End to end testing using integrated tools

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Da Zhang

Graduate Program in Computer Science and Engineering

The Ohio State University

2012

Dissertation Committee:

Rajiv Ramnath, Advisor

Jay Ramananathan

Thomas Bitterman
Abstract

Automated functional testing for web applications is a well-researched area. Most organizations confront serious challenges in testing their websites, and these challenges are becoming more and more serious because the number of clients who rely on the web to perform e-commerce activity is increasing. Therefore, thorough, automatic, regressive and lean website testing technology is required to maintain website quality. In this paper, we describe an environment for testing with Selenium and Nagios, as well as customization we develop to incorporate Selenium script into a Nagios executable library. Nagios is an open source framework for monitoring network hosts, services and other hardware conditions with the purpose of failure detection [29]. Based on plug-in mechanisms, each service within the Nagios executable library can be executed as a Nagios plug-in. Selenium is a set of different open source software tools, each with a different approach to supporting web application test automation and agile process automated testing [1]. In this paper, we introduce in the how we combine the Nagios monitoring tool and Selenium testing tool to realize end-to-end testing using integrated tools.
Dedication

This document is dedicated to my family and my friends.
Acknowledgments

I sincerely thank my professors, Dr. Rajiv Ramnath, Dr. Jay Ramanathan and Dr. Thomas Bitterman for guiding me throughout this research. I also thank my friends in the CETI (Centre for Enterprise Transformation and Innovation) group for all their help and cooperation.
Vita

September 2003 ........................................... Northeast Yucai High
2006 .......................................................... B.E. Dalian University of Technology
2010 to present ........................................... M.S. Computer Science, The Ohio State University

Publications


Fields of Study

Major Field: Computer Science and Engineering
Table of Contents

Chapter 1: Problem Statement ...........................................................................................................1

Chapter 2: Our Contribution .............................................................................................................4

Chapter 3: Selenium as a Web Application Testing Tool ..............................................................8

Chapter 4: Nagios as a Monitoring Tool .........................................................................................13

Chapter 5: Approach Solutions ......................................................................................................15
  5.1. Record Testing Commands....................................................................................................16
  5.2. Convert testing commands to testing scripts in Perl .........................................................17
  5.3. Report to administrator when problems happen .................................................................19

Chapter 6: Typical Application ......................................................................................................20
  6.1. Test Polymer Portal.............................................................................................................20
  6.2. Evaluate platform and browser compatibility ....................................................................32

Chapter 7: Evaluation ....................................................................................................................34
  7.1 Practical experience.............................................................................................................35
  7.2 Theoretical Model ...............................................................................................................38

Chapter 8: Conclusion ..................................................................................................................44
  A.1 Validate each primary menu on Polymer Portal in Perl testing script .......................53
A.2 Perl Scripts which covers a transaction of Polymer Portal..............................58

A.3 Customized selenium services as Nagios plug-in .............................................61
List of Tables

Table 1. Mainstream web browsers supported by Selenium RC .................................................. 9
Table 2. User Description .............................................................................................................. 22
Table 3. E-Catalog Service Description ......................................................................................... 23
Table 4. E-Commerce Service Description .................................................................................. 23
Table 5. Simulation Service Resource Provider Description ....................................................... 24
Table 6. Test Case For Login Activity .......................................................................................... 27
Table 7. Test Case for Browsing Activity ....................................................................................... 28
Table 8. Activity Coverage of t1 to t5 ......................................................................................... 31
Table 9. Checkout Form with Different Values ............................................................................. 31
Table 10. Information about what JavaScript technologies and Image format the browsers support ................................................................................................................................. 33
Table 11. Comparison of time spent before and after adopting new methodology ................. 37
Table 12. Two forms the user needs to fill in when use Polymer Portal ..................................... 41
List of Figures

Figure 1. Relationship between different Selenium tools .................................................. 8
Figure 2. Selenium RC components .................................................................................. 11
Figure 3. Nagios Executable Library ............................................................................... 14
Figure 4. Testing workflow ............................................................................................... 16
Figure 5. Selenium IDE ................................................................................................... 17
Figure 6. Various common languages supported by Selenium ......................................... 18
Figure 7. Components inside Polymer Portal ................................................................. 21
Figure 8. Polymer Portal workflow ................................................................................ 25
Figure 9. Polymer Portal shopping cart ........................................................................... 26
Figure 10. Test suites for purchase activity ................................................................. 30
Figure 11. Resource Allocation ....................................................................................... 34
Figure 12. Testing workflow ........................................................................................... 36
Automated functional testing for service oriented web applications is a well-reached area. Most organizations confront serious challenges in testing their web applications, and these challenges are becoming more and more serious because the number of clients who rely on these applications to perform e-commerce activity is increasing. However, due to inaccessibility of the internal source code of service oriented web applications, thorough, automatic, regressive and black box website testing technology is required to maintain the quality of any given web application. Automating functional and user acceptance tests are two of the traditional problems facing software development projects. At present, functional testing and user acceptance testing frequently are done manually. However, in the past year, FitNesse [1] has been developed to help functional testing due to the heterogeneity of web service-oriented applications. Heterogeneity means the applications need to cross multiple platforms, be compatible with various browsers, and also have the ability to test JavaScripts-based components. Some testing teams choose expensive tools such as Mercury Interactive’s WinRunner [37] or Rational Robot [38] (with mixed results), which, unfortunately, increases testing costs dramatically. However, testing reliability cannot be promised by just using testing tools; monitoring tools are still indispensable to quality of service (QoS). The fundamental concept of monitoring is to “observe and check the progress of quality of certain things over a period of time” [4].
This general definition can be applied to multiple areas. In the context of web application, it is primarily used in the areas of systems monitoring, network monitoring, and application performance monitoring. Besides being able to observe properties like availability and responsiveness, the observation of other characteristics of a system and the combination of these measurements enables the supervision of QoS of systems or applications [5]. Over the past few years, cloud computing has gained much attention and become an emerging field that has given service oriented web applications a novel perspective. Although cloud computing has been utilized widely in service-oriented architecture in recent years, the complexity of services adds enormous pressure in ensuring the QoS of service oriented web applications. Further Cloud Computing has introduced a deeper level of complexity for the infrastructure provider to ensure acceptable levels of QoS. The complexity results from the higher potential for heterogeneity of physical infrastructure resources and the introduction of the virtualization layer on top, which implies the management of not only the physical, but also the virtual set of resources [40] [41] [42]. In order to monitor service-oriented web applications, a well-designed motoring tool should has the features of 1) maximizing adaptability to rapid changes of the deployed services 2) notifying administrators immediately as events happen; and 3) offering an reliable measurements of and response times within the system. Considering these features, we use Nagios as a monitoring tool to facilitate the process of quality assurance. Recently, the open-source tool Selenium, developed originally by ThoughtWorks, has gained much attention as a possible silver bullet for the problems of automated testing for web applications. Given their separate
advantages, we propose a method for combining Nagios and Selenium to perform integrated testing.
Chapter 2: Our Contribution

In recent years, the typical way to test web applications has been based on protocol level with HttpUnit and HtmlUnit. During this process, usually, an HTTP request is established and then is sent to the web site under test. After a response is returned, it is analyzed by the protocol (usually in HTML) [8]. However, the advent of Rich Internet applications (RIAs) using Ajax, which is a group of interrelated web development techniques used on the client-side to create asynchronous web applications, made handling these classic tools much more tedious in testing web applications. Because internal source code is inaccessible for service oriented web applications, black box testing [34] technology is more feasible than white box testing technology [35] in this scenario. Typical testing tools and frameworks that perform black box testing include FitNesse, Fit (Frame for integrated test), Canoo WebTest, HttpUnit, Mercury Interactive’s WinRunner and Rational Robot (with mixed results). These testing tools or frameworks can be categorized as gadgets that support an agile style of black box testing, specifically in acceptance testing, regression testing and functional testing. Nonetheless, individual testing tools based on different programming languages have their advantages and disadvantages which result in certain limitations in specific testing environment. The open source tools Canoo WebTest and HttpUnit, for instance, are insufficient to handle most instances of built-in page JavaScript code. This inefficiency is compensated by the
proprietary tool QuickTest Professional (QTPro), which offers record-and-play of test scripts and also runs tests directly in a browser. However, some testing teams found that scripts in QTPro would break after small page changes [10], so a significant amount of script maintenance was necessary and also tedious. Therefore, it is not possible to write huge testing scripts in QTPro. In addition, some testing tools such as Mercury Interactive are not an open source tools, which requires the testing team to purchase the software license if they want to do testing and this is in turn results in a dramatic increase in cost.

Another disadvantage of these testing tools is their lack of ability to perform integrated testing. An integrated testing tool, not only tests functional requirements and user acceptance of certain web applications, but also can monitor other conditions of certain web applications, which include supervising the hardware situation, the network condition, the disk occupancy and so forth.

From previous experience, web applications can be tested manually or automatically. Manual testing is the process of executing the application and manually interacting with the application, specifying inputs and observing outputs [44]. Since manual testing is especially costly regarding labor cost, it is usually thought inefficient and not effective. Although manual testing is still indispensible under certain condition [46], how to minimize this manual degree is a problem we need to address. Additionally, testing need to be done repeatedly during development cycle to ensure the quality of service. For instance, we need to make sure that later code change does not influence previous code
that has already worked. Under this situation, automated regression [45] test is required since manually regression test is ineffective and unpractical. Therefore, it is essential to reduce the labor cost and improve the effectiveness of software testing by automating this regression testing process. In contrast to manual testing, automated testing automates not only test case execution, but also test case generation and test result verification [47]. Previous testers have put lots of effort on how to generate test cases automatically and how to generate random input values without users’ intervention [48].

Different from previous work, our contribution focuses on combining monitoring tools and automated testing tools to implement automated testing and monitoring without much interference from people. In the following part, we introduce contributions from three aspects:

1. Automation of generating, running and managing testing scripts using Selenium. We propose a methodology that not only shortens the time to generate test scripts but also automates this generating process and enables a regression test during the project development process. In our methodology, after recording different activities in testing web applications, we can generate testing scripts effortlessly and customize them as plug-ins and insert them into our monitoring executable library. Moreover, because sometimes a web application does not work due to network failure or database failure, in our methodology, we build a relationship is built between pair component plug-ins.
2. Automation of running testing cases.

In this paper, we propose a novel method that implements automation of running test cases with monitoring tools by taking advantage of the frequency setting ability of the monitoring system. After setting running frequency of certain services in monitoring tools, service checking is carried out automatically by the monitoring tool according to frequency configuration. A benefit of this property is that a regression test can be implemented in our method. During the development process, code changes may impact other components of the system as well as impact previous code. Regression testing inside our integrated testing system will run test cases recurrently to make sure that new changes will not alter the original source code. Therefore differentiating various kinds of failures is another contribution our integrated testing tools perform.

3. Automation of alerting administrators when problems happen or get resolved

If the test cases impact the original code, then the functional test on our web application components may fail. Under this situation, the monitoring system will notify the administrator or developer to fix the bugs. After the bugs are fixed, a recovery notification will also be sent to the related developers and testers together with the testing report. Therefore, our work not only automates the testing process, but also automates the process of generating testing reports and notifying users, allows for ease in locating bugs are and in turn reduces the time and labor for testers and developers to repair the bugs.
Chapter 3: Selenium as a Web Application Testing Tool

Selenium is a set of different open source software tools, each with a different approach to supporting web application test automation and agile process automated testing [1]. The fundamental part of this set of Selenium tools is Selenium Core which was developed with JavaScript. It can drive interactions with the page, allowing target users to automatically rerun tests against multiple browsers as long as they support JavaScript language. The relationship among different Selenium tools can be depicted as shown in following Figure 1.

Figure 1. Relationship between different Selenium tools
Based on Selenium Core, Selenium RC, Selenium Grid, and Selenium IDE were developed to facilitate distinct testing requirements, which provide various functions to meet the needs of web applications of all types. A key feature of Selenium lies in its support for multiple browsers for testing. Table 1 summarizes the key browsers supported by Selenium [1].

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

Table 1. Mainstream web browsers supported by Selenium RC
Originally being implemented as a Firefox extension, Selenium IDE provides testers an integrated environment to record, edit, debug and playback test commands. Using Selenium IDE, testers can either choose to use its recording capability, or may edit testing scripts manually from the Selenium IDE window. No matter what testing style the testers prefer, Selenium IDE can always be an optimal solution for creating testing commands. In addition, Selenium IDE sets the testers free from worrying about the actual internal code, which is beneficial for black box testing. In this paper, we use Selenium IDE to record sequences of testing commands for each testing cases and save them for later use.

Different from Selenium IDE, Selenium Remote Control (Selenium RC) was ground-breaking because no other product allowed testers control a browser from a language of the testers’ choice [1]. Selenium RC allows testers to write Selenium tests in other programming languages such as Perl, Python, Ruby etc., thus making Selenium compatible with traditional test environment. Additionally, testing AJAX-based web applications is a problem that can be addressed by Selenium RC.
Figure 2. Selenium RC components

Figure 2 shows how the client libraries communicate with the server by passing Selenium commands for execution [1]. There are three layers within Selenium RC. The fundamental part consists of testing scripts in common programming language; these will later be passed to upper layer, which is Remote Control Server. After the upper server layer receives a request from the lower level, those testing scripts will be transferred to JavaScript code and those testing codes are passed to the top layer, which includes multiple browsers with built-in JavaScript interpreter. After testing codes has been executed, a testing report will be generated to the testers. By examining the report, for
instance, we can verify whether a text area exists or not within a certain web page. If the report status of the text area verification line is OK, the text area exists; otherwise, it does not exist. In this paper, Selenium RC will be used after Selenium IDE to convert Selenium commands to testing scripts in certain programming languages.

Large enterprises tend to perform large scale of testing under distributed environment or via remote machines [1]. Different from Selenium IDE and Selenium RC, which are basically developed for performing testing on a single machine, Selenium-Grid allows Selenium RC to scale for large test suites and for test suites that must be run in distributed environments. The advantages of Selenium-Grid are two-fold: First, a large test suite can be divided into independent parts that run on multiple hosts at the same time, which will reduce testing time considerably. Second, if the test suite must run on multiple platforms, testers can have different remote machines supporting and running tests at the same time. In our paper, because the test suite is not large and it can be implemented within single machine, we did not choose the Selenium-Grid tool.

In our system, we intend to test one service-oriented web application on one host. We choose Selenium IDE to record test commands first and later use Selenium RC to execute testing scripts.
Nagios is an open source framework for monitoring network hosts, services and other hardware conditions with the purpose of failure detection [29]. Based on plug-in mechanisms, each service within the Nagios executable library can be executed as a Nagios plug-in. Example plug-ins can be network condition checking services, disk usage checking service, etc.

Nagios monitors entire the IT infrastructure to ensure systems, applications, services, and business processes are functioning properly as shown in Figure 3. Once failures happen, Nagios will alert technical staff to the problem by email, allowing them to begin remediation processes before outages affect business processes, end-users, or customers [29]. In addition, because Nagios is open source tool, users of Nagios can easily customize service configurations, which makes Nagios flexible in monitoring user-defined services. Statuses of various services can be displayed through web interface. Those statuses can be categorized as Critical, Warning, OK and Unknown. Among those four statuses, Critical represents serious problems happening to the system, which need to be fixed immediately; Warning means that certain services just exceed the boundary value required to alert administrators, OK represents everything works properly and Unknown tells administers that the system either does not know the status of the service,
or the status definition of the service is not recognizable. The benefit of Nagios’ plug-in mechanism is that is easy also easily to expand and customize Nagios’ executable library to meet an organization's needs. In this paper, we customize testing scripts with Nagios plug-in format and insert them into the Nagios library as testing services.

![Nagios Executable Library](image)

**Figure 3. Nagios Executable Library**

Different from other monitoring tools, which require a great deal of configuration, Nagios’ basic function is easy to configure and install in just 20 minutes [31], in which the basics components will work for 95% users. Therefore, in this paper, we choose Nagios as our monitoring tool. We will introduce in the next section how we combine the Nagios monitoring tool and Selenium testing tool to realize end-to-end testing using integrated tools.
Chapter 5: Approach Solutions

In this paper, we customize our testing script based on Nagios plug-in format and then insert it into the Nagios executable library, set the frequency rate, and run the testing script periodically. The testing scripts are generated based on testing cases that are derived from customers’ requirements. Via this process, regression and automated testing can be implemented. Figure 4 illustrates the process we perform in testing service oriented web applications. We test each case manually by recording testing commands through Selenium IDE (Figure 5). After we press the red button on the upper right corner, Selenium will begin to record whatever we operate within the website, including opening a website, typing content in text area, waiting for a web page to load, etc. Sections 5.1 to 5.3 show separate steps we perform for each web application.
5.1. Record Testing Commands

In the first step, we use Selenium IDE to record activity an sequence that consists of testing commands manually based on test case requirements The Selenium IDE window is shown in Figure 5.
5.2. Convert testing commands to testing scripts in Perl

After testing commands are generated, we are able to convert those testing commands to various testing scripts. After we finish recording each test case, we output those Selenium commands and then convert them into Perl script as shown in Figure 6, which is an add-on provided and supported by Selenium IDE. After this step, we have completed almost half of the task, because Nagios is an open source tool based on Perl scripts, it is easy to investigate the Nagios source code and follow the format and grammar defined by Nagios.
We have explained with definition of the four statuses Critical, Warning, OK and Unknown inside Nagios, and we follow the way Nagios defines these statuses. After this customization, the testing scripts are compatible with Nagios library. Then, in order for Nagios to recognize this service, we also need to define this service in the services’ library of Nagios, set this testing service name, set checking frequency and add email contact information. After these preparations, we can run the scripts periodically so that the testers and the developer will be informed when problems arise.

Figure 6 various common languages supported by Selenium
5.3. Report to administrator when problems happen

Because we run our testing commands periodically according to the testing frequency we defined previously, regression testing can be implemented. For instance, when we add some functions to our applications, and a if later code change influences our previous code, normal functions that have functioned properly should detect this influence and fix the bug caused by code change. In this situation, the signal bar of testing service inside Nagios library will turn to Critical. A feasible way to set the boundary for four statuses is to treat individual test cases as a whole piece. By this we mean that, for each test case, even though only one testing command fails the whole test case should fail. We can catch this failing information by writing Perl script with regular expression to catch error information. Then Nagios will notify administrators via email once the test case fails.
Chapter 6: Typical Application

6.1. Test Polymer Portal

In this chapter, we give an example of applying our testing methods on one of our existing service-oriented web applications Polymer Portal [49] which provides platform designed to integrate access to multiple modeling, simulation and training services. Polymer Portal provides a number of features including an e-commerce front end, common AAA service, and support for both cloud-hosted virtual machine (VM) images and high-performance computing (HPC) jobs. It has been deployed for six months and has been used successfully for a number of training and simulation activities [32].

Figure 7 gives us a preview of our service-oriented web applications, which can also be applied to our Polymer Portal. Usually, six components constitute service oriented web applications 1.) Users who utilize our portal to access simulation; 2.) E-Catalog Service, which gives a list of available services; 3) E-Commerce Service, which deals with transactions between submitted service purchase requests; 4) Simulation Service Provider, which integrates existing services and provides inventory to E-Catalog Services, and 5.) Independent Software Vendors, individuals who sell their software licenses and 6.)
Commercial License Brokers who negotiate between the Software Vendors and Service Providers. As Service Resource Provider, we deploy simulation software provided by independent vendors on our cloud computing environment, purchase software licenses from those vendors and sell those licenses as short term services to end users. Within the portal, the E-Catalog Service component renders available services to the user. The function, expected input, and expected output of each component is described as in Tables 2, 3, 4 and 5.

Figure 7. Components inside Polymer Portal
<table>
<thead>
<tr>
<th>Name</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Accesses simulations</td>
</tr>
<tr>
<td>Provides</td>
<td>1. Selection of service to E-Catalog</td>
</tr>
<tr>
<td></td>
<td>2. Payment to E-Commerce</td>
</tr>
<tr>
<td></td>
<td>3. Input and Data to Simulation Service</td>
</tr>
<tr>
<td>Receives</td>
<td>1. Simulation service list from E-Catalog</td>
</tr>
<tr>
<td></td>
<td>2. Billing information from E-Commerce</td>
</tr>
<tr>
<td></td>
<td>3. Results and visualizations from Simulation Service</td>
</tr>
</tbody>
</table>

Table 2. User Description

As Table 2 describes, Users represent target customers that include industrial users and academic users who need to access our services. They browse simulation services and training opportunities provided through E-Catalog component, select services they want and get charged through the E-Commerce component. After they purchase services, they are authorized to use the services during a certain period of time such as one week, one month, etc. From Figure 7, we can see that users interact with the three components: E-Catalog Service, E-Commerce Service and Service Resource Provider. These various interactions also require our test cases to cover all the communications between different components. Between User and E-Catalog Service, we need to guarantee that each link listed on the E-Catalog Service allows the end user to access actual services within a certain time. For instance, because Polymer Portal uses external Remote Desktop Service Moodle to allow users access external training courses, we need to make sure the end users can access Moodle when they select the service.
### E-Catalog Service Description

<table>
<thead>
<tr>
<th>Name</th>
<th>E-Catalog Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Single point-of-access for simulation services from multiple Resource Providers</td>
</tr>
<tr>
<td>Provides</td>
<td>1. Simulation browsing to User</td>
</tr>
<tr>
<td></td>
<td>2. Invoice of simulations to E-Commerce</td>
</tr>
<tr>
<td>Receives</td>
<td>1. Inventory of simulation services from Resource Providers</td>
</tr>
</tbody>
</table>

Table 3. E-Catalog Service Description

E-Catalog Service, which is shown in Table 3, provides a single point of access to users by letting them access multiple services that reside on individual platforms through a single gateway. This component is responsible for providing end users a list of available services and thus is the inventory of simulation services from Resource Providers. Additionally, E-Catalog can transfer an invoice of simulations to the E-Commerce component after the user purchases specific kind of service. Given that this component transfer invoice to the E-Commerce service, in our test cases, it needs to be guaranteed that the user enters a valid number of simulation licenses when they input the quantity of software licenses in the shopping cart.

### E-Commerce Service Description

<table>
<thead>
<tr>
<th>Name</th>
<th>E-Commerce Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Invoices users and receives payments.</td>
</tr>
<tr>
<td>Provides</td>
<td>1. Billing information to Users</td>
</tr>
<tr>
<td></td>
<td>2. Simulation authorization and Payments to Resource Providers</td>
</tr>
<tr>
<td></td>
<td>3. License usage reports and payments to License Broker</td>
</tr>
<tr>
<td>Receives</td>
<td>1. Invoice of purchased simulation services from E-Catalog</td>
</tr>
<tr>
<td></td>
<td>2. Payments from Users</td>
</tr>
<tr>
<td></td>
<td>3. Simulation service costs and usage from Resource Provider</td>
</tr>
<tr>
<td></td>
<td>4. Software licenses and license costs from License Broker</td>
</tr>
</tbody>
</table>

Table 4. E-Commerce Service Description
Basically, E-Commerce Service, shown in Table 4, deals with the transactions. It accepts invoices from E-Catalog service and sends billing information to users. Between the interaction of E-Catalog Service and E-Commerce Service, the user needs to provide valid billing information, which includes credit card information, email information, etc. Therefore, test cases should cover this validation. Additionally, after a user pays money for a certain service, a simulation authorization should be generated and given to Resource Providers. Then the user’s account will be added to corresponding groups of the Lightweight Directory Access Protocol (LDAP) server, which is the backend server responsible for managing users’ accounts in our system. After the initial use, when the user needs access to the service purchased, the user will be authenticated through LDAP server.

<table>
<thead>
<tr>
<th>Name</th>
<th>Simulation Service Resource Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Host simulation jobs and sessions for users.</td>
</tr>
</tbody>
</table>
| Provides | 1. Simulation sessions and jobs to Users  
2. Inventory of available simulation services to E-Catalog  
3. Per-unit costs and service usage to E-Commerce |
| Receives | 1. Invoice of purchased simulation services from E-Catalog  
2. Payments from Users  
3. Simulation service costs and usage from Resource Provider  
4. Software licenses and license costs from License Broker |

Table 5. Simulation Service Resource Provider Description

The Simulation Service Resource Provider described in Table 5 provides inventory to E-Catalog Service, simulation sessions and jobs to end users, and per-unit costs and service usage to E-Commerce.
Figure 8. Polymer Portal workflow

Figure 8 gives an overview of how different components of Polymer Portal interact with each other as a whole. The first time a person uses Polymer Portal, it is necessary for the user to register first with valid personal information. After registration, the backend LDAP server will assign an account to the user. Thereafter the user can log into the system with authentication. After the user logs in, he/she can browse available types of simulation services and training opportunities provided by Polymer Portal via E-Catalog components. For specific purposes, the user can select services with different time intervals, which limit how long the user can access the service. For instance, some academic users need to access a certain service for only one month; hence the user only purchases only one month service. For some industry users, they may need to use the service continuously in developing and testing their products for as long as one year.
which requires them to purchase one year of service. After the user purchases a certain service, the service will be added to the user’s shopping cart, which is shown in Figure 9.

![Figure 9. Polymer Portal shopping cart](image)

If the user commits to buy certain software licenses, as Figure 9 shows, E-Commerce Service will conduct this transaction and give the user corresponding authorization through the LDAP server. If this transaction is successful, the billing information will be sent to the user’s email address and the user is authorized to utilize the service. After being acquainted with interactions between pair components, we record testing commands according to different testing requirements for each interaction.

In the following section, we design a test case for each arrow in Figure 8. For the action of Login, we use different combinations of valid and invalid usernames and passwords to
verify that our Polymer Portal can catch invalid authentications. From Table 6, we can see that valid passwords and usernames are permitted through the authentication and invalid usernames and passwords are rejected.

<table>
<thead>
<tr>
<th>Username</th>
<th>Password</th>
<th>Expected Output</th>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid Value</td>
<td>Valid Value</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>Valid Value</td>
<td>Invalid Value</td>
<td>Rejected</td>
<td>Rejected</td>
</tr>
<tr>
<td>Invalid Value</td>
<td>Valid Value</td>
<td>Rejected</td>
<td>Rejected</td>
</tr>
<tr>
<td>Invalid Value</td>
<td>Invalid Value</td>
<td>Rejected</td>
<td>Rejected</td>
</tr>
</tbody>
</table>

Table 6. Test Case For Login Activity

For the action of browsing E-Catalog, we basically iterate each link listed within E-Catalog and make sure that the links can direct the user to related services. After logging in, the user can browse available services and training opportunities that are shown as a list from our portal. At this point, we need to make sure that every link directs the user to correct places allowing them utilize the service. As Table 7 shows, we list available services we will provide and also make a comparison between expected place and actual place to which the user is directed. From our testing results, we see that after clicking on each link, the user is directed to valid web pages.
<table>
<thead>
<tr>
<th>Products</th>
<th>Expected Link</th>
<th>Actual Link</th>
</tr>
</thead>
</table>

Table 7. Test Case for Browsing Activity
Following is the example Perl testing script that is generated for Login activity in Figure 8 running on Google Chrome browser. Other Example Testing Scripts can be found in the section of Appendix A.1, A.2 and A.3.

```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 => "*chrome",
    browser_url => "https://www.polymerportal.org/" );

$sel->open_ok("/");
$sel->click_ok("link=Login");
$sel->click_ok("css=span.tab");
$sel->type_ok("id=edit-name", "zhangda");
$sel->type_ok("id=edit-first-name", "da");
$sel->type_ok("id=edit-last-name", "zhang");
$sel->type_ok("id=edit-mail", "zhangda_1987\@hotmail.com");
$sel->type_ok("id=edit-password", "1987930");
$sel->type_ok("id=edit-password2", "1987930");

After deciding to buy certain software licenses for services, the user is required to enter the quantity he/she would like to purchase. We used a group of testing suites t1 to t5 in
Figure 10, to test purchase activity with different quantity values to make sure that our testing scripts can capture invalid values that are entered by the user, such as negative numbers or excessively large numbers.

For verification, we assigned different values for these test cases: Test case 1 (t1): Qty = 800, Test case 2 (t2): Qty = 4, Test case 3 (t3): Qty = 0, Test case 4 (t4): Qty = -1, Test
case 5 (t5): Qty = 700000. From Table 8, we notice that, for each activity. In addition, when the user fills in the billing information, there are six out of 11 fields that are required to be not null. Therefore, our testing scripts need to detect each field to make sure that none of these six fields can be null. As Table 9 shows, we set each field separately to be NULL, and test if a warning message will be shown to alert the user to fill in the corresponding fields.

<table>
<thead>
<tr>
<th>Activity</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>A7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 8. Activity Coverage of t1 to t5

<table>
<thead>
<tr>
<th>Field Name</th>
<th>None</th>
<th>Expected Output</th>
<th>Actual Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>First name</td>
<td>Yes</td>
<td>Warning Message</td>
<td>Warning Message</td>
</tr>
<tr>
<td>Last name</td>
<td>Yes</td>
<td>Warning Message</td>
<td>Warning Message</td>
</tr>
<tr>
<td>Street address</td>
<td>Yes</td>
<td>Warning Message</td>
<td>Warning Message</td>
</tr>
<tr>
<td>City</td>
<td>Yes</td>
<td>Warning Message</td>
<td>Warning Message</td>
</tr>
<tr>
<td>Country</td>
<td>Yes</td>
<td>Warning Message</td>
<td>Warning Message</td>
</tr>
<tr>
<td>State/Province</td>
<td>Yes</td>
<td>Warning Message</td>
<td>Warning Message</td>
</tr>
<tr>
<td>Postal code</td>
<td>Yes</td>
<td>Warning Message</td>
<td>Warning Message</td>
</tr>
</tbody>
</table>

Table 9. Checkout Form with Different Values
If the user is successful purchasing the software license, a unique order number will be generated and the receipt can be tracked through a web page, which allows the user to review the status of the transaction and get a printable receipt. The printable receipt indicates that the promised transaction is successful.

6.2. Evaluate platform and browser compatibility

From our previous experience, we noticed that we need to make our code work independently from various browsers and operating systems. Due to functions supported by our browsers, a code that works for this one browser may not work for another browser. For instance, some browsers do not support Document Object Model (DOM) due to security setting problems or old version problems of browsers, which result in failure to respond to certain events such as clicking highlighting and so forth. Additionally, display effects may vary from one browser to another due to image format differences. Even worse, some images can be viewed via browsers while others cannot.

Therefore, during our testing process, we need to test each main-stream web browser to make sure the actual code works well for each browser. We can benefit from our new methods by parallel testing in Nagios. As Table 10 illustrates, we investigated five kinds of main stream web browsers, including Internet Explorer (IE), Firefox, Chrome, Safari, and Netscape [43]. We ran the same testing scripts on individual browsers to verify that each browser works correctly. By inserting these scripts for different browsers into the
Nagios executable library and allowing them run in parallel, we reduced the time dramatically compared with running those tests for each browser sequentially and manually.

<table>
<thead>
<tr>
<th>Web browser</th>
<th>JavaScript 1</th>
<th>DOM 2</th>
<th>DOM 3</th>
<th>Rich Editing</th>
<th>ECMAScript 3</th>
<th>TIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internet Explorer</td>
<td>Yes</td>
<td>Partial</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Mozilla Firefox</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Google Chrome</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Safari</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Netscape Navigator</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Partial</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 10. Information about what JavaScript technologies and Image format the browsers support
Chapter 7: Evaluation

For many years the 40/20/40 rule has been found to typify the allocation of resources in software development: 40% of the resources are devoted to specifying the requirements and creating the design, 20% to coding and 40% to testing [25]. This is represented in Figure 11. We thus can conclude that testing consumes resources in real world development of software applications and therefore requires an efficient methodology to decrease testing cost.

![Resource Allocation Diagram](image)

Figure 11. Resource Allocation
Basically, there are two perspectives that are taken into account in calculating testing resources: resource costs and personnel costs [26]. The costs of testing are increasing dramatically as the systems to be tested become increasingly complex. Additionally, the costs of testing a service-oriented architecture are 2 to 3 times greater than testing a single stand-alone application [33], which adds more load in service-oriented web applications testing. In this section, we first calculate the time we spend prior to applying our method and then compare with the result after we apply the new method. Then we use the evaluation model [26], which divides testing costs into resource costs and personnel costs to estimate the benefit of our method.

7.1 Practical experience

From resource costs aspect [26], we have: communication costs, software costs and testing tool costs. In our Polymer Portal, the hardware of high performance computers (HPC) in the backend is ready to use, thus we do not need to investigate the communication costs. Because we use the open-source testing tool Selenium to perform testing and the open-source monitoring tool Nagios to perform monitoring, the test tool costs also do not to be investigated. Although we only use the basic functions that are free of these tools, 95 percent of requirements can be satisfied by these functions. Therefore, the test tool cost in our method goes down to zero. What we need to calculate in this paper then is mainly personnel cost, which is given in person day unit.
Before we use our methodology, the personnel costs of testing the Polymer Portal web application are calculated as follows. Because testing Polymer Portal is the pilot program, we only have one tester to write test scripts and run it in our environments. We perform testing according to Figure 12 and calculate personnel cost for each testing stage.

Figure 12. Testing workflow

According to the process in Figure 12, in the first stage of Test Case Design, we spent 3 person days working out test cases. After that, we write and verify test cases according to user requirements, which costs 5 person days. Based on these test cases, we began
writing and debugging testing scripts for each test case. On average, each test case costs 3 person days in writing testing scripts. Because we have 5 test cases, we will spend $5 \times 3 = 15$ person days in this stage. After that, we customize test scripts into Nagios plug-ins, which costs 2 person days per test case. Because we have 5 test cases, it will cost $2 \times 5 = 10$ person days. We spend 2 person days adding and configuring testing services to fit the Nagios executable library. We later summarize the time performed on testing as $3 + 5 + 15 + 10 + 2 = 35$ person days. After we applied our new method of generating test scripts automatically, although we need a day to install Selenium, we reduce the time of generating test scripts manually and the time to which is $3 \times 5$ and 3 days separately. Therefore, the time cost is $35 – 15 + 1 = 21$ days. Table 11 makes a comparison of the time spent at each stage before and after we apply new methodology.

<table>
<thead>
<tr>
<th>Testing Stages</th>
<th>Old method</th>
<th>New method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to install Selenium</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Test Case Design</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Test Case Verification</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Generate Test Scripts</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Customize to Nagios plug-ins</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Configure Adding Services</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Total Time</td>
<td>35 person days</td>
<td>21 person days</td>
</tr>
</tbody>
</table>

Table 11. Comparison of time spent before and after adopting new methodology
7.2 Theoretical Model

After we record personnel cost in our system, we then calculate the cost using the model from Hasitschka et al. [26] to verify that this formula is applicable to our application. In Hasitschka et al. [26], from a personnel cost aspect, the sum of all person days booked against the testing project is calculated using Formula 1. We use several terms to represent different coefficients that determine the personnel cost within a web application project. In most cases, testing ties up the capacity of system specialists for an indefinite length of time. Even if the testers perform testing systematically, they need time to go through all of the possible usage scenarios and work flow. Because testing is in principle never finished, it is difficult to determine when to stop. A possible stopping criterion we can use is coverage measures (COV) which determines the actual code covered by testing scripts. Another essential factor in testing is automation degree (TA), which represents the percentage of testing during a testing process and usually varies from 0 and 75%. Even if test data generation, test execution, and test validation are automated, testers are still required to analyze the validation reports. However, TA is undoubtedly an essential factor that influences the personnel cost. One concrete instance is that automated testing tools help GEOS product maintenance process reduce their testing staff from 92 to 24 with the TA degree of 75% [26]. In most development projects the degree of test automation is well under 33% and in many projects it is close to zero. Thus, the degree to which human labor can be replaced by automation tools is a vital parameter for
evaluating any test cost estimation equation. In this paper, we calculate the TA value by calculating the percentage of stages that can be automated during the workflow in Figure 5. As Figure 5 shows, there are 3 out of 6 steps that can be automated; therefore, we determine the TA value to be 50%. In formula (1), as Hasitschka et al. [26] defined, the scaling exponent is an exponent with a value range of 0.91 to 1.23 used to adjust the raw effort estimated. It is computed as the weighted average of five individual factors – team cohesion, team experience, environmental stability, supporting tools and process maturity [27]. However, instead of counting statements or function-points, this theory [26] assumes that one counts Number of test cases (TC) to represent the size of the project.

Formula (1)

\[
\text{Effort} = ST \times \left\{ \left[ \frac{TC \times COV}{TP \times (1-TA) + TP} \right]^TE \times \left( \frac{0.5}{TB} \right) \right\}
\]

Parameters set for Formula (1):

- Desired test coverage (COV)
- Number of test cases to be tested (TC)
- Manual test productivity in test cases per day (TP)
- Degree of test automation (TA)
- Testability metric (TB)
- Scaling exponent (TE)
- System type (ST)
In addition to being determined by the size and complexity of the system, the number of test cases is influenced by the coverage goal. Concerning coverage goal, suppose that 1,000 test cases would be needed to cover all functional and nonfunctional requirements. If the quality goal is 0.8, for instance, on a rational scale of 0 to 1, then 800 test cases would suffice to achieve that goal. If the quality goal is lowered to 0.6, then 600 test cases would suffice [27]. Testability metric (TB) is determined by the testability of a web form under the client side. Concerning determining TB value, a form with a larger number of unbounded fields could be considered to have lower testability, because generating meaningful values for these fields may be quite hard [35]. Within Polymer Portal, we have two forms, which are “Create a new account” and “Checkout,” that need to be filled in by the user through a transaction. In the form “Create a new account,” we have six fields: username, first name, last name, email address, password and reentered password. Among these fields Tonella and Ricca [35], define a function NUF (Number of Unbounded Fields) that returns the number of unbounded fields in a form as follows: 

\[
\text{NUF} (f) = |\text{unbounded (form)}|.
\]

Therefore, as shown in Table 12, in the form “Create a new account”, the number of unbounded fields is 3, which is calculated as follows: 

\[
\text{NUF} (\text{Create a new account}) = |\text{unbounded(\text{Create a new account})}| = 3.
\]

In addition to NUF, we define a function EFC (Enumerable Field Combinations), that calculates the number of combinations of choices for enumerable fields. We define the function as follows to calculate the EFC value within a web form:
In the form “Create a new account,” there are no enumerable fields, which results in an EFC value 0. In another form of “Checkout” of Polymer Portal, which consists of 11 fields with two enumerable fields “Country” and “State”, there are two options for the “Country” field and 65 + 15 options for “State,” which is determined by the country field. Thus, in the “Checkout” form, the value of EFC is 80.

Concerning how to calculate the Testability value, Tonella and Ricca [35] introduce a formula of Testability = (Bounded Field Combination / All Field combination). Therefore, the Testability value for the “Checkout” form is calculated as \( \frac{80}{8 \times 80} = 12.5\% \), while testability for “Create new user account” is \( \frac{1}{6} = 16.6\% \). Therefore we set the testability metrics in our system between 12.5 and 16.6%, which means the number of fields that can be tested inside our web application is really low. For convenience, we take the average of these two values which is \( \frac{12.5\% + 16.6\%}{2} = 14.55\% \). In our application we determined 75 test cases to be verified, which leading to the TC value of
Furthermore, desired testing coverage is assumed to be 67%, which means that 2/3 of the actual code can be tested successfully. According to our practical experience, we set the testing productivity to be four testing cases per day. For Scaling Exponent (TE), which is computed as the weighted average of five individual factors – team cohesion, team experience, environmental stability, tool support, and process maturity [27] we set an assessment of the test organization to a scaling exponent of 1.23 because Polymer Portal is our pilot application and we only have one tester investigating this testing method with our model. Before we utilize automated tools, the TA value is 0. For the value of System Type (ST), it is 1 for a standard integrated solution because we use centralized architecture in our system. Therefore, the testing effort prior to using the new method is calculated as shown in Formula 3.

\[
\text{TestEffort} = 1 \times \left\{ \frac{75 \times 0.67}{4 \times (1-0)+4} \right\}^{1.23} \times \left( \frac{0.5}{14.55\%} \right) = 36\text{PDS}
\]

PD is defined as Person Day which is the unit for estimate testing effort. After applying our new method, the TA value is changed to 0.5 and TestEffort is recalculated as shown in Formula 4.

\[
\text{TestEffort} = 1 \times \left\{ \frac{75 \times 0.67}{4 \times (1-0.5)+4} \right\}^{1.23} \times \left( \frac{0.5}{14.55\%} \right) = 20\text{PDS}
\]
From this point of view, we believe that the automation technology greatly saves the personnel labor by nearly two times in testing web applications. After we calculated the personnel cost using the given formula, we compared this with our practical experience. We found that the results calculated by Formula 1 are approximately the same as the value of our practical experience and therefore can be applied to calculate personnel cost in other projects. Furthermore, the results also show that automation degree can reduce the personnel cost dramatically during testing process.
Chapter 8: Conclusion

In this paper, we offer novel and effective integrated testing tools for service oriented web application testing, especially in maintaining and evolving test scripts so that testers can implement automatic generating testing scripts and notify users as exceptions occurs. Based on Nagios, which is a plug-in based monitoring environment, we insert a testing service as a plug-in into the Nagios executable library to implement our approach. The results of evaluation show that users spent less Personnel cost in generating test scripts with our tool than with previous approaches. We consider the combination of Nagios and Selenium tools as the first step towards developing different solutions for this large and pervasive problem of testing service oriented web application. In the future, we will continue digging into utilizing integrated testing tools to automate the testing process and expand the functions of testing. For instance, currently we cannot detect the issues incurred by opening a certain external applet. In this situation, although the link directs the user to the correct URL, the external applet still cannot be accessed by the user due to the problem caused by the applet itself. Facing this problem, our task in next stage is to find out how to customize our testing scripts so that they are able to locate the error. We believe that many solutions arise from a more holistic approach to software testing research; testing technology is really a field that deserves our contribution and attention.
References:


[17] Fu Jinwei; Sun Jianhong; Xu Haicheng  A Multi-sites Scheme Based on Open
Source CMS, Drupal Multimedia and Information Technology (MMIT), 2010 Second
International Conference on Volume: 2 Digital Object Identifier: 10.1109/MMIT.2010.68
Publication Year: 2010, Page(s): 239--241

[18] Rajan, A. “Coverage Metrics to Measure Adequacy of Black-Box Test Suites”
Automated Software Engineering, 2006. ASE '06. 21st IEEE/ACM International
Conference on Digital Object Identifier: 10.1109/ASE.2006.31 Publication Year: 2006 ,
Page(s): 335--338

[19] Laski, J. “TESTING IN TOP-DOWN PROGRAM DEVELOPMENT” Software
Testing, Verification, and Analysis, 1988, Proceedings of the Second Workshop on
Digital Object Identifier: 10.1109/WST.1988.5355 Publication Year: 1988, Page(s): 72--79

“Event- and Coverage-Based Testing of Web Services” Secure Software Integration and
Reliability Improvement Companion (SSIRI-C), 2010 Fourth International Conference
on Digital Object Identifier: 10.1109/SSIRI-C.2010.24 Publication Year: 2010 , Page(s): 62--69

[21] Mouy, P.; Marre, B.; Williams, N.; Le Gall, P. “Generation of all-paths unit test with
function calls” Software Testing, Verification, and Validation, 2008 1st International
Conference on Digital Object Identifier: 10.1109/ICST.2008.35 Publication Year: 2008 ,
Page(s): 32--41


[31] http://nagios.sourceforge.net/docs/3_0/quickstart.html


[34] Comparison of Five Black-box Testing Methods for Object-Oriented Software


[36] FitNesse (Web Site: http://www.fitnesse.org)


[38] Rational Robot (Web Site: http://www-01.ibm.com/software/awdtools/tester/robot/)

[40] Ferretti, S.; Ghini, V.; Panzieri, F.; Pellegrini, M.; Turrini, E.;, "QoS–Aware
no., pp.321--328, 5-10 July 2010.

[41] Jin Shao; Hao Wei; Qianxiang Wang; Hong Mei; , "A Runtime Model Based
International Conference on , vol., no., pp.313--320, 5-10 July 2010.

Centric End-to-End Monitoring Approach for Service Composites," Services Computing


[44] Salima, T.M.S. Ummu; Askarunisha, A.; Ramaraj, N. “Enhancing the Efficiency of
Regression Testing through Intelligent Agents”, Conference on Computational
Intelligence and Multimedia Applications, 2007. International Conference on Volume: 1
Digital Object Identifier: 10.1109/ICCIMA.2007.294 Publication Year: 2007, Page(s):
103--108.

[45] Dubey, Harsh Kumar; Kumar, Prashant; Singh, Rahul; Yadav, Santosh K; Yadav,
Students Conference on Digital Object Identifier: 10.1109/SCES.2012.6199072
Publication Year: 2012 , Page(s): 1--5.


Appendix A: Example Testing Scripts

A.1 Validate each primary menu on Polymer Portal in Perl testing script

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 => "*chrome",
                                       browser_url => "https://www.polymerportal.org/" );

$sel->open_ok("/");

$sel->click_ok("link=Home");

$sel->wait_for_page_to_load_ok("30000");

$sel->click_ok("link=Our Mission");

$sel->wait_for_page_to_load_ok("30000");

$sel->click_ok("css=div.breadcrumb > a");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=About the Polymer Portal");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=About PolymerOhio");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=About OSC");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=News");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=News");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("css=li.item99.sfHover > a > span");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Purchase Software Access");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=How to access your software");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Back up your files");
$sel->click_ok("link=Software Support");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Learn how Modeling & Simulation can provide your company with a competitive advantage");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Special Pricing");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Available Training Courses");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Training Course Access");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Certificate Programs");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Request Training Information");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("xpath=(//a[contains(text(),'Software Support'))[2]");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Contact Technical Support");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=FAQ");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=Contact Us");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("link=My Cart(0)");
$sel->wait_for_page_to_load_ok("30000");
Create new account testing scripts:

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 => "*chrome",
    browser_url => "https://www.polymerportal.org/" );

$sel->open_ok("/");
$sel->click_ok("link=Login");
$sel->wait_for_page_to_load_ok("30000");
$sel->click_ok("css=span.tab");
$sel->wait_for_page_to_load_ok("30000");
$sel->type_ok("id=edit-name", "zhangda");
$sel->type_ok("id=edit-first-name", "da");
$sel->type_ok("id=edit-last-name", "zhang");
$sel->type_ok("id=edit-mail", "zhangda_1987\@hotmail.com");
$sel->type_ok("id=edit-password", "1987930");
$sel->type_ok("id=edit-password2", "1987930");
$sel->click_ok("id=edit-create");
$sel->wait_for_page_to_load_ok("30000");
$sel->type_ok("id=edit-password", "Z1987930");

$sel->type_ok("id=edit-password2", "Z1987930");

$sel->click_ok("id=edit-create");

$sel->wait_for_page_to_load_ok("30000");
A.2 Perl Scripts which covers a transaction of Polymer Portal

use strict;

use warnings;

use Time::HiRes qw(sleep);

use Test::WWW::Selenium;

use Test::More tests => 5;

use Test::Exception;

my $sel = Test::WWW::Selenium->new( host => "localhost",
port => 4444,
browser => "*firefox",
browser_url => "https://eweldpredictor.ewi.org/" );

$sel->open_ok("/portal/home/");

$sel->click_ok("wrench_drawer");

$sel->wait_for_page_to_load_ok("30000");

$sel->click_ok("link=Pipe and plate weld simulations");

$sel->wait_for_page_to_load_ok("30000");

$sel->click_ok("/input[@value='Next']");

$sel->click_ok("/input[@value='Next' and @type='button' and 
@onclick="select_section('geometry')"]");
$sel->click_ok("/input[@value='Next' and @type='button' and
@onclick="select_section('weld')"]");

$sel->click_ok("create_bead_btn");

$sel->click_ok("LOCATOR_DETECTION_FAILED");

$sel->click_ok("/input[@value='Next' and @type='button' and
@onclick="select_section('material')"]");

$sel->click_ok("/input[@value='Next' and @type='button' and
@onclick="select_section('procedure')"]");

$sel->click_ok("/input[@value='Next' and @type='button' and
@onclick="select_section('fixture')"]");

$sel->click_ok("create_clamp_btn");

$sel->click_ok("create_cooling_btn");

$sel->click_ok("create_monitor_btn");

$sel->click_ok("/input[@value='Next' and @type='button' and
@onclick="select_section('submit')"]");

$sel->alert_is("One or more input parameters (highlighted in red)\nis outside the range supported by the simulator.\nPlease correct the value(s), and contact EWI if\nadditional assistance is required.");

$sel->click_ok("/input[@value='Next' and @type='button' and
@onclick="select_section('submit')"]");

59
$sel->alert_is("One or more input parameters (highlighted in red) is outside the range supported by the simulator. Please correct the value(s), and contact EWI if additional assistance is required.");

$sel->click_ok("//input[@value='Next' and @type='button' and @onclick="select_section('submit')"]");

$sel->alert_is("One or more input parameters (highlighted in red) is outside the range supported by the simulator. Please correct the value(s), and contact EWI if additional assistance is required.");

$sel->click_ok("trash_fixtures");

$sel->click_ok("//input[@value='Next' and @type='button' and @onclick="select_section('submit')"]");

$sel->click_ok("//input[@value='Submit']");

$sel->click_ok("//input[@value='Submit']");

$sel->alert_is("submit error! MochiKit.Async.XMLHttpRequestError("Request failed")");
A.3 Customized selenium services as Nagios plug-in

#!/usr/bin/perl -w

use strict;
use warnings;
use Time::HiRes qw(sleep);
use lib "/usr/local/nagios/libexec";
use Test::WWW::Selenium;
use Test::More tests=>2;
use utils qw($TIMEOUT %ERRORS &print_revision &support);
use Test::Exception;
use Getopt::Long;
use POSIX;
use File::Basename;

use vars qw(
  $opt_help
  $opt_usage
  $opt_version
  $opt_critical
  $opt_warning
);

61
sub print_usage();
sub print_help();

open (STDOUT, ">file1") || die ("open STDOUT failed");

print("hello world");

my $progname = basename($0);

my %ERRORS = ('UNKNOWN'  => '3',
              'OK'       => '0',
              'WARNING'  => '1',
              'CRITICAL' => '2');

Getopt::Long::Configure('bundling');

GetOptions
(
    "c=s"  => \$opt_critical,
    "critical=s" => \$opt_critical,
    "w=s"  => \$opt_warning,
    "warning=s" => \$opt_warning,
    "h"    => \$opt_help,
    "help" => \$opt_help,
    "usage" => \$opt_usage,
    "V"    => \$opt_version,
    "version" => \$opt_version
) || die "Try \$progname --help for more information.\n";

62
sub print_usage() {
    print "Usage: $progname -w WARNING -c CRITICAL\n";
    print "$progname --help\n";
    print "$progname --version\n";
}

sub print_help() {
    print "$progname - check current selenium requests\n";
    print "Options are:\n";
    print " -c, --critical \n";
    print " -w, --warning \n";
    print " -h, --help display this help and exit\n";
    print " --usage display a short usage instruction\n";
    print " -V, --version output version information and exit\n";
}

if ($opt_help) {
    print_help();
    exit $ERRORS{'OK'};
}
if ($opt_usage) {
    print_usage();
    exit $ERRORS{'OK'};
}

if ($opt_version) {
    print "$progname 0.0.1\n"
    exit $ERRORS{'OK'};
}

##### DO THE WORK
###############################################################
my $sel = Test::WWW::Selenium->new( host => "localhost",
    port => 4444,
    browser => "*firefox",
    browser_url => "https://eweldpredictor.ewi.org/" );

my $value = $sel->open_ok("/");
$sel->click_ok("wrench_drawer");
chomp($value);
if ($value == 1) {

    print "$value    &&&*****%current selenium test!!!!"."\n";
exit $ERRORS{'OK'};

}

else {

    print_usage();

    exit $ERRORS{'OK'};

}