Name of system: Selenium WebDriver

Reviewer: Thang Minh Le

Date: October 16, 2011

Author of software: Jason Huggins (Selenium) and Simon Stewart (WebDriver)

Author of book chapter: Simon Stewart

Five star rating of book chapter: 2

Purpose of the system: Selenium is a browser automation tool which is used to test a web application from its web pages. Doing user testing for a web application requires a tester to enter input data on each screen to test its functionality. This sort of testing is very time consuming especially on a large web application. It also requires human being involved, which prevents the testing process to be automated. Selenium is an effort to enable automated testing for a web application. Conceptually, the tool can be seen as a robot which simulates tester's activities such as typing input data, clicking buttons…
Selenium is a suit of three tools:

- Selenium IDE: an extension for FireFox which allows users to record and playback
- Selenium WebDriver: is the core the Selenium which provides API to control web browser
- Selenium Grid: makes it possible to perform browser automation over a grid of machines

Selenium has a very interesting design rooted from the history of Selenium. Selenium started off with the initial work from Jason Huggins during the time he was at ThoughtWorks. The tool was coded entirely in Javascript to synthesize user's events. It is easy to understand why Jason decided to use Javascript for his tool since almost all web browsers support Javascript embedded in a html page. Using Javascript provides a lot of advantages to Selenium in supporting multiple language bindings. However, the biggest drawback is all scripts loaded from a site are executed inside Javascript security sandbox, so-called "Same Origin Policy" of web browser. Independently, Simon Stewart came up later with WebDriver tool for the same purpose but in different direction. WebDriver tool attempts to simulate user's activities in a native way bound to web browsers. This allows the tool to side-step the security problem as in Selenium and has a better control over the browser. However, interacting with a browser in its native API is an unpleasant task to developers. Maintainability is also a big concern. In August 2009, both projects were merged to provide users to best of the two worlds.

Basic metrics

KLOC: Selenium server version 2.8.0:

Language	files	blank	comment	code
Java	931	19198	34383	80686
SUM:	931	19198	34383	80686

Selenium WebDriver Client version 2.8.0:

Language	files	blank	comment	code
Java	432	6970	11542	21868
SUM:	432	6970	11542	21868

Project start-up: initial work of Selenium was started in 2004. WebDriver was started in 2007. The merge of Selenium & WebDriver was in August 2009

Number of major releases: there have been two major releases 1.0 and 2.0

Number of developers: there are currently 1007 developers actively working on Selenium

Size of user community or number of installations: 615577

Major stakeholders: many companies are using this in their automated testing framework. Google is an example. They are user stakeholder of Selenium. Since the project is open source, developers are another major stakeholder group.

Use of concurrency: there is a small concurrency during setting up WebDriver instance when running test. This is in the case of FireFox WebDriver (http://code.google.com/p/selenium/wiki/FirefoxDriverInternals)
Flow of Control: Starting Firefox
The following steps are performed when instantiating an instance of the FirefoxDriver:

- Grab the "locking port"
- By default, the firefox driver attempts to use port 7055 to communicate
- Subtract 1 from the port (7055 - 1 = 7054) This is the locking port, which we use as a mutex to guarantee that only one firefox instance is started at the same time.
- Sit in a loop waiting for the locking port to become free.
- Exit the loop when we can establish a server socket (that is bind to) the locking port. This driver instance now has the mutex
- Find the next free port up from the locking port
- Identify the next free port above the locked one by attempting to bind to each of them in turn. Release the newly found port. For the first driver, this will be 7055. For the second, 7056.
- Locate the Firefox profile to use. This is normally an anonymous one
- Copy the existing profile to a temporary directory, ignoring any lock files that indicate that Firefox is actually running.
- Delete the current "extensions.cache" to force Firefox to look for newly installed extensions on start up.
- Update "user.js" with the preferences required to make WebDriver work as expected. Set the "webdriver_firefox_port" value to the one that we found in the step 2.
- Start Firefox with the "-silent" flag and telling it to use the freshly minted directory as the profile. This causes Firefox to start up, look at the profile and fix it up if necessary.
- Wait until that Firefox instance has finished and the process has died.
- Start firefox again and repeatedly attempt to connect until we're successful or too much time has passed.
- Release the locking port. The mutex has now been freed.

After the start-up, every running test is running in different instance and hence, there should have no concurrency.

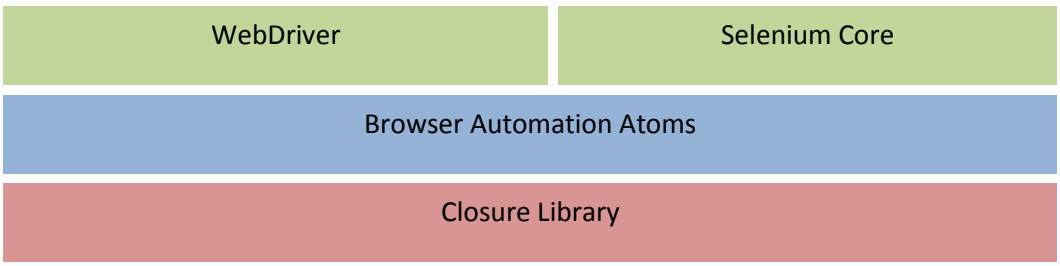
Implementation language: WebDriver (part of Selenium) tries to talk with browser in the browser's native language. Hence, the languages used in WebDriver are varied:

- FireFox WebDriver uses Java, JavaScript under XPCOM platform (Mozilla cross platform COM)
- IE WebDriver uses Visual C++, C#

Supporting software: Selenium can be run on many major operating systems including Windows, Linux, Mac. It also supports many programming languages to create test cases including Java, C#, Ruby, Python. Web browsers supported are FireFox, Internet Explorer, Chrome, Opera, iPhone, Android.

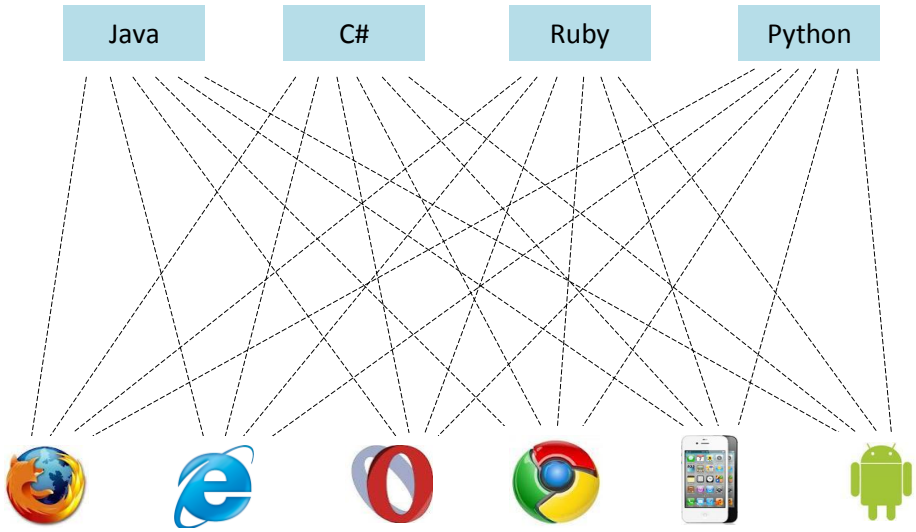
High level architecture

Diagram of software architecture

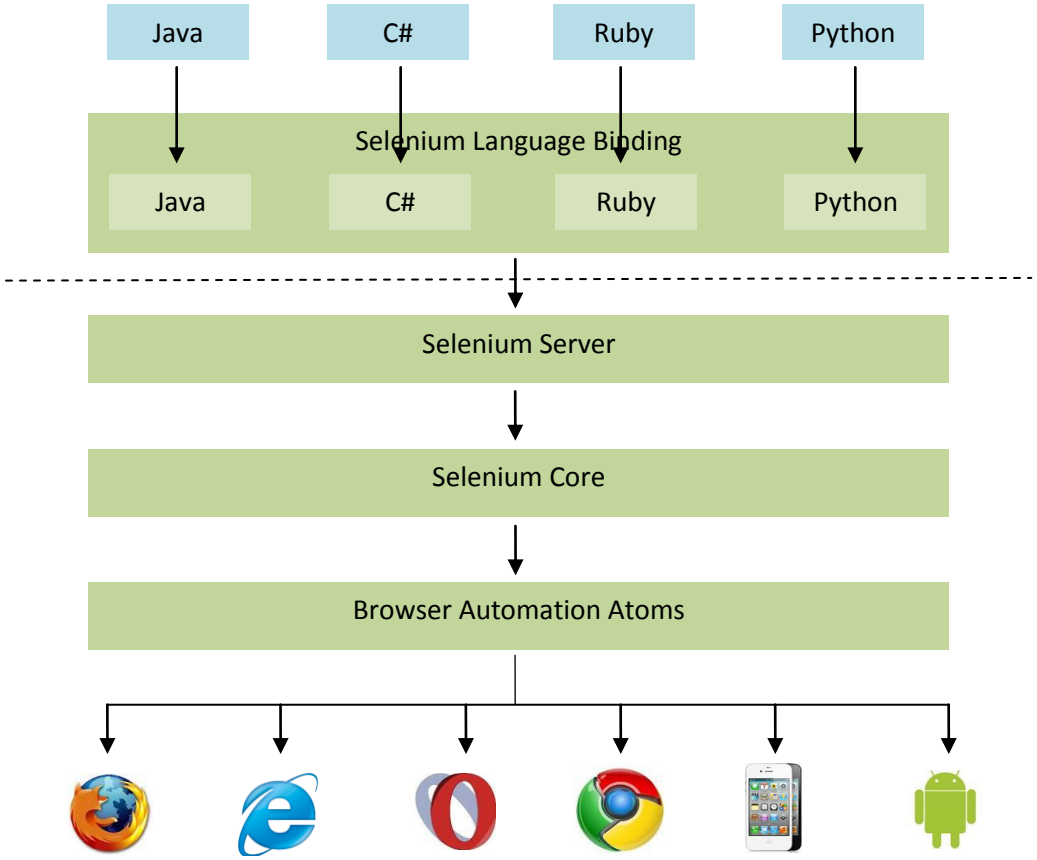


- Closure Library: a compiler targeting Javascript which performs organizing dependency order, pretty printing, optimization and de code removal.
- Browser Automation Atoms: a building block provides functions to interrogate the DOM. Atoms is mostly written in Javascript as a result of refactoring effort to put common functionalities from both WebDriver and Selenium Core into one place to reduce the cost of maintenance.
- WebDriver: a module contains implementation of web drivers for supported browsers. It also provides WebDriver API for users to build their tests interact with browsers.
- Selenium Core: the original Selenium which has most of its implementation written in Javascript.

High level scenarios:
Selenium has to support executing tests written in multiple languages target to multiple web browsers. So the combinatorial cases are as follow:



Selenium Core Solution:

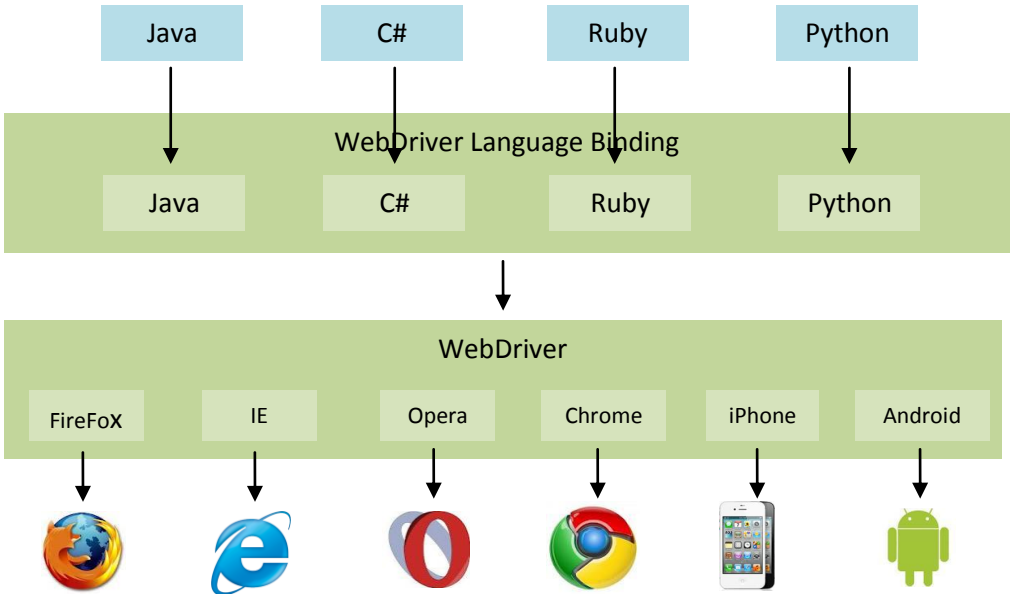


Selenium Core solution to the combinatorial explosion problem takes a client/server approach. Each programming language has a separated Selenium language binding. Each Selenium language binding is responsible to translate Selenium APIs to an appropriate Selenium commands which have the format similar to ActionFixture from FIT. Each command is split into three columns: name, element identifier and value. For example: the command belows is to type "Selenium WebDriver" to element name 'q'
Command: "type name=q Selenium WebDriver"

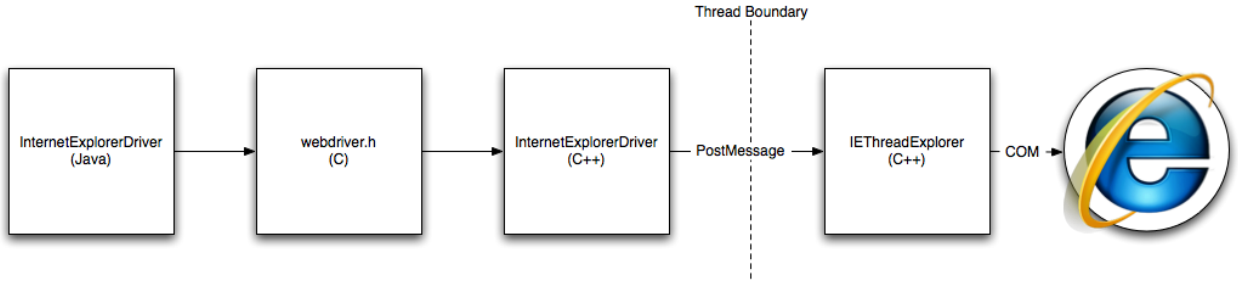
The test commands will be sent to Selenium Server using HTTP. Upon receiving the command, the server will pass it on to Selenium Core which then uses Browser Automation Atoms to translate the commands to an appropriate javascripts and will get fired by web browsers.

In this approach, each test case has the same set of javascripts for all web browsers.

WebDriver Solution:



WebDriver solution takes Object Orient approach to the combinatorial explosion. The solution provides a separated language binding for each supported language. In addition, it also has custom highly optimized web driver for each supported browser. The reason to built-in web driver for each browser is to fully control browser and have the ability to truly emulate user's activities. Initially, each running test will interact with appropriate WebDriver language binding for its programming language. The language binding will then contact with proper WebDriver client to initiate browser instance. Then, WebDriver client will communicate with the browser instance in the browser's native APIs. In cases when there is no suitable native APIs for a browser, WebDriver will attempt to generate user's activities at OS level. Such events are keyboard types and mouse clicks.



The case of IE is interesting. IE browser has its native API supporting web automation written in C++ which is wrapped in Windows COM packages. WebDriver for IE is written in pure Java. The difference in programming language is the big obstacle in communication between the driver and IE browser instance. In order to successfully communicate with IE browser, the IE WebDriver makes a native call to C wrapper using Java Native Architecture (JNA). The call will then be forwarded to a IE driver written in C++.

The C code in webdriver.h is ultimately a thin wrapper around C++ classes that model the the Object-based design of the Java code (the InternetExplorerDriver (C++) and IEThreadExplorer (C++) in the diagram) The underlying IE COM interfaces are designed for use in a Single Thread Apartment (STA)

model. That is, the COM object must only ever be accessed from a single thread. Unfortunately, we cannot control how many threads call into our implementing library, and there are perfectly reasonable occasions where more than one thread may attempt to call the underlying library, even if in serial fashion (for example, the RemoteWebDriver is hosted in a servlet container, where many threads may be in use) In order to isolate the COM interfaces, we pass messages across a thread boundary using the Win32 PostMessage API. (http://code.google.com/p/selenium/wiki/InternetExplorerDriverInternals)

Architectural style: Selenium is organized in layer architecture.

Major evolutionary changes:

- The merge of Selenium and WebDriver
- Change RPC communication to HTTP communicate in remote WebDriver

Performance bottlenecks: WebDriver controls browsers that are running in other processes. Although it's easy to overlook it, this means that every call that is made through its API is an RPC call and therefore the performance of the framework is at the mercy of network latency. In normal operation, this may not be terribly noticeable—most OSes optimize routing to localhost—but as the network latency between the browser and the test code increases, what may have seemed efficient becomes less so to both API designers and users of that API.

Real time: There is no real time aspect from Selenium.

Notation for architecture: it uses Booch notation

Methodology: not clear

Appendix

1. Size: M
2. Criticality: Med
3. Age of system: M
4. Rate of change: Hi
5. Business model: open source
6. Stable architecture: Med
7. Team distribution: VH
8. Governance: Hi=Distinct separate management and design teams