CONTENTS INCLUDE:

Selenium 2.0:

Using the Webdriver API to Create Robust User Acceptance Tests

By Matt Stine

WHAT IS SELENIUM 2.0?

Selenium 2.0 is a tool that makes the development of automated tests for web sites and web applications easier. It represents the merger of the original Selenium project with the WebDriver project. WebDriver contributes its object-oriented API for Document Object Model (DOM) interaction and browser control as well as its various browser driver implementations.

The WebDriver approach to browser control differs significantly from the Selenium approach. Whereas Selenium runs as a JavaScript application inside the targeted browser, WebDriver drives the browser directly through the most sensible means available. For Firefox, this meant using JavaScript wrapped in an XPCOM component; however, for Internet Explorer, this meant using C++ to drive IE's Automation APIs.

The merger of these two projects allows each technology's strengths to mitigate the other's weaknesses. For example, Selenium has good support for most common use cases, but WebDriver opens up many additional possibilities via its ability to step outside of the JavaScript sandbox. Going forward, both of these APIs will live side by side in a synergistic relationship, allowing automated test developers to easily leverage the right tool for the job.

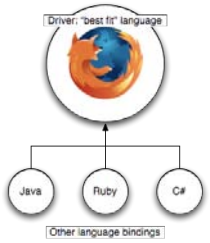
Selenium 2.0 currently supports writing cross-browser tests in Java (and other languages on the JVM including Groovy), Python, Ruby, and C#. It currently allows developers to target Firefox, Internet Explorer, and Chrome with automated tests. It also can be used with HtmlUnit for "headless" application testing. Furthermore, driver implementations exist for the iOS (iPhone/iPad), Android, and BlackBerry platforms.

The remainder of this Refcard will focus on the WebDriver contributions to the Selenium 2.0 project and will take a brief look at how to leverage the Selenium API via WebDriver.

ARCHITECTURE

The WebDriver team chose to focus on four primary architecture concerns:

1. **The User:** WebDriver drives the browser from your end user's point of view.
2. **A "Best Fit" Language:** each browser has a language that is most natural to use from an automation perspective. The various drivers are implemented as much as possible in that language.
3. **A Layered Design:** developers should be able to write their tests in the supported language of their choice, and these tests should work with all driver implementations. Therefore, the developer API exists as a thin wrapper around the core of each driver.



4. **Reduction in Cost of Change:** share as much code as possible between the driver projects. This is accomplished via the use of "Automation Atoms," which are JavaScript libraries implementing various read-only DOM query tasks. These Atoms are used for two purposes. The first is to compose a monolithic driver (such as the FireFox driver) written primarily in JavaScript. The second is to augment existing drivers written in other languages with tiny fragments of highly compressed JavaScript. This augmentation will reduce execution and parsing time.

INSTALLATION

Java

From http://code.google.com/p/selenium/downloads/list download the following two files and add them to your classpath (version numbers are the latest as of this writing):

- selenium-server-standalone-2.0a7.jar contains the Selenium server (required to use Selenium 1.x classes) and the RemoteWebDriver server, which can be executed via java -jar.

- selenium-java-2.0a7.zip contains the Java language bindings for Selenium 2.0. All of the necessary dependencies are bundled.

Java (Maven)

Add the following to your pom.xml:

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium</artifactId>
  <version>2.0a7</version>
</dependency>
```

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-server</artifactId>
  <version>2.0a7</version>
</dependency>
```

Ruby

Run the following command:

```
gem install selenium-webdriver
```

Python

Run one of the following commands:

```
easy_install selenium
```

or

```
pip install selenium
```

C#

Download selenium-dotnet-2.0a6.zip and add references to WebDriver.Common.dll and WebDriver.Firefox.dll (or your browser of choice) to your Visual Studio project.

DRIVER IMPLEMENTATIONS

There are currently four official driver implementations for desktop browsers:

Firefox

The Firefox driver is the most mature of the browser-based drivers.

From Java:

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

From Ruby:

```
driver = Selenium::WebDriver.for :firefox
```

From Python:

```
from selenium.firefox.webdriver import WebDriver
driver = WebDriver()
```

From C#:

```
IWebDriver driver = new FirefoxDriver();
```

Internet Explorer

The Internet Explorer driver has been tested and is known to work on IE 6, 7, and 8 on Windows XP and Vista. Compared to the other drivers, it is relatively slow.

From Java:

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

From Ruby:

```
driver = Selenium::WebDriver.for :internet_explorer
```

From Python:

```
from selenium.ie.webdriver import WebDriver
driver = WebDriver()
```

From C#:

```
IWebDriver driver = new InternetExplorerDriver();
```

Chrome

The Chrome driver is comparatively new. Because Chrome is based on Webkit, you may be able to verify that your application works in other Webkit-based browsers such as Safari. However, because Chrome uses V8 JavaScript engine rather than the Safari Nitro engine, you may experience some differences in behavior.

From Java:

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

From Ruby:

```
driver = Selenium::WebDriver.for :chrome
```

From Python:

```
from selenium.chrome.webdriver import WebDriver
driver = WebDriver()
```

From C#:

```
IWebDriver driver = new ChromeDriver();
```

HtmlUnit

The HtmlUnit driver is the fastest and most lightweight of the group. It is the only pure Java implementation, so it is the only solution that will run anywhere the JVM is available. Because it utilizes HtmlUnit to interact with your application instead of driving an actual browser, it carries along the associated baggage:

- You cannot visually inspect what's going on during your test.
- HtmlUnit uses Rhino as its JavaScript+DOM implementation. No major browser does this, and it is disabled by default.
- If you do enable JavaScript, the HtmlUnit Driver will emulate the behavior of Internet Explorer.

From Java:

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

Although it isn't recommended, it is possible to configure the HtmlUnitDriver to emulate a specific browser:

```
HtmlUnitDriver driver = new HtmlUnitDriver(BrowserVersion.FIREFOX_3);
```

SELENIUM RC EMULATION

It is possible to emulate Selenium Remote Control (RC) using the WebDriver Java implementation on any of its supported browsers. This allows you to maintain both WebDriver and Selenium test assets side by side and to engage in an incremental migration from Selenium to WebDriver. In addition, running the normal Selenium RC server is not required.

```
WebDriver driver = new FirefoxDriver();

String baseUrl = "http://downforeveryoneorjustme.com/";
Selenium selenium = new WebDriverBackedSelenium(driver, baseUrl);

selenium.open("http://downforeveryoneorjustme.com/");
selenium.type("domain_input", "google.com");
selenium.click("//div[@id='container']/form/a");

assertTrue(selenium.isTextPresent("It's just you"));
```

On the downside, you may notice some inconsistencies in behavior since the entire Selenium API is not implemented (eg., keyPressNative is unsupported) and Selenium Core is emulated. You may also experience slower performance in some cases.

PAGE INTERACTION MODEL

After navigating to a page…

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

…WebDriver offers a plethora of means for interacting with the elements on that page via its findElement methods. These methods accept an argument of type By, which defines several static methods implementing different means of element location:

Locators

Locator	Example (Java)
id attribute	By.id("myElementId")
name attribute	By.name("myElementName")
XPATH	By.xpath("//input[@id='myElementId']")
Class name	By.className("even-table-row")
CSS Selector	By.cssSelector("h1[title]")
Link Text	By.linkText("Click Me!") By.partialLinkText("ck M")
Tag Name	By.tagName("td")

In each case, findElement will return an instance of WebElement, which encapsulates all of the metadata for that DOM element and allows you to interact with it.

WebElement Methods

Method	Purpose
clear()	Clears all of the contents if the element is a text entity.
click()	Simulates a mouse click on the element.
getAttribute(String name)	Returns the value associated with the provided attribute name (if present) or null (if not present).
getTagName()	Returns the tag name for this element.
getText()	Returns the visible text contained within this element (including subelements) if not hidden via CSS.
getValue()	Gets the value of the element's "value" attribute.
isEnabled()	Returns true for input elements that are currently enabled; otherwise false.
isSelected()	Returns true if the element (radio buttons, options within a select, and checkboxes) is currently selected; otherwise false.
sendKeys(CharSequence… keysToSend)	Simulates typing into an element.
setSelected()	Select an element (radio buttons, options within a select, and checkboxes).
submit()	Submits the same block if the element if the element is a form (or contained within a form). Blocks until new page is loaded.
toggle()	Toggles the state of a checkbox element.

In addition to this set of methods designed for interacting with the element in hand, WebElement also provides two methods allowing you to search for elements within the current element's scope:

Method	Purpose
findElement(By by)	Finds the first element located by the provided method (see Locators table).
findElements(By by)	Finds all elements located by the provided method.

WebDriver provides a support class named Select to greatly simplify interaction with select elements and their association options:

Method	Purpose
selectByIndex(int index)/ deselectByIndex(int index)	Selects/deselects the option at the given index.
selectByValue(String value)/ deselectByValue(String value)	Selects/deselects the option(s) that has a value matching the argument.
selectByVisibleText(String text)/ deselectByVisibleTest(String text)	Selects/deselects the option(s) that displays text matching the argument.
deselectAll()	Deselects all options.
getAllSelectedOptions()	Returns a List<WebElement> of all selected options.
getFirstSelectedOption()	Returns a WebElement representing the first selected option.
getOptions()	Returns a List<WebElement> of all options.
isMultiple()	Returns true if this is a multi-select list; false otherwise.

You can create an instance of Select from a WebElement that is known to represent a select element:

```
WebElement element = driver.findElement(By.id("mySelect"));
Select select = new Select(element);
```

If the element does not represent a select element, this constructor will throw an UnexpectedTagNameException.

Interacting with Rendered Elements

If you're driving an actual browser such as Firefox, you also can access a fair amount of information about the rendered state of an element by casting it to RenderedWebElement. This is also how you can simulate mouse-hover events and perform drag-and-drop operations.

```
WebElement element = driver.findElement(By.id("header"));
RenderedWebElement renderedElement = (RenderedWebElement) element;
```

RenderedWebElement Methods

Method	Purpose
dragAndDropBy(int moveRightBy, int moveDownBy)	Drags and drops the element moveRightBy pixels to the right and moveDownBy pixels down. Pass negative arguments to move left and up.
dragAndDropOn(RenderedWeb Element element)	Drags and drops the element on the supplied element.
getLocation()	Returns a java.awt.Point representing the top left-hand corner of the element.
getSize()	Returns a java.awt.Dimension representing the width and height of the element.
getValueOfCssProperty(String propertyName)	Returns the value of the provided property.
hover()	Simulates a mouse hover event over the element.
isDisplayed()	Returns true if the element is currently displayed; otherwise false.

PAGE OBJECT PATTERN

The Page Object pattern is a useful tool for separating the orthogonal concerns of testing the logical functionality of an application from the mechanics of interacting with that functionality. The Page Object pattern acts as an API to an HTML page and describes to the test writer the "services" provided by that page, while at the same time not exposing the underlying implementation. The Page Object itself is the only entity that possesses deep knowledge of the HTML's structure.

Consider the Google home page. The user is able to enter search terms and perform two possible actions: to conduct the search or to automatically navigate to the

top hit ("I'm Feeling Lucky). We could model this with the following Page Object (Java):

```
public class GoogleHomePage {
  private final WebDriver driver;

  public GoogleHomePage(WebDriver driver) {
    this.driver = driver;

    //Check that we're actually on the Google Home Page.
    if (!"Google".equals(driver.getTitle())) {
      throw new IllegalStateException("This isn't Google's Home Page!");
    }
  }

  public GoogleResultsPage search(String searchTerms) {
    driver.findElement(By.name("q")).sendKeys(searchTerms);
    driver.findElement(By.name("btnG")).submit();

    return new GoogleResultsPage(driver);
  }

  public UnknownTopHitPage imFeelingLucky(String searchTerms) {
    driver.findElement(By.name("q")).sendKeys(searchTerms);
    driver.findElement(By.name("btnI")).submit();

    return new UnknownTopHitPage(driver);
  }
}
```

Hot Tip

These examples assume the behavior of the Google home page prior to the introduction of Google Instant. For them to work properly, create a custom FireFox profile turn instant off in that profile, and then run your tests using that profile by setting the system property "firefox.browser.profile" to the name of your custom profile.

To learn more about the Page Object pattern, visit: http://code.google.com/p/selenium/wiki/PageObjects

WebDriver provides excellent support for implementing Page Object's via its PageFactory. The PageFactory supports a "convention over configuration" approach to the Page Object pattern. By utilizing its initElements method, the driver element location code can be removed from the previous Page Object:

```
public class GoogleHomePage {
  private final WebDriver driver;

  private WebElement q;
  private WebElement btnG;
  private WebElement btnI;

  public GoogleHomePage(WebDriver driver) {
    this.driver = driver;

    //Check that we're actually on the Google Home Page.
    if (!"Google".equals(driver.getTitle())) {
      throw new IllegalStateException("This isn't Google's Home Page!");
    }
  }

  public GoogleResultsPage search(String searchTerms) {
    q.sendKeys(searchTerms);
    btnG.submit();

    return new GoogleResultsPage(driver);
  }

  public UnknownTopHitPage imFeelingLucky(String searchTerms) {
    q.sendKeys(searchTerms);
    btnI.submit();

    return new UnknownTopHitPage(driver);
  }
}
```

A fully initialized version of this object can then be created and used as in the following code snippet:

```
WebDriver driver = new HtmlUnitDriver(); //or your choice of driver
driver.get("http://www.google.com");
GoogleHomePage page = PageFactory.initElements(driver, GoogleHomePage.
class);
page.search("WebDriver");
```

This version of initElements will instantiate the class. It works only when there is a one-argument constructor that accepts a WebDriver instance or when the default no-argument constructor is present.

```
AnotherPageObject page = AnotherPageObject("testing", 1, 2, 3, driver);
PageFactory.initElements(driver, page);
```

We can further clean up the code by providing meaningful names for the form elements and then using annotations to declaratively specify the location strategy:

```
public class GoogleHomePage {
  private final WebDriver driver;

  @FindBy(name = "q")
  private WebElement searchBox;

  @FindBy(name = "btnG")
  private WebElement searchButton;

  @FindBy(name = "btnI")
  private WebElement imFeelingLuckyButton;

  public GoogleHomePage(WebDriver driver) {
    this.driver = driver;

    //Check that we're actually on the Google Home Page.
    if (!"Google".equals(driver.getTitle())) {
      throw new IllegalStateException("This isn't Google's Home Page!");
    }
  }

  public GoogleResultsPage search(String searchTerms) {
    searchBox.sendKeys(searchTerms);
    searchButton.submit();

    return new GoogleResultsPage(driver);
  }

  public UnknownTopHitPage imFeelingLucky(String searchTerms) {
    searchBox.sendKeys(searchTerms);
    imFeelingLuckyButton.submit();

    return new UnknownTopHitPage(driver);
  }
}
```

The @FindBy annotation supports the same location methods supported by the programmatic API (except for CSS selectors):

@FindBy Annotation Location Methods

Locator	Examples
id attribute	@FindBy(how = How.ID, using = "myElementId") @FindBy(id = "myElementId)
name attribute	@FindBy(how = How.NAME, using = "myElementName") @FindBy(name = "myElementName")
id or name attribute	@FindBy(how = How.ID_OR_NAME, using = "myElement")
XPATH	@FindBy(how = How.XPATH, using = "//input[@id='myElementId']") @FindBy(xpath = "//input[@id='myElementId']")
Class name	@FindBy(how = How.CLASS_NAME, using="even-table-row") @FindBy(className = "even-table-row")
Link Text	@FindBy(how = How.LINK_TEXT, using="Click Me!") @FindBy(linkText = "Click Me!")
Partial Link Text	@FindBy(how = How.PARTIAL_LINK_TEXT, using="ck M") @FindBy(partialLinkText = "ck M")
Tag Name	@FindBy(how = How.TAG_NAME, using="td") @FindBy(tagName = "td")

XPATH SUPPORT

WebDriver makes every effort to use a browser's native XPath capabilities wherever possible. For those browsers that do not have native XPath support, WebDriver provides its own implementation. If you're not familiar with the behavior of the various engines, this can lead to surprising results.

Driver	Tag/Attribute Names	Attribute Values	Native XPath Support
HtmlUnitDriver	Lower-cased	As they appear in HTML	Yes
InternetExplorerDriver	Lower-cased	As they appear in HTML	No
FirefoxDriver	Case insensitive	As they appear in HTML	Yes

Consider the following example:

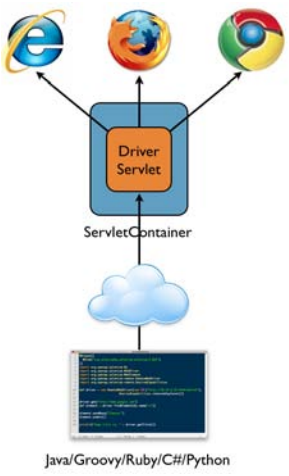
```
<input type="text" name="pizza"/>
<INPUT type="text" name="pie"/>
```

The following behavior will result:

XPath Expression	Number of Matches		
	HtmlUnitDriver	InternetExplorerDriver	FirefoxDriver
//input	1	2	1
//INPUT	0	2	0

REMOTE WEBDRIVER

WebDriver provides excellent capabilities around driving browers located on remote machines. This allows the tests to run in one environment while simultaneously driving a browser in a completely different environment. In turn, running your tests in a continuous integration environment deployed on a Linux system while driving Internet Explorer on various flavors of Microsoft Windows is a straightforward proposition.



The RemoteWebDriver consists of a client and server. The server is simply a Java Servlet running within the Jetty Servlet Container (but you can deploy it to your container of choice). This servlet interacts with the various browsers. The client is an instance of RemoteWebDriver, which communicates with the Server via a JSON-based wire protocol.

Using RemoteWebDriver

First, download selenium-server-standalone-2.0a7.jar and run it on the machine you want to drive:
java -jar selenium-server-standalone-2.0a7.jar

Next, implement a client in your language of choice (Groovy):

```
package com.deepsouthsoftware.seworkshop;

import java.net.URL;
import org.openqa.selenium.*;
import org.junit.*;
import org.openqa.selenium.remote.*;

import static org.junit.Assert.*;

public class RemoteWebDriverTest {
  private WebDriver driver;

  @Before
  public void setUp() throws Exception {
    driver = new RemoteWebDriver(new URL("http://127.0.0.1:4444/wd/hub"),
    DesiredCapabilities.firefox());
  }

  @Test
  public void testCheese() throws Exception {
    driver.get("http://www.google.com");
    WebElement element = driver.findElement(By.name("q"));
    element.sendKeys("Cheese!");
    Thread.sleep(1000); //Deal with Google Instant
    element = driver.findElement(By.name("btnG"));
    element.click();
    Thread.sleep(1000); //Deal with Google Instant
    assertEquals("cheese! - Google Search", driver.getTitle());
  }

  @After
  public void tearDown() throws Exception {
    driver.quit();
  }
}
```

The constructor for RemoteWebDriver accepts two arguments:

1) The URL for the remote server instance.
2) A Capabilities object, which specifies the target platform and/or browser. DesiredCapabilities provides static factory methods for the commonly used choices (e.g., DesiredCapabilities.firefox()).

MOBILE DEVICE SUPPORT

WebDriver provides excellent support for testing Web applications on modern mobile device platforms. Support for the following platforms are provided:

- iOS (iPhone/iPad)
- Android
- Blackberry (5.0+)
- Headless WebKit

WebDriver is capable of running tests both in the platform emulators and the devices themselves. The driver implementations employ the same JSON-based wire protocol utilized by the RemoteWebDriver.

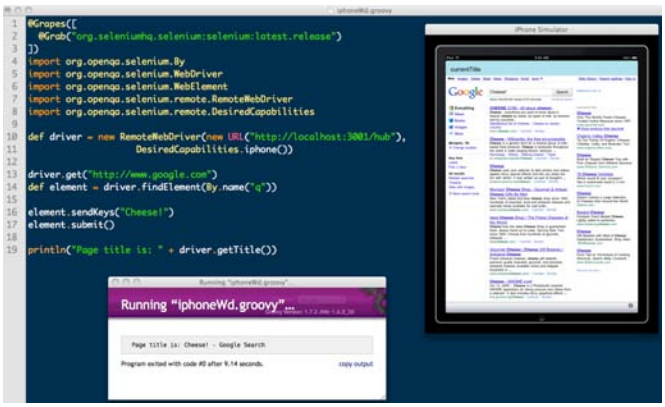
Driving the iOS Platform

The iPhone driver works via an iPhone application that implements the JSON-based wire protocol and then drives a UIWebView, which is a WebKit browser embeddable in iPhone applications.

1) Install the iOS SDK from http://developer.apple.com/ios.
2) Download the WebDriver source from http://code.google.com/p/webdriver/source/checkout.
3) Open iphone/iWebDriver.xcodeproj in XCode.
4) Set the build configuration's active executable to iWebDriver.
5) Click Build and Go.

The iOS simulator will launch with the iWebDriver app installed. You can then connect to the iWebDriver app from your language of choice:

```
package com.deepsouthsoftware.seworkshop;

import java.net.URL;
import org.openqa.selenium.*;
import org.junit.*;
import org.openqa.selenium.remote.*;

import static org.junit.Assert.*;

public class IphoneWebDriverTest {
  private WebDriver driver;

  @Before
  public void setUp() throws Exception {
    driver = new RemoteWebDriver(new URL("http://localhost:3001/hub"),
    DesiredCapabilities.iphone());
  }

  @Test
  public void testCheese() throws Exception {
    driver.get("http://www.google.com");
    WebElement element = driver.findElement(By.name("q"));
    element.sendKeys("Cheese!");
    element.submit();
    assertEquals("Cheese! - Google Search", driver.getTitle());
  }

  @After
  public void tearDown() throws Exception {
    driver.quit();
  }
}
```



Driving the Android Platform

The Android driver works via an Android application that implements the JSON-based wire protocol and then drives an Android WebView, which is a WebKit browser embeddable in Android applications.

1) Install the Android SDK from http://developer.android.com/sdk/index.html.
2) Run ./android and install a target API (e.g. 2.2).
3) Execute ./android create avd -n my_android -t 1 -c 100M to create a new Android Virtual Device targeting Android 2.2 with a 100M SD card. Do not create a custom hardware profile.
4) Start the emulator: ./emulator -avd my_android &
5) Install the Android WebDriver Application: ./adb -e install -r android-server.apk. You can download this from: http://code.google.com/p/selenium/downloads/list.
6) Set up port forwarding: ./adb forward tcp:8080 tcp:8080

You can then connect to the AndroidDriver app from your language of choice:

```
package com.deepsouthsoftware.seworkshop;

import java.net.URL;
import org.openqa.selenium.*;
import org.junit.*;
import org.openqa.selenium.android.AndroidDriver;

import static org.junit.Assert.*;

public class AndroidWebDriverTest {
  private WebDriver driver;

  @Before
  public void setUp() throws Exception {
    driver = new AndroidDriver();
  }

  @Test
  public void testCheese() throws Exception {
    driver.get("http://www.google.com");
    WebElement element = driver.findElement(By.name("q"));
    element.sendKeys("Cheese!");
    element.submit();
    assertEquals("Cheese! - Google Search", driver.getTitle());
  }

  @After
  public void tearDown() throws Exception {
    driver.quit();
  }
}
```

ABOUT THE AUTHOR

Matt Stine is the Group Leader of Research Application Development at St. Jude Children's Research Hospital in Memphis, TN. For the last decade he has been developing and supporting enterprise Java applications in support of life sciences research. Matt appears frequently on the No Fluff Just Stuff symposium series tour, as well as at other conferences such as JavaOne, SpringOne/2GX, The Rich Web Experience, and The Project Automation Experience. He is an Agile Zone Leader for DZone, and his articles also appear in GroovyMag and NFJS the Magazine. When he's not on the road, Matt also enjoys his role as President of the Memphis/Mid-South Java User Group. His current areas of interest include lean/agile software development, modularity and OSGi, Groovy/Grails, JavaScript development, and automated testing of modern web applications. Find him on Twitter at http://www.twitter.com/mstine and read his blog at http://www.mattstine.com.

RECOMMENDED BOOK

Two of the industry's most experienced agile testing practitioners and consultants, Lisa Crispin and Janet Gregory, have teamed up to bring you the definitive answers to these questions and many others. In *Agile Testing*, Crispin and Gregory define agile testing and illustrate the tester's role with examples from real agile teams. They teach you how to use the agile testing quadrants to identify what testing is needed, who should do it, and what tools might help. The book chronicles an agile software development iteration from the viewpoint of a tester and explains the seven key success factors of agile testing.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
140 Preston Executive Dr.
Suite 100
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-03-5
ISBN-10: 1-936502-03-8
50795
9 781936 502035
$7.95

Version 1.0