each test.
  def setup

    # This array is used to capture errors and display them at the
    # end of the test run.
```

```
    @verification_errors = []

    # Create a new instance of the Selenium-Client driver.
    @selenium = Selenium::Client::Driver.new \
      :host => "localhost",
      :port => 4444,
      :browser => "*chrome",
      :url => "http://www.google.com/",
      :timeout_in_second => 60

    # Start the browser session
    @selenium.start

    # Print a message in the browser-side log and status bar
    # (optional).
    @selenium.set_context("test_untitled")

  end

  # The teardown method is called after each test.
  def teardown

    # Stop the browser session.
    @selenium.stop

    # Print the array of error messages, if any.
    assert_equal [], @verification_errors
  end

  # This is the main body of your test.
  def test_untitled

    # Open the root of the site we specified when we created the
    # new driver instance, above.
    @selenium.open "/"

    # Type 'selenium rc' into the field named 'q'
    @selenium.type "q", "selenium rc"

    # Click the button named "btnG"
    @selenium.click "btnG"

    # Wait for the search results page to load.
    # Note that we don't need to set a timeout here, because that
    # was specified when we created the new driver instance, above.
    @selenium.wait_for_page_to_load

    begin

      # Test whether the search results contain the expected text.
      # Notice that the star (*) is a wildcard that matches any
      # number of characters.
      assert @selenium.is_text_present("Results * for selenium rc")

    rescue Test::Unit::AssertionFailedError

      # If the assertion fails, push it onto the array of errors.
```

```
      @verification_errors << $!

    end
  end
end
```

6.5.5 Perl, PHP

The members of the documentation team have not used Selenium RC with Perl or PHP. If you are using Selenium RC with either of these two languages please contact the Documentation Team (see the chapter on contributing). We would love to include some examples from you and your experiences, to support Perl and PHP users.

6.6 Learning the API

The Selenium RC API uses naming conventions that, assuming you understand Selenese, much of the interface will be self-explanatory. Here, however, we explain the most critical and possibly less obvious aspects.

6.6.1 Starting the Browser

In C#:

```
selenium = new DefaultSelenium("localhost", 4444, "*firefox", "http://www.google.com/
selenium.Start();
```

In Java:

```
setUp("http://www.google.com/", "*firefox");
```

In Perl:

```
my $sel = Test::WWW::Selenium->new( host => "localhost",
                                    port => 4444,
                                    browser => "*firefox",
                                    browser_url => "http://www.google.com/" );
```

In PHP:

```
$this->setBrowser("*firefox");
$this->setBrowserUrl("http://www.google.com/");
```

In Python:

```
self.selenium = selenium("localhost", 4444, "*firefox",
                         "http://www.google.com/")
self.selenium.start()
```

In Ruby:

```
@selenium = Selenium::ClientDriver.new("localhost", 4444, "*firefox", "http://ww
@selenium.start
```

Each of these examples opens the browser and represents that browser by assigning a "browser instance" to a program variable. This program variable is then used to call methods from the browser. These methods execute the Selenium commands, i.e. like *open* or *type* or the *verify* commands.

The parameters required when creating the browser instance are:

host Specifies the IP address of the computer where the server is located. Usually, this is the same machine as where the client is running, so in this case *localhost* is passed. In some clients this is an optional parameter.

port Specifies the TCP/IP socket where the server is listening waiting for the client to establish a connection. This also is optional in some client drivers.

browser The browser in which you want to run the tests. This is a required parameter.

url The base url of the application under test. This is required by all the client libs and is integral information for starting up the browser-proxy-AUT communication.

Note that some of the client libraries require the browser to be started explicitly by calling its *start()* method.

6.6.2 Running Commands

Once you have the browser initialized and assigned to a variable (generally named "selenium") you can make it run Selenese commands by calling the respective methods from the browser variable. For example, to call the *type* method of the selenium object:

```
selenium.type("field-id","string to type")
```

In the background the browser will actually perform a *type* operation, essentially identical to a user typing input into the browser, by using the locator and the string you specified during the method call.

6.7 Reporting Results

Selenium RC does not have its own mechanism for reporting results. Rather, it allows you to build your reporting customized to your needs using features of your chosen programming language. That's great, but what if you simply want something quick that's already done for you? Often an existing library or test framework can meet your needs faster than developing your own test reporting code.

6.7.1 Test Framework Reporting Tools

Test frameworks are available for many programming languages. These, along with their primary function of providing a flexible test engine for executing your tests, include library code for reporting results. For example, Java has two commonly used test frameworks, JUnit and TestNG. .NET also has its own, NUnit.

We won't teach the frameworks themselves here; that's beyond the scope of this user guide. We will simply introduce the framework features that relate to Selenium along with some techniques you can apply. There are good books available on these test frameworks however along with information on the internet.

6.7.2 Test Report Libraries

Also available are third-party libraries specifically created for reporting test results in your chosen programming language. These often support a variety of formats such as HTML or PDF.

6.7.3 What's The Best Approach?

Most people new to the testing frameworks will being with the framework's built-in reporting features. From there most will examine any available libraries as that's less time consuming than developing your own. As you begin to use Selenium no doubt you will start putting in your own "print statements" for reporting progress. That may gradually lead to you developing your own reporting, possibly in parallel to using a library or test framework. Regardless, after the initial, but short, learning curve you will naturally develop what works best for your own situation.

6.7.4 Test Reporting Examples

To illustrate, we'll direct you to some specific tools in some of the other languages supported by Selenium. The ones listed here are commonly used and have been used extensively (and therefore recommended) by the authors of this guide.

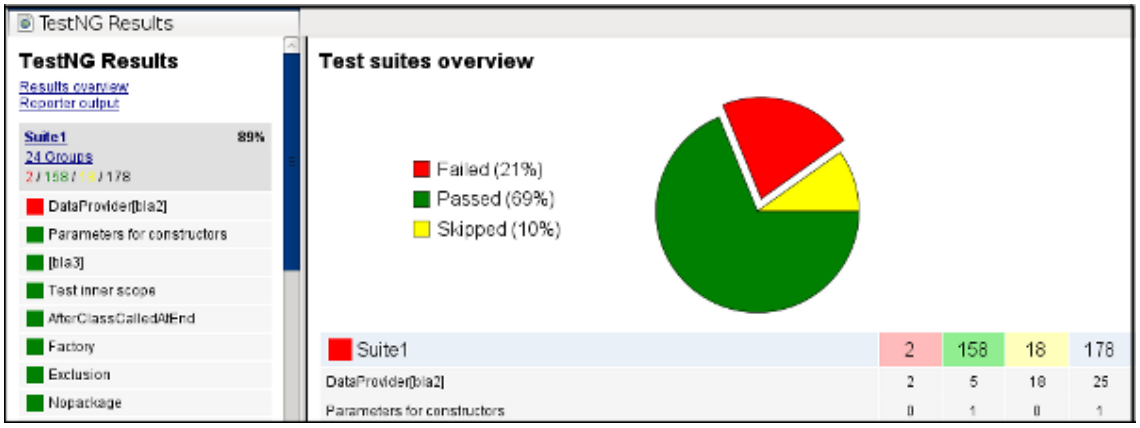
Test Reports in Java

- If Selenium Test cases are developed using JUnit then JUnit Report can be used to generate test reports. Refer to JUnit Report for specifics.

- If Selenium Test cases are developed using TestNG then no external task is required to generate test reports. The TestNG framework generates an HTML report which list details of tests. See TestNG Report for more.

- ReportNG is a HTML reporting plug-in for the TestNG framework. It is intended as a replacement for the default TestNG HTML report. ReportNG provides a simple, colour-coded view of the test results. See ReportNG for more.

- Also, for a very nice summary report try using TestNG-xslt. A TestNG-xslt Report looks like this.



See TestNG-xslt for more.

Logging the Selenese Commands

- Logging Selenium can be used to generate a report of all the Selenese commands in your test along with the success or failure of each. Logging Selenium extends the Java client driver to add this Selenese logging ability. Please refer to Logging Selenium.

Test Reports for Python

- When using Python Client Driver then HTMLTestRunner can be used to generate a Test Report. See HTMLTestRunner.

Test Reports for Ruby

- If RSpec framework is used for writing Selenium Test Cases in Ruby then its HTML report can be used to generate a test report. Refer to RSpec Report for more.

Note: If you are interested in a language independent log of what's going on, take a look at Selenium Server Logging

6.8 Adding Some Spice to Your Tests

Now we'll get to the whole reason for using Selenium RC, adding programming logic to your tests. It's the same as for any program. Program flow is controlled using condition statements and iteration. In addition you can report progress information using I/O. In this section we'll show some examples of how programming language constructs can be combined with Selenium to solve common testing problems.

You will find as you transition from the simple tests of the existence of page elements to tests of dynamic functionality involving multiple web-pages and varying data that you will require programming logic for verifying expected results. Basically, the Selenium-IDE does not support iteration and standard condition statements. You can do some conditions by embedding javascript in Selenese parameters, however iteration is impossible, and most conditions will be much easier in a programming language. In addition, you may need exception handling for error recovery. For these reasons and others, we have written this section to illustrate the use of common programming techniques to give you greater 'verification power' in your automated testing.

The examples in this section are written in C# and Java, although the code is simple and can be easily adapted to the other supported languages. If you have some basic knowledge of an object-oriented programming language you shouldn't have difficulty understanding this section.

6.8.1 Iteration

Iteration is one of the most common things people need to do in their tests. For example, you may want to to execute a search multiple times. Or, perhaps for verifying your test results you need to process a "result set" returned from a database.

Using the same Google search example we used earlier, let's check the Selenium search results. This test could use the Selenese:

open	/	
type	q	selenium rc
clickAndWait	btnG	
assertTextPresent	Results * for selenium rc	
type	q	selenium ide
clickAndWait	btnG	
assertTextPresent	Results * for selenium ide	
type	q	selenium grid
clickAndWait	btnG	
assertTextPresent	Results * for selenium grid	

The code has been repeated to run the same steps 3 times. But multiple copies of the same code is not good program practice because it's more work to maintain. By using a programming language, we can iterate over the search results for a more flexible and maintainable solution.

In C#:

```
// Collection of String values.
String[] arr = {"ide", "rc", "grid"};

// Execute loop for each String in array 'arr'.
foreach (String s in arr) {
    sel.open("/");
    sel.type("q", "selenium " +s);
    sel.click("btnG");
    sel.waitForPageToLoad("30000");
    assertTrue("Expected text: " +s+ " is missing on page."
    , sel.isTextPresent("Results * for selenium " + s));
 }
```

6.8.2 Condition Statements

To illustrate using conditions in tests we'll start with an example. A common problem encountered while running Selenium tests occurs when an expected element is not available on page. For example, when running the following line:

```
selenium.type("q", "selenium " +s);
```

If element 'q' is not on the page then an exception is thrown:

```
com.thoughtworks.selenium.SeleniumException: ERROR: Element q not found
```

This can cause your test to abort. For some tests that's what you want. But often that is not desirable as your test script has many other subsequent tests to perform.

A better approach is to first validate if the element is really present and then take alternatives when it it is not. Let's look at this using Java.

```
// If element is available on page then perform type operation.
if(selenium.isElementPresent("q")) {
    selenium.type("q", "Selenium rc");
} else {
    System.out.printf("Element: " +q+ " is not available on page.")
}
```

The advantage of this approach is to continue with test execution even if some UI elements are not available on page.

6.8.3 Executing JavaScript from Your Test

JavaScript comes very handy in exercising an application which is not directly supported by selenium. The **getEval** method of selenium API can be used to execute JavaScript from selenium RC.

Consider an application having check boxes with no static identifiers. In this case one could evaluate JavaScript from selenium RC to get ids of all check boxes and then exercise them.

```
public static String[] getAllCheckboxIds () {
            String script = "var inputId  = new Array();";// Create array in java scri
            script += "var cnt = 0;"; // Counter for check box ids.
            script += "var inputFields  = new Array();"; // Create array in java scrip
            script += "inputFields = window.document.getElementsByTagName('input');";
            script += "for(var i=0; i<inputFields.length; i++) {"; // Loop through the
            script += "if(inputFields[i].id !=null " +
            "&& inputFields[i].id !='undefined' " +
            "&& inputFields[i].getAttribute('type') == 'checkbox') {"; // If input fie
            script += "inputId[cnt]=inputFields[i].id ;" + // Save check box id to inp
            "cnt++;" + // increment the counter.
            "}" + // end of if.
            "}"; // end of for.
            script += "inputId.toString();" ;// Convert array in to string.
            String[] checkboxIds = selenium.getEval(script).split(","); // Split the s
            return checkboxIds;
 }
```

To count number of images on a page:

```
selenium.getEval("window.document.images.length;");
```

Remember to use window object in case of DOM expressions as by default selenium window is referred to, not the test window.

6.9 Server Options

When the server is launched, command line options can be used to change the default server behaviour.

Recall, the server is started by running the following.

```
$ java -jar selenium-server-standalone-<version-number>.jar
```

To see the list of options, run the server with the `-h` option.

```
$ java -jar selenium-server-standalone-<version-number> -h
```

You'll see a list of all the options you can use with the server and a brief description of each. The provided descriptions will not always be enough, so we've provided explanations for some of the more important options.

6.9.1 Proxy Configuration

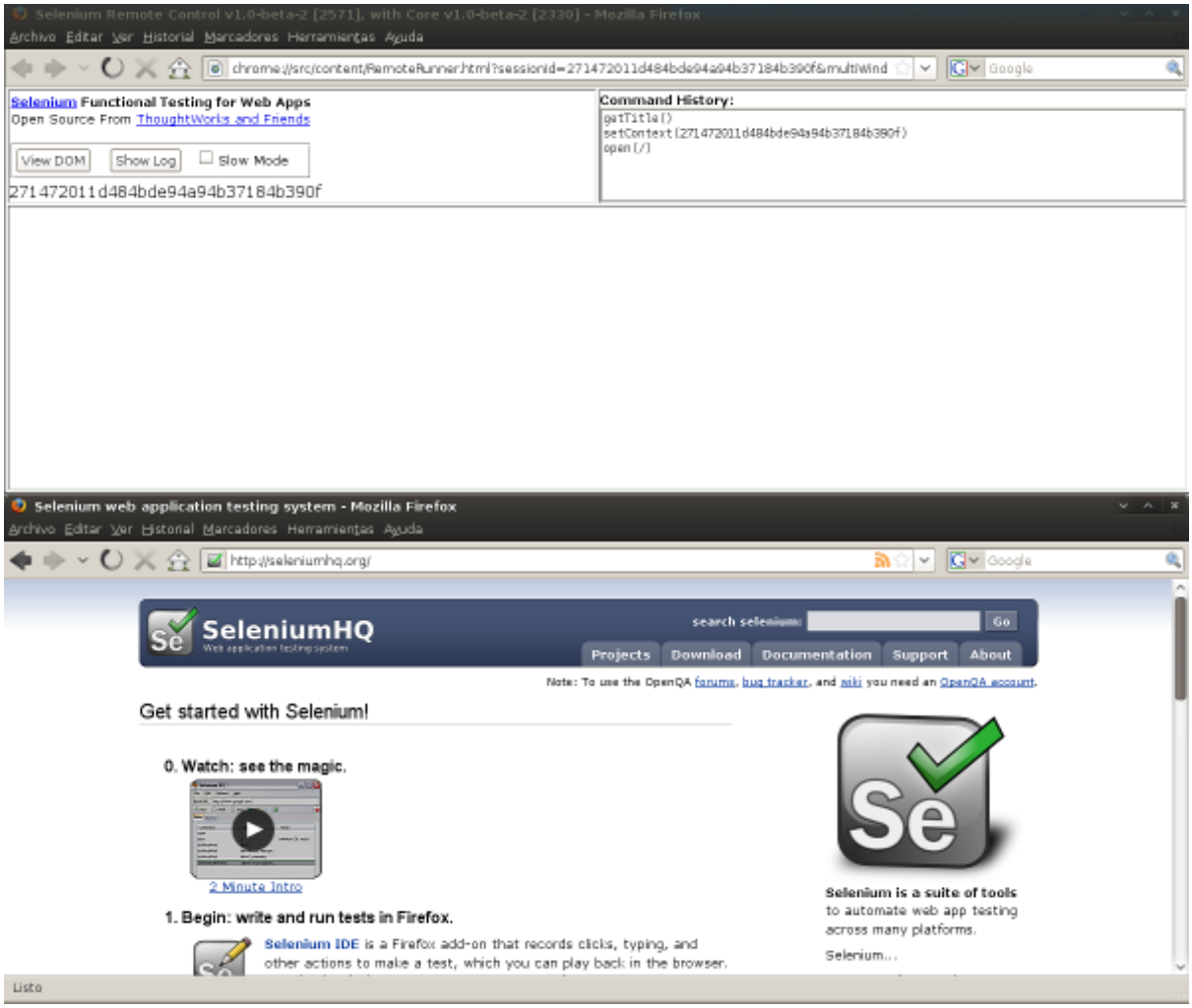
If your AUT is behind an HTTP proxy which requires authentication then you should configure http.proxyHost, http.proxyPort, http.proxyUser and http.proxyPassword using the following command.

```
$ java -jar selenium-server-standalone-<version-number>.jar -Dhttp.proxyHost=proxy.com -
```

6.9.2 Multi-Window Mode

If you are using Selenium 1.0 you can probably skip this section, since multiwindow mode is the default behavior. However, prior to version 1.0, Selenium by default ran the application under test in a sub frame as shown here.

Some applications didn't run correctly in a sub frame, and needed to be loaded into the top frame of the window. The multi-window mode option allowed the AUT to run in a separate window rather than in the default frame where it could then have the top frame it required.



For older versions of Selenium you must specify multiwindow mode explicitly with the following option:

```
-multiwindow
```

As of Selenium RC 1.0, if you want to run your test within a single frame (i.e. using the standard for earlier Selenium versions) you can state this to the Selenium Server using the option

```
-singlewindow
```

6.9.3 Specifying the Firefox Profile

Firefox will not run two instances simultaneously unless you specify a separate profile for each instance. Selenium RC 1.0 and later runs in a separate profile automatically, so if you are using Selenium 1.0, you can probably skip this section. However, if you're using an older version of Selenium or if you need to use a specific profile for your tests (such as adding an https certificate or having some addons installed), you will need to explicitly specify the profile.

First, to create a separate Firefox profile, follow this procedure. Open the Windows Start menu, select "Run", then type and enter one of the following:

```
firefox.exe -profilemanager
```

```
firefox.exe -P
```

Create the new profile using the dialog. Then when you run Selenium Server, tell it to use this new Firefox profile with the server command-line option *-firefoxProfileTemplate* and specify the path to the profile using it's filename and directory path.

```
-firefoxProfileTemplate "path to the profile"
```

> **Warning:** Be sure to put your profile in a new folder separate from the default!!! The Firefox profile manager tool will delete all files in a folder if you delete a profile, regardless of whether they are profile files or not.

More information about Firefox profiles can be found in Mozilla's Knowledge Base

6.9.4 Run Selenese Directly Within the Server Using -htmlSuite

You can run Selenese html files directly within the Selenium Server by passing the html file to the server's command line. For instance:

```
java -jar selenium-server-standalone-<version-number>.jar -htmlSuite "*firefox" "http://
```

This will automatically launch your HTML suite, run all the tests and save a nice HTML report with the results.

Note: When using this option, the server will start the tests and wait for a specified number of seconds for the test to complete; if the test doesn't complete within that amount of time, the command will exit with a non-zero exit code and no results file will be generated.

This command line is very long so be careful when you type it. Note this requires you to pass in an HTML Selenese suite, not a single test. Also be aware the -htmlSuite option is incompatible with `-interactive` You cannot run both at the same time.

6.9.5 Selenium Server Logging

Server-Side Logs

When launching selenium server the **-log** option can be used to record valuable debugging information reported by the Selenium Server to a text file.

```
java -jar selenium-server-standalone-<version-number>.jar -log selenium.log
```

This log file is more verbose than the standard console logs (it includes DEBUG level logging messages). The log file also includes the logger name, and the ID number of the thread that logged the message. For example:

```
20:44:25 DEBUG [12] org.openqa.selenium.server.SeleniumDriverResourceHandler -
Browser 465828/:top frame1 posted START NEW
```

The message format is

```
TIMESTAMP(HH:mm:ss) LEVEL [THREAD] LOGGER - MESSAGE
```

This message may be multiline.

Browser-Side Logs

JavaScript on the browser side (Selenium Core) also logs important messages; in many cases, these can be more useful to the end-user than the regular Selenium Server logs. To access browser-side logs, pass the **-browserSideLog** argument to the Selenium Server.

```
java -jar selenium-server-standalone-<version-number>.jar -browserSideLog
```

-browserSideLog must be combined with the **-log** argument, to log browserSideLogs (as well as all other DEBUG level logging messages) to a file.

6.10 Specifying the Path to a Specific Browser

You can specify to Selenium RC a path to a specific browser. This is useful if you have different versions of the same browser and you wish to use a specific one. Also, this is used to allow your tests to run against a browser not directly supported by Selenium RC. When specifying the run mode, use the *custom specifier followed by the full path to the browser's executable:

```
*custom <path to browser>
```

6.11 Selenium RC Architecture

Note: This topic tries to explain the technical implementation behind Selenium RC. It's not fundamental for a Selenium user to know this, but could be useful for understanding some of the problems you might find in the future.

To understand in detail how Selenium RC Server works and why it uses proxy injection and heightened privilege modes you must first understand the same origin policy.

6.11.1 The Same Origin Policy

The main restriction that Selenium faces is the Same Origin Policy. This security restriction is applied by every browser in the market and its objective is to ensure that a site's content will never be accessible by a script from another site. The Same Origin Policy dictates that any code loaded within the browser can only operate within that website's domain. It cannot perform functions on another website. So for example, if the browser loads JavaScript code when it loads www.mysite.com, it cannot run that loaded code against www.mysite2.com–even if that's another of your sites. If this were possible, a script placed on any website you open would be able to read information on your bank account if you had the account page opened on other tab. This is called XSS (Cross-site Scripting).

To work within this policy, Selenium-Core (and its JavaScript commands that make all the magic happen) must be placed in the same origin as the Application Under Test (same URL).

Historically, Selenium-Core was limited by this problem since it was implemented in JavaScript. Selenium RC is not, however, restricted by the Same Origin Policy. Its use of the Selenium Server as a proxy avoids this problem. It, essentially, tells the browser that the browser is working on a single "spoofed" website that the Server provides.

Note: You can find additional information about this topic on Wikipedia pages about Same Origin Policy and XSS.

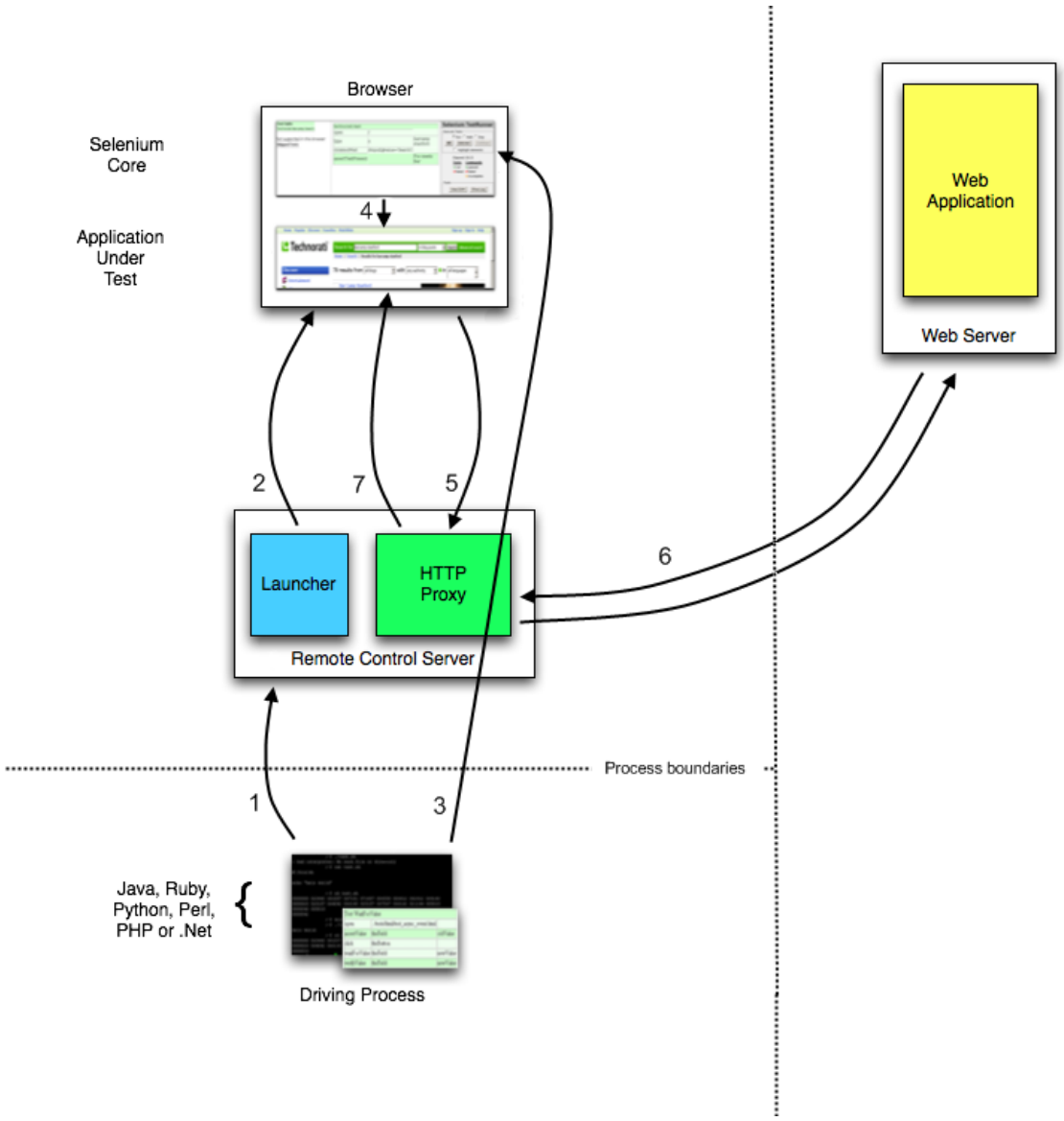
6.11.2 Proxy Injection

The first method Selenium used to avoid the The Same Origin Policy was Proxy Injection. In Proxy Injection Mode, the Selenium Server acts as a client-configured [1] **HTTP proxy** [2], that sits between the browser and the Application Under Test. It then masks the AUT under a fictional URL (embedding Selenium-Core and the set of tests and delivering them as if they were coming from the same origin).

Here is an architectural diagram.

[1] The proxy is a third person in the middle that passes the ball between the two parts. It acts as a "web server" that delivers the AUT to the browser. Being a proxy gives Selenium Server the capability of "lying" about the AUT's real URL.

[2] The browser is launched with a configuration profile that has set localhost:4444 as the HTTP proxy, this is why any HTTP request that the browser does will pass through Selenium server and the response will pass through it and not from the real server.



As a test suite starts in your favorite language, the following happens:

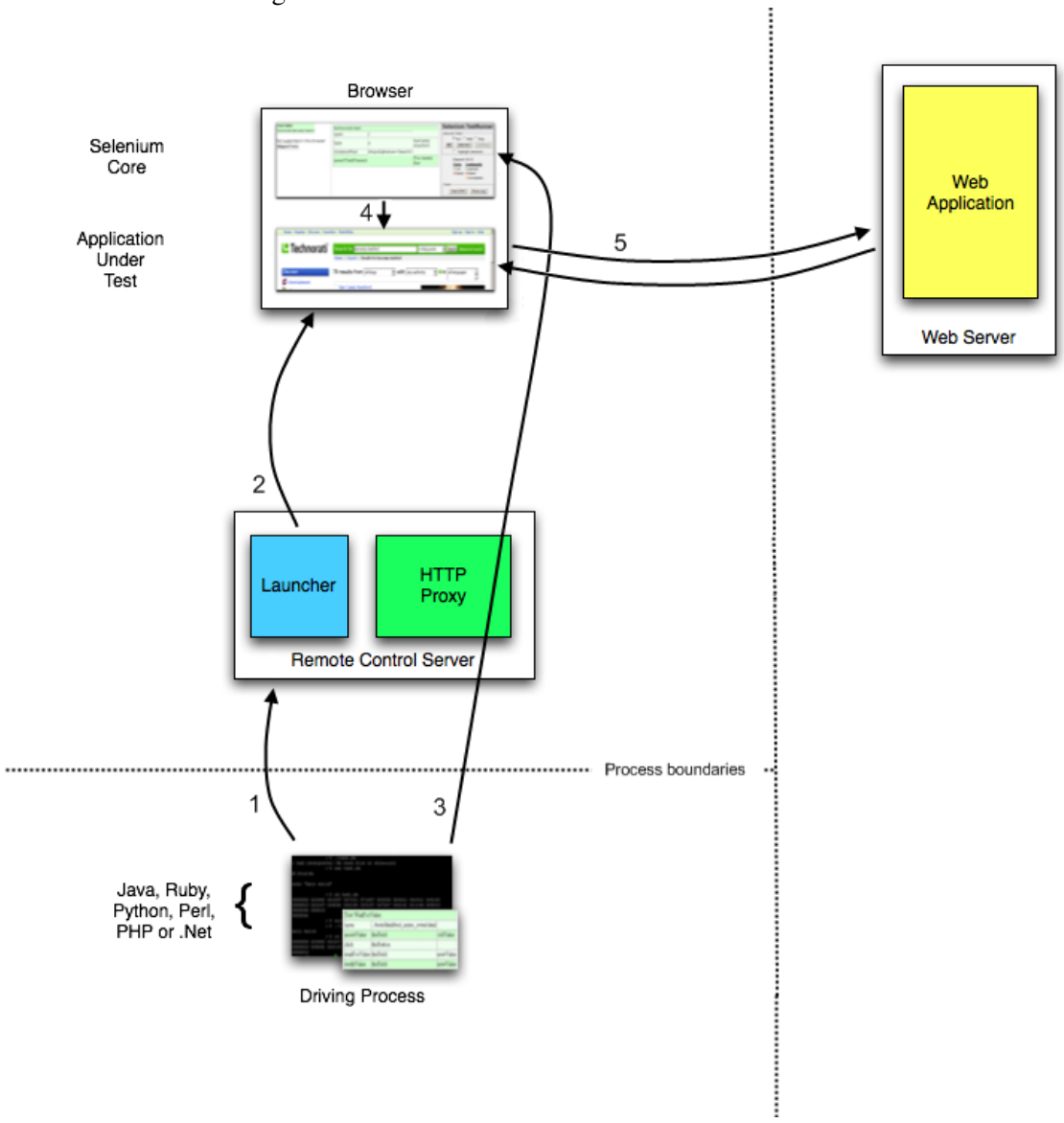
1. The client/driver establishes a connection with the selenium-RC server.
2. Selenium RC server launches a browser (or reuses an old one) with a URL that injects Selenium-Core's JavaScript into the browser-loaded web page.
3. The client-driver passes a Selenese command to the server.
4. The Server interprets the command and then triggers the corresponding JavaScript execution to execute that command within the browser.
5. Selenium-Core instructs the browser to act on that first instruction, typically opening a page of the AUT.
6. The browser receives the open request and asks for the website's content from the Selenium RC server (set as the HTTP proxy for the browser to use).
7. Selenium RC server communicates with the Web server asking for the page and once it receives it, it sends the page to the browser masking the origin to look like the page comes from the same server as Selenium-Core (this allows Selenium-Core to comply with the Same Origin Policy).

8. The browser receives the web page and renders it in the frame/window reserved for it.

6.11.3 Heightened Privileges Browsers

This workflow in this method is very similar to Proxy Injection but the main difference is that the browsers are launched in a special mode called *Heightened Privileges*, which allows websites to do things that are not commonly permitted (as doing XSS, or filling file upload inputs and pretty useful stuff for Selenium). By using these browser modes, Selenium Core is able to directly open the AUT and read/interact with its content without having to pass the whole AUT through the Selenium RC server.

Here is the architectural diagram.



As a test suite starts in your favorite language, the following happens:

1. The client/driver establishes a connection with the selenium-RC server.
2. Selenium RC server launches a browser (or reuses an old one) with a URL that will load Selenium-Core in the web page.
3. Selenium-Core gets the first instruction from the client/driver (via another HTTP request made to the Selenium RC Server).

4. Selenium-Core acts on that first instruction, typically opening a page of the AUT.

5. The browser receives the open request and asks the Web Server for the page. Once the browser receives the web page, renders it in the frame/window reserved for it.

6.12 Handling HTTPS and Security Popups

Many applications switch from using HTTP to HTTPS when they need to send encrypted information such as passwords or credit card information. This is common with many of today's web applications. Selenium RC supports this.

To ensure the HTTPS site is genuine, the browser will need a security certificate. Otherwise, when the browser accesses the AUT using HTTPS, it will assume that application is not 'trusted'. When this occurs the browser displays security popups, and these popups cannot be closed using Selenium RC.

When dealing with HTTPS in a Selenium RC test, you must use a run mode that supports this and handles the security certificate for you. You specify the run mode when your test program initializes Selenium.

In Selenium RC 1.0 beta 2 and later use *firefox or *iexplore for the run mode. In earlier versions, including Selenium RC 1.0 beta 1, use *chrome or *iehta, for the run mode. Using these run modes, you will not need to install any special security certificates; Selenium RC will handle it for you.

In version 1.0 the run modes *firefox or *iexplore are recommended. However, there are additional run modes of *iexploreproxy and *firefoxproxy. These are provided for backwards compatibility only, and should not be used unless required by legacy test programs. Their use will present limitations with security certificate handling and with the running of multiple windows if your application opens additional browser windows.

In earlier versions of Selenium RC, *chrome or *iehta were the run modes that supported HTTPS and the handling of security popups. These were considered 'experimental modes although they became quite stable and many people used them. If you are using Selenium 1.0 you do not need, and should not use, these older run modes.

6.12.1 Security Certificates Explained

Normally, your browser will trust the application you are testing by installing a security certificate which you already own. You can check this in your browser's options or Internet properties (if you don't know your AUT's security certificate ask your system administrator). When Selenium loads your browser it injects code to intercept messages between the browser and the server. The browser now thinks untrusted software is trying to look like your application. It responds by alerting you with popup messages.

To get around this, Selenium RC, (again when using a run mode that support this) will install its own security certificate, temporarily, to your client machine in a place where the browser can access it. This tricks the browser into thinking it's accessing a site different from your AUT and effectively suppresses the popups.

Another method used with earlier versions of Selenium was to install the Cybervillians security certificate provided with your Selenium installation. Most users should no longer need to do this however; if you are running Selenium RC in proxy injection mode, you may need to explicitly install this security certificate.

6.13 Supporting Additional Browsers and Browser Configurations

The Selenium API supports running against multiple browsers in addition to Internet Explorer and Mozilla Firefox. See the SeleniumHQ.org website for supported browsers. In addition, when a browser is not directly supported, you may still run your Selenium tests against a browser of your choosing by using the "*custom" run-mode (i.e. in place of *firefox or *iexplore) when your test application starts the browser. With this, you pass in the path to the browsers executable within the API call. This can also be done from the Server in interactive mode.

```
cmd=getNewBrowserSession&1=*custom c:\Program Files\Mozilla Firefox\MyBrowser.exe&2=h
```

6.13.1 Running Tests with Different Browser Configurations

Normally Selenium RC automatically configures the browser, but if you launch the browser using the "*custom" run mode, you can force Selenium RC to launch the browser as-is, without using an automatic configuration.

For example, you can launch Firefox with a custom configuration like this:

```
cmd=getNewBrowserSession&1=*custom c:\Program Files\Mozilla Firefox\firefox.exe&2=htt
```

Note that when launching the browser this way, you must manually configure the browser to use the Selenium Server as a proxy. Normally this just means opening your browser preferences and specifying "localhost:4444" as an HTTP proxy, but instructions for this can differ radically from browser to browser. Consult your browser's documentation for details.

Be aware that Mozilla browsers can vary in how they start and stop. One may need to set the MOZ_NO_REMOTE environment variable to make Mozilla browsers behave a little more predictably. Unix users should avoid launching the browser using a shell script; it's generally better to use the binary executable (e.g. firefox-bin) directly.

6.14 Troubleshooting Common Problems

When getting started with Selenium RC there's a few potential problems that are commonly encountered. We present them along with their solutions here.

6.14.1 Unable to Connect to Server

When your test program cannot connect to the Selenium Server, an exception will be thrown in your test program. It should display this message or a similar one:

```
"Unable to connect to remote server....Inner Exception Message:
    No connection could be made because the target machine actively
    refused it...."

    (using .NET and XP Service Pack 2)
```

If you see a message like this, be sure you started the Selenium Server. If so, then there is a problem with the connectivity between the Selenium Client Library and the Selenium Server.

When starting with Selenium RC, most people begin by running thier test program (with a Selenium Client Library) and the Selenium Server on the same machine. To do this use "localhost" as your connection parameter. We recommend beginning this way since it reduces the influence of potential networking problems which you're getting started. Assuming your operating system has typical networking and TCP/IP settings you should have little difficulty. In truth, many people choose to run the tests this way.

If, however, you do want to run Selenium Server on a remote machine, the connectivity should be fine assuming you have valid TCP/IP connectivity between the two machines.

If you have difficulty connecting, you can use common networking tools like *ping*, *telnet*, *ifconfig(Unix)/ipconfig* (Windows), etc to ensure you have a valid network connection. If unfamilar with these, your system administrator can assist you.

6.14.2 Unable to Load the Browser

Ok, not a friendly error message, sorry, but if the Selenium Server cannot load the browser you will likley see this error.

```
(500) Internal Server Error
```

This could be caused by

- Firefox (prior to Selenium 1.0) cannot start because the browser is already open and you did not specify a separate profile. See the section on Firefox profiles under Server Options.
- The run mode you're using doesn't match any browser on your machine. Check the parameters you passed to Selenium when you program opens the browser.
- You specified the path to the browser explicitly (using "*custom"–see above) but the path is incorrect. Check to be sure the path is correct. Also check the user group to be sure there are no known issues with your browser and the "*custom" parameters.

6.14.3 Selenium Cannot Find the AUT

If your test program starts the browser successfully, but the browser doesn't display the website you're testing, the most likely cause is your test program is not using the correct URL.

This can easily happen. When you use Selenium-IDE to export your script, it inserts a dummy URL. You must manually change the URL to the correct one for your application to be tested.

6.14.4 Firefox Refused Shutdown While Preparing a Profile

This most often occurs when your run your Selenium RC test program against Firefox, but you already have a Firefox browser session running and, you didn't specify a separate profile when you started the Selenium Server. The error from the test program looks like this:

```
Error:  java.lang.RuntimeException: Firefox refused shutdown while
preparing a profile
```

Here's the complete error message from the server:

```
16:20:03.919 INFO - Preparing Firefox profile...
16:20:27.822 WARN - GET /selenium-server/driver/?cmd=getNewBrowserSession&1=*fir
efox&2=http%3a%2f%2fsage-webapp1.qa.idc.com HTTP/1.1
java.lang.RuntimeException: Firefox refused shutdown while preparing a profile
        at org.openqa.selenium.server.browserlaunchers.FirefoxCustomProfileLaunc
her.waitForFullProfileToBeCreated(FirefoxCustomProfileLauncher.java:277)
.........................
Caused by: org.openqa.selenium.server.browserlaunchers.FirefoxCustomProfileLaunc
her$FileLockRemainedException: Lock file still present! C:\DOCUME~1\jsvec\LOCALS
~1\Temp\customProfileDir203138\parent.lock
```

To resolve this, see the section on Specifying a Separate Firefox Profile

6.14.5 Versioning Problems

Make sure your version of Selenium supports the version of your browser. For example, Selenium RC 0.92 does not support Firefox 3. At times you may be lucky (I was). But don't forget to check which browser versions are supported by the version of Selenium you are using. When in doubt, use the latest release version of Selenium with the most widely used version of your browser.

6.14.6 Error message: "(Unsupported major.minor version 49.0)" while starting server

This error says you're not using a correct version of Java. The Selenium Server requires Java 1.5 or higher.

To check double-check your java version, run this from the command line.

```
java -version
```

You should see a message showing the Java version.

```
java version "1.5.0_07"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_07-b03)
Java HotSpot(TM) Client VM (build 1.5.0_07-b03, mixed mode)
```

If you see a lower version number, you may need to update the JRE, or you may simply need to add it to your PATH environment variable.

6.14.7 404 error when running the getNewBrowserSession command

If you're getting a 404 error while attempting to open a page on "http://www.google.com/selenium-server/", then it must be because the Selenium Server was not correctly configured as a proxy. The "selenium-server" directory doesn't exist on google.com; it only appears to exist when the proxy is properly configured. Proxy Configuration highly depends on how the browser is launched with *firefox, *iexplore, *opera, or *custom.

- *iexplore: If the browser is launched using *iexplore, you could be having a problem with Internet Explorer's proxy settings. Selenium Server attempts To configure the global proxy settings in the Internet Options Control Panel. You must make sure that those are correctly configured when Selenium Server launches the browser. Try

looking at your Internet Options control panel. Click on the "Connections" tab and click on "LAN Settings".

  - If you need to use a proxy to access the application you want to test, you'll need to start Selenium Server with "-Dhttp.proxyHost"; see the Proxy Configuration for more details.
  - You may also try configuring your proxy manually and then launching the browser with *custom, or with *iehta browser launcher.

- ***custom: When using *custom you must configure the proxy correctly(manually),** otherwise you'll get a 404 error. Double-check that you've configured your proxy settings correctly. To check whether you've configured the proxy correctly is to attempt to intentionally configure the browser incorrectly. Try configuring the browser to use the wrong proxy server hostname, or the wrong port. If you had successfully configured the browser's proxy settings incorrectly, then the browser will be unable to connect to the Internet, which is one way to make sure that one is adjusting the relevant settings.
- For other browsers (*firefox, *opera) we automatically hard-code the proxy for you, and so ther are no known issues with this functionality. If you're encountering 404 errors and have followed this user guide carefully post your results to user group for some help from the user community.

6.14.8 Permission Denied Error

The most common reason for this error is that your session is attempting to violate the same-origin policy by crossing domain boundaries (e.g., accesses a page from http://domain1 and then accesses a page from http://domain2) or switching protocols (moving from http://domainX to https://domainX).

This error can also occur when JavaScript attempts to find UI objects which are not yet available (before the page has completely loaded), or are no longer available (after the page has started to be unloaded). This is most typically encountered with AJAX pages which are working with sections of a page or subframes that load and/or reload independently of the larger page.

This error can be intermittent. Often it is impossible to reproduce the problem with a debugger because the trouble stems from race conditions which are not reproducible when the debugger's overhead is added to the system. Permission issues are covered in some detail in the tutorial. Read the section about the The Same Origin Policy, Proxy Injection carefully.

6.14.9 Handling Browser Popup Windows

There are several kinds of "Popups" that you can get during a Selenium test. You may not be able to close these popups by running selenium commands if they are initiated by the browser and not your AUT. You may need to know how to manage these. Each type of popup needs to be addressed differently.

- HTTP basic authentication dialogs: These dialogs prompt for a username/password to login to the site. To login to a site that requires HTTP basic authentication, use a username and password in the URL, as described in RFC 1738, like this: open("http://myusername:myuserpassword@myexample.com/blah/blah/blah").
- SSL certificate warnings: Selenium RC automatically attempts to spoof SSL certificates when it is enabled as a proxy; see more on this in the section on HTTPS. If your browser is configured correctly, you should never see SSL certificate warnings, but you may need to configure your

browser to trust our dangerous "CyberVillains" SSL certificate authority. Again, refer to the HTTPS section for how to do this.

- modal JavaScript alert/confirmation/prompt dialogs: Selenium tries to conceal those dialogs from you (by replacing window.alert, window.confirm and window.prompt) so they won't stop the execution of your page. If you're seeing an alert pop-up, it's probably because it fired during the page load process, which is usually too early for us to protect the page. Selenese contains commands for asserting or verifying alert and confirmation popups. See the sections on these topics in Chapter 4.

6.14.10 On Linux, why isn't my Firefox browser session closing?

On Unix/Linux you must invoke "firefox-bin" directly, so make sure that executable is on the path. If executing Firefox through a shell script, when it comes time to kill the browser Selenium RC will kill the shell script, leaving the browser running. You can specify the path to firefox-bin directly, like this.

```
cmd=getNewBrowserSession&1=*firefox /usr/local/firefox/firefox-bin&2=http://www.google.c
```

6.14.11 Firefox *chrome doesn't work with custom profile

Check Firefox profile folder -> prefs.js -> user_pref("browser.startup.page", 0); Comment this line like this: "//user_pref("browser.startup.page", 0);" and try again.

6.14.12 Is it ok to load a custom pop-up as the parent page is loading (i.e., before the parent page's javascript window.onload() function runs)?

No. Selenium relies on interceptors to determine window names as they are being loaded. These interceptors work best in catching new windows if the windows are loaded AFTER the onload() function. Selenium may not recognize windows loaded before the onload function.

6.14.13 Problems With Verify Commands

If you export your tests from Selenium-IDE, you may find yourself getting empty verify strings from your tests (depending on the programming language used).

Note: This section is not yet developed.

6.14.14 Safari and MultiWindow Mode

Note: This section is not yet developed.

6.14.15 Firefox on Linux

On Unix/Linux, versions of Selenium before 1.0 needed to invoke "firefox-bin" directly, so if you are using a previous version, make sure that the real executable is on the path.

On most Linux distributions, the real *firefox-bin* is located on:

```
/usr/lib/firefox-x.x.x/
```

Where the x.x.x is the version number you currently have. So, to add that path to the user's path. you will have to add the following to your .bashrc file:

```
export PATH="$PATH:/usr/lib/firefox-x.x.x/"
```

If necessary, you can specify the path to firefox-bin directly in your test, like this:

```
" *firefox /usr/lib/firefox-x.x.x/firefox-bin "
```

6.14.16 IE and Style Attributes

If you are running your tests on Internet Explorer and you cannot locate elements using their *style* attribute. For example:

```
//td[@style="background-color:yellow"]
```

This would work perfectly in Firefox, Opera or Safari but not with IE. IE interprets the keys in *@style* as uppercase. So, even if the source code is in lowercase, you should use:

```
//td[@style="BACKGROUND-COLOR:yellow"]
```

This is a problem if your test is intended to work on multiple browsers, but you can easily code your test to detect the situation and try the alternative locator that only works in IE.

6.14.17 Where can I Ask Questions that Aren't Answered Here?

Try our user group

CHAPTER

SEVEN

TEST DESIGN CONSIDERATIONS

7.1 Introducing Test Design

We've provided in this chapter information that will be useful to both, those new to test automation and for the experienced QA professional. Here we describe the most common types of automated tests. We also describe 'design patterns' commonly used in test automation for improving the maintenance and extensibily of your automation suite. The more experienced reader will find these interesting if not already using these techniques.

7.2 Types of Tests

What parts of your application should you test? That depends on aspects of your project: user expectations, time allowed for the project, priorities set by the project manager and so on. Once the project boundaries are defined though, you, the tester, will certainly make many decisions on what to test.

We've created a few terms here for the purpose of categorizing the types of test you may perform on your web application. These terms are by no means standard, although the concepts we present here are typical for web-application testing.

7.2.1 Testing Static Content

The simplest type of test, a *content test*, is a simple test for the existence of a static, non-changing, UI element. For instance

- Does each page have its expected page title? This can be used to verify your test found an expected page after following a link.
- Does the application's home page contain an image expected to be at the top of the page?
- Does each page of the website contain a footer area with links to the company contact page, privacy policy, and trademarks information?
- Does each page begin with heading text using the <h1> tag? And, does each page have the correct text within that header?

You may or may not need content tests. If your page content is not likely to be affected then it may be more efficient to test page content manually. If, for example, your application involves files being moved to different locations, content tests may prove valuable.

7.2.2 Testing Links

A frequent source of errors for web-sites is broken links or missing pages behind links. Testing involves clicking each link and verifying the expected page. If static links are infrequently changed then manual testing may be sufficient. However if your web designers frequently alter links, or if files are occasionally relocated, link tests should be automated.

7.2.3 Function Tests

These would be tests of a specific function within your application, requiring some type of user input, and returning some type of results. Often a function test will involve multiple pages with a form-based input page containing a collection of input fields, Submit and Cancel operations, and one or more response pages. User input can be via text-input fields, check boxes, drop-down lists, or any other browser-supported input.

Function tests are often the most complex tests you'll automate, but are usually the most important. Typical tests can be for login, registration to the site, user account operations, account settings changes, complex data retrieval operations, among others. Function tests typically mirror the user-scenarios used to specify the features and design or your application.

7.2.4 Testing Dynamic Elements

Often a web page element has a unique identifier used to uniquely locate that element within the page. Usually these are implemented using the html tag's 'id' attribute or it's 'name' attribute. These names can be a static, i.e unchanging, string constant. They can also be dynamically generated values that vary each instance of the page. For example, some web servers might name a displayed document doc3861 one instance of a page, and 'doc6148' on a different instance of the page depending on what 'document' the user was retrieving. This means your test script which is verify that a document exists may not have a consistent identifier to user for locating that document. Often, dynamic elements with varying identifiers are on some type of result page based on a user action. Thing though certainly depends on the function of the web application.

Here's an example.

```
<input type="checkbox" value="true" id="addForm:_ID74:_ID75:0:_ID79:0:
    checkBox" />
```

This shows an HTML tag for a check box. Its ID (addForm:_ID74:_ID75:0:_ID79:0:checkBox) is a dynamically generated value. The next time the same page is opened it will likely be a different value.

7.2.5 Ajax Tests

Ajax is a technology which supports dynamically changing user interface elements which can dynamically change without the browser having to reload the page, such as animation, RSS feeds, and real-time data updates among others. There's a countless ways Ajax can be used to update elements on a web page. But, the easy way to think of this is that in Ajax-driven applications, data can retrieved from the application server and then displayed on the page without reloading the entire page. Only a portion of the page, or strictly the element itself is reloaded.

7.3 Validating Results

7.3.1 Assert vs. Verify

When should you use an assert command and when should you use a verify command? This is up to you. The difference is in what you want to happen when the check fails. Do you want your test to terminate, or to continue and simply record that the check failed?

Here's the trade-off. If you use an assert, the test will stop at that point and not run any subsequent checks. Sometimes, perhaps often, that is what you want. If the test fails you will immediately know the test did not pass. Test engines such as TestNG and JUnit have plugins for commonly used development environments (Chap 5) which conveniently flag these tests as failed tests. The advantage: you have an immediate visual of whether the checks passed. The disadvantage: when a check does fail, there are other checks which were never performed, so you have no information on their status.

In contrast, verify commands will not terminate the test. If your test uses only verify commands you are guaranteed (assuming no unexpected exceptions) the test will run to completion whether the checks find defects or not. The disadvantage: you have to do more work to examine your test results. That is, you won't get feedback from TestNG or JUnit. You will need to look at the results of a console printout or a log output. And you will need to take the time to look through this output every time you run your test. If you are running hundreds of tests, each with it's own log, this will be time-consuming, and the immediate feedback of asserts will be more appropriate. Asserts are more commonly used then verifys due to their immediate feedback.

7.3.2 Trade-offs: *assertTextPresent*, *assertElementPresent*, *assertText*

You should now be familiar with these commands, and the mechanics of using them. If not, please refer to Chapter 3 first. When constructing your tests, you will need to decide

- Do I only check that the text exists on the page? (*verify/assertTextPresent*)
- Do I only check that the HTML element exists on the page? That is, the text, image, or other content is not to be checked, only the HTML tag is what is relevant. (*verify/assertElementPresent*)
- Must I test both, the element and it's text content? (*verify/assertText*)

There is no right answer. It depends on the requirements for your test. Which, of course, depend on the requirements for the application you're testing. If in doubt, use *assertText* since this is the strictest type of checkpoint. You can always change it later but at least you won't be missing any potential failures.

Verify/assertText is the *most specific test* type. This can fail if either the HTML element (tag) OR the text is not what your test is expecting. Perhaps your web-designers are frequently changing the page and you don't want your test to fail every time they do this because the changes themselves are expected periodically. However, assume you still need to check that *something* is on the page, say a paragraph, or heading text, or an image. In this case you can use *verify/assertElementPresent*. It will ensure that a particular type of element exists (and if using XPath can ensure it exists relative to other objects within the page). But you don't care what the content is. You only care that a specific element, say, an image, is at a specific location.

Getting a feel for these types of decisions will come with time and a little experience. They are easy concepts, and easy to change in your test.

7.4 Location Strategies

7.4.1 Choosing a Location Strategy

There are multiple ways of selecting an object on a page. But what are the trade offs of each of these locator types? Recall we can locate an object using

- the element's ID
- the element's name attribute
- an XPath statement
- by a links text
- document object model (DOM)

Using an element ID or name locator is the most efficient in terms of test performance, and also makes your test code more readable, assuming the ID or name within the page source is well-named. XPath statements take longer to process since the browser must run it's XPath processor. Xpath has been known to be especially slow in Internet Explorer version 7. Locating via a link's text is often convenient and performs well. This technique is specific to links though. Also, if the link text is likely to change frequently, locating by the <a> element would be the better choice.

Sometimes though, you must use an XPath locator. If the page source does not have an ID or name attribute you may have no choice but to use an XPath locator. (DOM locators are no longer commonly used since Xpath can do everything they can and more. DOM locators are available simply to support legacy tests.)

There is an advantage to using XPath that locating via ID or name attributes do not have. With XPath (and DOM) you can locate an object with respect to another object on the page. For example, if there is a link that must occur within the second paragraph within a section, you can use XPath to specify this. With ID and name locators, you can only specify that they occur on the page that is, somewhere on the page. If you must test that an image displaying the company logo appears at the top of the page within a header section XPath may be the better locator.

7.4.2 Locating Dynamic Elements

As was described earlier in the section on types of tests, a dynamic element is a page element whose identifer varies with each instance of the page. For example,

```
<a class="button" id="adminHomeForm" onclick="return oamSubmitForm('adminHomeForm',
    'adminHomeForm:_ID38');" href="#">View Archived Allocation Events</a>
```

This HTML anchor tag defines a button with an ID attribute of "adminHomeForm". It's a fairly complex anchor tag when compared to most HTML tags, but it is still a static tag. The HTML will be the same each time this page is loaded in the browser. Its ID remains constant with all instances of this page. That is, when this page is displayed, this UI element will always have this Identifier. So, for your test script to click this button you simply need to use the following selenium command.

```
click       adminHomeForm
```

Or, in Selenium 1.0

```
selenium.click( "adminHomeForm" );
```

Your application, however, may generate HTML dynamically where the identifier varies on different instances of the webpage. For instance, HTML for a dynamic page element might look like this.

```
<input type= "checkbox"  value= "true"  id= "addForm:_ID74:_ID75:0:_ID79:0:checkBox"
    name= "addForm:_ID74:_ID75:0:_ID79:0:checkBox" />
```

This defines a checkbox. Its ID and name attributes (both addForm:_ID74:_ID75:0:_ID79:0:checkBox) are dynamically generated values. In this case, using a standard locator would look something like the following.

```
click       addForm:_ID74:_ID75:0:_ID79:0:checkBox
```

Or, again in Selenium-RC

```
selenium.click("addForm:_ID74:_ID75:0:_ID79:0:checkBox);
```

Given the dynamically generated Identifier, this approach would not work. The next time this page is loaded the Identifier will be a different value from the one used in the Selenium command and therefore, will not be found. The click operation will fail with an "element not found" error.

To correct this, a simple solution would be to just use an XPath locator rather than trying to use an ID locator. So, for the checkbox you can simply use

```
click       //input
```

Or, if it is not the first input element on the page (which it likely is not) try a more detailed XPath statement.

```
click       //input[3]
```

Or

```
click       //div/p[2]/input[3]
```

If however, you do need to use the ID to locate the element, a different solution is needed. You can capture this ID from the website before you use it in a Selenium command. It can be done like this.

```
String[] checkboxids  = selenium.getAllFields(); // Collect all input IDs on page.
             for(String checkboxid:checkboxids) {
                    if(checkboxid.contains( "addForm" )) {
                selenium.click(expectedText);
            }
             }
```

This approach will work if there is only one check box whose ID has the text 'expectedText' appended to it.

7.4.3 Locating Ajax Elements

As was presented in the Test Types subsection above, a page element implemented with Ajax is an element that can be dynamically refreshed without having to refresh the entire page. The best way to locate and verify an Ajax element is to use the Selenium 2.0 WebDriver API. It was specifically designed to address testing of Ajax elements where Selenium 1 has some limitations.

In Selenim 2.0 you use the waitFor() method to wait for a page element to become available. The parameter is a By object which is how WebDriver implements locators. This is explained in detail in the WebDriver chapters.

To do this with Selenium 1.0 (Selenium-RC) a bit more coding is involved, but it isn't difficult. The approach is to check for the element, if it's not available wait for a predefined period and then again recheck it. This is then executed with a loop with a predetermined time-out terminating the loop if the element isn't found.

Let's consider a page which brings a link (link=ajaxLink) on click of a button on page (without refreshing the page) This could be handled by Selenium using a *for* loop.

```
// Loop initialization.
for (int second = 0;; second++) {

     // If loop is reached 60 seconds then break the loop.
     if (second >= 60) break;

     // Search for element "link=ajaxLink" and if available then break loop.
     try { if (selenium.isElementPresent("link=ajaxLink")) break; } catch (Exception e)

     // Pause for 1 second.
     Thread.sleep(1000);

}
```

This certainly isn't the only solution. Ajax is a common topic in the user forum and we recommend searching previous discussions to see what others have done.

7.5 Wrapping Selenium Calls

As with any programming, you will want to use utility functions to handle code that would otherwise be duplicated throughout your tests. One way to prevent this is to wrap frequently used selenium calls with functions or class methods of your own design. For example, many tests will frequently click on a page element and wait for page to load multiple times within a test.

```
selenium.click(elementLocator);
selenium.waitForPageToLoad(waitPeriod);
```

Instead of duplicating this code you could write a wrapper method that performs both functions.

```
/**
 * Clicks and Waits for page to load.
 *
 * param elementLocator
 * param waitPeriod
 */
```

```
public void clickAndWait(String elementLocator, String waitPeriod) {
        selenium.click(elementLocator);
        selenium.waitForPageToLoad(waitPeriod);
}
```

7.5.1 'Safe Operations' for Element Presence

Another common usage of wrapping selenium methods is to check for presence of an element on page before carrying out some operation. This is sometimes called a 'safe operation'. For instance, the following method could be used to implement a safe operation that depends on an expected element being present.

```
/**
 * Selenum-RC -- Clicks on element only if it is available on page.
 *
 * param elementLocator
 */
public void safeClick(String elementLocator) {
        if(selenium.isElementPresent(elementLocator)) {
                selenium.click(elementLocator);
        } else {
                // Using the TestNG API for logging
                Reporter.log( "Element: " +elementLocator+ ", is not available on page
                                +selenium.getLocation());
        }
}
```

This example uses the Selenium 1 API but Selenium 2 also supports this.

```
/**
 * Selenium-WebDriver -- Clicks on element only if it is available on page.
 *
 * param elementLocator
 */
public void safeClick(String elementLocator) {
        WebElement webElement = getDriver().findElement(By.XXXX(elementLocator));
        if(webElement != null) {
                selenium.click(elementLocator);
        } else {
                // Using the TestNG API for logging
                Reporter.log( "Element: " +elementLocator+ ", is not available on page
                                + getDriver().getUrl());
        }
}
```

In this second example 'XXXX' is simply a placeholder for one of the multiple location methods that can be called here.

Using safe methods is up to the test developer's discretion. Hence, if test execution is to be continued, even in the wake of missing elements on the page, then safe methods could be used, while posting a message to a log about the missing element. This, essentially, implements a 'verify' with a reporting mechanism as opposed to an abortive assert. But if element must be available on page in order to be able to carry out further operations (i.e. login button on home page of a portal) then this safe method technique should not be used.

7.6 UI Mapping

A UI map is a mechanism that stores all the locators for a test suite in one place for easy modification when identifiers or paths to UI elements change in the AUT. The test script then uses the UI Map for locating the elements to be tested. Basically, a UI map is a repository of test script objects that correspond to UI elements of the application being tested.

What makes a UI map helpful? Its primary purpose for making test script management much easier. When a locator needs to be edited, there is a central location for easily finding that object, rather than having to search through test script code. Also, it allows changing the Identifier in a single place, rather than having to make the change in multiple places within a test script, or for that matter, in multiple test scripts.

To summarize, a UI map has two significant advantages.

- Using a centralized location for UI objects instead of having them scattered throughout the script. This makes script maintenance more efficient.
- Cryptic HTML Identifiers and names can be given more human-readable names improving the readability of test scripts.

Consider the following, difficult to understand, example (in java).

```
public void testNew() throws Exception {
              selenium.open("http://www.test.com");
              selenium.type("loginForm:tbUsername", "xxxxxxxx");
              selenium.click("loginForm:btnLogin");
              selenium.click("adminHomeForm:_activitynew");
              selenium.waitForPageToLoad("30000");
              selenium.click("addEditEventForm:_IDcancel");
              selenium.waitForPageToLoad("30000");
              selenium.click("adminHomeForm:_activityold");
              selenium.waitForPageToLoad("30000");
}
```

This script would be hard to follow for anyone not familiar with the AUT's page source. Even regular users of the application might have difficulty understanding what thus script does. A better script could be:

```
public void testNew() throws Exception {
              selenium.open("http://www.test.com");
              selenium.type(admin.username, "xxxxxxxx");
              selenium.click(admin.loginbutton);
              selenium.click(admin.events.createnewevent);
              selenium.waitForPageToLoad("30000");
              selenium.click(admin.events.cancel);
              selenium.waitForPageToLoad("30000");
              selenium.click(admin.events.viewoldevents);
              selenium.waitForPageToLoad("30000");
}
```

Now, using some comments and whitespace along with the UI Map identifiers makes a very readable script.

```
public void testNew() throws Exception {

            // Open app url.
            selenium.open( "http://www.test.com" );

            // Provide admin username.
            selenium.type(admin.username,  "xxxxxxxx" );

            // Click on Login button.
            selenium.click(admin.loginbutton);

            // Click on Create New Event button.
            selenium.click(admin.events.createnewevent);
            selenium.waitForPageToLoad( "30000" );

            // Click on Cancel button.
            selenium.click(admin.events.cancel);
            selenium.waitForPageToLoad( "30000" );

            // Click on View Old Events button.
            selenium.click(admin.events.viewoldevents);
            selenium.waitForPageToLoad( "30000" );
}
```

There are various ways a UI Map can be implemented. One could create a class or struct which only stores public String variables each storing a locator. Alternatively, a text file storing key value pairs could be used. In Java, a properties file containing key/value pairs is probably best method.

Consider a property file *prop.properties* which assigns as 'aliases' reader-friendly identifiers for UI elements from the previous example.

```
admin.username = loginForm:tbUsername
admin.loginbutton = loginForm:btnLogin
admin.events.createnewevent = adminHomeForm:_activitynew
admin.events.cancel = addEditEventForm:_IDcancel
admin.events.viewoldevents = adminHomeForm:_activityold
```

The locators will still refer to html objects, but we have introduced a layer of abstraction between the test script and the UI elements. Values are read from the properties file and used in the Test Class to implement the UI Map. For more on Java properties files refer to the following link.

7.7 Page Object Design Pattern

Page Object is a Design Pattern which has become popular in test automation for enhancing test maintenance and reducing code duplication. A page object is an object-oriented class that serves as an interface to a page of your AUT. The tests then use the methods of this page object class whenever they need to interact with that page of the UI. The benefit is that if the UI changes for the page, the tests themselves don't need to change, only the code within the page object needs to change. Subsequently all changes to support that new UI are located in one place.

The Page Object Design Pattern provides the following advantages.

1. There is clean separation between test code and page specific code such as locators (or their use if you're using a UI map) and layout.

2. There is single repository for the services or operations offered by the page rather than having these services scattered through out the tests.

In both cases this allows any modifications required due to UI changes to all be made in one place. Useful information on this technique can be found on numerous blogs as this 'test design pattern' is becoming widely used. *We encourage the reader who wishes to know more to search the internet for blogs on this subject.* Many have written on this design pattern and can provide useful tips beyond the scope of this user guide. To get you started, though, we'll illustrate page objects with a simple example.

First, consider an example, typical of test automation, that does not use a page object.

```
/***
 * Tests login feature
 */
public class Login {

        public void testLogin() {
                selenium.type("inputBox", "testUser");
                selenium.type("password", "my supersecret password");
                selenium.click("sign-in");
                selenium.waitForPageToLoad("PageWaitPeriod");
                Assert.assertTrue(selenium.isElementPresent("compose button"),
                                "Login was unsuccessful");
        }
}
```

There are two problems with this approach.

1. There is no separation between the test method and the AUTs locators (IDs in this example); both are intertwined in a single method. If the AUT's UI changes its identifiers, layout, or how a login is input and processed, the test itself must change.
2. The id-locators would be spread in multiple tests, all tests that had to use this login page.

Applying the page object techniques this example could be rewritten like this in the following example of a page object for a Sign-in page.

```
/**
 * Page Object encapsulates the Sign-in page.
 */
public class SignInPage {

        private Selenium selenium;

        public SignInPage(Selenium selenium) {
                this.selenium = selenium;
                if(!selenium.getTitle().equals("Sign in page")) {
                        throw new IllegalStateException("This is not sign in page, curr
                                        +selenium.getLocation());
                }
        }

        /**
         * Login as valid user
         *
```

```
         * @param userName
         * @param password
         * @return HomePage object
         */
        public HomePage loginValidUser(String userName, String password) {
                selenium.type("usernamefield", userName);
                selenium.type("passwordfield", password);
                selenium.click("sign-in");
                selenium.waitForPageToLoad("waitPeriod");

                return new HomePage(selenium);
        }
}
```

and page object for a Home page could look like this.

```
/**
 * Page Object encapsulates the Home Page
 */
public class HomePage {

        private Selenium selenium;

        public HomePage(Selenium selenium) {
                if (!selenium.getTitle().equals("Home Page of logged in user")) {
                        throw new IllegalStateException("This is not Home Page of logge
                                        "is: " +selenium.getLocation());
                }
        }

        public HomePage manageProfile() {
                // Page encapsulation to manage profile functionality
                return new HomePage(selenium);
        }

        /*More methods offering the services represented by Home Page
        of Logged User. These methods in turn might return more Page Objects
        for example click on Compose mail button could return ComposeMail class object*/

}
```

So now, the login test would use these two page objects as follows.

```
/***
 * Tests login feature
 */
public class TestLogin {

        public void testLogin() {
                SignInPage signInPage = new SignInPage(selenium);
                HomePage homePage = signInPage.loginValidUser("userName", "password");
                Assert.assertTrue(selenium.isElementPresent("compose button"),
                                "Login was unsuccessful");
        }
}
```

There is a lot of flexibility in how the page objects may be designed, but there are a few basic rules for getting the desired maintainability of your test code. Page objects themselves should never be make verifications or assertions. This is part of your test and should always be within the test's code, never in an page object. The page object will contain the representation of the page, and the services the page provides via methods but no code related to what is being tested should be within the page object.

There is one, single, verification which can, and should, be within the page object and that is to verify that the page, and possibly critical elements on the page, were loaded correctly. This verification should be done while instantiating the page object. In the examples above, both the SignInPage and HomePage constructors check that the expected page is available and ready for requests from the test.

A page object does not necessarily need to represent an entire page. The Page Object design pattern could be used to represent components on a page. If a page in the AUT has multiple components, it may improved maintainability if there was a separate page object for each component.

There are other design patterns that also may be used in testing. Some use a Page Factory for instantiating their page objects. Discussing all of these is beyond the scope of this user guide. Here, we merely want to introduce the concepts to make the reader aware of some of the things that can be done. As was mentioned earlier, many have blogged on this topic and we encourage the reader to search for blogs on these topics.

7.8 Data Driven Testing

Data Driven Testing refers to using the same test (or tests) multiple times with varying data. These data sets are often from external files i.e. .csv file, text file, or perhaps loaded from a database. Data driven testing is a commonly used test automation technique used to validate an application against many varying input. When the test is designed for varying data, the input data can expand, essentially creating additional tests, without requiring changes to the test code.

In Python:

```
# Collection of String values
source = open("input_file.txt", "r")
values = source.readlines()
source.close()
# Execute For loop for each String in the values array
for search in values:
    sel.open("/")
    sel.type("q", search)
    sel.click("btnG")
    sel.waitForPageToLoad("30000")
    self.failUnless(sel.is_text_present("Results * for " + search))
```

The Python script above opens a text file. This file contains a different search string on each line. The code then saves this in an array of strings, and iterates over the array doing a search and assert on each string.

This is a very basic example, but the idea is to show that running a test with varying data can be done easily with a programming or scripting language. For more examples, refer to the Selenium RC wiki for examples of reading data from a spreadsheet or for using the data provider capabilities of TestNG. Additionally, this is a well-known topic among test automation professionals including those who don't use Selenium so searching the internet on "data-driven testing" should reveal many blogs on this topic.

7.9 Database Validation

Another common type of testing is to compare data in the UI against the data actually stored in the AUT's database. Since you can also do database queries from a programming language, assuming you have database support functions, you can use them to retrieve data and then use the data to verify what's displayed by the AUT is correct.

Consider the example of a registered email address to be retrieved from a database and then later compared against the UI. An example of establishing a DB connection and retrieving data from the DB could look like this.

In Java:

```
// Load Microsoft SQL Server JDBC driver.
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

// Prepare connection url.
String url = "jdbc:sqlserver://192.168.1.180:1433;DatabaseName=TEST_DB";

// Get connection to DB.
public static Connection con =
DriverManager.getConnection(url, "username", "password");

// Create statement object which would be used in writing DDL and DML
// SQL statement.
public static Statement stmt = con.createStatement();

// Send SQL SELECT statements to the database via the Statement.executeQuery
// method which returns the requested information as rows of data in a
// ResultSet object.

ResultSet result =  stmt.executeQuery
("select top 1 email_address from user_register_table");

// Fetch value of "email_address" from "result" object.
String emailaddress = result.getString("email_address");

// Use the emailAddress value to login to application.
selenium.type("userID", emailaddress);
selenium.type("password", secretPassword);
selenium.click("loginButton");
selenium.waitForPageToLoad(timeOut);
Assert.assertTrue(selenium.isTextPresent("Welcome back" +emailaddress), "Unable to log
```

This is a simple Java example of data retrieval from a database.

CHAPTER

EIGHT

SELENIUM-GRID

Please refer to the Selenium Grid website

http://selenium-grid.seleniumhq.org/how_it_works.html

This section is not yet developed. If there is a member of the community who is experienced in Selenium-Grid, and would like to contribute, please contact the Documentation Team. We would love to have you contribute.

CHAPTER

NINE

USER-EXTENSIONS

NOTE: This section is close to completion, but it has not been reviewed and edited.

9.1 Introduction

It can be quite simple to extend Selenium, adding your own actions, assertions and locator-strategies. This is done with JavaScript by adding methods to the Selenium object prototype, and the PageBot object prototype. On startup, Selenium will automatically look through methods on these prototypes, using name patterns to recognize which ones are actions, assertions and locators. The following examples give an indication of how Selenium can be extended with JavaScript.

9.2 Actions

All methods on the Selenium prototype beginning with "do" are added as actions. For each action foo there is also an action fooAndWait registered. An action method can take up to two parameters, which will be passed the second and third column values in the test. Example: Add a "typeRepeated" action to Selenium, which types the text twice into a text box.

```
Selenium.prototype.doTypeRepeated = function(locator, text) {
    // All locator-strategies are automatically handled by "findElement"
    var element = this.page().findElement(locator);

    // Create the text to type
    var valueToType = text + text;

    // Replace the element text with the new text
    this.page().replaceText(element, valueToType);
};
```

9.3 Accessors/Assertions

All *get*Foo and *is*Foo methods on the Selenium prototype are added as accessors (storeFoo). For each accessor there is an assertFoo, verifyFooa nd waitForFoo registered. An assert method can take up to 2 parameters, which will be passed the second and third column values in the test. You can also define your own assertions literally as simple "assert" methods, which will also auto-generate "verify" and "waitFor" commands. Example: Add a valueRepeated assertion, that makes sure that the element

value consists of the supplied text repeated. The 2 commands that would be available in tests would be assertValueRepeated and verifyValueRepeated.

```
Selenium.prototype.assertValueRepeated = function(locator, text) {
    // All locator-strategies are automatically handled by "findElement"
    var element = this.page().findElement(locator);

    // Create the text to verify
    var expectedValue = text + text;

    // Get the actual element value
    var actualValue = element.value;

    // Make sure the actual value matches the expected
    Assert.matches(expectedValue, actualValue);
};
```

9.3.1 Automatic availability of storeFoo, assertFoo, assertNotFoo, waitForFoo and waitForNotFoo for every getFoo

All *get*Foo and *is*Foo methods on the Selenium prototype automatically result in the availability of storeFoo, assertFoo, assertNotFoo, verifyFoo, verifyNotFoo, waitForFoo, and waitForNotFoo commands. Example, if you add a `getTextLength()` method, the following commands will automatically be available: `storeTextLength`, `assertTextLength`, `assertNotTextLength`, `verifyTextLength`, `verifyNotTextLength`, `waitForTextLength`, and `waitForNotTextLength` commands.

```
Selenium.prototype.getTextLength = function(locator, text) {
    return this.getText(locator).length;
};
```

Also note that the `assertValueRepeated` method described above could have been implemented using `isValueRepeated`, with the added benefit of also automatically getting assertNotValueRepeated, storeValueRepeated, waitForValueRepeated and waitForNotValueRepeated.

9.4 Locator Strategies

All locateElementByFoo methods on the PageBot prototype are added as locator-strategies. A locator strategy takes 2 parameters, the first being the locator string (minus the prefix), and the second being the document in which to search. Example: Add a "valuerepeated=" locator, that finds the first element a value attribute equal to the the supplied value repeated.

```
// The "inDocument" is a the document you are searching.
PageBot.prototype.locateElementByValueRepeated = function(text, inDocument) {
    // Create the text to search for
    var expectedValue = text + text;

    // Loop through all elements, looking for ones that have
    // a value === our expected value
    var allElements = inDocument.getElementsByTagName("*");
    for (var i = 0; i < allElements.length; i++) {
        var testElement = allElements[i];
```

```
        if (testElement.value && testElement.value === expectedValue) {
            return testElement;
        }
    }
    return null;
};
```

9.5 Using User-Extensions With Selenium-IDE

User-extensions are very easy to use with the selenium IDE.

1. Create your user extension and save it as user-extensions.js. While this name isn't technically necessary, it's good practice to keep things consistent.
2. Open Firefox and open Selenium-IDE.
3. Click on Tools, Options
4. In Selenium Core Extensions click on Browse and find the user-extensions. js file. Click on OK.
5. Your user-extension will not yet be loaded, you must close and restart Selenium-IDE.
6. In your empty test, create a new command, your user-extension should now be an options in the Commands dropdown.

9.6 Using User-Extensions With Selenium RC

If you Google "Selenium RC user-extension" ten times you will find ten different approaches to using this feature. Below, is the official Selenium suggested approach.

9.6.1 Example

C#

1. Place your user extension in the same directory as your Selenium Server.
2. If you are using client code generated by the Selenium-IDE you will need to make a couple small edits. First, you will need to create an `HttpCommandProcessor` object with class scope (outside the SetupTest method, just below `private StringBuilder verificationErrors;`)

```
HttpCommandProcessor proc;
```

1. Next, instantiate that `HttpCommandProcessor` object as you would the `DefaultSelenium` object. This can be done in the test setup.

```
proc = new HttpCommandProcessor("localhost", 4444, "*iexplore", "http://google.ca/");
```

1. Instantiate the DefaultSelenium object using the `HttpCommandProcessor` object you created.

```
selenium = new DefaultSelenium(proc);
```

1. Within your test code, execute your user-extension by calling it with the `DoCommand()` method of `HttpCommandProcessor`. This method takes two arguments: a string to identify the user-extension method you want to use and string array to pass arguments. Notice that the first letter of your function is lower case, regardless of the capitalization in your user-extension. Selenium automatically does this to keep common JavaScript naming conventions. Because JavaScript is case sensitive, your test will fail if you begin this command with a capital. inputParams is the array of arguments you want to pass to the JavaScript user-extension. In this case there is only one string in the array because there is only one parameter for our user extension, but a longer array will map each index to the corresponding user-extension parameter. Remember that user extensions designed for Selenium-IDE will only take two arguments.

```
string[] inputParams = { "Hello World" };
proc.DoCommand( "alertWrapper" , inputParams);
```

1. Start the test server using the `-userExtensions` argument and pass in your `user-extensinos.js` file.

```
java -jar selenium-server.jar -userExtensions user-extensions.js
```

```
using System;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading;
using NUnit.Framework;
using Selenium;

namespace SeleniumTests
{
        [TestFixture]
        public class NewTest
        {

                private ISelenium selenium;
                private StringBuilder verificationErrors;
                private HttpCommandProcessor proc;

                [SetUp]
                public void SetupTest()
                {
                        proc = new HttpCommandProcessor( "localhost" , 4444,  "*iexplore"
                        selenium = new DefaultSelenium(proc);
                        //selenium = new DefaultSelenium("localhost", 4444, "*iexplore",
                        selenium.Start();
                        verificationErrors = new StringBuilder();

                }
```

```
        [TearDown]
        public void TeardownTest()
        {
                try
                {
                        selenium.Stop();
                }
                catch (Exception)
                {
                        // Ignore errors if unable to close the browser
                }
                Assert.AreEqual("", verificationErrors.ToString());
        }

        [Test]
        public void TheNewTest()
        {
                selenium.Open("/");
                string[] inputParams = {"Hello World",};
                proc.DoCommand("alertWrapper", inputParams);
        }
    }
}
```

Appendixes:

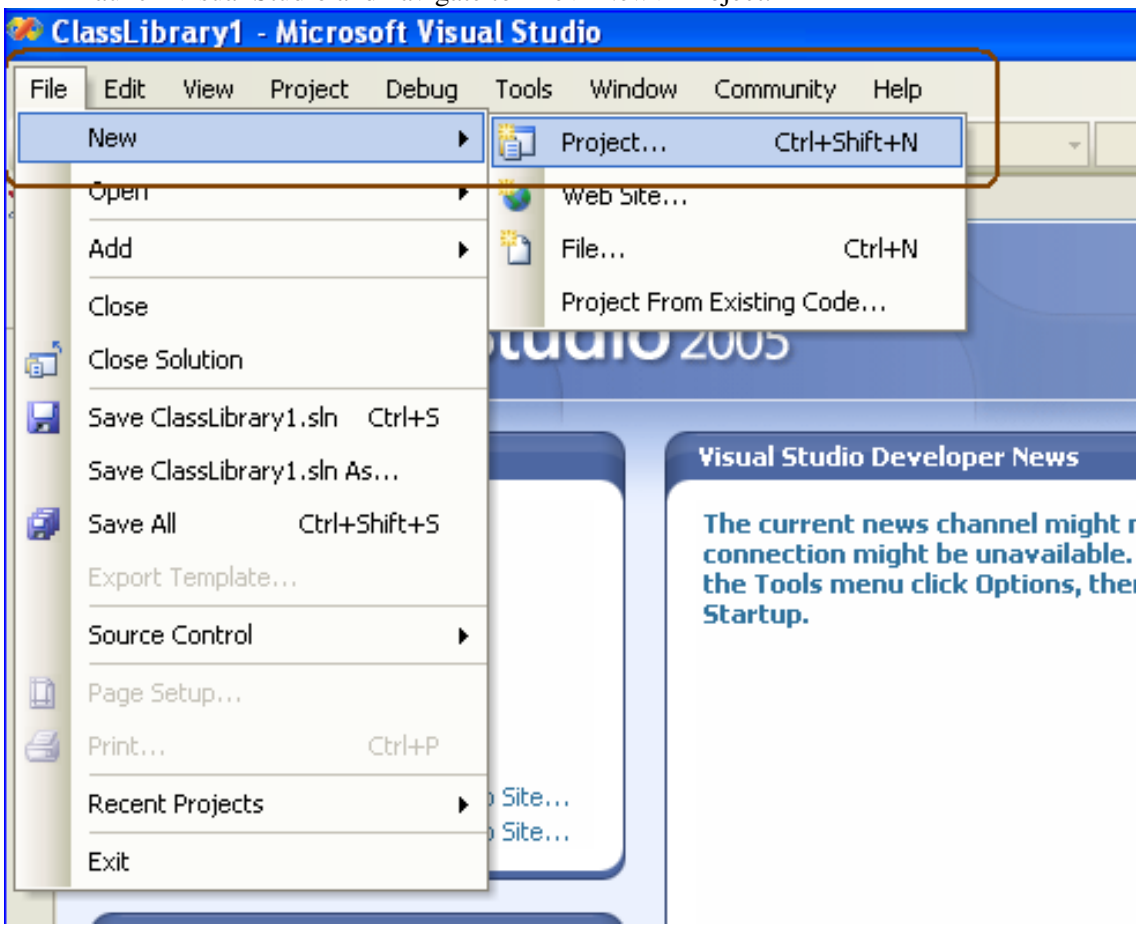
CHAPTER

TEN

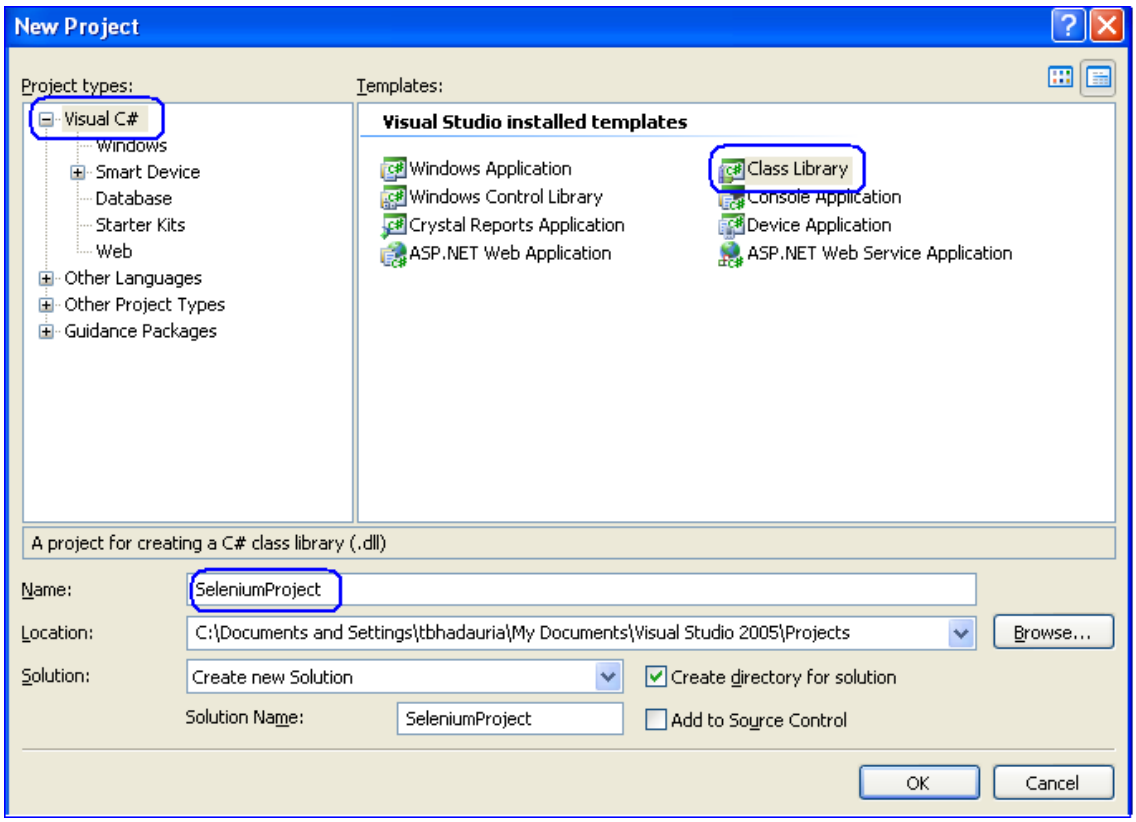
.NET CLIENT DRIVER CONFIGURATION

.NET client Driver can be used with Microsoft Visual Studio. To Configure it with Visual Studio do as Following.

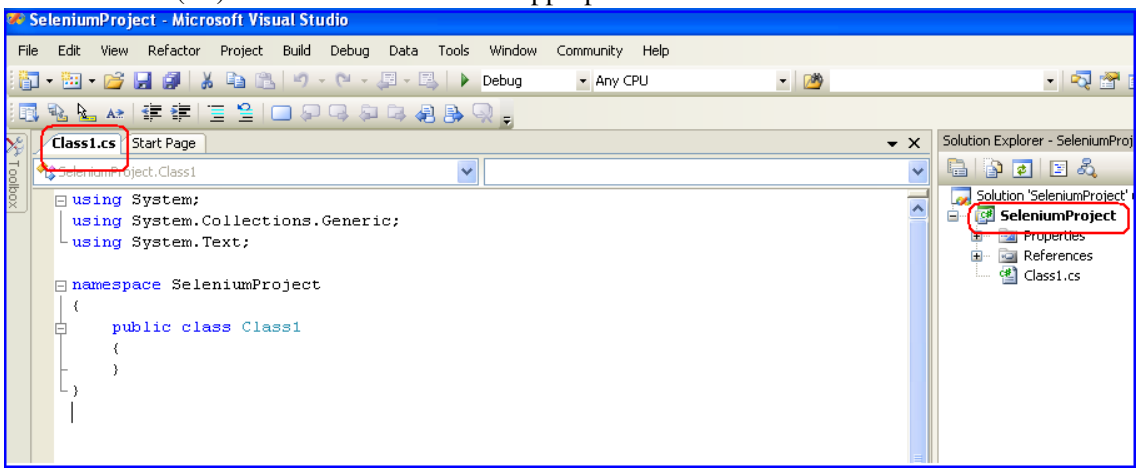
- Launch Visual Studio and navigate to File > New > Project.



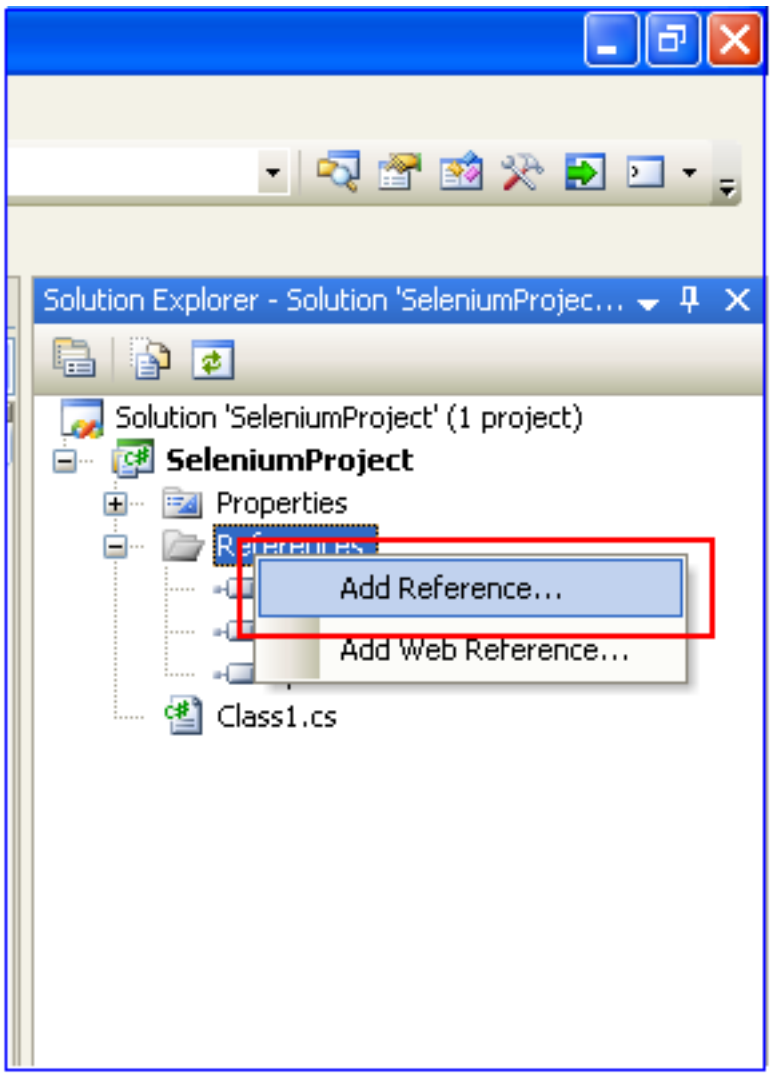
- Select Visual C# > Class Library > Name your project > Click on OK button.



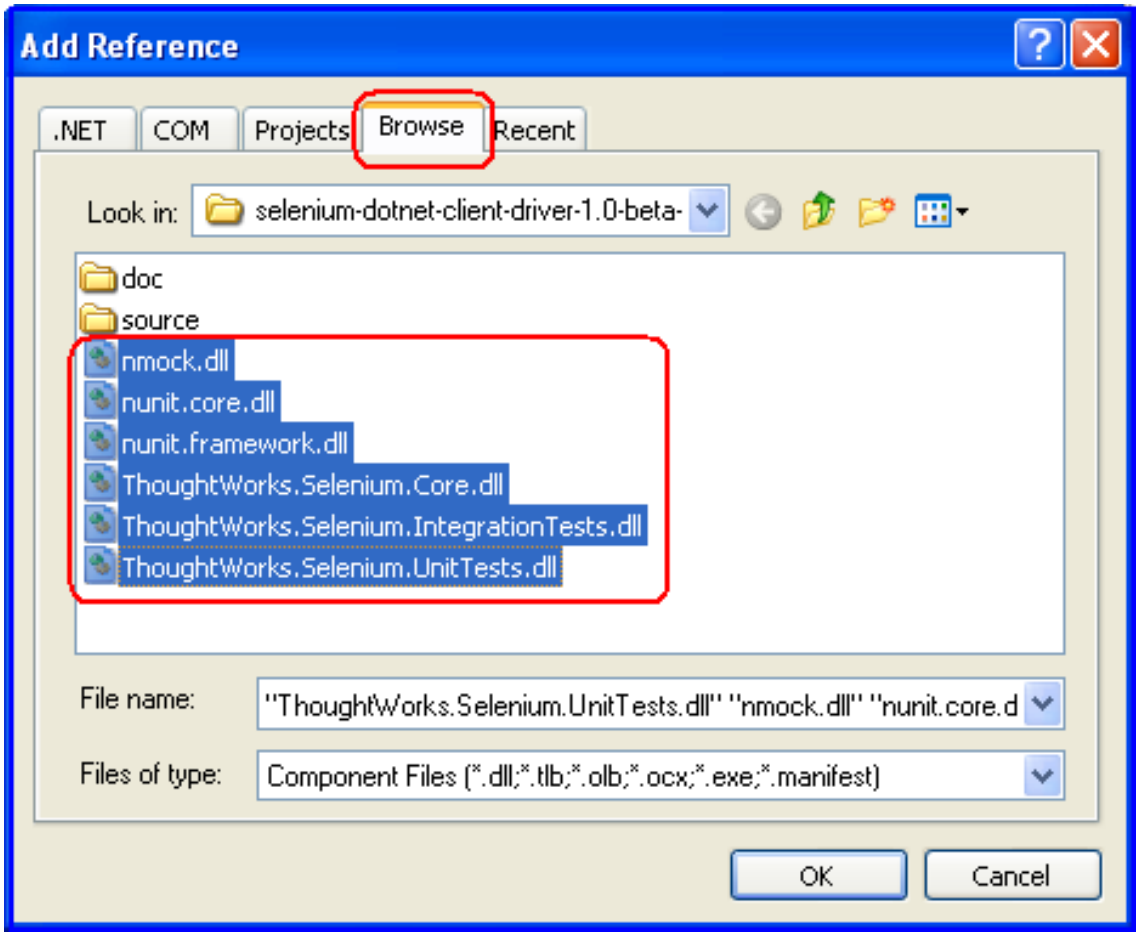
- A Class (.cs) is created. Rename it as appropriate.



- Under right hand pane of Solution Explorer right click on References > Add References.



- Select following dll files - nmock.dll, nunit.core.dll, nunit.framework.dll,ThoughtWorks.Selenium.Core.dll, ThoughtWorks.Selenium.IntegrationTests.dll, ThoughtWorks.Selenium.UnitTests.dll and click on Ok button



With This Visual Studio is ready for Selenium Test Cases.

CHAPTER

ELEVEN

JAVA CLIENT DRIVER CONFIGURATION

In General configuration of Selenium-RC with any java IDE would have following steps:

- Download Selenium-RC from the SeleniumHQ downloads page
- Start any java IDE
- Create new project
- Add "selenium-java-<version-number>.jar" to your project classpath
- Record your test from Selenium-IDE and translate it to java code (Selenium IDE has automatic translation feature to generate tests in variety of languages)
- Run selenium server from console
- Run your test in the IDE

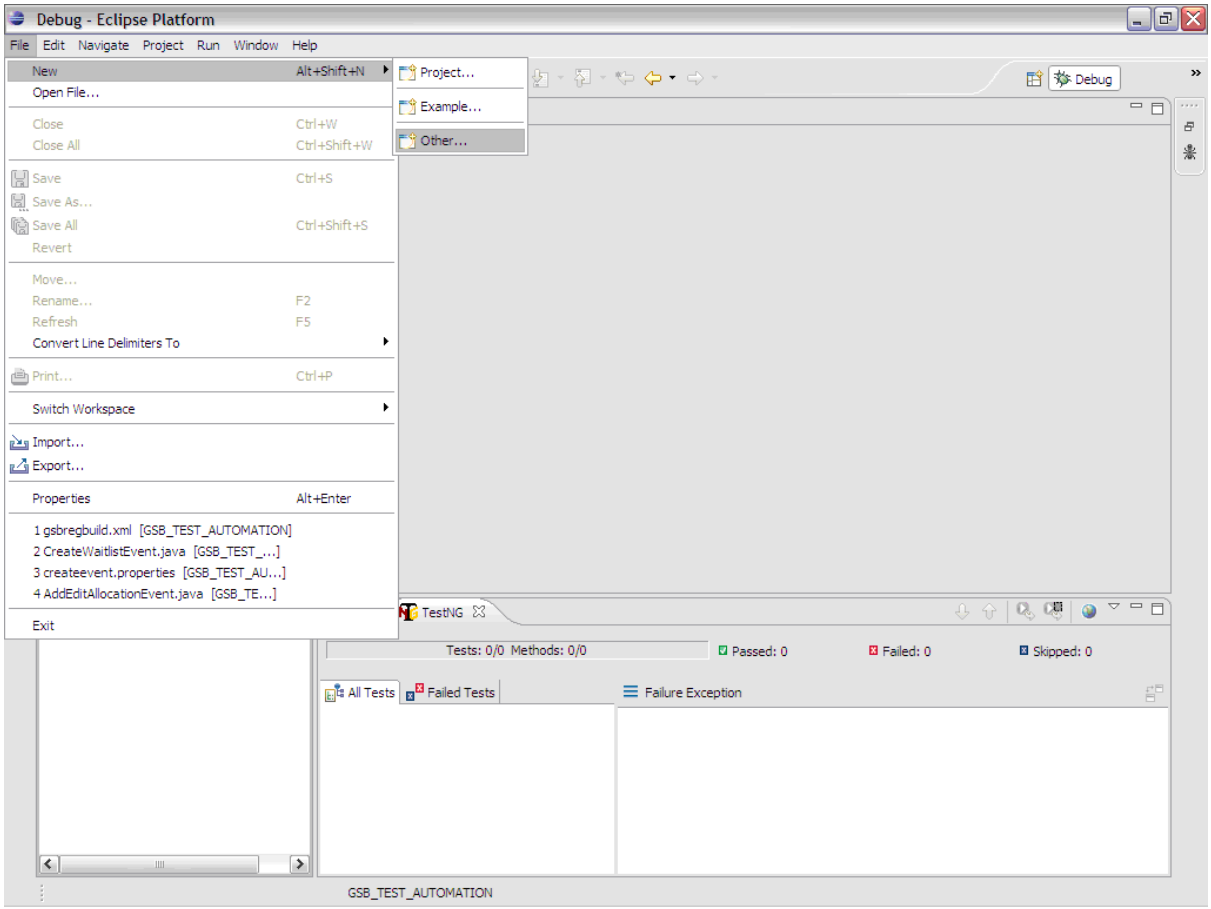
These points have been delineated below with reference to Eclipse and IntelliJ:

11.1 Configuring Selenium-RC With Eclipse

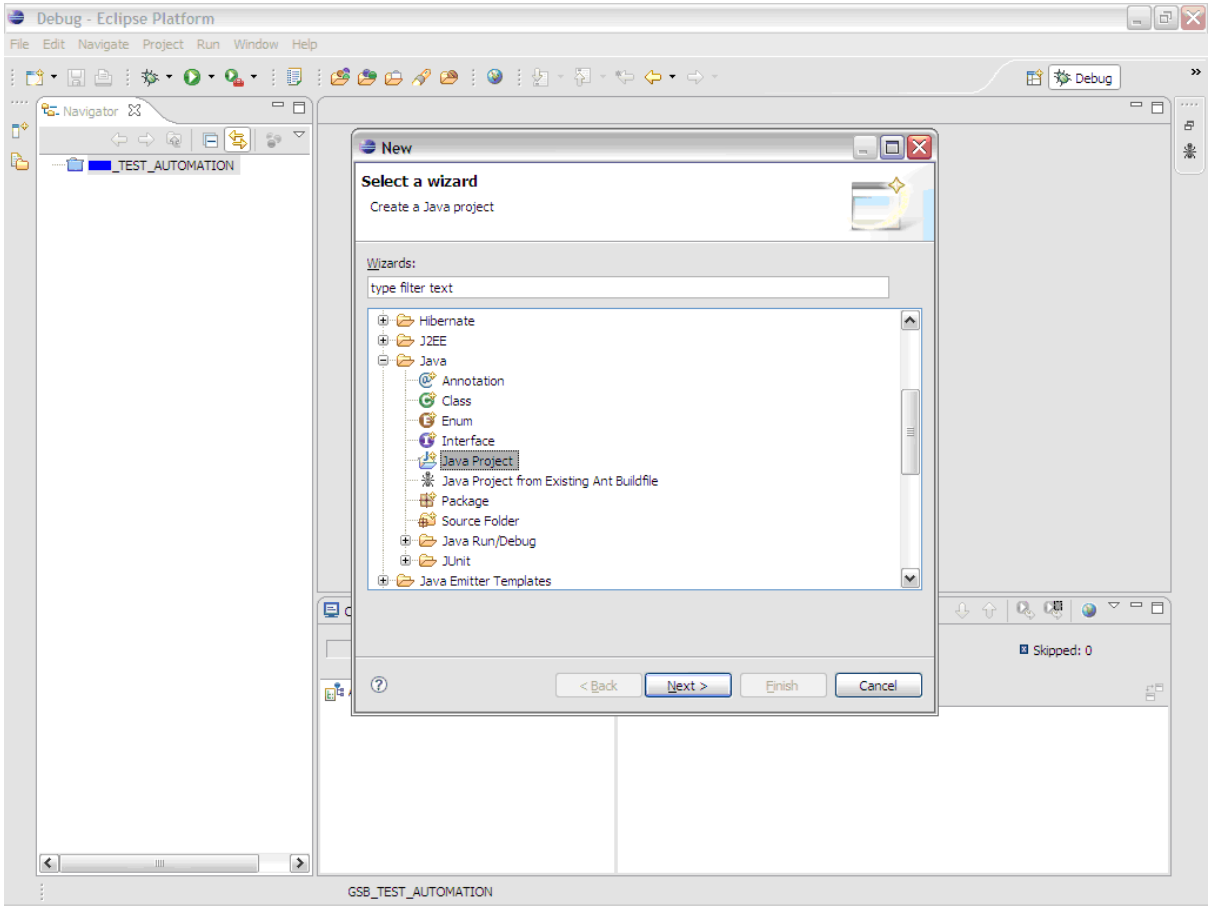
Eclipse is a multi-language software development platform comprising an IDE and a plug-in system to extend it. It is written primarily in Java and is used to develop applications in this language and, by means of the various plug-ins, in other languages as well as C/C++, Cobol, Python, Perl, PHP and more.

Following lines describes configuration of Selenium-RC with Eclipse - Version: 3.3.0. (Europa Release). It should not be too different for higher versions of Eclipse

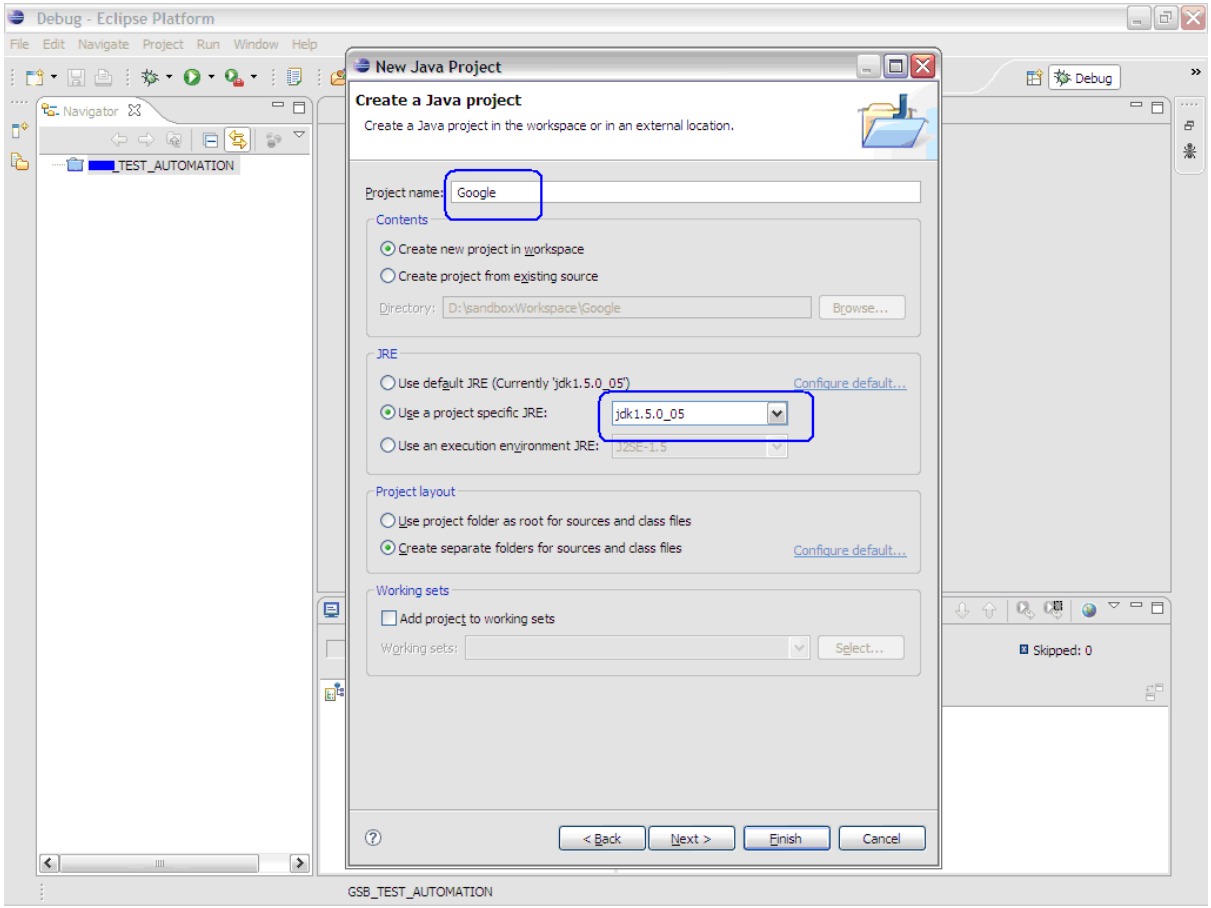
- Launch Eclipse.
- Select File > New > Other.



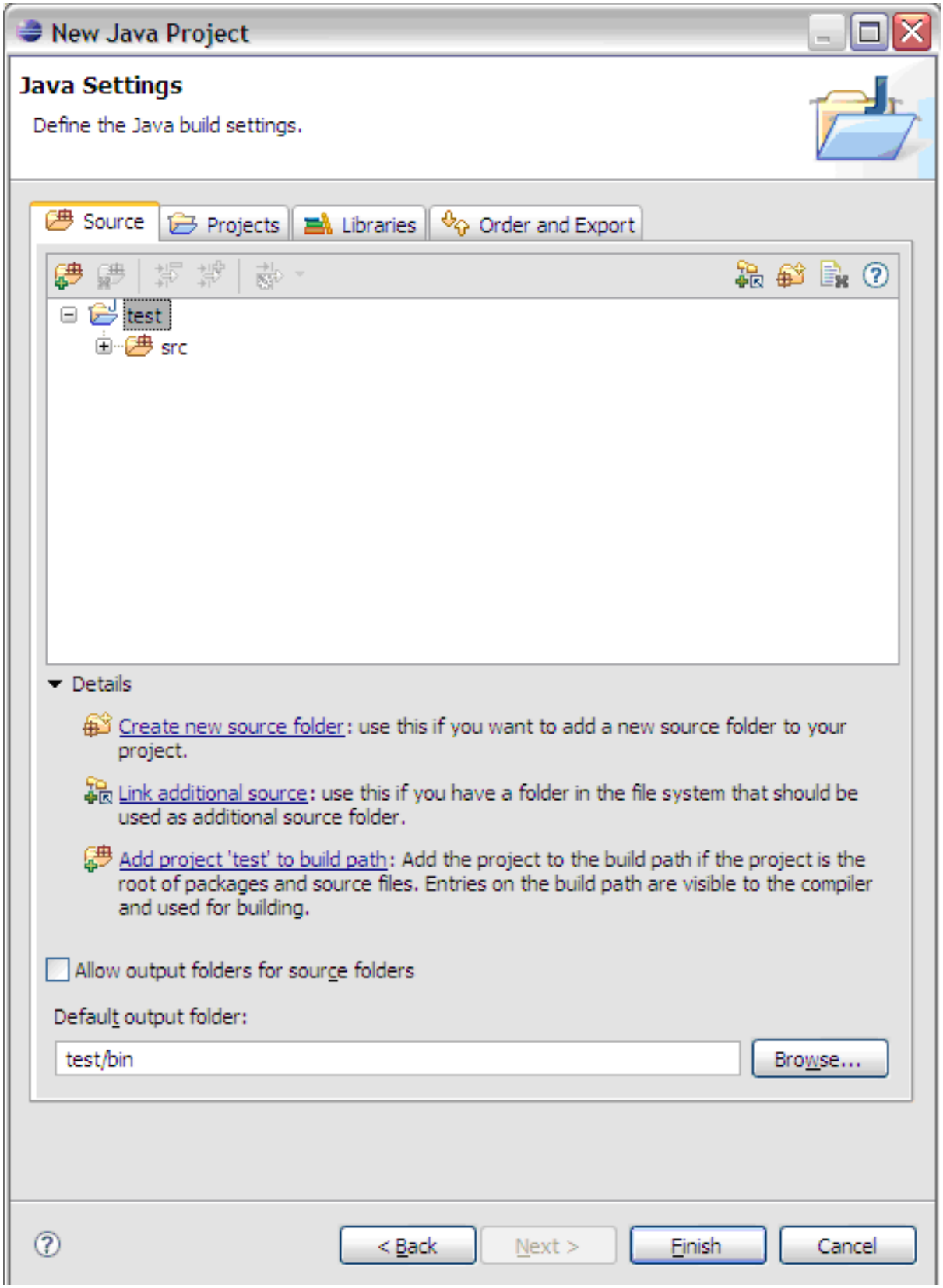
- Java > Java Project > Next



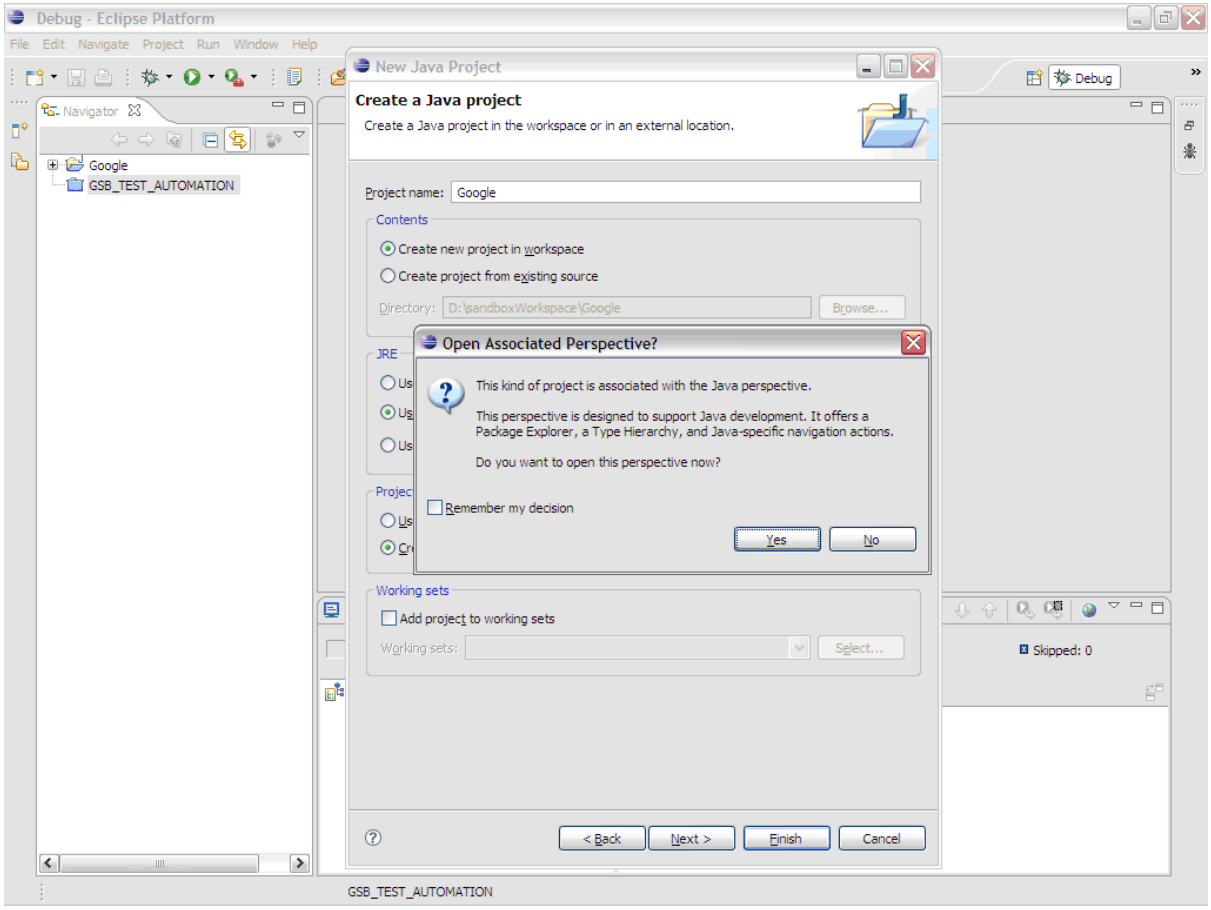
- Provide Name to your project, Select JDK in 'Use a project Specific JRE' option (JDK 1.5 selected in this example) > click Next



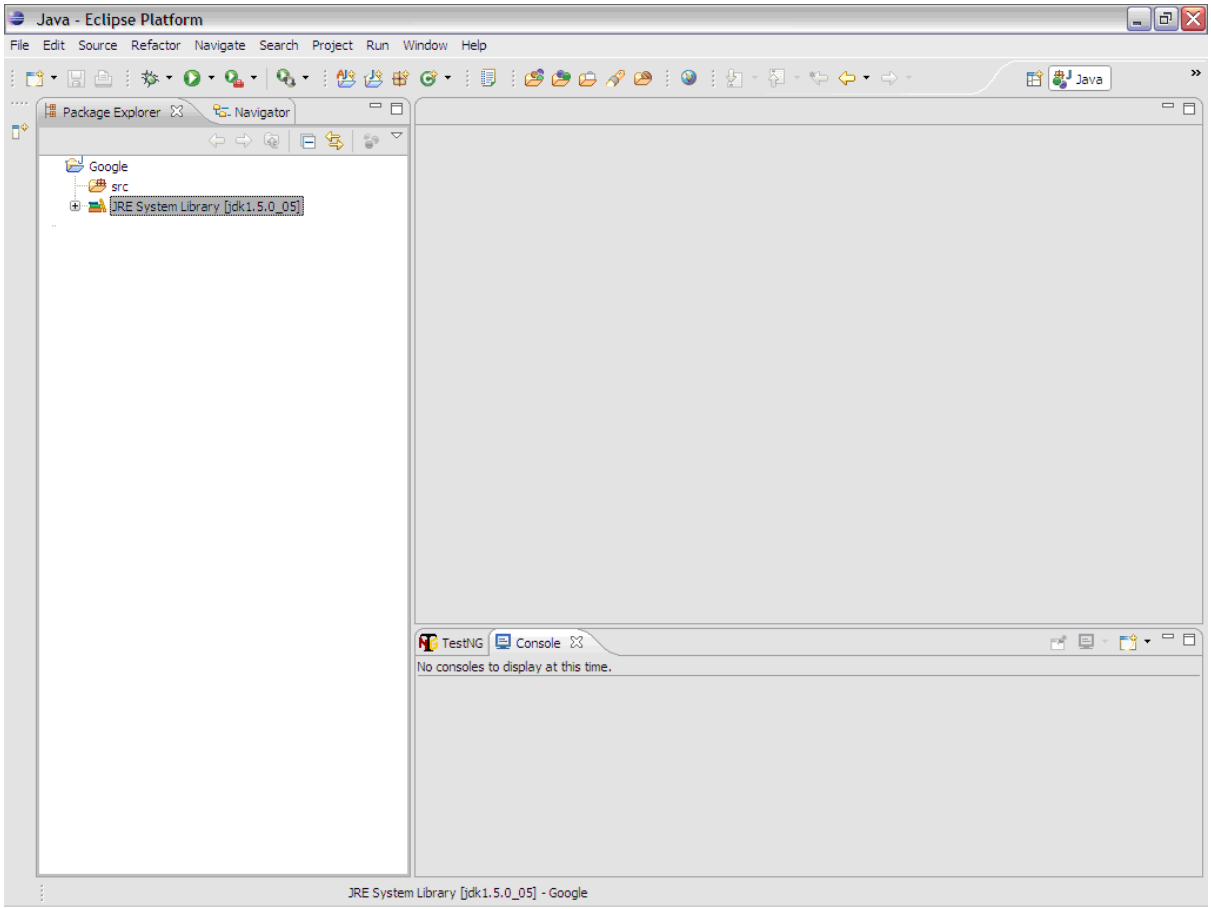
- Keep 'JAVA Settings' intact in next window. Project specific libraries can be added here. (This described in detail in later part of document.)



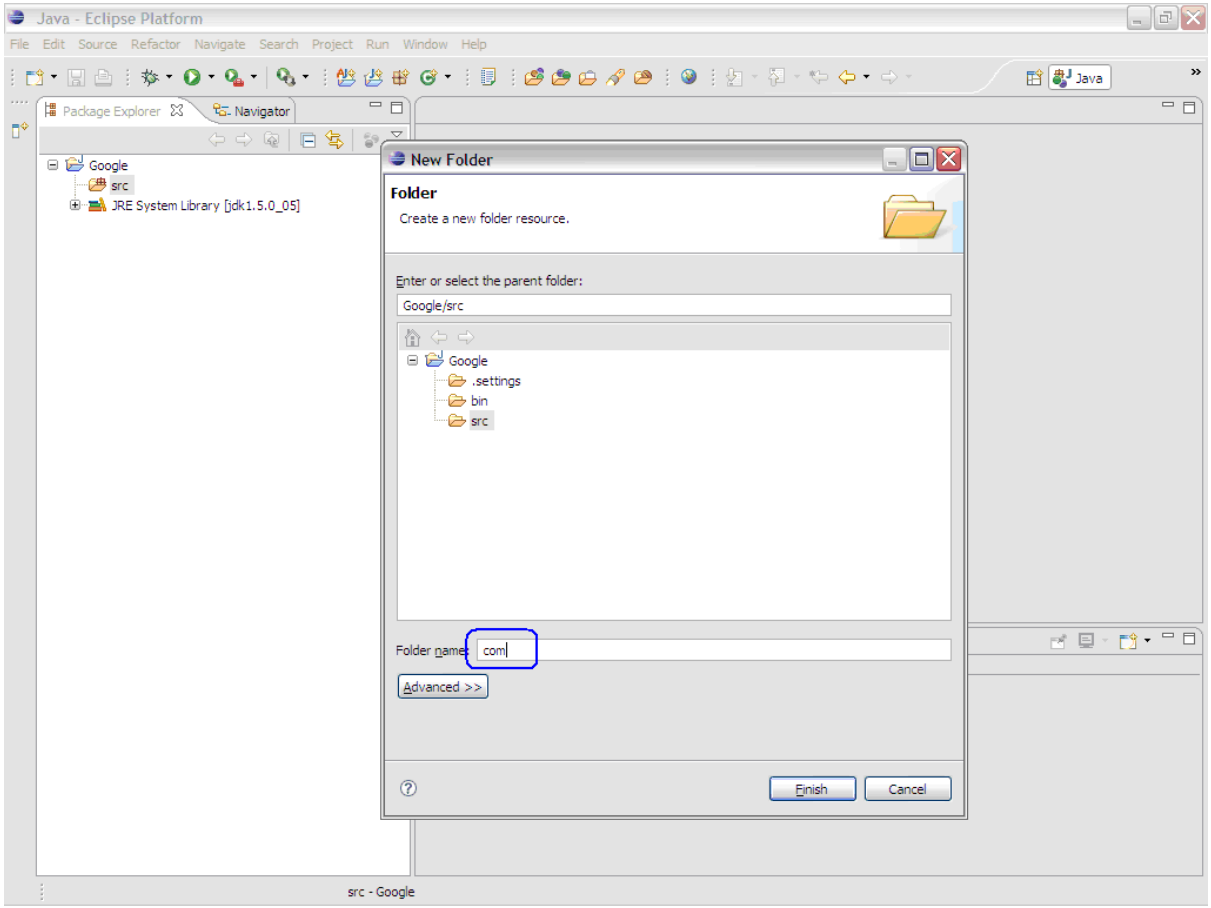
- Click Finish > Click on Yes in Open Associated Perspective pop up window.



This would create Project Google in Package Explorer/Navigator pane.

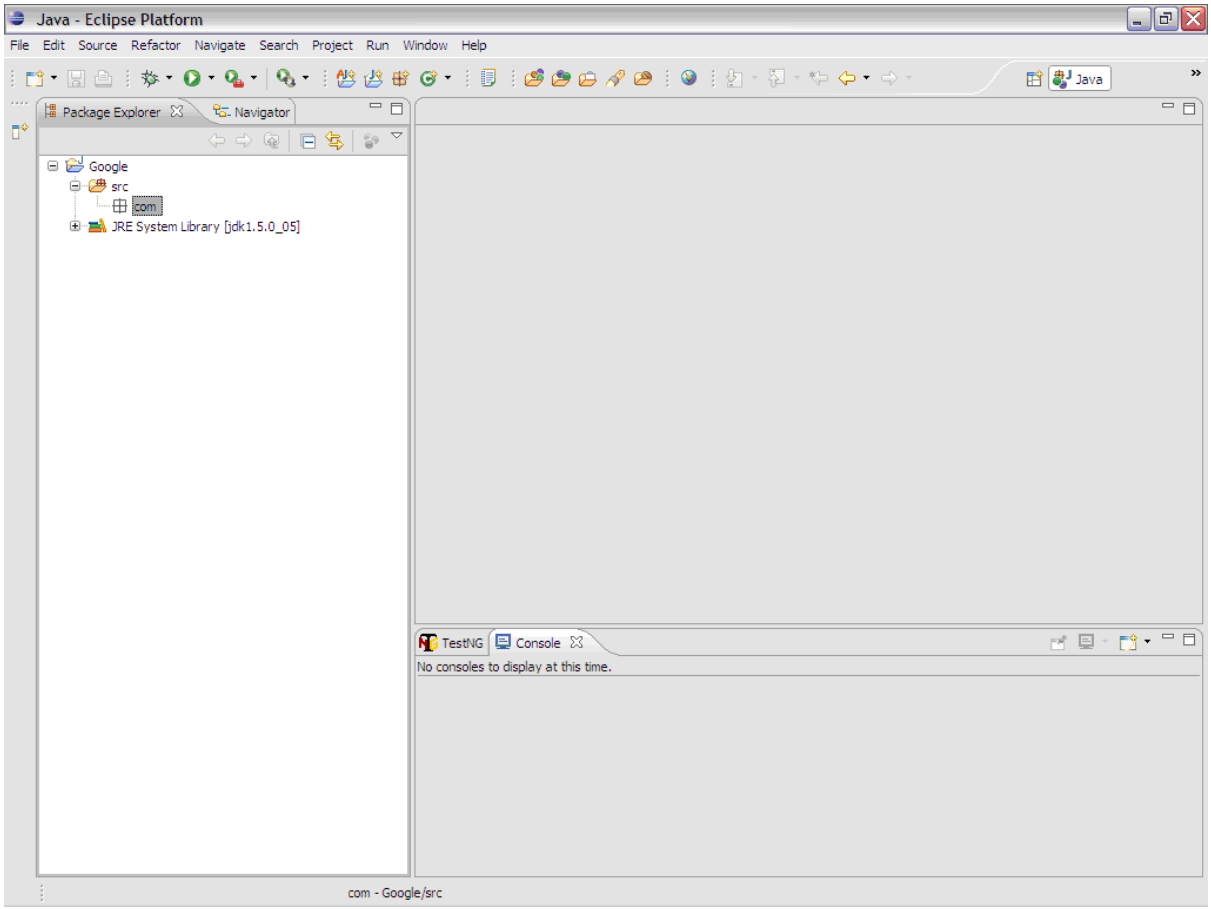


- Right click on src folder and click on New > Folder

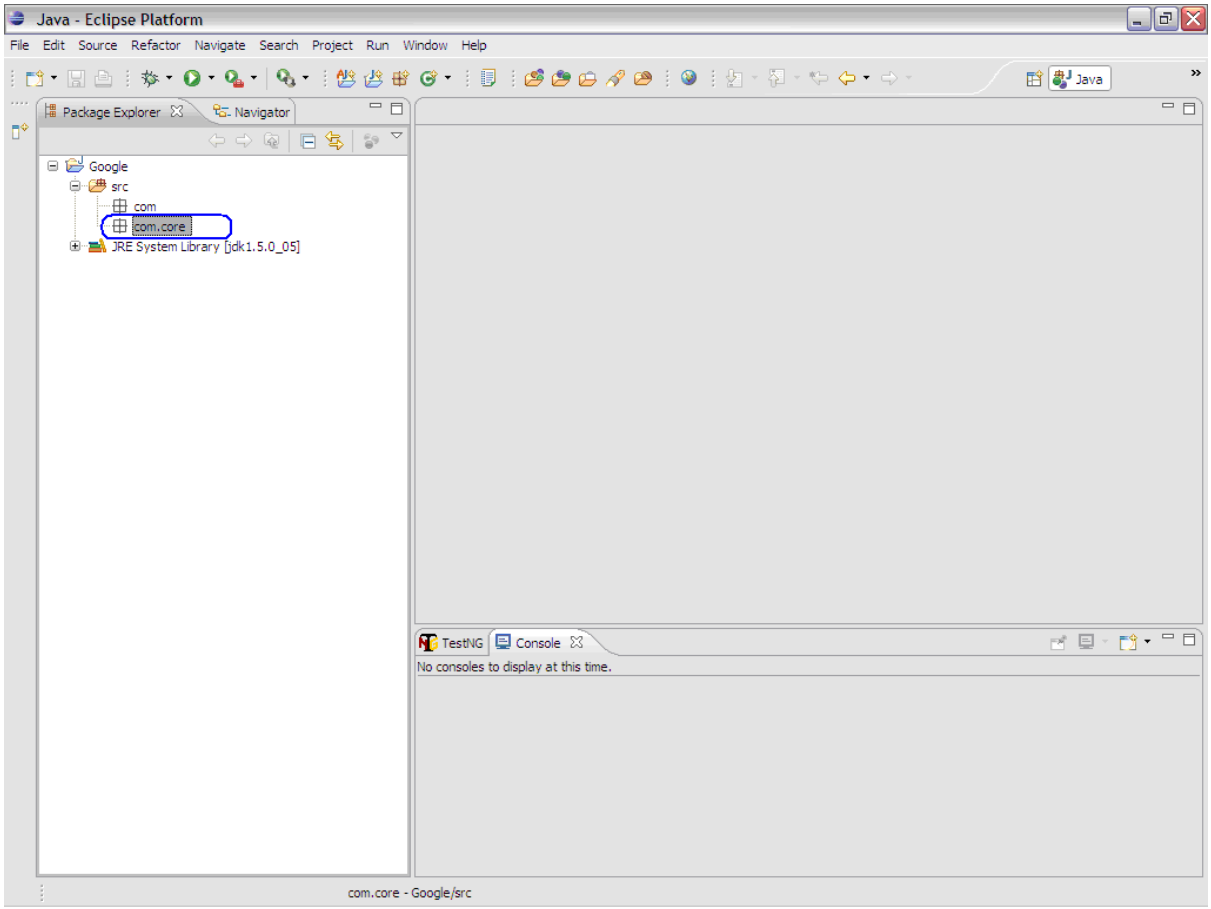


Name this folder as com and click on Finish button.

- This should get com package insider src folder.



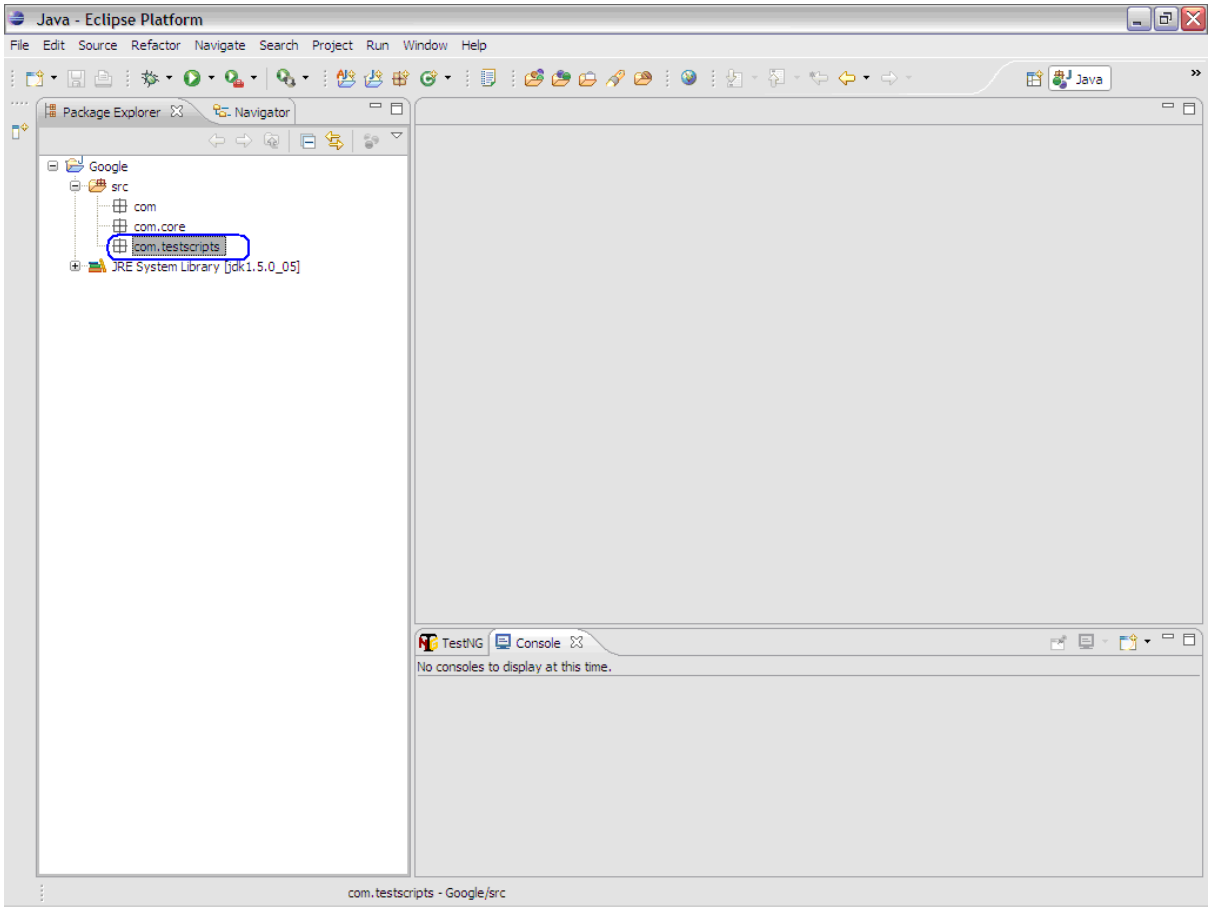
- Following the same steps create *core* folder inside *com*



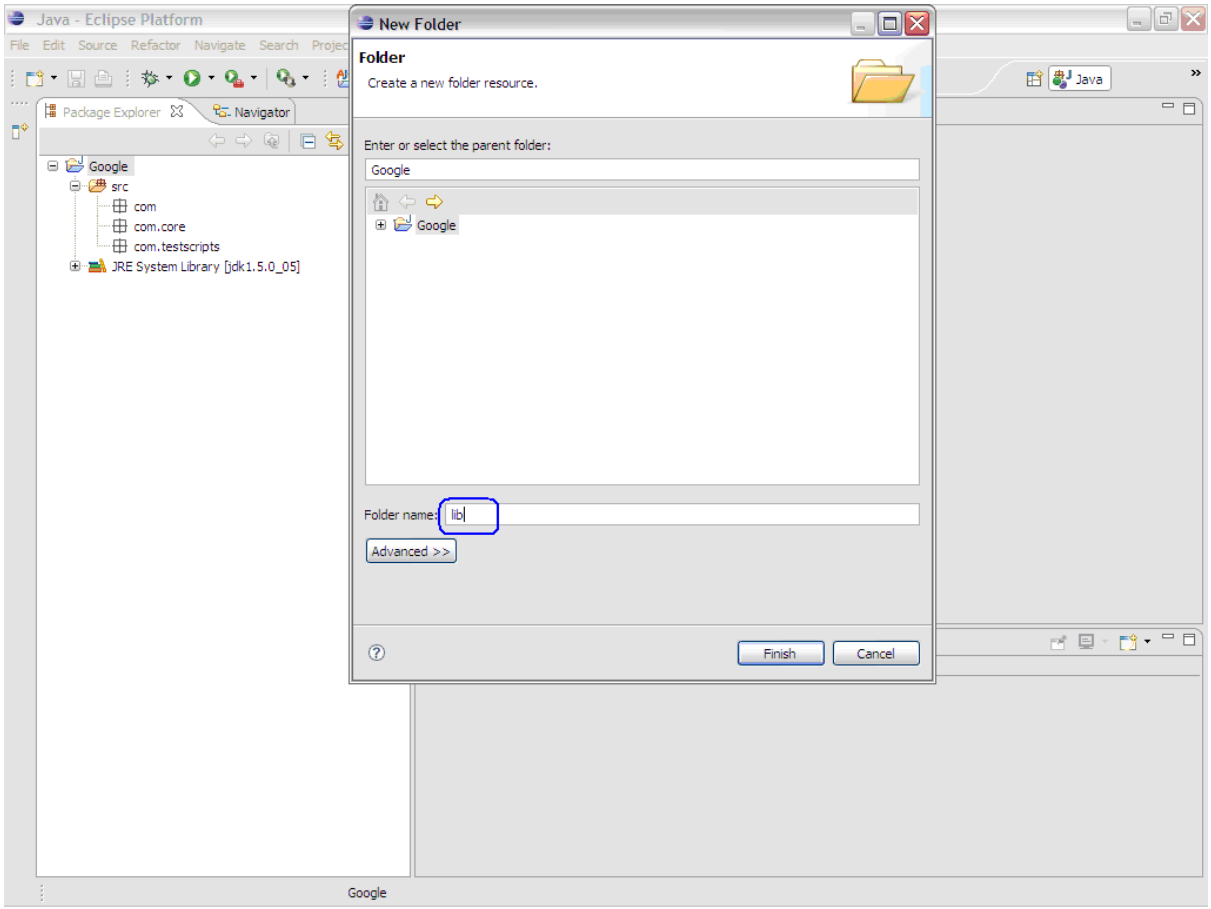
SelTestCase class can be kept inside *core* package.

Create one more package inside *src* folder named *testscripts*. This is a place holder for test scripts.

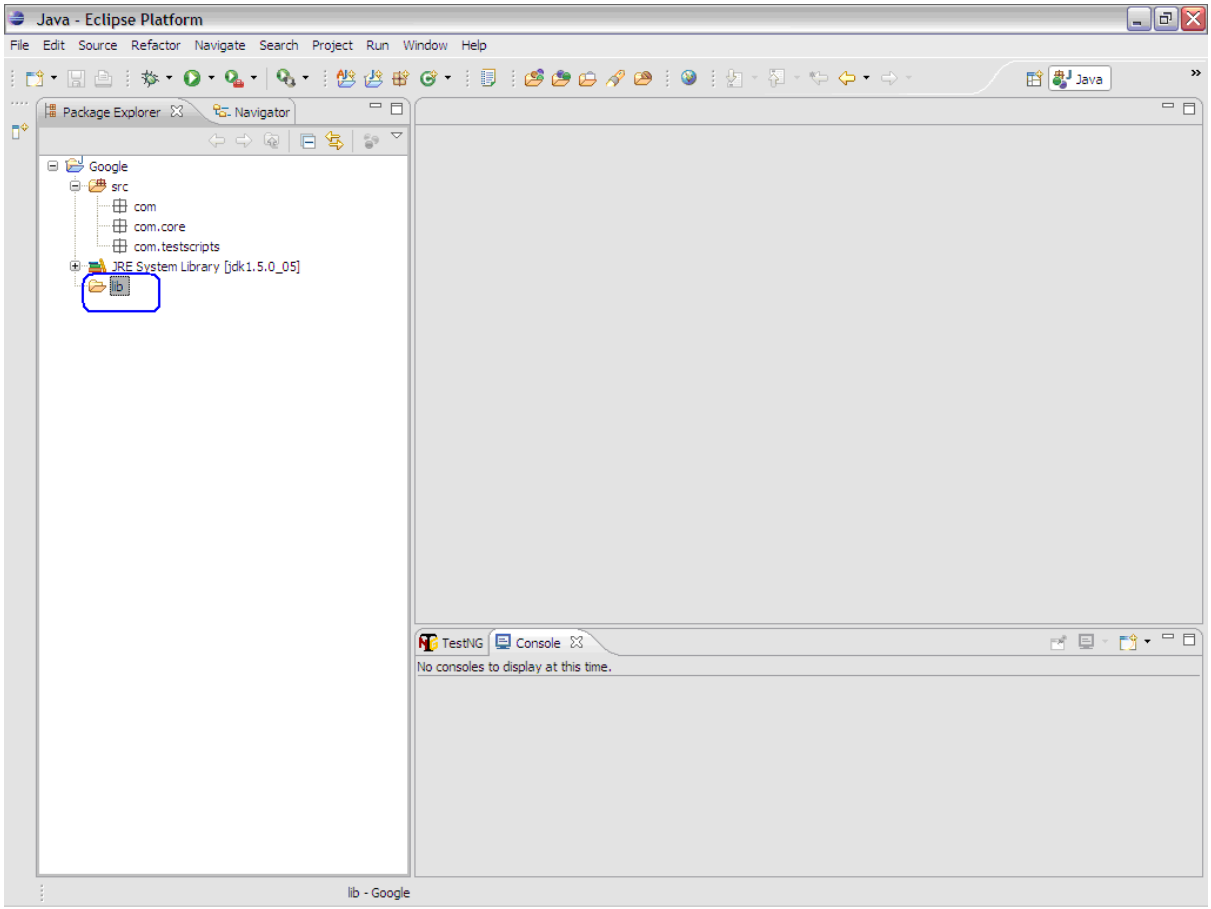
Please notice this is about the organization of project and it entirely depends on individual's choice / organization's standards. Test scripts package can further be segregated depending upon the project requirements.



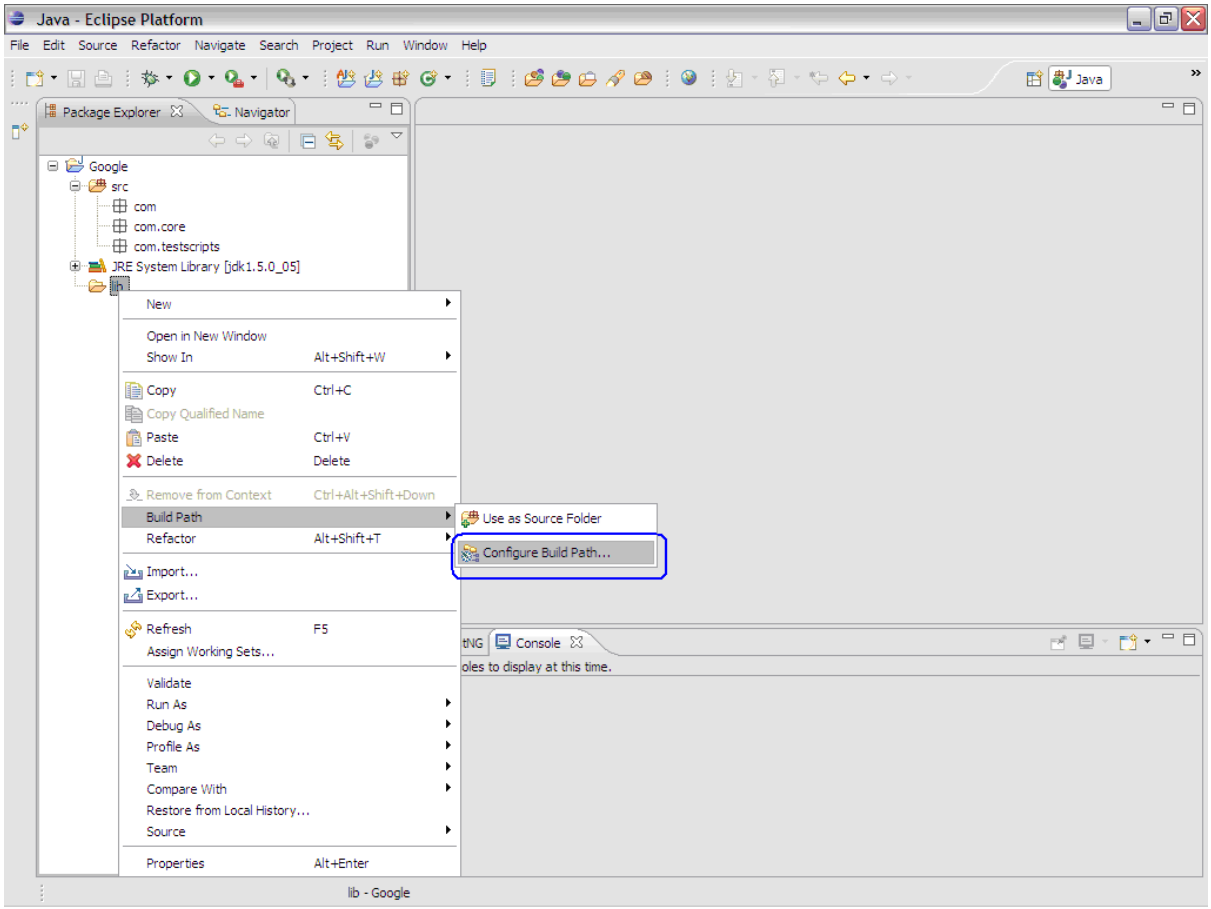
- Create a folder called lib inside project Google. Right click on Project name > New > Folder. This is a place holder for jar files to project (i.e. Selenium client driver, selenium server etc)



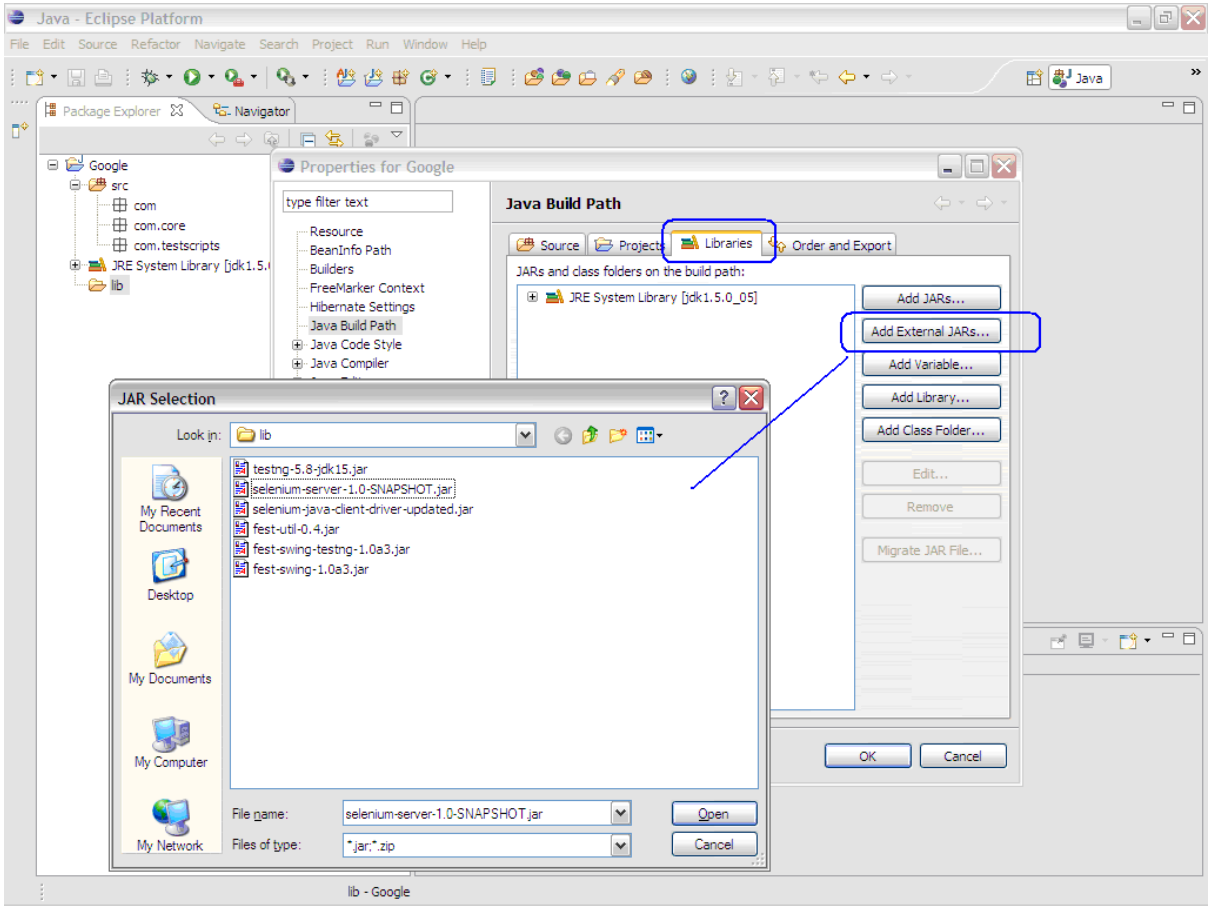
This would create lib folder in Project directory.



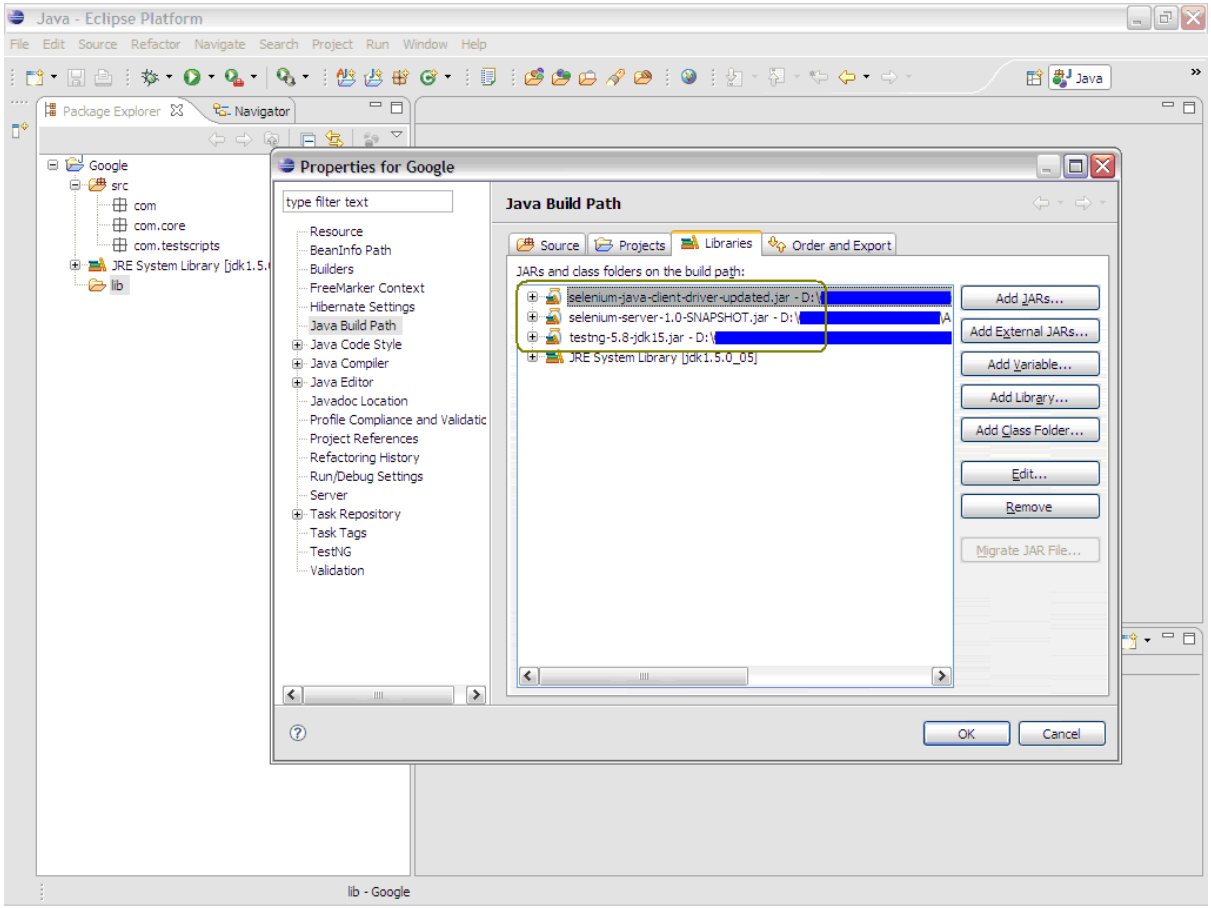
- Right click on *lib* folder > Build Path > Configure build Path



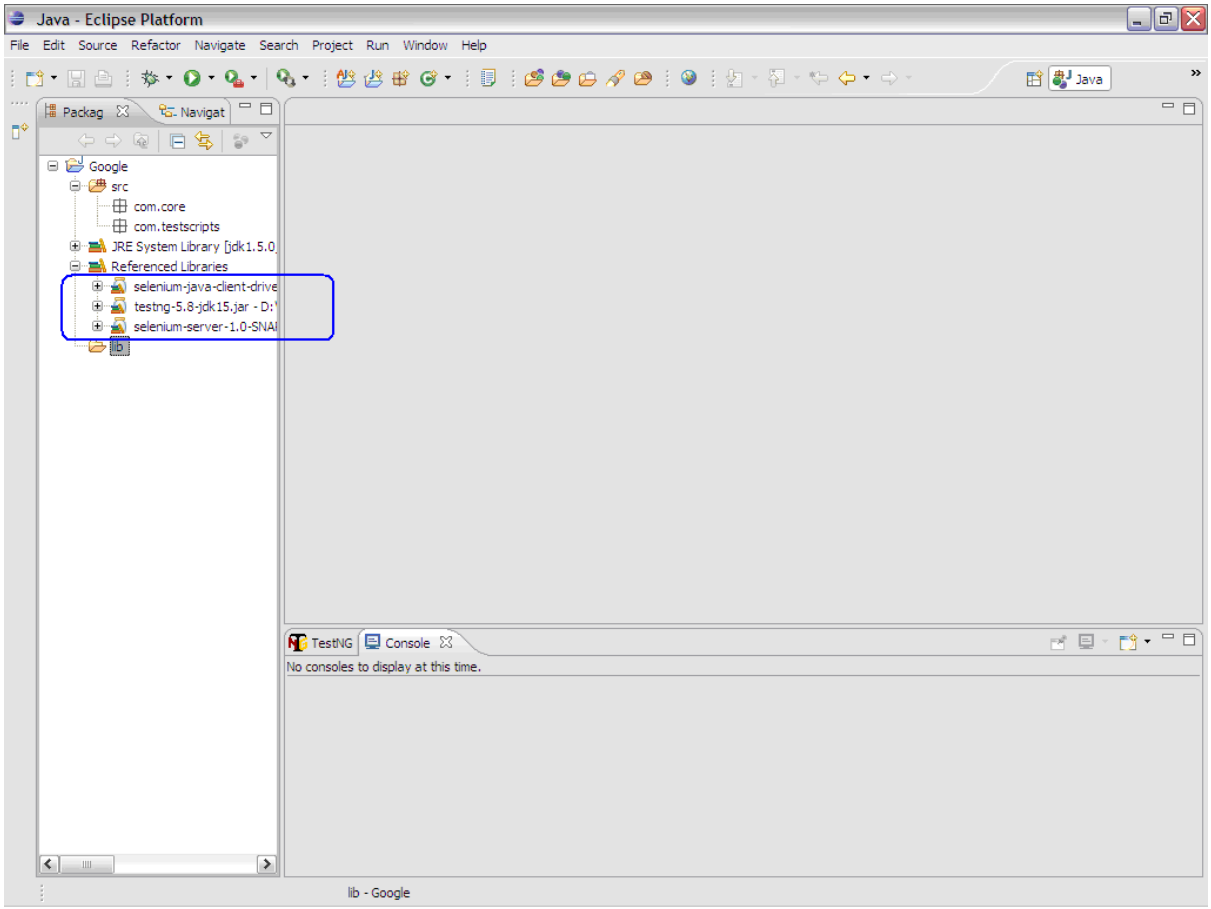
- Under Library tab click on Add External Jars to navigate to directory where jar files are saved. Select the jar files which are to be added and click on Open button.



After having added jar files click on OK button.



Added libraries would appear in Package Explorer as following:

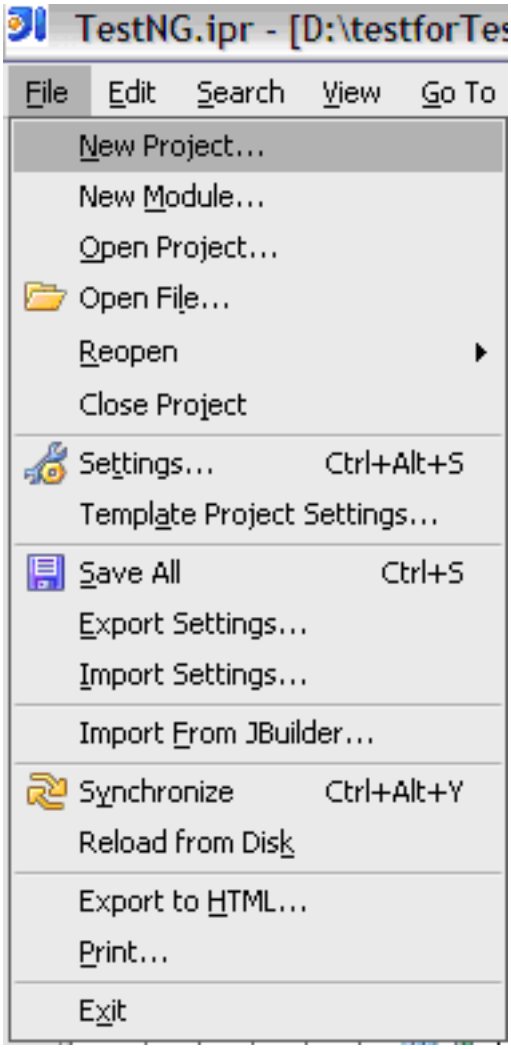


11.2 Configuring Selenium-RC With Intellij

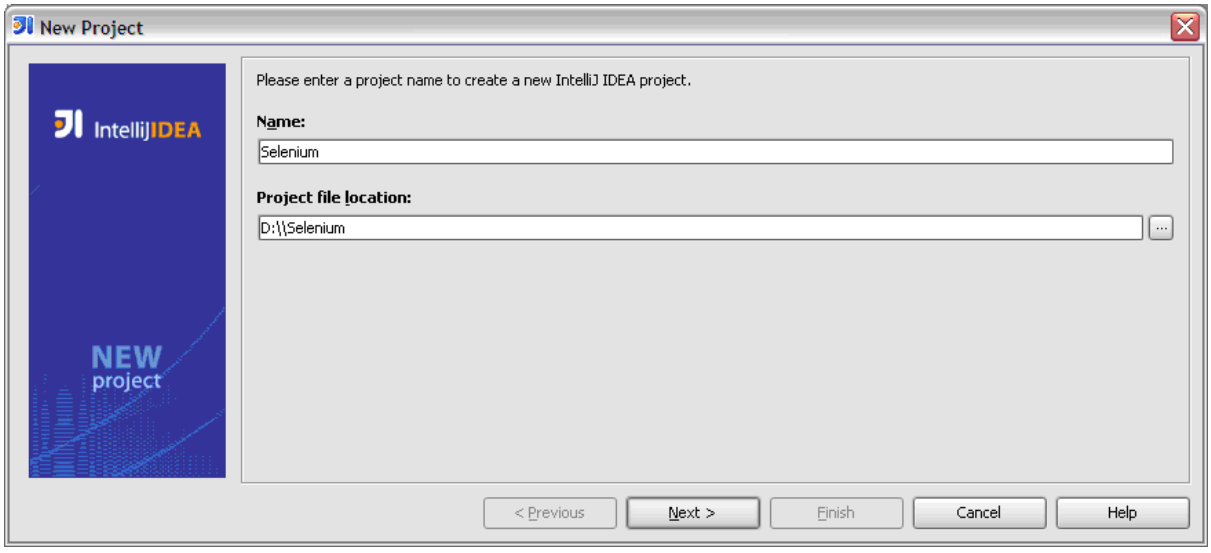
IntelliJ IDEA is a commercial Java IDE by the company JetBrains. Intellij provides a set of integrated refactoring tools that allow programmers to quickly redesign their code. IntelliJ IDEA provides close integration with popular open source development tools such as CVS, Subversion, Apache Ant and JUnit.

Following lines describes configuration of Selenium-RC with IntelliJ 6.0 It should not be very different for higher version of intelliJ.

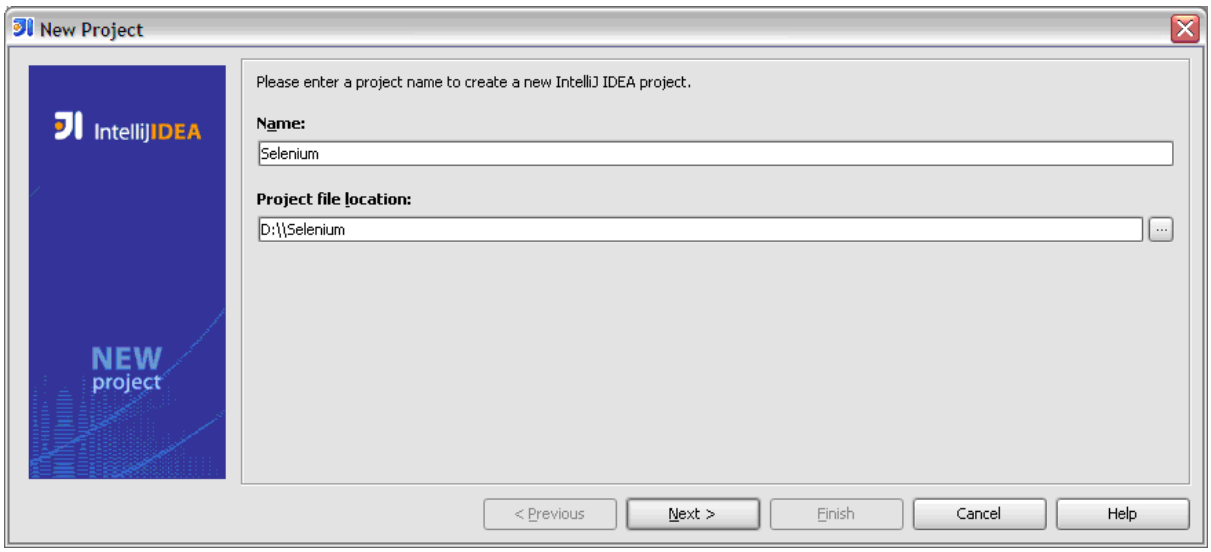
- Open a New Project in IntelliJ IDEA.



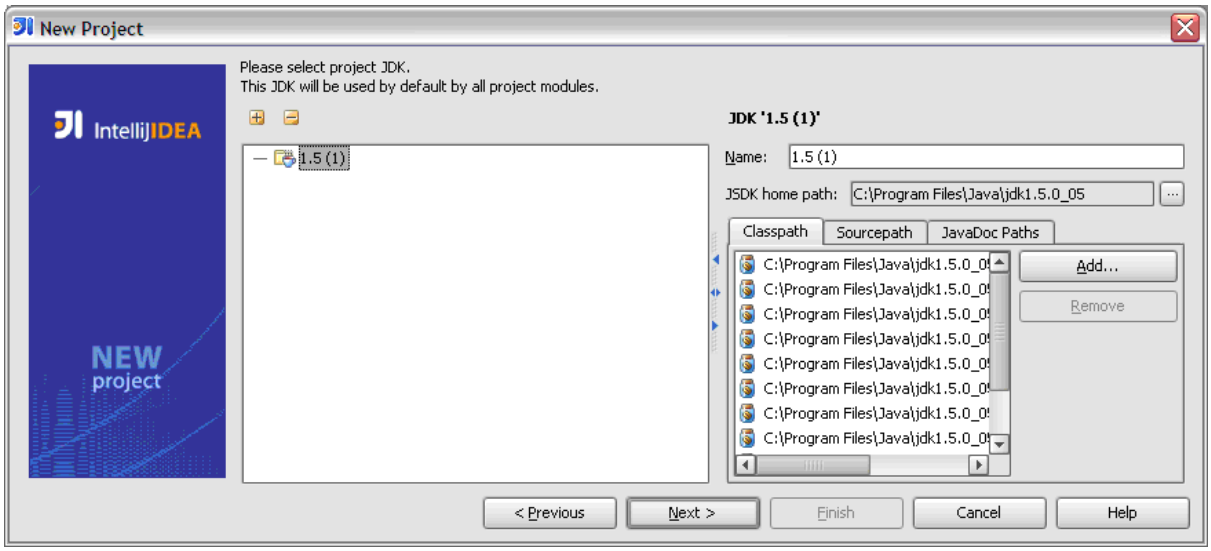
- Provide name and location to Project.



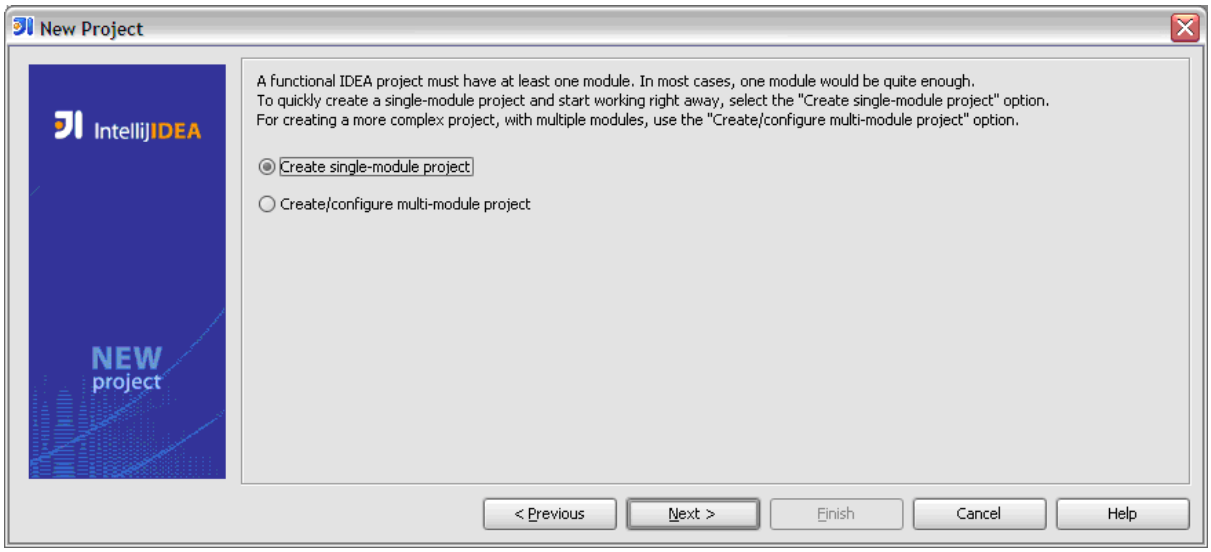
- Click Next and provide compiler output path.



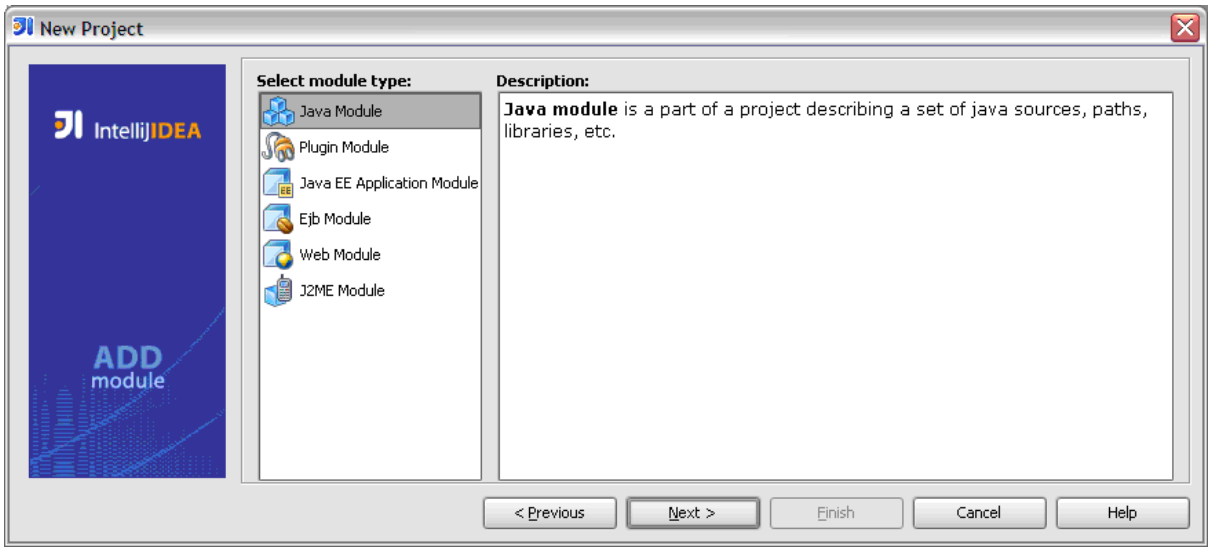
- Click Next and select the JDK to be used.



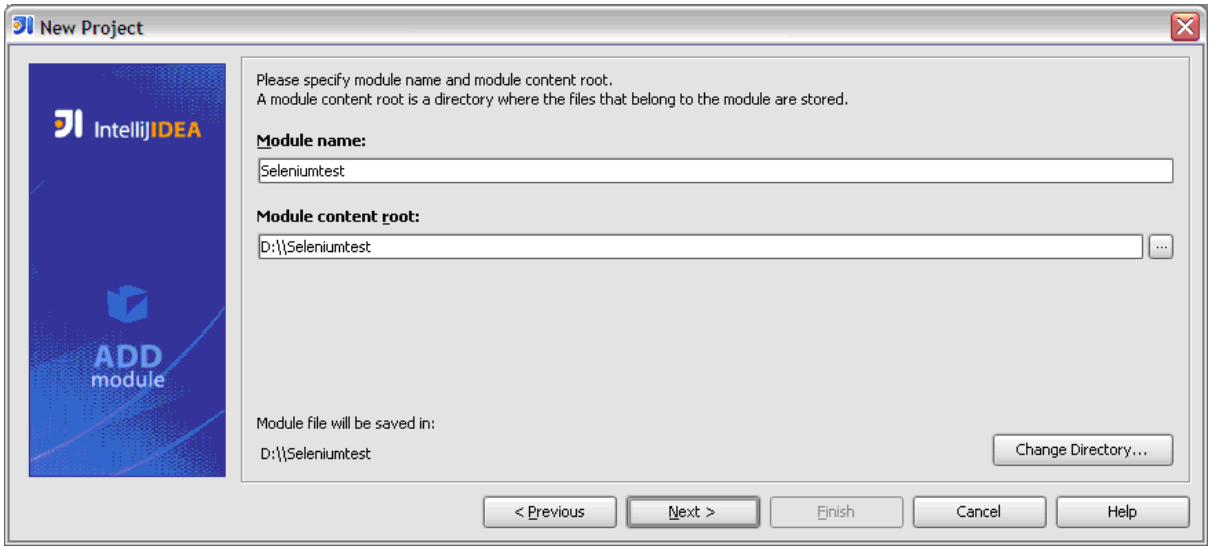
- Click Next and select Single Module Project.



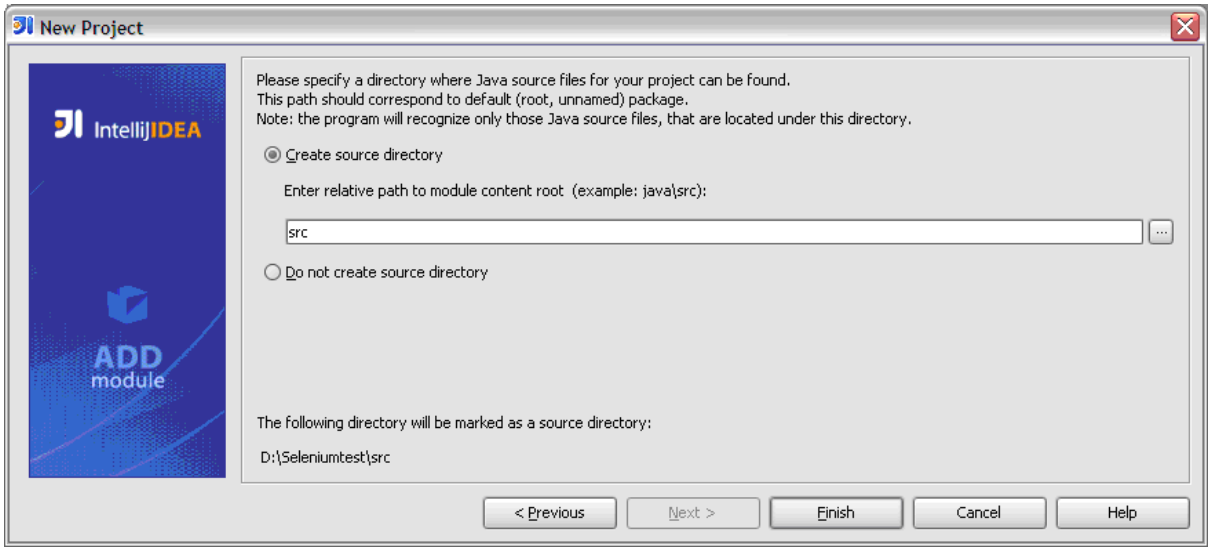
- Click Next and select Java module.



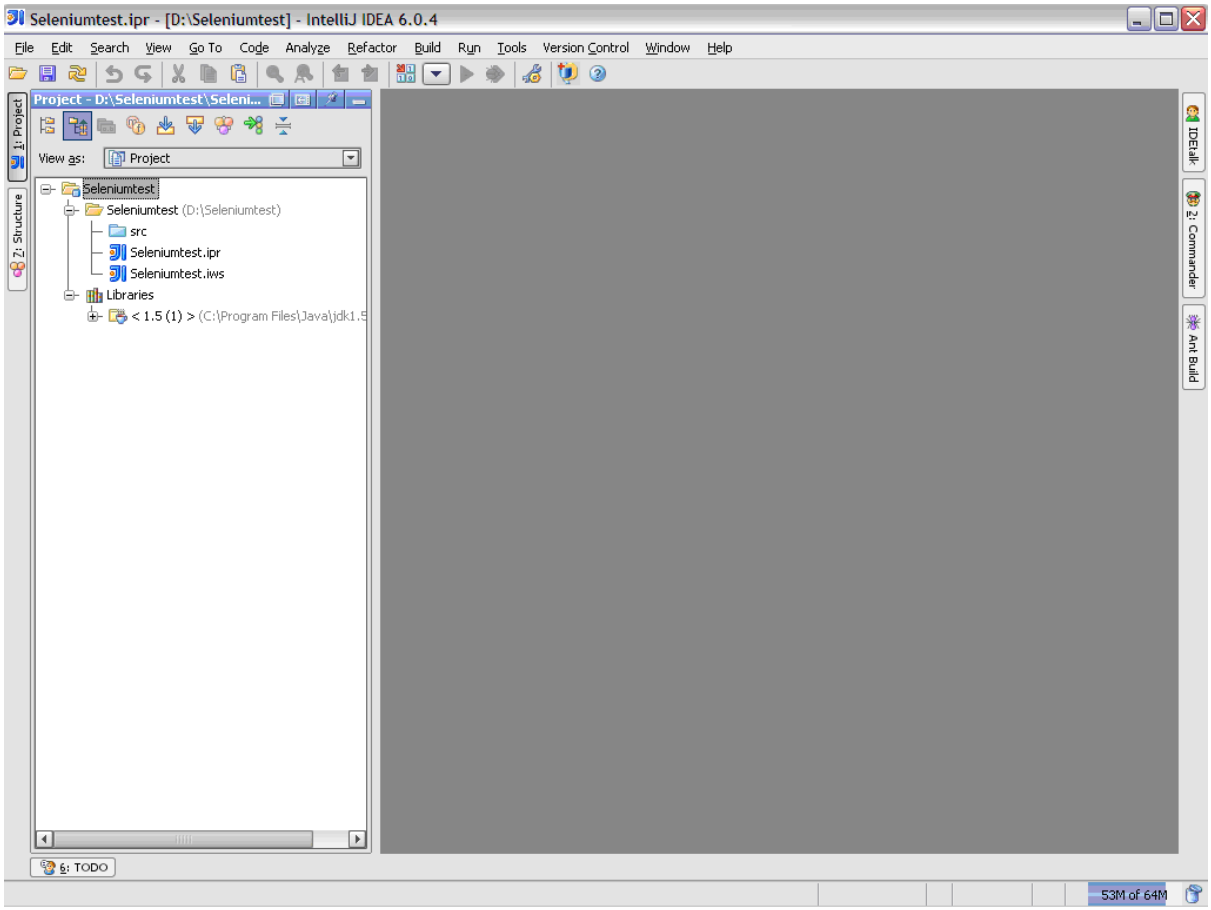
- Click Next and provide Module name and Module content root.



- Click Next and select Source directory.

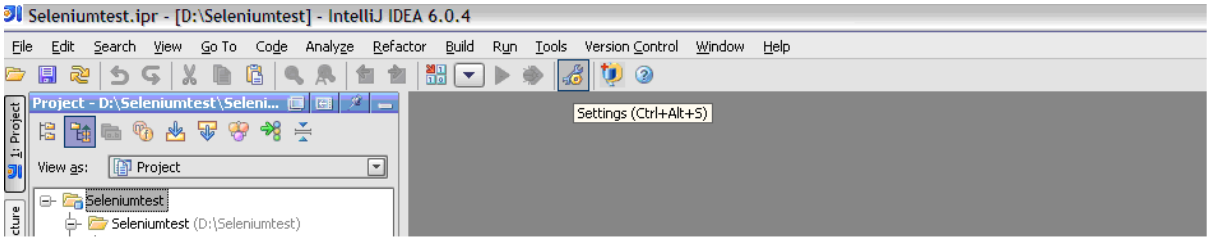


- At last click Finish. This will launch the Project Pan.

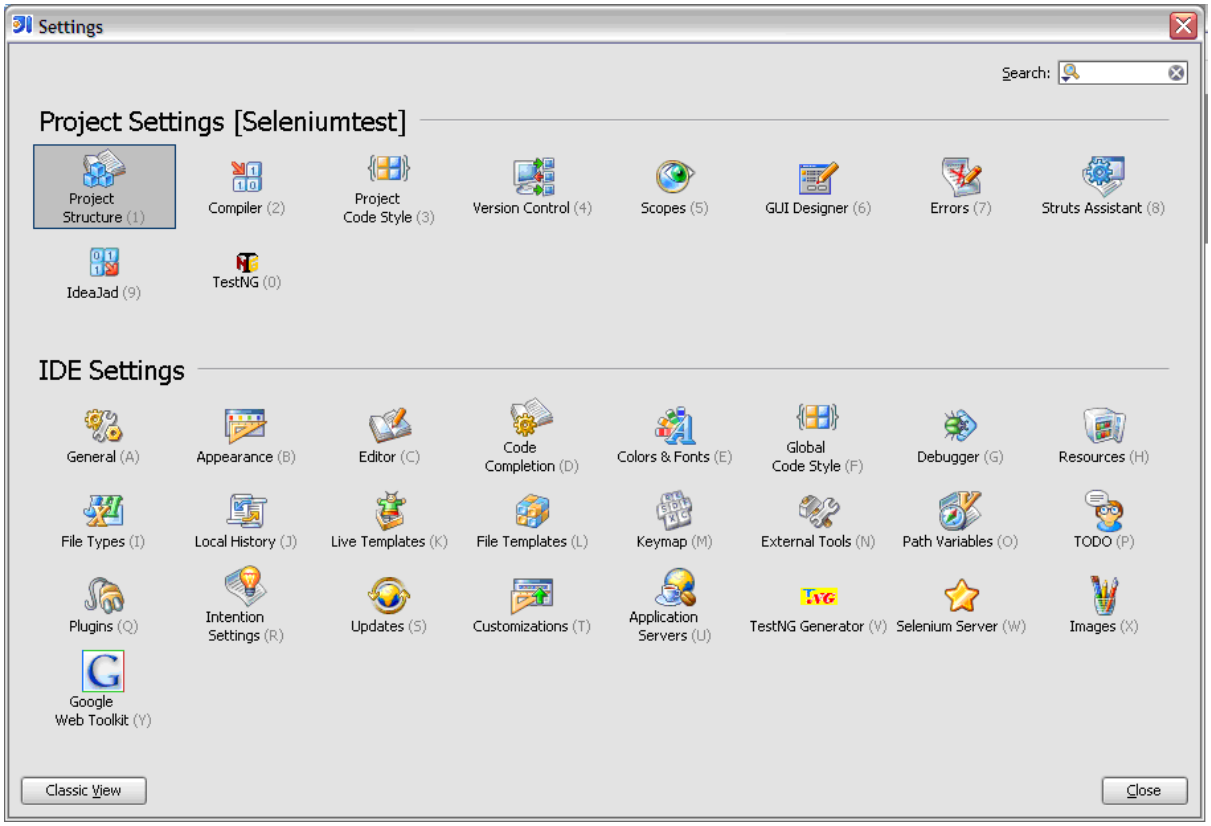


Adding Libraries to Project:

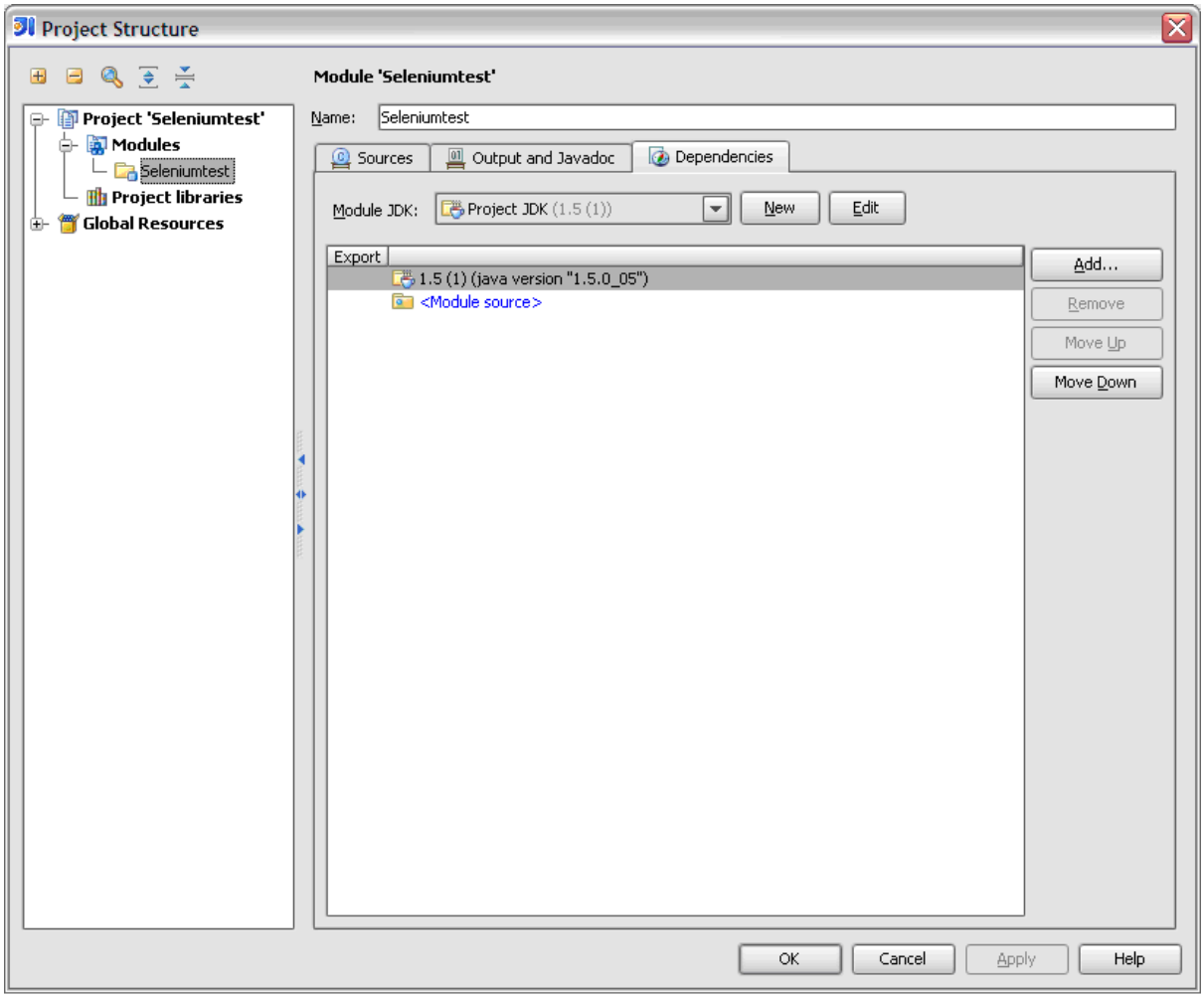
- Click on *Settings* button in the Project Tool bar.



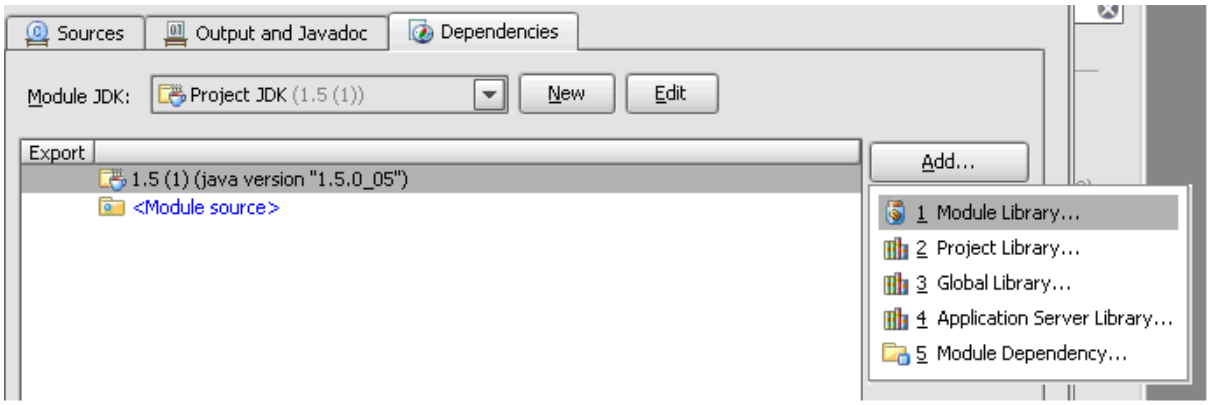
- Click on *Project Structure* in Settings pan.



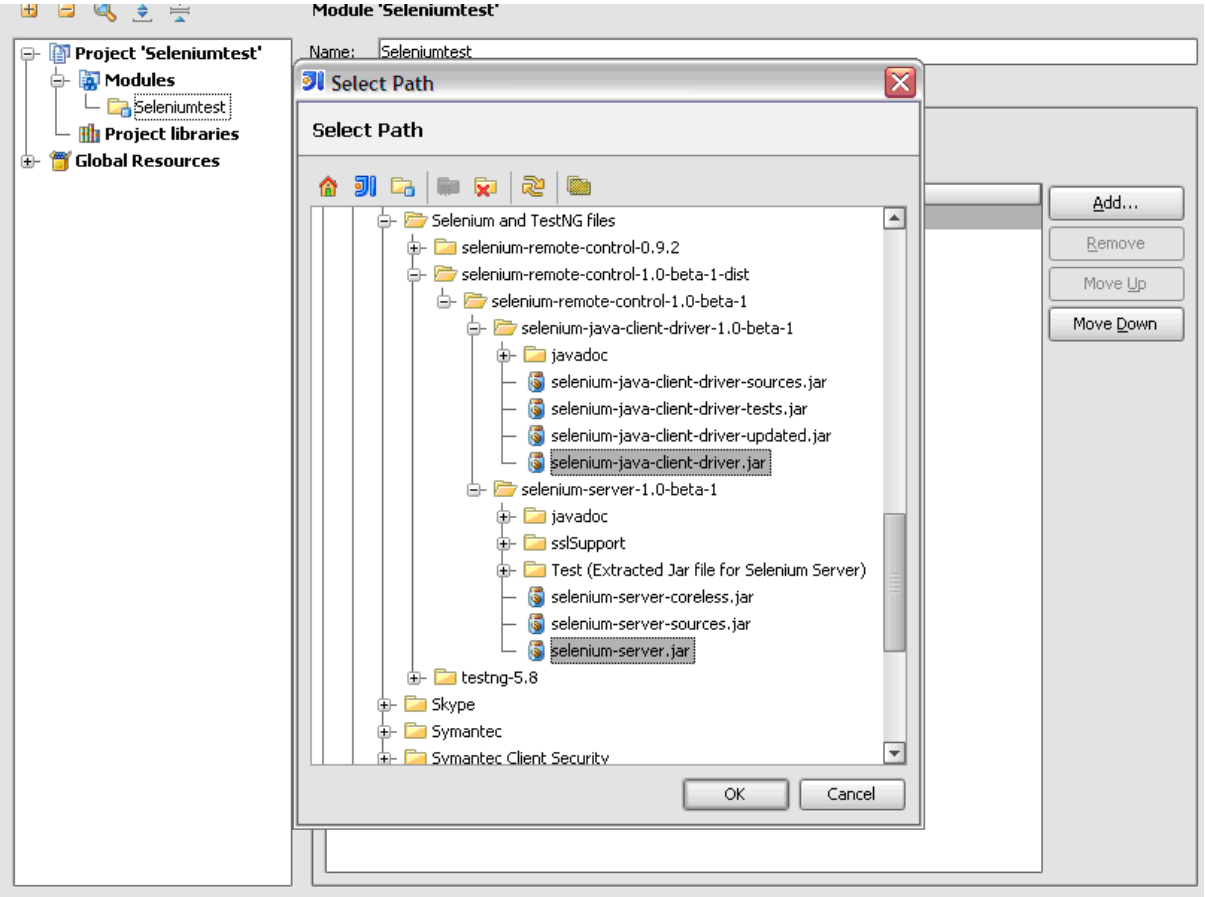
- Select *Module* in Project Structure and browse to *Dependencies* tab.



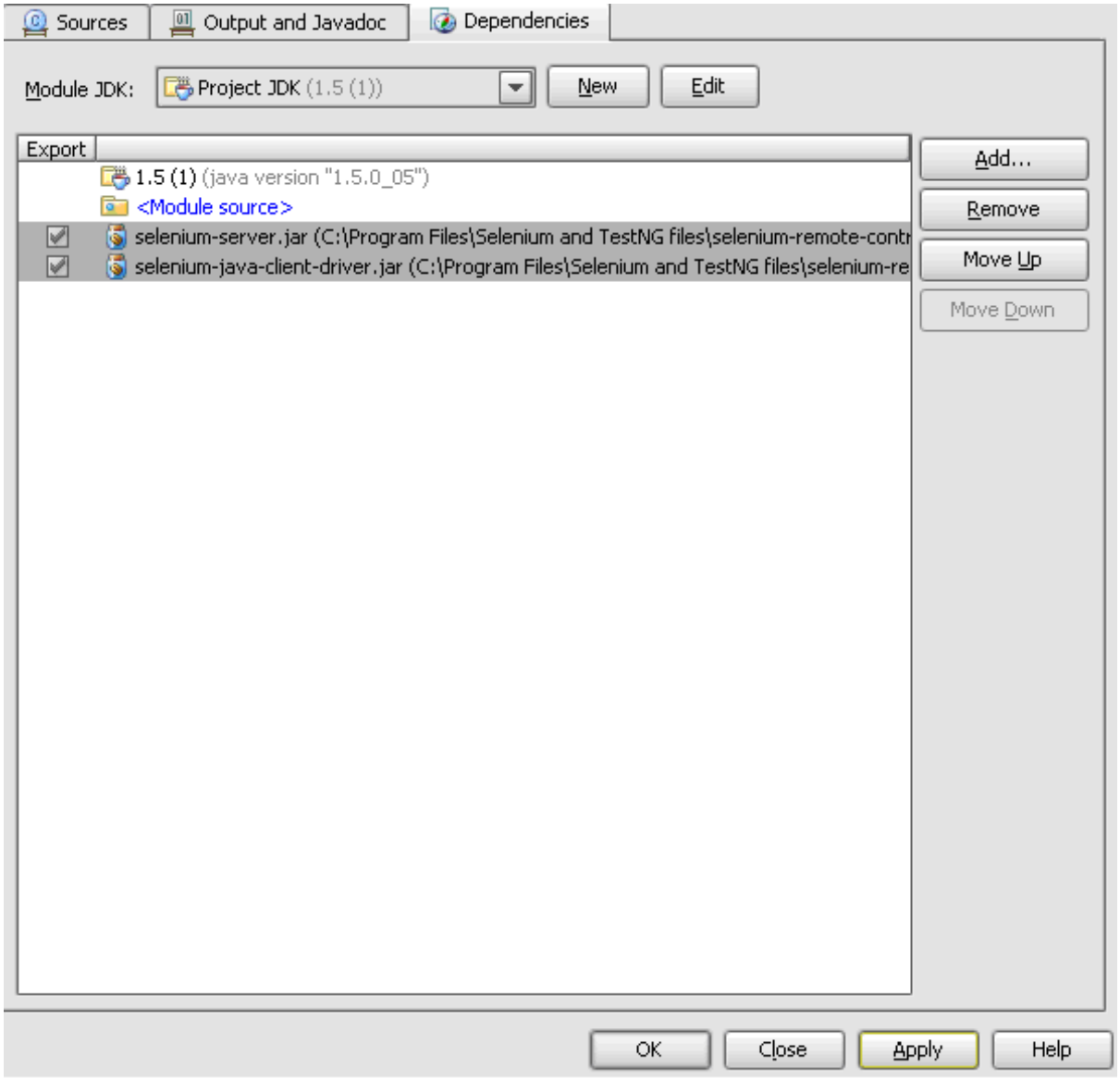
- Click on Add button followed by click on Module Library.



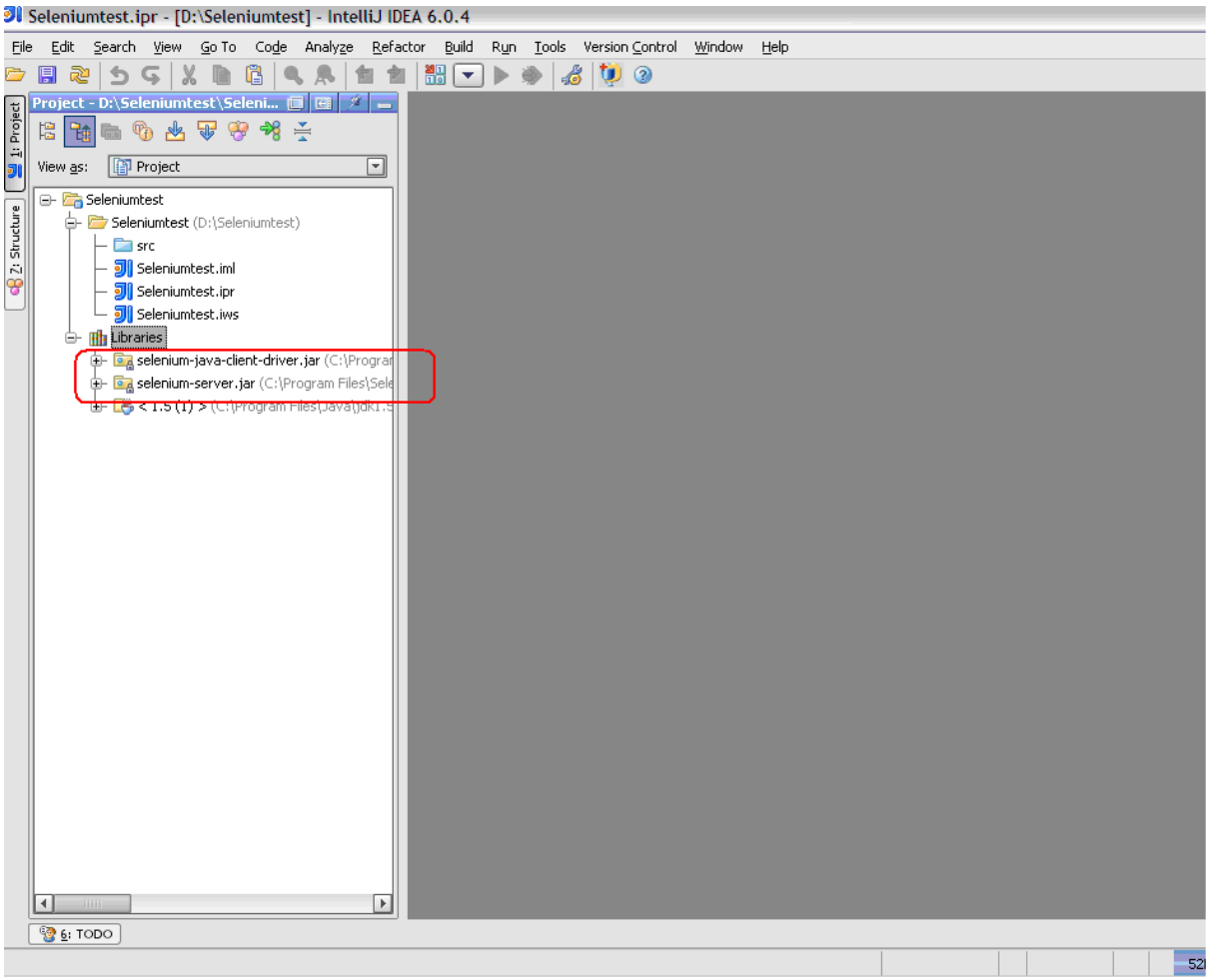
- Browse to the Selenium directory and select selenium-java-client-driver.jar and selenium-server.jar. (Multiple Jars can be selected b holding down the control key.).



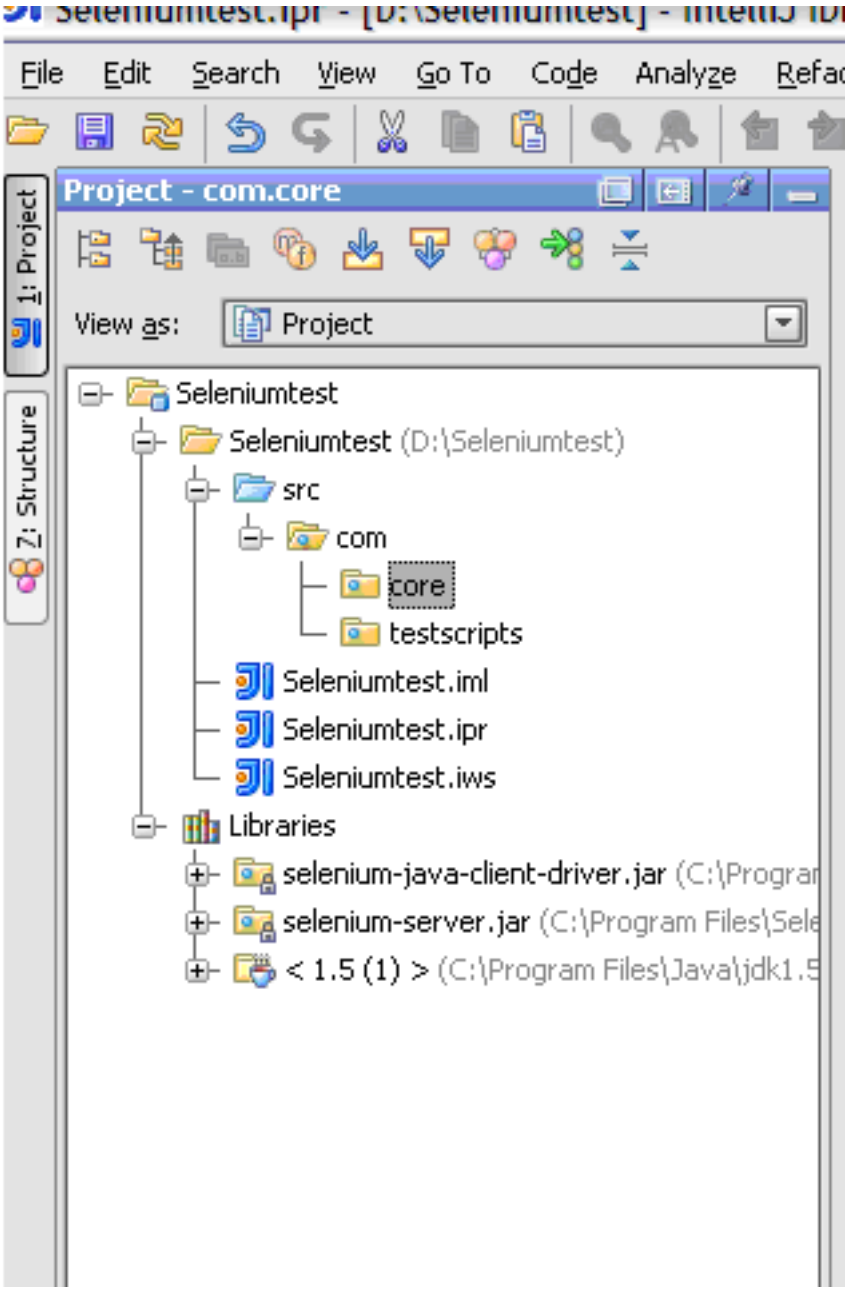
- Select both jar files in project pan and click on *Apply* button.



- Now click ok on Project Structure followed by click on Close on Project Settings pan. Added jars would appear in project Library as following.

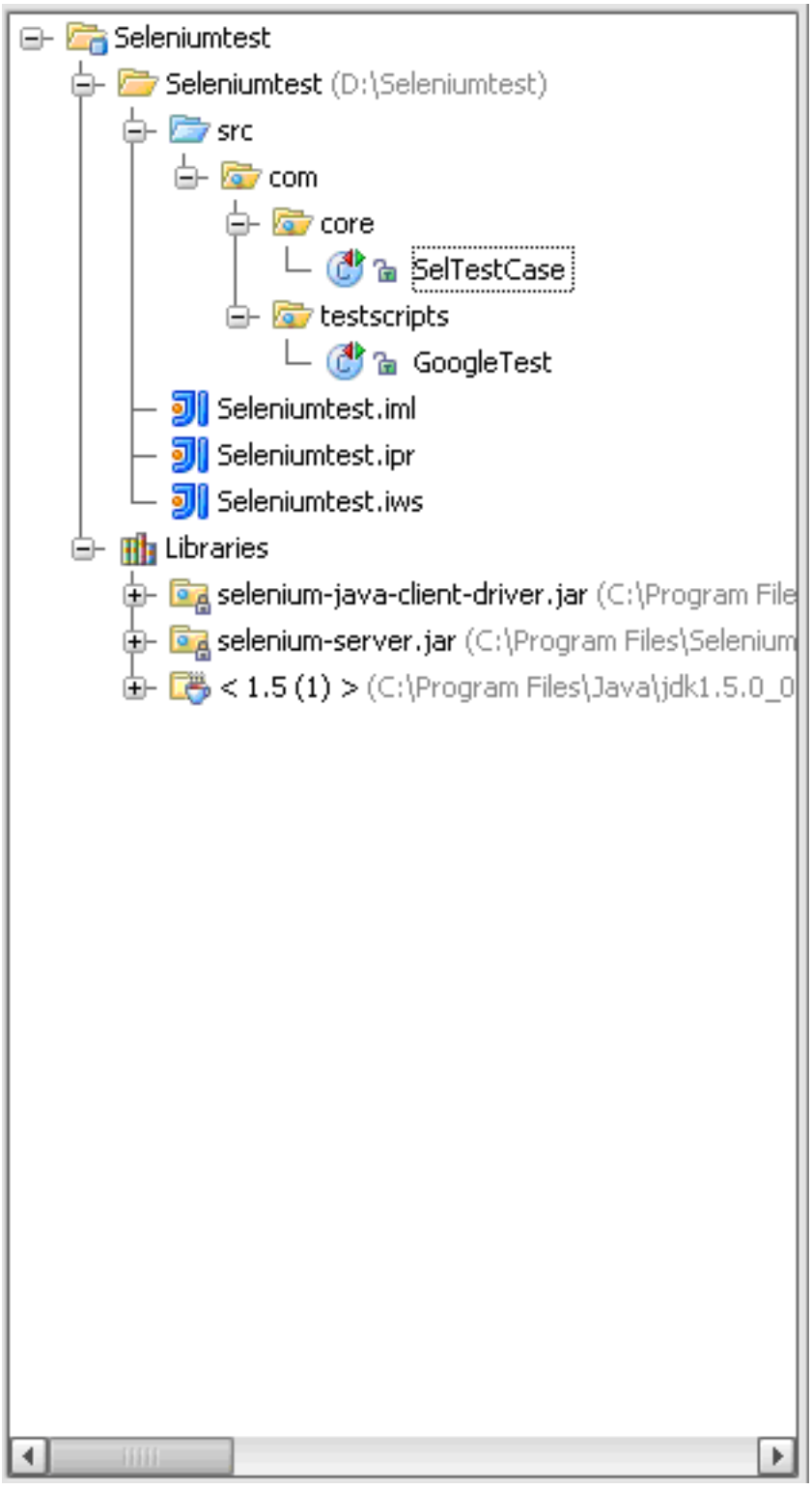


- Create the directory structure in src folder as following.



Note: This is not hard and fast convention and might very from project to project.

- Herein *core* contains the SelTestCase class which is used to create Selenium object and fire up the browser. *testscripts* package contains the test classes which extend the SelTestCase class. Hence extended structure would look as following.


Seleniumtest
Seleniumtest (D:\Seleniumtest)
src
com
core
SelTestCase
testscripts
GoogleTest
Seleniumtest.iml
Seleniumtest.ipr
Seleniumtest.iws
Libraries
selenium-java-client-driver.jar (C:\Program File
selenium-server.jar (C:\Program Files\Selenium
< 1.5 (1) > (C:\Program Files\Java\jdk1.5.0_0

CHAPTER

TWELVE

PYTHON CLIENT DRIVER CONFIGURATION

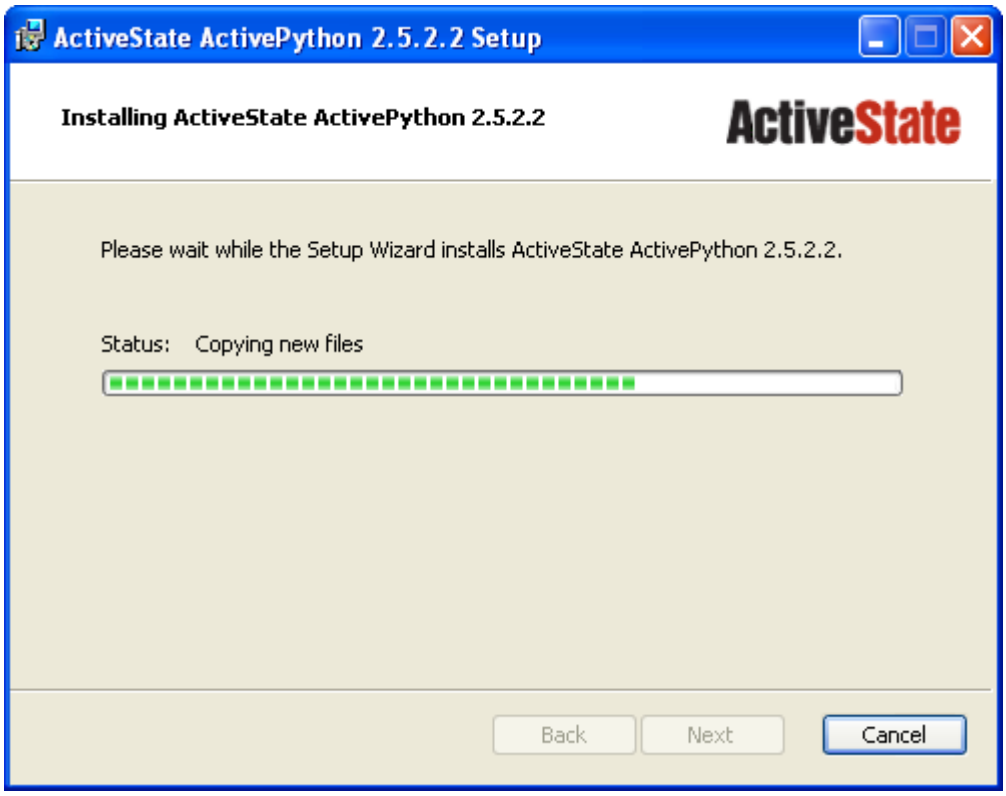
- Download Selenium-RC from the SeleniumHQ downloads page
- Extract the file *selenium.py*
- Either write your Selenium test in Python or export a script from Selenium-IDE to a python file.
- Add to your test's path the file *selenium.py*
- Run Selenium server from the console
- Execute your test from a console or your Python IDE

The following steps describe the basic installation procedure. After following this, the user can start using the desired IDE, (even write tests in a text processor and run them from command line!) without any extra work (at least on the Selenium side).

- Installing Python

  Note: This will cover python installation on Windows and Mac only, as in most linux distributions python is already pre-installed by default.

  - Windows
    1. Download Active python's installer from ActiveState's official site: http://activestate.com/Products/activepython/index.mhtml
    2. Run the installer downloaded (ActivePython-x.x.x.x-win32-x86.msi)



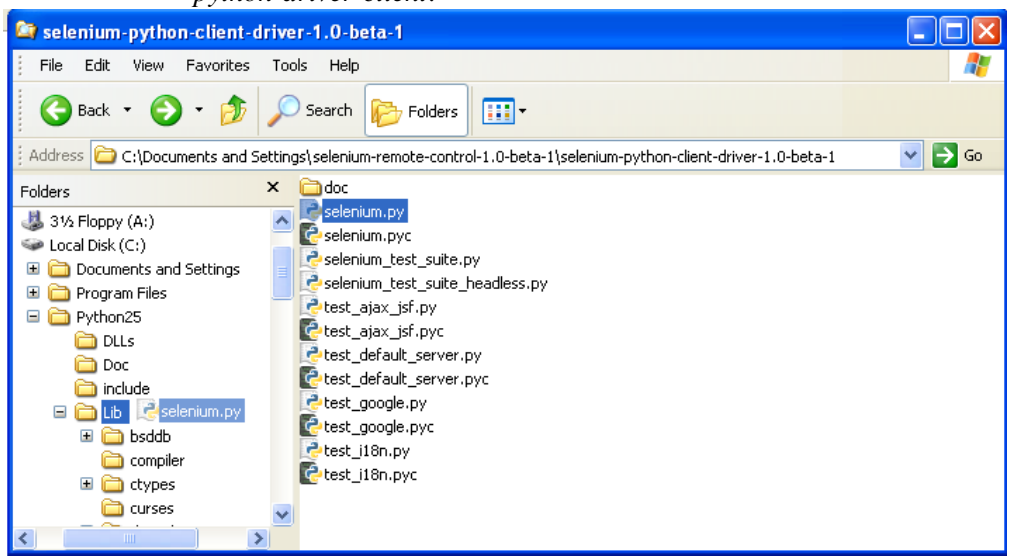
- Mac

  The latest Mac OS X version (Leopard at this time) comes with Python pre-installed. To install an extra Python, get a universal binary at http://www.pythonmac.org/ (packages for Python 2.5.x).

You will get a .dmg file that you can mount. It contains a .pkg file that you can launch.

- Installing the Selenium driver client for python

    1. Download the last version of Selenium Remote Control from the downloads page
    2. Extract the content of the downloaded zip file
    3. Copy the module with the Selenium's driver for Python (selenium.py) in the folder *C:/Python25/Lib* (this will allow you to import it directly in any script you write). You will find the module in the extracted folder, it's located inside *selenium-python-driver-client*.



Congratulations, you're done! Now any python script that you create can import selenium and start interacting with the browsers.

CHAPTER

THIRTEEN

LOCATING TECHNIQUES

13.1 Useful XPATH patterns

13.1.1 text

Not yet written - locate elements based on the text content of the node.

13.1.2 starts-with

Many sites use dynamic values for element's id attributes, which can make them difficult to locate. One simple solution is to use XPath functions and base the location on what you do know about the element. For example, if your dynamic ids have the format `<input id="text-12345" />` where `12345` is a dynamic number you could use the following XPath: `//input[starts-with(@id, 'text-')]`

13.1.3 contains

If an element can be located by a value that could be surrounded by other text, the contains function can be used. To demonstrate, the element `<span class="top heading bold">` can be located based on the 'heading' class without having to couple it with the 'top' and 'bold' classes using the following XPath: `//span[contains(@class, 'heading')]`. Incidentally, this would be much neater (and probably faster) using the CSS locator strategy `css=span.heading`

13.1.4 siblings

Not yet written - locate elements based on their siblings. Useful for forms and tables.

13.2 Starting to use CSS instead of XPATH

13.2.1 Locating elements based on class

In order to locate an element based on associated class in XPath you must consider that the element could have multiple classes and defined in any order, however with CSS locators this is much simpler (and faster).

- XPath: `//div[contains(@class, 'article-heading')]`

- CSS: `css=div.article-heading`

CHAPTER

FOURTEEN

MIGRATING FROM SELENIUM RC TO SELENIUM WEBDRIVER

A common question when adopting Selenium 2 is what's the correct thing to do when writing new tests in code. Users who are new to the framework can begin by using the new WebDriver APIs for writing their tests. But what of users who already have suites of existing tests? This guide is designed to demonstrate how to migrate your existing tests to the new APIs, allowing all new tests to be written using the new features offered by WebDriver.

The method presented here describes a piecemeal migration to the WebDriver APIs without needing to rework everything in one massive push. This means that the process may take a significant amount of time, and makes it easier for you to decide where to spend your effort.

This guide is written using Java, because this has the best support for making the migration. As we provide better tools for other languages, this guide shall be expanded to include those languages.

14.1 Why Migrate to WebDriver

Moving a suite of tests from one API requires an enormous amount of effort. Why would you and your team consider making this move?

- Smaller, compact API. WebDriver's API is more Object Oriented than the original Selenium RC API. This can make it easier to work with.
- Better emulation of user interactions. Where possible, WebDriver makes use of native events in order to interact with a web page. This more closely mimics the way that your users work with your site and apps. In addition, WebDriver offers the advanced user interactions APIs which allow you to model complex interactions with your site.
- Support by browser vendors. Opera, Mozilla and Google are all active participants in WebDriver's development, and each have engineers working to improve the framework. Often, this means that support for WebDriver is baked into the browser itself: your tests run as fast and as stably as possible.

14.2 Before Starting

In order to make the process of migrating as painless as possible, make sure that all your tests run properly with the latest Selenium release. This may sound obvious, but it's best to have it said!

14.3 Getting Started

The first step when starting the migration is to change how you obtain your instance of Selenium. When using Selenium RC, this is done like so:

```
Selenium selenium = new DefaultSelenium(
    "localhost", 4444, "*firefox", "http://www.yoursite.com");
selenium.start();
```

This should be replaced like so:

```
WebDriver driver = new FirefoxDriver();
Selenium selenium = new WebDriverBackedSelenium(driver, "http://www.yoursite.com");
```

Once you've done this, run your existing tests. This will give you a fair idea of how much work needs to be done. The Selenium emulation is good, but it's not completely perfect, so it's completely normal for there to be some bumps and hiccups.

14.4 Next Steps

Once your tests are running green again, the next stage is to migrate the actual test code to use the WebDriver APIs. Depending on how well abstracted your code is, this might be a short process or a long one. In either case, the approach is the same and can be summed up simply: modify code to use the new API when you come to edit it.

14.5 Common Problems

Fortunately, you're not the first person to go through this migration, so here are some common problems that others have seen, and how to solve them.

14.5.1 Clicking and Typing is More Complete

A common pattern in a Selenium RC test is to see something like:

```
selenium.type("name", "exciting tex");
selenium.keyDown("name", "t");
selenium.keyPress("name", "t");
selenium.keyUp("name", "t");
```

This relies on the fact that "type" simply replaces the content of the identified element without also firing all the events that would normally be fired if a user interacts with the page. The final direct invocations of "key*" cause the JS handlers to fire as expected.

When using the WebDriverBackedSelenium, the result of filling in the form field would be "exciting texttt": not what you'd expect! The reason for this is that WebDriver more accurately emulates user behavior, and so will have been firing events all along.

This same fact may sometimes cause a page load to fire earlier than it would do in a Selenium 1 test. You can tell that this has happened if a "StaleElementException" is thrown by WebDriver.

14.5.2 WaitForPageToLoad Returns Too Soon

Discovering when a page load is complete is a tricky business. Do we mean "when the load event fires", "when all AJAX requests are complete", "when there's no network traffic", "when document.readyState has changed" or something else entirely?

WebDriver attempts to simulate the original Selenium behavior, but this doesn't always work perfectly for many reasons. The most common reason is that it's hard to tell the difference between a page load not having started yet, and a page load having completed between method calls. This sometimes means that control is returned to your test before the page has finished (or started!) loading.

The solution to this is to wait on something specific. Commonly, this might be for the element you want to interact with next, or for some Javascript variable to be set to a specific value. An example would be:

```
Wait<WebDriver> wait = new WebDriverWait(driver, 30);
WebElement element= wait.until(visibilityOfElementLocated(By.id( "some_id" )));
```

Where "visibilityOfElementLocated" is implemented as:

```
public ExpectedCondition<WebElement> visibilityOfElementLocated(final By locator) {
  return new ExpectedCondition<WebElement>() {
    public WebElement apply(WebDriver driver) {
      WebElement toReturn = driver.findElement(locator);
      if (toReturn.isDisplayed()) {
        return toReturn;
      }
      return null;
    }
  };
}
```

This may look complex, but it's almost all boiler-plate code. The only interesting bit is that the "ExpectedCondition" will be evaluated until the "apply" method returns something that is neither "null" nor Boolean.FALSE.

Of course, adding all these "wait" calls may clutter up your code. If that's the case, and your needs are simple, consider using the implicit waits:

```
driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
```

By doing this, every time an element is located, if the element is not present, the location is retried until either it is present, or until 30 seconds have passed.

14.5.3 Finding By XPath or CSS Selectors Doesn't Always Work, But It Does In Selenium 1

In Selenium 1, it was common for xpath to use a bundled library rather than the capabilities of the browser itself. WebDriver will always use the native browser methods unless there's no alternative. That means that complex xpath expressions may break on some browsers.

CSS Selectors in Selenium 1 were implemented using the Sizzle library. This implements a superset of the CSS Selector spec, and it's not always clear where you've crossed the line. If you're using the WebDriverBackedSelenium and use a Sizzle locator instead of a CSS Selector for finding elements, a

warning will be logged to the console. It's worth taking the time to look for these, particularly if tests are failing because of not being able to find elements.

14.5.4 There is No Browserbot

Selenium RC was based on Selenium Core, and therefore when you executed Javascript, you could access bits of Selenium Core to make things easier. As WebDriver is not based on Selenium Core, this is no longer possible. How can you tell if you're using Selenium Core? Simple! Just look to see if your "getEval" or similar calls are using "selenium" or "browserbot" in the evaluated Javascript.

You might be using the browserbot to obtain a handle to the current window or document of the test. Fortunately, WebDriver always evaluates JS in the context of the current window, so you can use "window" or "document" directly.

Alternatively, you might be using the browserbot to locate elements. In WebDriver, the idiom for doing this is to first locate the element, and then pass that as an argument to the Javascript. Thus:

```
String name = selenium.getEval(
    "selenium.browserbot.findElement('id=foo', browserbot.getCurrentWindow()).tagName");
```

becomes:

```
WebElement element = driver.findElement(By.id("foo"));
String name = (String) ((JavascriptExecutor) driver).executeScript(
    "return arguments[0].tagName", element);
```

Notice how the passed in "element" variable appears as the first item in the JS standard "arguments" array.

14.5.5 Executing Javascript Doesn't Return Anything

WebDriver's JavascriptExecutor will wrap all JS and evaluate it as an anonymous expression. This means that you need to use the "return" keyword:

```
String title = selenium.getEval("browserbot.getCurrentWindow().title");
```

becomes:

```
((JavascriptExecutor) driver).executeScript("return window.title;");
```