Yeager: An Annotation-Based Framework for the Generation of Automated Long Sequence Regression Tests in Python

by

Casey Doran

Bachelor of Science
Software Engineering
Florida Institute of Technology
2015

A thesis
submitted to the School of Computing at
Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Software Engineering

Melbourne, Florida
October 2017

We the undersigned committee hereby recommend
that the attached document be accepted as fulfilling in
part the requirements for the degree of
Master of Science in Software Engineering.

"Yeager: An Annotation-Based Framework for the Generation of Automated Long Sequence Regression Tests in Python",
a thesis by Casey Doran

Keith Brian Gallagher, Ph.D.
Associate Professor, Computer Sciences and Cybersecurity
Thesis Advisor

William H. Allen III, Ph.D.
Associate Professor, Harris Institute for Assured Information
Committee Member

Anthony Smith, Ph.D.
Assistant Professor, Electrical and Computer Engineering
Outside Member

Ronaldo Menezes, Ph.D.
Professor and Director, School of Computing

Abstract

TITLE:	Yeager: An Annotation-Based Framework for the Generation of Automated Long Sequence Regression Tests in Python
AUTHOR:	Casey Doran
MAJOR ADVISOR:	Keith Brian Gallagher, Ph.D.

This work presents a Python software package, Yeager, designed to enable the generation and execution of high-volume automated long-sequence regression tests. Users apply the package to existing suites of automated regression tests by annotating individual test methods as state changes for the Software Under Test. Given a sufficiently connected state model (as inferred from these annotations), it becomes possible to generate and execute configurable random walks through the SUT's various states instead of simple regression suites as originally written.

Divided into three sections, this thesis provides a concise overview of an exemplar regression test suite in Python for a web application, a guide to the usage of Yeager itself within the context of the aforementioned regression test suite, and an extensive discussion of the benefits and drawbacks of High Volume Automated Testing in general, and Long Sequence Regression Testing in particular, within the scope of a typical software development organization.

Contents

Acknowledgements

This thesis would not exist if not for the assistance of:

- Dr. Cem Kaner and the Center For Software Testing Education and Research, for taking a chance on an enthusiastic freshman and introducing me to the world of software testing, as well as years of world-class training beyond valuation.

- The wider context-driven testing community, particularly the participants in the Workshops on Teaching Software Testing 11 and 12, my first exposure to the considerable ups and downs of academia.

- Ana Marafuga, Mike DeCabia, Jeff Farr, and Curtis Chambers, the team at Dycom Industries, the most formative internship I've ever had. They let an intern design their entire corporate test automation strategy, most of which informed this thesis, brave souls all.

- Dr. Richard Ford, whose infectious enthusiasm for others' learning and discovery knows no bounds.

- The Samuels family, particularly Bill Jr. and Rob, and Dave Pickerell, for their helpful contributions through most of my academic tribulations.

- The members of the Ruckus and the Harbor City Hooligans. Soccer clubs typically lack rigor, but on the space coast everybody's a rocket scientist.

- My immediate and adopted family, for loving me to where I am today, including

- kbg, for reminding me to thank them for it.

Dedication

TBD, TBH.

Chapter 1

A Concise Overview of A Python Regression Test Suite For a Web Application

This thesis proposes a general-purpose python module for the implementation of high volume automated tests. To properly discuss the nuanced uses of the module, it is first critical to establish a ”typical” industrial usage scenario.

To that end, this chapter describes the state of the art in the web test automation field, and walks through the construction of a web test suite for a popular open source relationship management site, Monica, available for use from the website `https://monicahq.org` as well for self-hosting from `https://github.com/monicahq/monica`. Later chapters will discuss implementation of the module for high volume long sequence regression testing as well as the industrial and academic context surrounding the practice of high volume automated testing.

The test suite discussed in this chapter is published in its entirety online at `https://github.com/elementc/monica-tests-traditional`. They are written against the 0.6.5 release of the Monica software, and may be run using Python 3.

1.1 Technologies

There are a considerable number of tools and libraries used in the development and execution of web application tests. Regardless of actual platform, there must be at least a browser driver, a test runner, and probably some set of inspection tools. As later chapters will use a Python library, the following Python-friendly libraries have been selected.

1.1.1 Selenium

The Selenium open source project is a library which permits the programatic control of a web browser. This library is ostensibly designed for automated testing purposes, but it may be used in any case where automated browser interaction is critical, including secretarial desktop automation, the development of testing tools, malicious purposes, and niche industrial purposes. It has a number of supported platforms, including Python and a purpose-built IDE. The general usage operational cycle is:

1. Instantiate a browser driver, selecting the type of web browser to be driven.

2. Load specific URLs using the driver's `get` method.

3. Query the loaded page using the driver's `find_element` methods.

4. Interact with page components using the element objects and associated methods returned from the above step.

[Holmes and Kellogg, 2006; Bruns et al., 2009; Razak and Fahrurazi, 2011; Wang and Xu, 2009; Kaur and Gupta, 2013; Kongsli, 2007; Artzi et al., 2011]

1.1.2 Python Test Runners

Test runners are executables that load test suites, execute selected subsets, and then report results. There are a number of different test runners in the Python ecosystem, varying in their usage, provided test libraries, and reporting capabilities. Common test runners like pytest and nose live in the Python package archive, and have many users. However, there is a test runner built into the Python standard library named unittest. In the interest of keeping the dependencies of this test suite down (and taking advantage of familiar, high quality documentation), we have selected the unittest library for the runner of this test suite. [Nielsen, 2014; Pajankar, 2017]

1.1.3 Developer Tools and Resources

Test authors need to be able to inspect the web application under test from the UI perspective in order to effectively use Selenium. Historically, a web debugger such as Firefox's Firebug tool has been used to fill this requirement. In modern web development, however, the debugger and inspector are built directly into the desktop web browser. These are all roughly equivalent in capability, but for the purposes of this document we'll use terminology consistent with Google Chrome's Inspector toolkit, which can be accessed with the F12 key.[Odell, 2014]

1.2 Architecture

A typical web application test suite is built from three components: a collection of page object models, a set of configuration parameters, and a set of test sequence scripts. In most cases, these will be stored in similarly named directories (/pages/, /config/, and /tests/).

1.2.1 Page Objects

To abstract out much of the low level work associated with interacting with the system under test, a typical usage case is to write a Python class for each "page" of the web application, and for each class to have a function related to each of the low level interactions, eg setting a field to a value or pulling a string from the page title. In the constructor of the page object, common sanity checks are often run to ensure the system is in a good state. [Liu et al., 2000; Kung et al., 2000; Leotta et al., 2013a; Marchetto et al., 2008]

1.2.2 Configuration

It is common for different environments to have different credentials and settings, for instance, a continuous integration server might deploy with one set of passwords while a developer workstation has another and a user acceptance test server has a third set. To that end, it is critical that such variances are captured corectly, often in a configuration file or by reading correct values from execution environment variables. This may also be sensitive to certain differences in environments, for instance the need to skip verification of emails by system or application of different sets of mock interfaces.

1.2.3 Test Sequences

With page interfaces well defined and a suite of configuration details available, actual test authorship becomes fairly simple, with files conformant to the selected test runner's interface being filled with sequences of fairly easy to understand, high-level steps. Most of the time, a test script will be authored for each user story in the requirements of the system under test. Scenario-based tests is usually the easiest for novice programmers to write, but advanced methods may include detailed tours of the system or even exhausive tests of certain featue sets in isolation: test scripts for each of a calendar, a mail client, a presentation tool, and a contacts manager in an office productivity suite.[Leotta et al., 2013b]

1.3 Building The Test Suite

This section is still under development and depends of the completion of the "traditional" test suite for monica. It may need additional restructuring or refocusing depending on what the committee thinks, but in general this will at least detail the test authorship process. [Sandström, 2015].

1.3.1 Planning A Set Of Tests

walkthrough of how to identify and abstract the list of program features to test, including building a list of actions to write [Nguyen, 2001]

1.3.2 Determining DOM Object Identification Methods

In order to be able to construct a proper page action, it is critical that our test code is able to interact with the right specific parts of the page under test.

In Selenium, we use the driver's `find_element_by_*` methods to do so. There are methods for finding page elements by many methods, including html ID, html name, link text and partial link text, css selectors, and several other more unique methods like an xpath string or just the tag's name. All of these methods are convenience wrappers for a base method named find_element that takes a special constant from the `selenium.webdriver.common.by.By` class, such as `By.ID` or `By.CSS_SELECTOR`.

While it may be more immediately readable to write test code using the convenience methods, this does have an effect on maintainability in that if a particular field must change the method it is found by, many function calls will need to be replaced. To prevent such a replacement nightmare, we can use python tuples and the unzip (splat) operator to combine a method of selection with a string constant as a single "element selector" field which may universally be consumed by a `find_element` function call.

Consider this HTML tag:

```
<input
  type="email"
  class="form-control"
  id="email"
  name="email"
  value=""
>
```

This has a number of useful atributes we could use as a selector, but the best of all is the id field. HTML ids must be unique in an html document [Web

Hypertext Application Technologies Working Group, 2017b] so selection by ID is extremely resillient. Here's a selector and a call to `find_element` for the id "email":

```
email_sel = (By.ID, "email")
email_field = driver.find_element(*email_sel)
```

While selecting by the id field is comparatively simple (application authors may wish to give constant ids to parts of their applications they know will be involved in testing), selection by other methods is more complex. Consider this HTML tag:

```
<button
  type="submit"
  class="btn btn-primary"
>
  Login
</button>
```

This button is critical, it must be clicked in order to complete a login! However, it lacks a unique ID. It is tempting to use the `By.LINK_TEXT` selection method since it has a fairly concise body text ("Login"), but this won't work since it's a `<button>` tag and not an `<a>` (anchor, a hyperlink base) tag. The next logical option is to select by class, the class attribute of html being whitespace-separated tags which are not guaranteed to be unique. If the software is designed to use classes in a way that relevant tags will be unique, this is an option, but it is not typical. In this case, the button has the btn and btn-primary classes, an identically styled button would have the same set of classes. This, then, is a candidate for improvement in the system under test, to at least provide a cleaner testing interface in the form of a unique id or class on this button, but in the interim we can fall back to HTML's built-in selection system, the CSS selector.

A CSS selector is a string confomrant to the CSS selection grammar [World Wide Web Consortium CSS Working Group, 2017] which enables detailed selection of DOM element or elements. It can combine an element's tag, id, class, parents, children, or even position. For the purposes of this login button's selection, we require the following features of candidate elements:

- Tag named button
- Classes btn and btn-primary
- First on the page

The following CSS selector satisfies these requirements:

`button.btn.btn-primary:nth-of-type(1)`.

Here's a python example:

```
login_sel = (By.CSS_SELECTOR, "button.btn.btn-primary:nth-of-type(1)")
login_button = driver.find_element(*login_sel)
```

[Gupta et al., 2003; Web Hypertext Application Technologies Working Group, 2017a; Nicholus, 2016]

1.3.3 Scripting Actions

Now that each relevant element of the web page under test has a unique identifier for our use, the next step is to write the code that actually triggers the interactions with them. This is fairly straightforward, we use the Page Object's driver field to retrieve elements using these identifiers, then take actions on those elements.

The following snippet from our login page test retrieves an email text field and a password text field by their html IDs, as well as a login button by a css selector. These elements are then interacted with via the `send_keys()` and `click()` methods. Note that `self.username` and `self.password` are defined in the object constructor from some secure source of testing account credentials.

```
email_sel = (By.ID, "email")
password_sel = (By.ID, "password")
login_btn_sel = (By.CSS_SELECTOR, "button.btn.btn-primary")
def log_in_correctly(self):
  email_field = self.driver.find_element(*self.email_sel)
  password_field = self.driver.find_element(*self.password_sel)
  login_button = self.driver.find_element(*self.login_btn_sel)
  email_field.send_keys(self.username)
  password_field.send_keys(self.password)
  login_button.click()
```

1.3.4 Asserting Validity

Consider what an assertion (the `assert` statement in python) does: it takes one mandatory argument, a statement that boils down to true or false, and if the statement evaluates to anything but true, the system halts immediately with a message (an optional second argument or a default one) as to what assertion failed and in what way. They explicate and then enforce contracts in programming systems and are a staple of high quality programming.

Since contracts are a core part of a test, much of the actual important work of a test suite consists of well-placed assertions about the state of the system under test. While the assert statement is the most obvious and common way to designate an assertion, any snippet of code which does not change the state of the running test and raises an exception if some assumption is not true can fulfill the same purpose- I call these "assert-alikes". For instance, if a Selenium webdriver find_element fails to find the element described, it simply raises an exception. There's no need to do something complex like

```
dash = self.driver.find_element(*self.dashboard_sel) or False
assert dash, "Couldn't find the dashboard."
```

when, for the purposes of asserting some page element is present, the element finding call itself (`self.driver.find_element(*self.dashboard_sel)`) can stand alone.

Within the page object model, two key places for the insertion of assertions becomes apparent: first, at the initialization of the page object model itself, and second, as necessary during action execution. In the `monicatests-traditional` repository, page object models are derived from a PageBase class in the page_base.py file. This includes an overridable function, `self.initial_status`, which is called in the lowest inherited class definition- and can be selectively chained upwards- after object construction is complete. This method is filled with a number of assertions and assertion-alikes for the given page object model at the expected starting state in that page. Methods on the page object mdoel also include assertions as necessary to verify successful operation, including constructing and returning a page model for a new page when a method causes a transition for that new page- therefore triggering the new page's initial_status method all over again.

1.3.5 Assembling The Final Test Scripts

building (and running) a suite

Chapter 2

Using Yeager To Generate Long Sequence Regression Tests

The test suite assembled in the previous chapter is a great way for a software development team to verify that the core functionality of the system under test is fundamentally operational. When executed, it will test the few well-understood scenarios we have outlined consistently and, assuming enough assertions are present, thoroughly. In fact, the suite requires the entire process from the previous chapter in order to accomodate the additon of new scenarios.

It's a boring, tedious, and repetitious task that can be the entire career of a test engineer. However, as any test automator will know, tasks which are boring, tedious, and repetitious are ripe targets for computer automation, and the task of scenario authorship is no different.

This chapter will outline a method for adapting the existing test suite explored in the previous chapter, using a tool of our own authorship named Yeager, to enable the computer to generate scenarios automatically. Yeager is an MIT-open sourced python version 3 module, with source available at `https://github.com/elementc/yeager`. It provides a python annotation and a set of utility functions. Usage of Yeager's state transition annotation allows testers to quickly and easily map an existing suite of test code onto a state machine, in the form of a graph. This graph can then be traversed using the utility functions, thereby generating new test scenarios from the existing code.

The resultant adapted test suite is published online at `https://github.com/elementc/monica-tests-yeagerized` for your convenience.

2.1 Software As A State Machine

Consider the system under test, Monica. As a relationship management web site, it has a few obvious states it can be in: logged out and on the landing page, logged in and on the dashboard, viewing a list of contacts, viewing a list of journal entries, or viewing the settings page. This maps nicely to the page objects we defined in the previous chapter. Actions on those page objects assume a current state (eg, we're logged in and on the dashboard) and after execution are in a new state which may or may not be the same state (eg, the `Dashboard.click_contacts_button()` method transitions from the dashboard to the contacts list, while the `LoginPage.log_in_incorrectly()` method should result in the system being in the same login page it was before the method was run).

In fact, most modern programs can be looked at as systems composed of a finite set of states (pages, in this case) with some state transitions (links) and a data context (the stuff you've already typed into the system in those states).

Yeager uses this fact to enable automated test sequence generation.

2.1.1 States in Our Example System

Let's consider Monica's pages, which are already built into our test suite, to be states.

We have: the login page (`Login`) and logging in takes us to the `Dashboard` which has tabs for the `Contacts` list and the `Journal` log. There's also a `Settings` page which has subpages for `Import`, `Export`, `Users`, and `Tags`.

The Dashboard and Contacts list both let us `AddAContact`, while the Journal tab lets us `AddAJournalEntry`. From a given `Contact`, one can `AddASignificantOther`, `AddAChild`, `UpdateJobInformation`, `AddANote`, `AddAnActivity`, `AddAReminder`, `AddAGift`, and `AddADebt`.

For the purpose of our discussions, these pages will constitute the entire set of states in the system under test. Conveniently, each of them is a python class.

2.1.2 State Transitions As Actions In Our Example System

A graph consists of a set of nodes and a set of edges. If our nodes are the states the system under test may be in, the edges are the actions that may be taken from those states, possibly resulting in a state transition. It is certainly possible for an edge to be a loop connecting the starting state to itself. In the particular case of testing web applications, note that though it's reasonable to author a page object model with each method corresponding to an edge, this is not an assumption that is necessary to make, and would-be Yeager adopters may choose to lump lower-level page object methods into clusters of function calls in new functions and treat those higher-level functions as edges instead.

2.1.3 Our Example System, Illustrated

overview of the system as a whole, fully rendered and illustrated

2.1.4 Graph Connectedness

"Is it possible to get from here to here?" and other questions, probably will introduce Dijkstra.

2.1.5 Capturing Contextual State

Yeah, getting past the login screen is cool, but there's other outside influences on the output of the program than just which page we're on

2.1.6 Taking A Walk On The Graph: Long Sequence Testing

introduce the concept of long sequence testing

2.2 Yeager State Transition Annotations

how to use yeager: mark up your existing code

2.2.1 State Identifiers

Anything that can be a Python dictionary key can serve as a state identifier. For simplicity's sake we use strings, but as long as Python will allow it, so will Yeager. Enterprising Yeager hackers may use the actual Python page object model class from the test suite to be adapted, often in combination with a custom random walk algorithm and the use of Python's reflection toolkit.

2.2.2 Basic State Transition Annotations

The fastest way to get started with using Yeager is to define a function for each of the state transitions you wish to use in the test. These will probably be short snippets from the traditional-style test sequences. Then, for each of these functions, you should use the `yeager.annotations.state_transition` annotation to mark the transition of that function. Here's an example using some of our Monica test code from the previous chapter:

```
from pages.login import LoginPage
from pages.dashboard import DashboardPage
from yeager.annotations import state_transition


@state_transition(None, "login-page")
def open(driver):
    driver = webdriver.Chrome()
    driver.get("https://app.monicahq.com/")


@state_transition("login-page", "dashboard-page")
```

```
def log_in(driver):
    login = LoginPage(driver)
    login.log_in_correctly()


@state_transition("dashboard-page", "login-page")
def log_out(driver):
    dashboard = DashboardPage(driver)
    dashboard.log_out()
```

Note that we use the Python `None` constant as a reference to the uninitialized system. Yeager treats `None` as a special node in the implied state model our annotations provide: it's assumed to be the entry point.

2.2.3 Using The Yeager Connectedness Tester

Yeager provides a utility function to check for misconfigured annotations. The function, `yeager.orphaned_states`, takes one optional argument (the starting state, it defaults to `None`), and returns a list of all states that yeager knows about but doesn't know how to get to. The inverse, the known states, is also provided as a utility function with the same optional argument, as `yeager.reachable_states`. Though the orphaned states function is useful for debugging, it can be used in other automated ways, for instance as a test coverage check or a way to automate `walk()` calls with each of the "orphaned" states actually being new entry points. These are hacker's utility tools, and can be used as such.

2.3 Yeager Test Harnesses

in more advanced scenarios, we need to assist Yeager's execution

2.3.1 Test Setup and Entry Point

It's up to testers to generate python scripts that start up and execute a yeager test, but the process is very easy.

The first step is to cause the python interpreter to parse all of the relevant yeager annotations. In simple test scripts, it's enough to simply write the test code and annotations at the top of the file, but in large test suites, it may be necessary to simply import those python files at the top of the yeager test script instead. Critically, yeager annotation metadata exists as long as the python interpreter instance does, so it doesn't matter what modules or other structure applies to the code the yeager annotations are spread around in. If it has been parsed, yeager knows about it.

To actually start taking a walk on the state model, simply call yeager's `walk` function with an integer representing how many steps to take.

2.3.2 Exit Point

many scenarios won't deal with this, but how to note ways tests can end successfully.

2.3.3 Application Context Storage

sometimes a test needs more information than just what state we're in. this overviews how to store things relevant to tests (who's expected to be logged in, how many emails they have, how many contacts, etc for an email client app)

2.3.4 Test Method Helpers

special args to an annotation that specify a caller which pulls data from App Context Storage

2.3.5 Yeager-Only Assertions

hooks provided for each state transition which can make additonal assertions not in the original test

2.3.6 The Yeager Logger

Yeager doesn't make any particular assumptions about the logging toolkit that you use. It uses standard output to print log data, but it can be configured to use any arbitrary function the user supplies instead. For Long Sequence Regression Testing, it is very important to log with vigor, as a failure is often the result of many consecutive steps instead of one instant.

2.3.7 Advanced State Transition Annotations (With Context From Harness)

using the stuff from above to enable more rich/complex state transitions

2.4 Yeager Test Plans

the bread and butter, informing the test generator what you're wanting to do

2.4.1 Run-To-Crash vs. Run-Finitely

discussion of a couple scenarios the tester may wish to choose between

2.4.2 Controlling The Path: Blacklists

how to inform a test to NOT go to certain states

2.4.3 Controlling The Path: Weights

how to inform a test to prefer (or shun) certain states

2.4.4 Controlling The Path: Visitation Limits

how to limit the number of times a particular state should be visited (for instance, dont go to the logout state in this run, stay logged in)

2.4.5 Additional Configuration

tbd during Yeager development

2.4.6 Executing Test Plans

```
python -m yeager run yplan.py
```

2.4.7 Interpreting Results And Logs

what do logs look like anyways?

Chapter 3

High Volume Automated Testing And Long Sequence Regression Testing In Context

This is probably an article unto itself. This lends a ”why” to the development of Yeager.

3.1 A Note On The Recorded History Of High Volume Automated Testing

what we know about HiVAT

3.1.1 High Volume Automated Testing Has Been Invented Six Times

and here's where we list all the inventors we can find. [Miller et al., 1990]

3.1.2 Every Industrial Inventor Thinks It's A Trade Secret

which is why I'm apologizing that this is sourced from a bunch of talks and interviews and less-than-academic sourcing.

3.1.3 A Call For HiVAT Documentation and Academic Consideration

so that the next poor sap who writes about it isn't going to have to do so much archaeology.

3.2 Anatomy Of A High Volume Automated Test

Let's look at the different legos we can play with

3.2.1 Driver: What Actions Are Taken

how to generate things. random entirely? random from list? build and run?

3.2.2 Interface: Black Box vs. White Box (And Shades Of Grey)

are you acting on the disassembled source or are you acting on the running end-user program, or something in between (like sending http requests to a ui-based app)

3.2.3 Oracle: Determining Correct Behaviour

how do you know things are going ok?

3.2.4 Logger: Figuring Out What Happened

how does the test report the results?

3.2.5 Testing Context: Cornering vs. Surveying vs. Abusing

what are you trying to do with this HiVAT anyways

3.2.6 Scalability: Parallelized vs. Sequential

how are you breaking down the work (and why should you care)

3.3 The High Volume Test Automation Family Tree

let's walk through some well-documented techniques

3.3.1 Long Sequence Regression Testing

uh, this is the one we're talking about [Lee and Yannakakis, 1996]

3.3.2 API Testing

i'm not sure if this belongs but i've seen it on some lists

3.3.3 Exhaustive Testing

ditto

3.3.4 "Fuzzing" And Other Monkey-Based Testing

"throw a fuzzer at it and see what happens"

3.3.5 Load-Based Testing

put one of the above techniques in a thread pool of a million or so

3.3.6 Testing In Production (Safely!)

Microsoft does this, siphons some user input from Bing to the live search engine and the next version of the search engine, comparing output from both versions. Sometimes users get output from the test version, even.

3.3.7 A/B Testing

An aggressive version of TIP invented by marketers to compare multiple versions of the same ad campaign.

3.3.8 Synthetic HiVAT Techniques

This is where I will wildly speculate about techniques not listed in above subsections (and therefore not discovered in literature review), but would make sense to implement in a context, as built from combinations of the building blocks listed in the Anatomy section.

3.4 High Volume Automated Testing Benefits and Drawbacks

this section might be merged into the above section simply due to the uniqueness of benefits and drawbacks among all the various HiVAT techniques. If, however, trends are apparent, they'll be discussed here.

3.5 The Case For Long Sequence Regression Testing

if there's something you could call a "conclusion", it's probably here. LSRT is a powerful, easy-to-adopt form of HiVAT in some scenarios, with otherwise-elusive bug discovery an eminently attainable outcome.

3.6 Scenarios For Yeager Adoption

A shameless ad for different ways Yeager can be adopted by different groups (a subsection per scenario)

Bibliography

Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Moller, and Frank Tip. A framework for automated testing of javascript web applications. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 571–580. IEEE, 2011.

Andreas Bruns, Andreas Kornstadt, and Dennis Wichmann. Web application tests with selenium. *IEEE software*, 26(5), 2009.

Suhit Gupta, Gail Kaiser, David Neistadt, and Peter Grimm. Dom-based content extraction of html documents. In *Proceedings of the 12th international conference on World Wide Web*, pages 207–214. ACM, 2003.

Antawan Holmes and Marc Kellogg. Automating functional tests using selenium. In *Agile Conference, 2006*, pages 6–pp. IEEE, 2006.

Harpreet Kaur and Gagan Gupta. Comparative study of automated testing tools: Selenium, quick test professional and testcomplete. *International Journal of Engineering Research and Applications*, 3(5):1739–43, 2013.

Vidar Kongsli. Security testing with selenium. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 862–863. ACM, 2007.

David Chenho Kung, Chien-Hung Liu, and Pei Hsia. An object-oriented web test model for testing web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 111–120. IEEE, 2000.

David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 108–113. IEEE, 2013a.

Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 272–281. IEEE, 2013b.

Chien-Hung Liu, David Chenho Kung, and Pei Hsia. Object-based data flow testing of web applications. In *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, pages 7–16. IEEE, 2000.

Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. State-based testing of ajax web applications. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 121–130. IEEE, 2008.

Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

Hung Q Nguyen. *Testing applications on the Web: Test planning for Internet-based systems*. John Wiley & Sons, 2001.

Ray Nicholus. Understanding the web api and vanilla javascript. In *Beyond jQuery*, pages 19–29. Springer, 2016.

Finn Årup Nielsen. Python programmingtesting. 2014.

Den Odell. Browser developer tools. In *Pro JavaScript Development*, pages 423–437. Springer, 2014.

Ashwin Pajankar. *Python Unit Test Automation: Practical Techniques for Python Developers and Testers*. Apress, 2017.

Rosnisa Abdull Razak and Fairul Rizal Fahrurazi. Agile testing with selenium. In *Software Engineering (MySEC), 2011 5th Malaysian Conference in*, pages 217–219. IEEE, 2011.

Martin Sandström. marteinn/selenium-python-boilerplate: A boilerplate for running selenium tests with python. `https://github.com/marteinn/Selenium-Python-Boilerplate`, September 2015. (Accessed on 07/05/2017).

Xinchun Wang and Peijie Xu. Build an auto testing framework based on selenium and fitnesse. In *Information Technology and Computer Science, 2009. ITCS 2009. International Conference on*, volume 2, pages 436–439. IEEE, 2009.

Web Hypertext Application Technologies Working Group. Document object model standard. https://dom.spec.whatwg.org/, June 2017a. (Accessed on 07/04/2017).

Web Hypertext Application Technologies Working Group. Html standard. https://html.spec.whatwg.org/, September 2017b. (Accessed on 09/14/2017).

World Wide Web Consortium CSS Working Group. Selectors level 4. https://drafts.csswg.org/selectors/, August 2017. (Accessed on 09/14/2017).