Evaluating the effectiveness of web application testing techniques using automated tools

EVALUATING THE EFFECTIVENESS OF WEB APPLICATION TESTING TECHNIQUES USING AUTOMATED TOOLS

BY

WEAAM A. ALRASHED

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTING
AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED
SCIENCE

Master of Applied Science (2018) McMaster University
(Software Engineering) Hamilton, Ontario, Canada

TITLE: Evaluating the effectiveness of web application testing techniques using automated tools

AUTHOR: Weaam A. Alrashed
B.Sc (Information Technology)
King Saud University
Riyadh, Kingdom of Saudi Arabia

SUPERVISOR: Dr. Douglas G. Down

NUMBER OF PAGES: xi, 93.

To my beloved Father "AbdulMohsen"

Wish you were here..

To my wonderful Mother "Madawey"

Who always encourages me to pursue my dreams..

To "AbdulAziz Al-Muammar"

You shall never be forgotten..

Abstract

The heterogeneous structure and dynamic nature of web applications have made the testing procedure a challenge. Producing high-quality web applications can be performed by conducting appropriate testing techniques. As a result, several white-box and session-based testing techniques have been proposed in the literature. In this work, the performance and effectiveness of these testing techniques are evaluated in terms of fault detection on a simulated PHP online bookstore. The testing techniques are examined with the use of PHPUnit, xDebug and Selenium automated testing tools. We believe that combining the testing techniques with appropriate automated testing tools (PHPUnit and Selenium) can be effective in terms of fault detection and time spent to construct and run test cases on PHP web applications. The results show that some testing techniques are preferred. We also identify categories of faults that are amenable to detection by each of the techniques, as well as categories of faults that are difficult to detect by any of the techniques. Moreover, using the automated tools has helped in automating the conduct of the tests and in reducing the time required to perform them.

Acknowledgements

I would like to acknowledge the people who have helped me throughout the exceptional journey of my Master degree. I am grateful to so many for their guidance and support.

I extend my sincerest gratitude to Dr.Douglas Down, my supervisor, for his guidance, encouragement and valuable critiques. His willingness to give his time so generously has been crucial in the completion of my thesis and it was a real privilege for me to share his exceptional knowledge.

My sincere thanks also go to Dr.Areej AlWabil, who has always been an inspiration to me for her dedication and hard work.

I would like to offer my special thanks to Matthew Dawson, who helped in completing an important phase of the testing procedure. Also, special thank you for all the participants who have taken out the time to complete the study.

Similarly, I thank my support system: Hailah, Manoree, Mishary, Abdulrahman and Feras who provided emotional support that has enriched all facets of my life, and for that I am so thankful.

To my mother, Madawey, I sincerely thank you - your belief in me has made this journey possible.

To my father, Abdulmohsen, who is always in my heart- you are missed.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In the past decade, web applications have been adopted in various domains because of their availability and extensive range of functionalities. It has been reported in [4] that there were around 3.26 billion Internet users in 2016. That number has grown in 2017 to be about 3.74 billion users. This huge number of Internet users shows the importance of providing dependable and high-quality web applications. Performing adequate and appropriate testing using different testing techniques can reveal faults and ensure the provision of reliable web applications to end users.

Web applications tend to generate dynamic web pages and elements. To accommodate these dynamic features some techniques and tools have been proposed to aid in the automated testing of the PHP scripting language, such as PHPUnit [12] and Selenium [10]. The former is used as a unit testing framework specialized for testing applications written in PHP, whereas the

latter is used in the acceptance testing phase to replay the behaviour of a user automatically by controlling the browser, filling input fields of forms with values and navigating links between pages. It has several components that include: Selenium Webdriver, Selenium RC, Selenium Grid and Selenium IDEs. Selenium has been introduced comprehensively in [8] where its various components that support different testing scenarios are illustrated.

We next provide a problem statement, an overview of the thesis and a literature review that presents similar testing techniques and related work in this domain.

1.1 Problem statement

Testing of web applications has always been a challenging task, due to their complex and compound structure. This heterogeneous structure is largely due to a combination of different languages interacting with each other when developing the client and server sides. The client-side is often composed of HTML to display the content of web pages, CSS to manage the layout and style of HTML pages and JavaScript to contribute in creating dynamic and animated content. On the server-side, PHP is one of the most popular scripting languages because it supports the creation of interactive pages [5], [18]. Testing these dynamic web applications can be costly and time consuming. Thus, a need has arisen for specialized and automated testing techniques and tools that take into account this dynamic nature. Moreover, studies have not adequately covered the techniques and tools that are specialized in detecting

specific types of faults in PHP web applications.

The main contribution of this thesis is to provide recommendations on the efficacy of some previously proposed testing techniques used in the detection of specific faults of different types and severities. Also, an overview of the automated tools used during the testing process is provided to help developers and testers interested in the testing of web applications written in the PHP language.

1.2 Thesis overview

In this work, the performance of some previously proposed testing techniques is evaluated in terms of fault detection on a simulated online bookstore written in PHP when used with the PHPUnit, xDebug and Selenium testing tools. We believe that combining the testing techniques with the appropriate automated testing tools can be effective in terms of fault detection and time spent to construct and run test cases on PHP web applications. These techniques are two white-box techniques (WB1 and WB2) introduced by Ricca and Tonella in [2] and two additional session-based testing techniques (US1 and US2) proposed by Elbaum et al. in [1] and [26] with one more hybrid approach that combines white-box and session-based techniques (WB2+US1).

Our approach is as follows. First, a UML model is manually generated to construct the test cases for two white-box testing techniques (WB1 and

WB2). Although the tools ReWeb and TestWeb used by Ricca and Tonella in [2] and [3] to create the UML model automatically are not publicly available, the guidelines described in [3] on how to create and produce the model are manually followed. Second, path expressions are extracted from the model to generate test cases for white-box techniques. Third, the simulated online bookstore is seeded with different types of faults to be examined by users in order to generate the test cases for user session-based techniques (US1 and US2). Finally, a hybrid (HYB) technique that is a combination of white-box and session-based techniques is evaluated.

The coverage metric is considered in measuring and reporting the covered test cases in a test suite and the covered lines of code in an application using PHPUnit and xDebug [19], respectively. The number and types of faults that have been discovered by each technique are reported and faults that have not been revealed by each technique are discussed in detail, with a goal of identifying the kinds of faults that may be difficult to detect with the proposed techniques.

The contribution of the employed tools to the proposed techniques is clearly visible in that they conduct the tests automatically, in particular they are fast and reliable. The coverage for both the test suite and the code is done automatically with the help of the PHPUnit and xDebug [19] tools. PHPUnit asserts that the content of a web page or the elements of a form exist and whether URL links, input fields and messages from an application are present as expected. xDebug is used as an extension for the debugging of PHP applications in general and provides code coverage results for web-based applications in particular.

The tools are extremely helpful, especially with the WB2 and US1 techniques, since they are the most efficient techniques in revealing faults when conducting the automated tests. Although it takes some time to write the tests in PHPUnit and Selenium, the benefits are significant if the web application will not undergo frequent modifications. The code of the test case itself is reproducible, it can be modified for multiple uses and previously written test cases can be reused. If there is a need to simply change the input values of a test case, then the old values of the input fields in previous tests can be replaced by the new values without the need to write a new test case. In addition, human intervention is not required as the tools help in automating the conduct of the tests. The overall approach is time efficient, accurate and cost-effective.

To the best of our knowledge, this approach has not been implemented before and reporting results can provide insight on its effectiveness for PHP web applications and in which scenarios each technique would be beneficial.

1.3 Literature Review

In any software development life cycle, testing is considered to be a fundamental procedure. A product must not be delivered without being tested. This ensures the delivery of a high-quality product by discovering and eliminating faults of different severities. However, some organizations could be under market pressure to release software or an application as soon as possible. This could affect the testing phase by not being able to test the product

properly. Therefore, a need for reliable and automated testing methodologies arises.

During the past years, several tools and techniques have been introduced for testing web applications. These testing procedures and tools concentrate more on the validation of non-functional requirements while only a few of the available techniques validate functional requirements (as reported by Elbaum et al. in [1]).

In efforts to fill this gap, a number of tools and techniques have been proposed. Capture and replay methodology is one of the popular examples of these tools that validate functional requirements. This approach records user behaviour for an application. When the program starts running, the input and interaction of a human tester or a user is recorded and is automatically replayed later in regression or functional testing without human intervention. Some examples of the type of tools that support capture and replay are IBM Rational Functional tester [17] and Selenium IDE [10].

The study in [22] proposed a web application test model, WATM, for representing, analyzing and testing the data flow of web applications. The proposed model is composed of the object model, where each web application artifact is represented as an object, and the structural model showing the data flow between the web application artifacts. After the WATM is constructed, a data flow testing methodology is implemented, and test cases are extracted. This approach aims to ease the testing of complex web applications.

Benedikt et al. [6] presented a testing tool, VeriWeb, that traverses the paths in a web site automatically and discovers the dynamic pages and their content, such as the elements and components of forms. It also uses the *SmartProfiles* concept, which fills out the elements of a form with values automatically. This concept was proposed to aid in filling forms without the need for a human tester to do so. The limitation of VeriWeb is that it can fail in detecting some navigational paths because it has a limit on the number of paths that it can traverse.

Another tool, Apollo, by Artzi et al. [7], has been implemented to help in the discovery of faults in PHP web applications. The concept is based on a technique that automates the creation of tests. This technique uses symbolic execution, where a program is examined to determine which input values resulted in the execution of a certain segment of a program. The technique also supports concrete execution where a program navigates one control flow path with a particular input value. In addition, Apollo is used to automatically provide input values for web application elements then report their behaviour for any potential issues that may occur such as execution failures (missing file or incorrect MySQL query, etc.) or HTML failures that will be detected by the use of an HTML validator. Finally, the tool uses an oracle (HTML validator) to compare the output of a given HTML document to predefined HTML syntax and report the errors in a bug report repository. Apollo was tested on six PHP web applications available online and resulted in effectively revealing some execution and HTML malformed faults, including invalid HTML syntax (an open tag without a closing tag or missing tags). However, the tool has some limitations such as it can be tested only on PHP web applications and is not applicable with other

scripting languages. Also, it considers restricted sources of input parameters as it obtains them from POST, GET and REQUEST arrays only.

Another automated system was proposed by Girgis et al. in [9] for testing web applications. This automated approach collects the content of HTML pages consisting of HTML tags, links and page names. After that, the data is retrieved and examined for possible errors. The errors include defective links (broken links or links that redirect to invalid pages), orphan pages or cookies errors. The system is made up of four main units, Web navigation, HTML analyzer, link execution and error checking. This system was tested on a web application that was seeded with certain types of faults and resulted in revealing all of the injected faults. This system, however, has not been tested against errors related to web forms as this feature is not supported. The effectiveness of the system cannot be generalized to all web applications as it was only tested on one case study.

Ricca and Tonella [2] have proposed a white-box testing technique which is based on UML (Unified Modeling Language). This model-based technique automatically transforms the complex structure of a web application into a model in order to ease its validation by generating test cases based on the representation of the model. The technique utilizes a UML model and data flow capabilities and uses a basic path testing approach to generate the test cases. It consists of two tools, ReWeb and TestWeb, where the former is used to produce a UML model of a given web application and the latter is implemented to help in generating the test cases from the model provided by the ReWeb tool. Although the tools appear to be powerful, no available online releases of them have been provided. This technique, however, requires

the intervention of a human tester in generating the input values to run the test cases. In some cases, this might be costly.

Another technique, proposed by Elbaum et al. in [1], is based on the use of logged user session data to create test cases and input values to carry out tests, removing the need for a human tester to produce the test cases and the input values. This approach has been suggested to be used as a new method when testing web applications without the necessity to inspect the source code of the product under consideration. The concept of the technique relies on user interactions in the system – the visited links and provided input. The logged user session data consists of the URLs that have been visited along with the (name-value) attributes that are recorded in the access log of the server. The study compares the effectiveness of two white-box testing techniques proposed previously in [2], two session-based techniques and one hybrid approach that combines the capabilities of both techniques. The study showed that the white-box techniques and session-based techniques are effective in terms of fault detection and that they complement each other. This user session-based approach provides an important contribution to the generation of test cases. Its main strength can be seen through the time and energy required to produce input values for test cases. Also, it does not depend completely on the source code of the web application as it gathers data from user sessions, in comparison to white-box testing that depends entirely on the source code to create the test cases, see Mendes and Mosley [11].

Further studies by Sampath et al. in [15] and [21] have considered the approach of using user sessions that were logged by the server. The studies have analyzed user session data to investigate the efficacy of concept analysis. This approach reduces the test suite size by selecting some of the logged user sessions. The test suite size is reduced by choosing user sessions (test cases) that cover all of the system's requests. Also, test cases that cover requests that were not covered by the previously selected user sessions were also considered. The authors have also introduced some heuristics such as the *one-per-node* heuristic, the *test-all-exec-requests* and the *k-limited* heuristic for test case selection. The studies conclude that the result of the coverage of this approach is similar when compared to the original test suite, which consists of the original (unreduced) set of user sessions. They also find that their approach fails to detect some of the faults detected by the original test suite.

In [13], De Jesus et al. have proposed an additional technique that is based on the tasks to be performed by a web application. This approach generates test cases by analyzing a web application model that includes all of the available valid paths to perform a specific task. It leverages the use of the logged sessions of the developer when conducting different tasks in the GUI (Graphical User interface) of an application with the help of the UsaTasker tool [14]. Several graphs are produced by extracting the navigational routes performed by the developer and documented by the tool when testing the application. The study concludes that this task-based approach can provide relatively high coverage in relation to functional requirements.

In [20], Di Lucca et al. have developed an object-oriented model structure for web applications. This model representation facilitates the testing of each unit of a web application and provides an approach to test the integration of these units as a group. A set of tools for testing the web application has also been proposed, composed of an analyzer, a repository for storing the web application model, test cases, test log and associated database, a test case generator and a test case executor. The approach and tools have been evaluated on a case study to examine their effectiveness. The study concludes that the presented approach and tools have helped in conducting some of the testing events in an automated manner. However, the general correctness and usefulness has not been verified beyond a single case study.

Another study, performed by Milani et al. in [16], explored the efficiency of combining automated crawler abilities to automatically traverse the web application and provide the available paths. The test cases were written manually by a tester using Selenium [10] tool. The proposed approach has been executed on four open source web applications through the development of a tool called Testilizer. It has been reported that the proposed tool and methodology provided an increase in fault detection and coverage when compared to the original test suite, consisting of test cases (web application paths) constructed and written manually.

Chapter 2

Methodology

2.1 Procedures

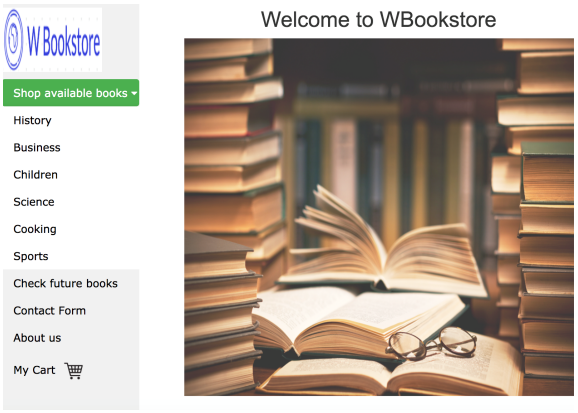
In this section an overview of the development of a web application, the UML model creation, the fault seeding procedure, recruiting participants and the web application testing methodology are discussed.

2.1.1 Web site creation

A simulation of an online bookstore is constructed using the PHP scripting language, MySQL for database management, HTML for the layout of the web pages and JavaScript for creating dynamic content. The web application is

hosted locally on an Apache web server. A number of functionalities have been built into the web application. Users can navigate different categories: History, Business, Children, Science, Cooking, and Sports. They can also purchase the available books in these categories. Moreover, they can check the future books that will be available in three different categories, and can contact the bookstore via the "Contact Form" page. In addition, the website provides the possibility to navigate to an "About us" page, to view the "My Cart" page and to purchase books by navigating to the "Checkout" page. Finally, a user can delete any previously added books in their cart. Some screenshots of the web application can be seen in Figure 2.1. The first screenshot in Figure 2.1a shows the available functionalities of the web application while Figure 2.1b shows the available books for purchase in the "Business" category. This page shows the book cover, the name of the book, its price and a quantity field to add the required quantity of the desired books in the cart. The size of the web application is considered to be small. It has basic functionalities but is still sufficient to conduct the tests and evaluate the testing techniques. There are 21 pages with around 3000 lines of code. Among these pages, 14 are written in the PHP language with a total of 387 lines of code.

Two procedures are conducted in order for the study to be performed: first, fault seeding the web pages then testing of the faulty web application by users.



(a) Index page.

(b) "Business" category page.

Figure 2.1: Screenshots of the web application.

2.1.2 Model creation

A UML model for the web application is created manually based on the guidelines provided by Ricca and Tonella in [3] to represent the functionalities in a clear and simplified manner. The model consists of nodes and edges, where the former represents the web application objects such as the web pages and forms while the latter represents the relationships among these

objects and how pages are generated dynamically. Appendix B shows the constructed model representing the whole web application. After the creation of the model, path expressions are produced to represent the routes that the user might visit when exploring the web application as proposed by Beizer in [27]. The partial UML model shown in Figure 2.2 represents the "Business" category page of the web application that was provided earlier in Figure 2.1b.

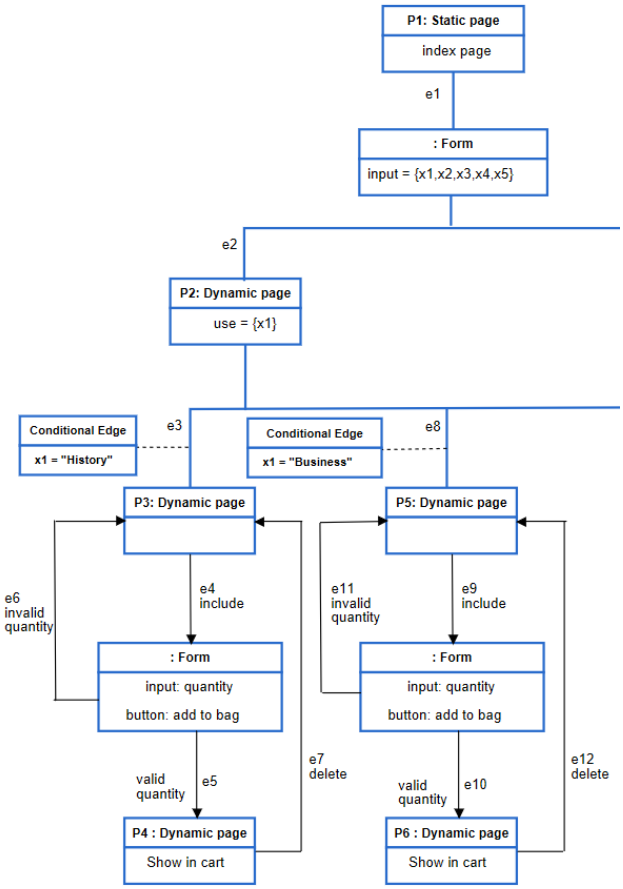


Figure 2.2: A partial view of the constructed UML model for the "Business" category page.

In Figure 2.2, x1 represents the option of "Shop available books" from the side menu, which can be one of the six categories (History, Business, Children, Science, Cooking, and Sports). These paths help in creating the test suite, so the tester will not neglect any potential route when writing the test cases. A detailed description of the path expressions is outlined in the next section.

2.1.3 Path expressions

The path expressions are generated from the UML model given in Figure 2.3.

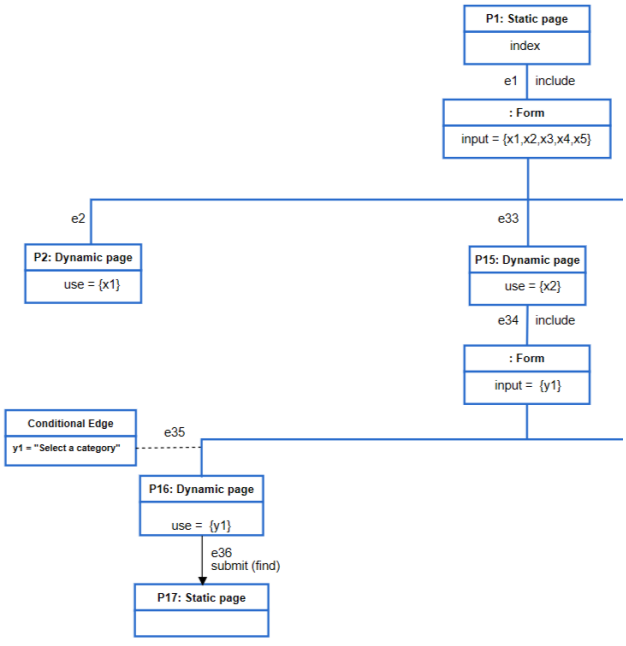


Figure 2.3: A partial view of the UML model constructed for the "Check future books" page.

This model represents a partial representation of the "Check future books" functionality in the web application and its path expression can be constructed as follows: $e1\ e33\ e34\ e35\ e36$. After that, the path expression is then transformed to a test requirement which is a set of URLs and (name-value) pairs to form a test case to be executed on the web application.

1. $e1$: `http://localhost:8888/online_shopping/index.php` (open the main page)
2. $e33$: `http://localhost:8888/online_shopping/Book_Search.php` (select "Check future books" option)
3. $e34$ $e35$: `http://localhost:8888/online_shopping/Search.php?query=0` (select "Select a category" option)
4. $e36$: `http://localhost:8888/online_shopping/Search.php?query=0&find=Find` (click "Find" button)

2.1.4 Fault seeding

In order to evaluate the testing techniques, a web application with some faults is needed. An experienced student participated in the fault seeding procedure. The need for a participant who has no knowledge about the testing techniques and the project is essential to minimize any types of bias that may occur if the faults were to be seeded with prior knowledge about the test cases. If the tester is the same individual who is seeding the faults and writing the test cases, then these injected faults may be selected and added unconsciously in a manner that is more likely to be detected by these written test cases. This process involved modifying some parts of the code of

the simulated online bookstore in order to create a failure. The participant was instructed on how to perform this procedure by providing a detailed description of the required types of faults (Table 2.1). These fault types follow those recommended in [1] and [26].

Table 2.1: Types of fault considered in web application testing.

Fault types	**Definition**
Scripting faults	1. Addition/deletion of variable definitions (such as. variable type and links). 2. Changing the values and rearranging the conditions of execution. 3. Changing the predefined values of variables. 4. Function calls addition/removal.
Forms faults	The name of the form or predefined values for a name.
Database query faults	The modification of a query expression. Such modifications could have an effect on which table to access and which fields within a table to retrieve data from, or in searching for values related to key or record values.

The participant was also encouraged to consider seeding faults where the detection is at different levels (simple, medium, hard) and to keep them as realistic as possible. These modifications have caused different types of faults in the website. This procedure is essential to evaluate the effectiveness of the techniques in terms of fault detection.

The participant effectively seeded 63 faults. Three of these faults were ignored as they were not relevant (an explicit list of the seeded faults is pro-

vided in Appendix A). Also, nine more faults were discovered while revising the website and were added to the seeded faults. So, the total number of faults was 69.

It is a challenge to find a knowledgeable participant who can understand, analyze and examine the source code and the different types of faults then inject these faults in the appropriate locations in the web application. This critical task, if conducted properly, can provide high quality results about the performance of the testing techniques applied. The knowledge of the participant in the various fault types and the language of the written code is crucial in performing the study and evaluating the testing techniques accurately. If this task is not conducted correctly, seeded faults could be injected such that they all represent one level of severity (all the seeded faults are simple to detect or very hard to detect). Also, the number of faults injected could be small due to the minimal knowledge that the seeder has about the fault types and where they should properly be seeded. As a result, the evaluation of the testing techniques could be compromised. It has been observed that describing the code for the participant, providing clear guidelines on how to conduct the fault seeding procedure and explaining the different types of fault have been beneficial in achieving the desired results and have greatly helped in expediting the time spent to perform this procedure.

One important observation about the fault seeding procedure was that, the work that has been conducted on fault classification for web-based applications seems to be limited. A clear and descriptive standard list of the different fault types for web applications would be a valuable addition.

2.1.5 Participants

Twenty-one participants were recruited via an email that was sent to all students in the Department of Computing and Software to request participation in this study. The experiment received ethics approval and took place in an office at McMaster University. The volunteers fulfilled two roles. One participant performed the fault seeding of the web application while the rest conducted the testing of the faulty web application. Prior to conducting the study, the participants were provided with a consent form that describes the study, their role and that their participation was to be conducted anonymously. The participants were given the option to withdraw from the study at any time. At the end of the study, the participants were provided with an electronic copy of their consent.

2.1.6 Testing the web application

Twenty participants were asked to examine and test the faulty web application. They were asked to enter random and non-personal data in the input fields of the website. Each session lasted no longer than 15 minutes. After that, the navigated URL and (name-value) pairs that have been entered during the session are extracted from the Apache access log. The data is then used to evaluate the effectiveness of the fault detection capability of user session testing techniques and whether these different inputs can reveal any of the injected faults. The user session testing techniques include US1 and US2. The US1 testing technique directly replays user sessions performed on

a web application. The US2 technique combines different user interactions and produces new artificial user sessions to be replayed.

Conducting a study that requires the recruitment of participants is not always a trivial task. However, the use of participants can provide some useful insights on the effectiveness of the testing techniques in a more natural way as the participants are real users of the web application. During the conduct of the study, some participants tried to provide valid data in spite of the fact that they were informed of the possibility of using erroneous data in the input fields. This in turn resulted in a minimal fault detection rate. User sessions could have detected more faults if the participants had consider the use of erroneous data in the input fields without specifying the exact types of these invalid data. Doing so can affect the correctness of the outcomes of the tests. Furthermore, users who were asked to perform typical behaviour had minimal fault detection rates when compared to users who were informed of the possibility of providing invalid data that can cause the web application to act in an abnormal manner. Users who provided invalid values managed to reveal some faults related to unacceptable input values. However, it is worth noting that natural user behaviour was able to reveal faults related to invalid links and navigational issues.

The number of conducted sessions could have been increased if the web application were hosted publicly and a link for the web application was sent to participants in order for them to use the website at their convenience.

Lessons learned for conducting a study that requires the involvement of participants The following points summarize the key lessons for studies specialized in testing web applications involving participants.

- Studies that are based on user participation require ethics approval that can take quite a while. Planning ahead regarding completing such documents is advisable.
- It is important to explain the study to participants before the start of the session. Making them understand the study itself is crucial to obtain good results.
- The server should be configured to allow capturing user interactions in the Apache access log.
- The web application should be hosted on a public or local server in order to conduct the study.
- The process of seeding faults should be performed on a separate day from when the user sessions are conducted. The web application should be prepared before participants start to use it.
- Cookies regarding each user session should be erased before the start of the next user session (this applies if the study is conducted locally on one computer). This allows each user to have a unique session id that differentiate each user.

2.2 Testing techniques

In this section the five techniques used for testing the web application are explained.

2.2.1 WB1

This technique has been proposed by Ricca and Tonella in [2]. It constructs the path expressions for the UML model and uses a tester to manually provide values for the input fields. In this technique, textual differences are ignored when testing dynamically generated pages. As a result, it only tests the correctness of the generated web pages via the URL link without looking at or comparing the content of the page itself to the original valid web pages. For testing the input fields, one random input value is selected when testing each of the valid and invalid situations. The acceptable input values for an input field that accepts three digits only are any set of numbers of length three. The invalid values that can be tested are any letter from the alphabet, a special character and digits of length four or more. The invalid test cases depend on the knowledge of the tester and testing plan as there is no specific methodology to be followed when constructing invalid test cases. This is because the WB1 technique creates test cases based on the generated path expressions that take into account testing one valid case and another path that represents an invalid case. The input selection for each test case is performed randomly by the tester based on the type of the input field (numeric or letter input field). So, if the input field accepts only numeric values

then the tester selects one case that exercises the valid test case (a random numeric value is chosen based on the knowledge of the tester). Another test case examines the invalid situation where values are selected from the letters or special characters category. An example can be seen in Figure 2.4, a screenshot of the "Checkout" page that shows an input field *First Name* that accepts only letters, where the minimum acceptable number of letters is three and the maximum acceptable number of letters is five.

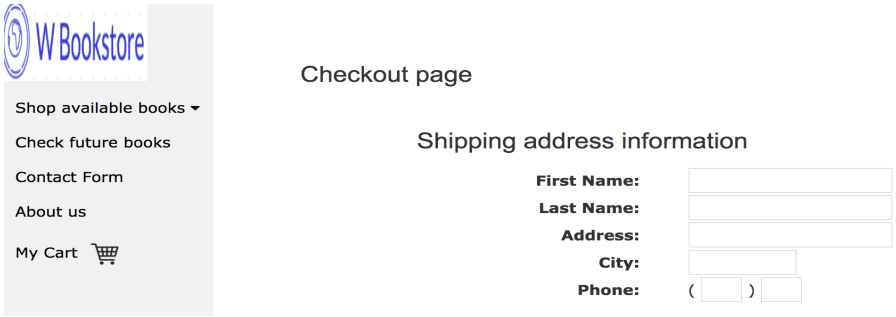


Figure 2.4: A screenshot of the "Checkout" page.

The valid test case for the *First Name* input field is shown in Listing 2.1.

Listing 2.1: A valid test case for the *First Name* input field.

```
public function test_Checkout_fname_valid()
{
$driver ->get("http://localhost:8888/online_shopping/Faulty/index.
    ↪ php");
$driver ->findElement(WebDriverBy::XPath("//a[@href='MyCart.php']"))
    ↪  ->click();
$driver ->findElement(WebDriverBy::name('checkout')) ->click();
$fname= $driver->findElement(WebDriverBy::name("fname")) ->sendKeys(
    ↪ 'Weaam');
}
```

The invalid test cases are shown in Listing 2.2, which exercises the input field against special characters and numbers. The invalid test cases can be separated into two test cases but here they have been combined into one test case.

Listing 2.2: An invalid test case for the *First Name* input field.

```
 public function test_Checkout_fname_invalid()
{
$driver ->get("http://localhost:8888/online_shopping/Faulty/index.
    ↪ php");
$driver ->findElement(WebDriverBy::XPath("//a[@href='MyCart.php']"))
    ↪  ->click();
$driver ->findElement(WebDriverBy::name('checkout')) ->click();
$fname= $driver->findElement(WebDriverBy::name("fname")) ->sendKeys(
```

```
    ↪ '1234%$#');
}
```

Null values (the empty string) are ignored in the case of an invalid state. Circular links and navigational paths that appear in each page are also not considered. Circular links connect the nodes in a model in a manner such that the nodes could eventually form an infinite loop of not being able to provide the needed data or perform their required tasks. Navigational paths are the links that are usually provided in the left-side or on the top of the page to facilitate the movement between pages.

The test cases are generated according to the path expressions that were produced based on the UML model that was constructed for the web application used in this study (the complete model is provided in Appendix B). Path expressions are essential for creating the test cases. A partial view of the UML model can be seen in Figure 2.5. Values are generated and assigned to the input fields by a human tester based on the aforementioned guidelines that describe acceptable and unacceptable values for testing input fields. Then after generating all the test cases and assigning values to their input fields, the test suite is executed against the faulty web application.

This partial model represents two test cases and has been added here to provide a clear description of the generation of these test cases. The first test case exercises the available books for purchase in the "History" category. The second test case exercises the available books for purchase in the "Business" category and is described in detail in the WB2 technique discussed in the next section. The path expression for the first test case is: $e1\ e2\ e3\ e4\ e5$,

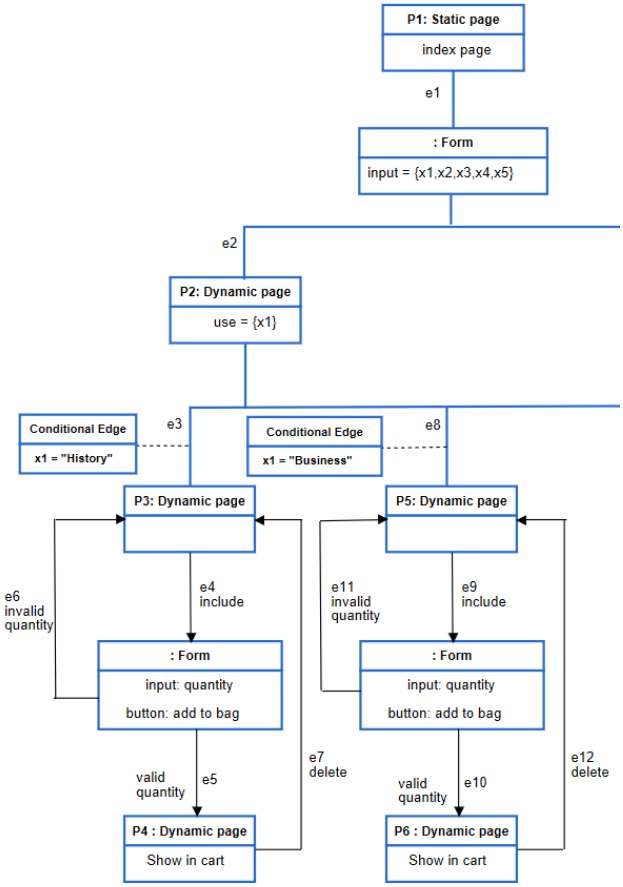


Figure 2.5: A partial view of the constructed UML model for the "History" and "Business" category page.

where $e1$ is the path from the entry node (index page), $e2$ is the selection of the first choice "Shop available books" from the main menu, $e3$ represents the selection of the "History" category page, $e4$ provides a valid input value, 1, in the quantity input field and finally, $e5$ is the valid submit by clicking on the "Add to cart" button.

The test cases are written in PHPUnit and Selenium, see Listing 2.3.

Listing 2.3: WB1 code sample.

```
1  public function test_History_Success()
2  {
3  $host = 'http://localhost:4444/wd/hub';
4  $capabilities = DesiredCapabilities::chrome();
5  $driver = RemoteWebDriver::create($host, $capabilities, 3000);
6  $driver->get("http://localhost:8888/online_shopping/Faulty/index.php
       ↪ ");
7  $driver->findElement(WebDriverBy::XPath("//button[contains(.,'Shop
       ↪ available books')]"))->click();
8  $driver->findElement(WebDriverBy::XPath("//a[@href='History.php']"))
       ↪  ->click();
9  $driver->findElement(WebDriverBy::name("quantity"))->sendKeys(1);
10 $driver->findElement(WebDriverBy::name('add'))->click();
11 $driver->manage()->timeouts()->implicitlyWait = 10;
12 $currentURL1 = $driver->getCurrentURL();
13 $exp_url1 = "http://localhost:8888/online_shopping/Faulty/History.
       ↪ php";
14 $this->assertEquals($exp_url1, $currentURL1);
15 }
```

Line 1 defines the name of the function, lines 3, 4 and 5 connect to the server and launch the Chrome web browser to open the web page. Lines 6, 7, 8, 9 and 10 open the website index page by providing the URL, select the menu option "Shop available books", select the "History" category, add the value 1 to the quantity input field and click on the "Add" button which adds

the desired book to the cart. Line 11 instructs the web driver to wait for 10 seconds, then line 12 gets the URL of the current page. In line 13, a variable *$exp_ url* is initialized and assigned a URL. Finally, line 14 compares the two URLs and asserts if they are equal or not by providing a boolean result (true if equal, false otherwise).

This technique is straightforward to implement. Nonetheless, providing values manually by a human tester requires time and effort, especially if there are many input fields in the web pages and the web application size is large. Using this technique with PHPUnit and Selenium is sometimes a challenging task. Instead of spending time in installing these testing tools, learning them and writing code to open, navigate pages and provide input values, this could be done manually by a tester. However, if there is a need to repeat the same tests, then doing it automatically through these tools will definitely save time, cost and effort. In particular, the savings could be significant with medium and large scale web applications.

One of the limitations of this technique is that it does not consider textual differences and only compares the URL of the dynamically generated page. This can be problematic because if the content of the generated web page is incorrect, it will compare the URLs only (URL of the valid original page and URL of the generated page) and will decide that the generated page is correct when it is not. Another limitation is that there are no provided guidelines for the input values that should be used for both invalid and valid cases. The tester is relied upon to select the data and perform the tests. Also, null values (the empty string) are not considered, it is for the tester to decide whether to include them or not.

This technique can be helpful if the tested website needs only to verify the URLs of the pages. Implementing this with the aid of the automated testing tools PHPUnit and Selenium can be beneficial.

2.2.2 WB2

Conceptually, this technique does not differ much when compared to WB1 except that it provides more thorough test cases and considers exercising a random value that represents each group of the valid and invalid data sets. These data sets are the predefined acceptable and unacceptable values in the input field which can include for example, letters category, digits and special characters categories. For example, if there is an input field that accepts 10 letters only, then the valid test case exercises one combination of any 10 characters. On the other hand, the invalid cases exercise a mixture of letters less than 10 from the alphabet, combination of letters that are more than 10, any number of the special characters class and any number of the digits class. So, it takes into account testing various cases of input values for the acceptable and unacceptable values in each input field. Moreover, the textual differences when generating the dynamic pages are considered. This is an important testing strategy as it assures that the generated page is valid by checking its content and comparing it to the original page without making a decision based only on its URL. The details on how the textual differences were measured have not been provided. So in the absence of such information, assumptions have been made to detect textual differences by asserting that the desired text appears without looking at any additional content such as the images of books. This of course requires a human tester to compare the

content of the generated page with the original saved web page. However, comparing the textual content can be performed automatically with the use of the automated tools PHPUnit and Selenium. The images included in the web pages require a human tester to determine their presence or absence because such a feature is not yet provided by PHPUnit or Selenium.

This technique also adopts the "boundary values testing" methodology for testing the input fields. The selection of input values is within the boundaries of the data domain such as values that can be found in the maximum and minimum of the boundaries of a given input domain [23]. So, if an input field accepts numbers between two and 12 then the boundary test cases exercise four valid situations and two invalid. For the valid test cases, the first case tests the minimum acceptable number which is two, the second case tests a number above the minimum such as three, the third case tests the maximum acceptable value which is 12 and the fourth case tests a value below the maximum number, such as 11. As for the two invalid test cases, one test case that tests a value below the minimum such as one and another tests a value above the maximum such as 13. The advantage of this methodological approach is that it is time and cost effective as there is no need to test all the values in the data set from 2 until 12 for the valid test cases. So, if the test is successful/unsuccessful for the chosen values that lie within particular input boundaries, then the test should pass/fail for all the other values that are within these particular boundaries.

Testing null values is important to avoid any missing required data from users. In addition, the approach "each condition/all conditions" is considered as it tests the input fields against null values (representing the empty string).

In particular, the test suite is produced by covering the input fields in which for each input field, a test case is generated that has a random acceptable value assigned to the first input field while the other fields are assigned the null value. The random acceptable value is any value that is valid to be used in the input field and is accepted by predefined instructions such as if there is an input field that accepts digits only, then the random accepted value is any digit. This step is repeated for all of the input fields found in the page. After that, one more test case is added to the test suite where all of the input fields are assigned a random acceptable value. Finally, the test suite is executed against the web application. This method is beneficial in detecting the fields that accept null values when they should not. It also provides the tester with clear instructions on how to test the input fields against the null value. It is thus easier for the tester to conduct the tests in a systematic manner.

This procedure requires time, cost and provides extra test cases for every input field. If the web application is large, then this can be both time consuming and costly.

Figure 2.6 shows a partial view of the UML model. The full UML model can be seen in Appendix B. This partial model represents the task of adding a book from the "Business" category. The path expression is $e1\ e2\ e8\ e9\ e10$, which illustrates the paths needed for navigating to the "Business" page. The test requirement for the path expression is as follows:

1. $e1$: `http://localhost:8888/online_shopping/index.php` (open the main page).

2. $e2$: click on the option "Shop available books" from the menu.
3. $e8$: http://localhost:8888/online_shopping/Business.php (select the "Business" category).
4. $e9$: http://localhost:8888/online_shopping/shopB.php?quantity=1&name=Predictably+Irrational+Book (select the desired book by adding the valid quantity, 1, in the quantity field).
5. $e10$: (click on the "Add" button to add the book to the cart).

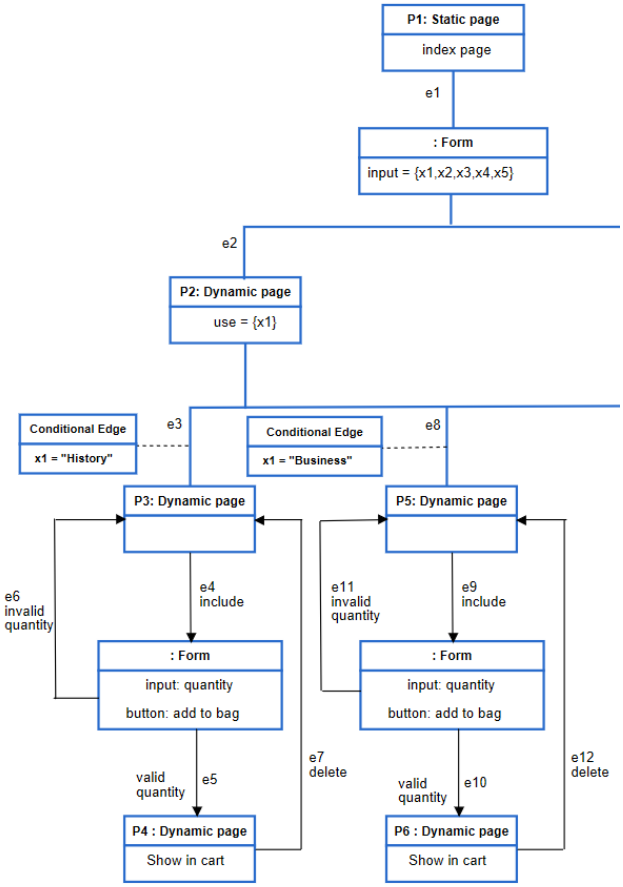


Figure 2.6: A partial view of the UML model for "Shop available books" from the "Business" category page.

The test suite is written in PHPUnit and Selenium, a sample of the code is given in Listing 2.4. This code shows one test case for successfully adding a book from the "Business" category. Line 1 defines the name of the function, lines 3, 4 and 5 connect to the server and launch the Chrome web browser to open the web page. Lines 6, 7 and 8 open the website index page by providing the URL, select the menu option "Shop available books", then select the "Business" category. Lines 9-14 check if the page contains the given book names. Lines 15-20 assert that the book names in the page match the given names via the *"assertEquals"* method. Line 21 adds the first book to the cart. After that, lines 22-24 compare the retrieved URL with the expected URL that the web page should navigate to after adding a book.

Listing 2.4: WB2 code sample.

```
1  public function test_Business_Success()
2  {
3  $host = 'http://localhost:4444/wd/hub';
4  $capabilities = DesiredCapabilities::chrome();
5  $driver = RemoteWebDriver::create($host, $capabilities, 3000);
6  $driver->get("http://localhost:8888/online_shopping/Faulty/index.php
       ↪ ");
7  $driver->findElement(WebDriverBy::XPath("//button[contains(.,'Shop
       ↪ available books')]"))->click();
8  $driver->findElement(WebDriverBy::XPath("//a[@href='Business.php']")
       ↪ ) ->click();
9  $bookName1= $driver->findElement(WebDriverBy:: XPath("//h5[contains
       ↪ (.,'Predictably Irrational Book')]"));
10 $bookName2= $driver->findElement(WebDriverBy::XPath("//h5[contains
       ↪ (.,'The Obstacle is the way')]"));
```

```
11 $bookName3= $driver->findElement(WebDriverBy::XPath("//h5[contains
       ↪ (.,'The Silent Language of Leaders Book')]"));
12 $bookName4= $driver->findElement(WebDriverBy::XPath("//h5[contains
       ↪ (.,'Good to Great Book')]"));
13 $bookName5= $driver->findElement(WebDriverBy::XPath("//h5[contains
       ↪ (.,'The undoing project A friendship that changed our minds')
       ↪ ]"));
14 $bookName6= $driver->findElement(WebDriverBy::XPath("//h5[contains
       ↪ (.,'This I know marketing lessons from under the influence')]
       ↪ "));
15 $this->assertEquals('Predictably Irrational Book', $bookName1->
       ↪ getText());
16 $this->assertEquals('The Obstacle is the way', $bookName2->getText()
       ↪ );
17 $this->assertEquals('The Silent Language of Leaders Book',
       ↪ $bookName3->getText());
18 $this->assertEquals('Good to Great Book', $bookName4->getText());
19 $this->assertEquals('The undoing project A friendship that changed
       ↪ our minds', $bookName5->getText());
20 $this->assertEquals('This I know marketing lessons from under the
       ↪ influence', $bookName6->getText());
21 $driver->findElement(WebDriverBy::name('add'))->click();
22 $currentURL1 = $driver->getCurrentURL();
23 $exp_url1 = "http://localhost:8888/online_shopping/Faulty/Business.
       ↪ php";
24 $this->assertEquals($exp_url1, $currentURL1);
25 }
```

The WB2 technique is powerful in that it considers various cases when testing the web application, in particular when testing the input fields. These cases include: considering null values when testing the input fields as part of the "each condition/all conditions" strategy, the performance of "boundary values testing" for input fields and the textual differences are taken into account to avoid a "true negative" situation. This situation means that when comparing two pages with the same URLs but different content, this can result in a positive outcome which indicates that the result is true but in fact it should be false. However, if the URLs and content are both being compared to the original page then this situation can be avoided. Comparing the URLs only for the generated pages is an insufficient indicator and textual difference should be considered.

One limitation of this methodology is that when comparing images in the web pages as part of the textual differences, the presence of a human tester is required to assure the validity and existence of the images in the pages. Moreover, writing these thorough test cases requires time and effort. With large scale projects, this technique can be overwhelming even though it provides high coverage for all possible test cases. Another limitation is the production of input values for testing the input fields as this can take some time to be generated. Using the automated testing tools PHPUnit and Selenium with this technique can save time, cost and effort if the web application is planned to be tested again in the future.

2.2.3 US1

This technique relies on collecting user sessions after using the web application. All user requests (the URL and name-value pairs) are recorded in the Apache access log as http requests, see Figure 2.7. This is performed by configuring the server to enable the collection of Get requests in the access log, where the activities of users on the server are reported.

```
:1-"GET/Shop/Children.php" sessionid:50efe364372e
:1-"GET/Shop/shopC.php?id=11&quantity=1&name=Harry+potter&
    price=15" sessionid:50efe364372e
```

Figure 2.7: User session request sample.

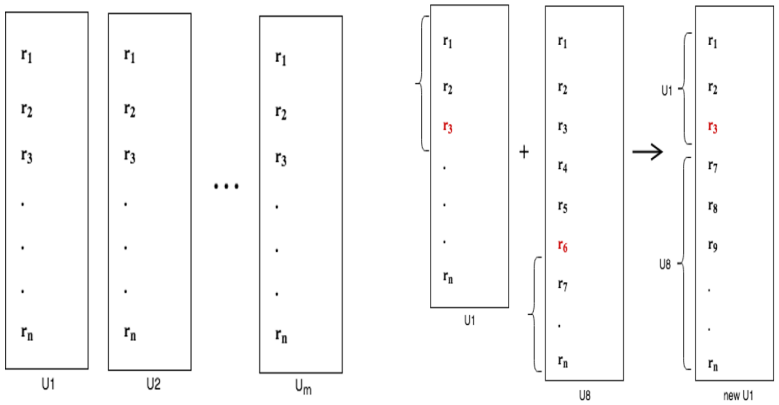
This user session-based approach has been presented by Elbaum et al. in [1] to overcome the limitation of the high cost in generating different input values by a human tester when exercising the web application, as in the WB1 and WB2 techniques. The US1 technique provides a variety of input selections that exercise the input fields in the web application without the need for a human tester to generate inputs. This avoids the potential high cost and is time-efficient. After users complete the sessions, the collection of their requests is done by viewing the server access log then replaying the sessions against the faulty web application. The test suite is constructed by writing the sessions in PHPUnit and Selenium, see a sample of a test case for a user session in Appendix C.1.

This code represents a replay of a user session that has been extracted from the server's access log after it has been analyzed. After the analysis phase, the required data is considered and the session is transformed to be written in PHPUnit and Selenium code. Finally, during the execution of this test case, a human tester monitors the session and reports any discovered faults.

The US1 technique provides the benefit of having available test data without the need to generate it. Having said that, extracting the data from the access log and filtering the required (name-value) pairs are not trivial tasks. Also, one disadvantage of this technique is that it relies mainly on user input. As a consequence, if users provide input values that are not relevant or are not beneficial for the test then this technique can be ineffective. Moreover, having a huge number of sessions does not guarantee effective fault detection. Some user sessions can be redundant as similar actions are often performed. It has been observed that users who were encouraged to provide erroneous data in the input fields, their sessions detected more faults than the users who were not encouraged to do so. Replaying the sessions with the help of the automated tools is an overwhelming task, especially with user session-based techniques. An alternative approach is to use the automated Selenium IDE to view the required website by providing its URL, then perform the tests and save them for future use. This IDE, however, can only be installed as an add-on in the Firefox browser. This tool can help in avoiding the enormous time spent in writing all user interactions in PHPUnit and Selenium.

2.2.4 US2

Conceptually, US2 is analogous to US1, in that it relies on user sessions to test the web application. However, US2 does not replay user sessions directly but mixes requests from two sessions together in order to generate a new artificial test case to replay. The reason for this is to find if there are any undetected faults that might be uncovered through the use of conflict users requests and whether they find hidden faults that the aforementioned techniques (WB1, WB2 and US1) cannot reveal. The conflict users requests include conflicting data that are simultaneously provided by different users when using the web application. In our case this situation is not considered due to how the sessions were conducted. That is, only one user at a time can use the web application and after the session ends another user can start. Although the conflict users situation does not apply to our web application, the evaluation of this technique is still considered to explore the other types of fault that it might discover.



(a) User sessions requests. (b) Mixing two sessions requests.

Figure 2.8: Example of US2 procedure.

Let U represent a set of sessions $U_1 \ldots U_m$. U_x, where $x \in \{ 1 \ldots m \}$, contains several requests $r \in \{ r_1 \cdots r_n \}$ as can be seen in Figure 2.8a.

The US2 procedure consists of the following steps:

1. Select a random session U_a from the pool of user sessions.
2. Select a random request r_i, where $1 < i < n$, in session U_a then copy all of the requests beginning from request r_1 to r_i. The request r_i can be a URL only or a URL along with its (name-value) pairs.
3. Select a random session U_b, where $b \neq a$ and search for a request r_j with identical URL as request r_i.
4. Copy all of the requests that follow request r_j and append them after request r_i.
5. Finally, session U_a is marked as used. This procedure will be repeated until all of the sessions in U are marked as used. See Figure 3.1 for a visual description of this procedure.

Note that when applying the US2 procedure after selecting session U_a, the requests that follow r_i are ignored when combined with r_j. These requests can differ from the given user requests and can be beneficial in revealing some types of fault that the other requests cannot. Applying this technique on a large number of sessions with an enormous number of requests can be impractical due to the time required to implement it.

It may have been worthwhile to modify the study to allow several participants to use the web application simultaneously. However, this requires having several computers available on site because in our case the web ap-

plication was hosted locally. One solution to this issue could be considered which is performing the study remotely via hosting the web application on a public server. Doing so eliminates the need for multiple computers at one location. Also, the number of available books in the database should be modified and an appropriate validation for the number of allowable books for purchase should be added.

2.2.5 Hybrid approach HYB (WB2+US1)

In this technique, presented in [1] and in [26], user session data from US1 is used along with the test requirements generated by the WB2 technique. In the first study [1], WB1 is used with US1, not WB2. However, in the second study [26], WB2 is used with US1. Here, we opted to use WB2 (because it provides thorough test cases) with US1 to perform the HYB approach.

Each of these generated test cases in the WB2 technique is translated into a URL request. After that, each URL is matched with similar requests in user sessions. Finally, the empty attributes of the URLs are filled with the appropriate values obtained from requests in US1 sessions. This procedure is performed on all of the WB2 path expressions until all of these test requirements are marked as used. Thus, a set of executable test cases is produced. The purpose of combining the two techniques (WB2 and US1) is to maximize the use of white-box testing techniques by using the comprehensive test cases. Filling the empty attributes in the test requirements of WB2 with US1 values might minimize the cost of generating input values by a tester. Moreover, it is of interest to explore the fault detection effectiveness of using

two approaches together. Also, it is of interest to examine if the HYB technique can detect not only the faults that have been detected by the WB2 and US1 techniques individually but other faults that these techniques separately could not reveal.

Implementing the HYB technique does not involve a lot of effort because the same code written in PHPUnit and Selenium for the test cases that belong to WB2 is used by simply replacing the input values to match those provided by US1.

2.3 Data Collection Tools

In this section a description is provided of the tools used in the case study.

2.3.1 PHPUnit

PHPUnit [12] is a testing framework for PHP applications that was released in the year 2004. Its main speciality is in Unit testing, where each individual unit of the source code is tested to ensure that it is working as expected. PHPUnit provides assertions methods used to state if content exists and functions that decide if the result is correct or not based on given values. Another feature that PHPUnit provides is *Data providers*. *Data providers* are test methods which accept arbitrary arguments of data in an array to test input fields automatically. It also provides links navigation where each link

can be visited by providing its URL. Some of the other testing frameworks are described in Table 2.2.

Table 2.2: Different testing frameworks for PHP.

Framework name	**Description**
Codeception [24]	A PHP testing framework that enables the conduct of Unit tests, Functional tests and Acceptance tests. Its code is more descriptive than PHPUnit.
SimpleTest [25]	A PHP framework that is similar to PHPUnit in its functionalities. It conducts Unit tests only and is easy to install and use when compared with PHPUnit. However, unlike PHPUnit it is not integrated with many PHP IDEs such as Netbeans and PHPStorm.

PHPUnit is chosen as it is the standard framework when testing PHP applications. Furthermore, it is stable, well maintained and supports both Windows and Linux operating systems.

2.3.2 Selenium

Selenium [10] is a tool developed for web application testing. It is commonly used in the acceptance testing phase to replay the behaviour of a user automatically by controlling the browser, filling input fields of forms with values and navigating links. It is composed of various software tools that include:

Selenium WebDriver which launches the browser and controls it automatically, Selenium RC which is a remote controller, and Selenium Grid which can be used to run tests simultaneously on different machines and different browsers with different versions, as it supports the execution of distributed tests. Selenium IDE is an add-on in the Firefox browser to conduct, capture and replay user interactions. Selenium has been introduced comprehensively in [8] where its various components that support different testing scenarios are explained.

2.3.3 xDebug

xDebug [19] is a PHP framework used as an extension for debugging PHP applications. It provides the capability to automatically record and report code coverage data. It is the most popular tool used to measure code coverage of web applications. This tool is independent of both platform and operating system. Other tools such as PHPUnit measure the code coverage for PHP applications that are not considered to be web-based. xDebug is believed to be the ideal tool for measuring and reporting code coverage for web applications. The code coverage is calculated automatically by adding a snippet of code that consists of two functions. The first function is provided in Listing 2.5 and needs to be located at the beginning of the source code or document to start calculating the code coverage data automatically. The second function shown in Listing 2.6 is placed at the end of the page to stop calculating the code coverage data. After the test suite finishes, the result of the coverage data is retrieved by providing the function in Listing 2.7.

Listing 2.5: xDebug code sample to start code coverage.

```
<?php
xdebug_start_code_coverage(XDEBUG_CC_UNUSED | XDEBUG_CC_DEAD_CODE);
?>
```

Listing 2.6: xDebug code sample to stop code coverage.

```
<?php
xdebug_stop_code_coverage();
?>
```

Listing 2.7: xDebug code sample to retrieve the code coverage data.

```
<?php
xdebug_get_code_coverage();
?>
```

The result of the code coverage reports the lines of code that have been executed or not by the test suite. It provides the line number and next to it the following numbers: [1] indicates that this line has been executed and [-1] means that the line has not been executed.

2.4 Data Analysis

2.4.1 User Data

User activities, including navigating the web pages, adding/deleting a book and providing input data in the web page fields when using the website were

logged in an Apache access log file as URL and (name-value) pairs. This has been achieved by adding a session/cookie snippet in the PHP source code to log user activities and identify each session by providing a unique session id for each user using the website. This procedure is essential to distinguish each user session and to log the URLs and (name-value) pairs used in the session-based testing techniques.

2.4.2 Test suite creation

To eliminate any kind of bias, the test cases have been created before looking at the faults that have been seeded in the website. These test cases are written in the PHP programming language via PHPUnit and executed using Selenium server. These tools are used to conduct the tests automatically. Human intervention has not been eliminated completely as it is essential in monitoring the tests and writing down the encountered faults during and after the testing process.

Chapter 3

Discussion

3.1 Testing techniques results

This section provides the outcomes achieved by each testing technique. These testing techniques represent Ricca & Tonella white-box (WB1 and WB2), user session (US1 and US2) and hybrid HYB (WB2+US1) approaches. Also, the effectiveness of each testing technique is discussed with respect to the following metrics: test suite coverage, code coverage and fault detection.

3.1.1 WB1 testing technique

By applying the guidelines described in Section 2.2.1 on how to create the required test cases, a test suite of size 47 was constructed for the WB1 technique. This test suite size was relatively small when compared to the WB2 testing technique and the other session-based testing techniques.

The result of running the test suite for the WB1 technique against the faulty web application discovered 28 faults from a total of 69 faults. The types of faults detected include: the invalid links to navigate from one page to the other forming an independent path (by looking at the UML model, links that include at least one new path through the graph) and illegal input values in some of the input fields. The illegal input values were some special characters such as ($&#% ...) in text or numeric input fields, numbers in text input fields and letters in numeric input fields. Moreover, buttons with faulty behaviour (buttons that redirect to a wrong page after clicking on them) were also detected.

On the other hand, the number of faults that were not detected totalled 41. These faults include: latency in loading the page (the loading of a page takes longer than expected after performing a task such as after clicking on the "Add to cart" button in the "Sports" page to add a book, the page takes too long to add the book in the cart and refresh the page), the generation of the invalid content that was retrieved from a table in the database to be displayed in the dynamically generated pages, and circular links. In addition, cascading faults that depend on performing a specific task first then conducting another task could not be discovered. This is because the first

defective task masked the later task. An example of this can be described as follows: testing the deletion of a book from the "History" category was not possible because the addition of a book task could not be performed, due to a fault in the add button where books could not be added to the cart. Furthermore, defective and broken links between pages were difficult to detect because links between pages are not tested as per the limitations of the WB1 testing technique. Also, each independent path is tested in isolation (in a separate test case), making it impossible to detect such faults. Finally, null values and boundary values for each input field were not discovered as the technique does not exercise such types of input values. By boundary values, we mean that each input field accepts a specific length of input values and any other length is considered a fault.

3.1.2 WB2 testing technique

As expected, the test suite size of WB2 was larger than WB1. The WB2 technique produced 98 test cases, the larger size being due to the adoption of the input selection approaches "boundary values" and "each condition/all conditions". The result of running the test suite against the faulty web application revealed a total of 61 faults. In addition to the faults that WB1 revealed, WB2 discovered some additional faults, including circular links between pages and faults related to database requests and data retrieval, as it considers textual differences, which helps in detecting these faults. Furthermore, the null values and the unacceptable lengths of values for each input field were also discovered via the "each condition/all conditions" and "boundary values" testing strategies.

Eight faults were not detected, representing cascading faults in the functionality of the links or buttons within a page, latency in loading of a page and navigational errors. The other undetected faults were related to unvalidated buttons, logic faults and input types.

White-box (WB1 and WB2) testing techniques summary

WB1 detected faults: This technique has effectively detected the faults related to the invalid links found in the independent paths. So, if only the independent paths of the web application need to be tested, then WB1 is an appropriate approach.

WB2 detected faults: The strength of this technique is in its ability to uncover all unacceptable lengths of values used as inputs in the input fields. Also, invalid values of input fields (unacceptable characters and null values) and wrongly generated pages (pages with content different than the original page) were also effectively discovered. This strength is due to having a clear methodical approach when testing the web application functionalities.

WB2 unique detected faults: Nine unique[1] faults were detected using the WB2 technique, divided between invalid input values and invalid boundary values.

[1]unique: a fault that has been detected by this technique only.

Faults that WB2 could not reveal: Our work suggests that the following categories of faults cannot be detected by the WB2 technique.

1. Cascading faults.
2. Some logic faults related to the session/cookie data that require the user to set the session then perform a task. The user will encounter an error if the session data is not set. Otherwise, they will not encounter any error and will be able to perform the required task. One example of this case can be seen in the "History" category when the user fails to add a book. However, if the user starts the session by first adding a book from the "Business" category and then trying to add another book from the "History" category in the same test case (session), then the user will be able to add a book from the "History" category because the session was already set.

   The WB2 technique cannot detect such faults because it tests independent paths and each test case represents an individual session where each category is tested independently.
3. Navigational errors in links when navigating between pages. If there are broken links or links that redirect to incorrect pages, then these faults cannot be detected due to the testing of independent paths only.
4. Input field type. If a hidden input field is changed to a different type then this cannot be detected because the technique will check whether the field is present or not (textual differences) without considering its type.
5. Unvalidated buttons that redirect to a page/link when a prerequisite step should be performed before proceeding such as allowing the redirection to the "Checkout" page in the case of an empty cart.

6. Latency faults.

Enhancements to the WB2 technique to reveal undetected faults
The following are suggested enhancements to the WB2 technique that may address the undetected faults described earlier.

1. Cascading faults. It seems to be difficult to discover such faults in general not because of a limitation in the testing technique or the automated tools but because of the nature of such faults. However, tools (such as Eclipse) that can trace faults in the order that they appear could be beneficial in revealing these types of faults.
2. Some logic faults related to the session/cookie data. These faults could be revealed if the test case can include two or more independent paths in the same test case. The test can be done twice with the path sequence being tested in different order. This can be performed by first opening the page of the "History" category then adding the desired book by clicking on the "Add to cart" button. Continuing in the same test case, proceed by opening the "Business" category page and adding any book by clicking on the "Add to cart" button. In the second test case, start with adding the desired book from the last selected category in the first test case which is the "Business" page. Then end the second test case by adding any book from the first selected category "History" in the first test case. This strategy uncovers the session data issue if found in one of the pages.
3. Navigational errors between pages. Testing the navigational links can be considered when preparing the test cases to be able to reveal such

faults. This suggestion can be applied by testing the links related to other pages that appear in one page. For example, all the links related to different pages that appear in the "History" page can be tested by traversing these links when conducting the "History" test case. Testing these links can reveal whether it is possible or not to reach the other pages from the "History" category page without any issues.

4. Input field type.

Listing 3.1: Testing variable type.

```
1 public function test_variable_type()
2 {
3 $driver-> get("http://localhost:8888/online_shopping/index.php"
    ↪ );
4 $currentvalue = $driver-> findElement(WebDriverBy::id('element'
    ↪ ))-> getAttribute('type');
5 $exp_value = "number"; \\or could be "text"
6 $this->assertEquals($exp_value, $currentvalue);
7 }
```

A methodology for testing the type of input fields should be added to the testing strategies adopted by the WB2 technique in order to reveal such faults. This methodology can be performed automatically by the automated testing tools (PHPUnit and Selenium) where functions that are specialised in checking the type of an input field are supported. The method "*findElement*" in Listing 3.1, line 4, locates a specific element in a web page by its id or name. After locating the desired element, the "*getAttribute*" method retrieves the type of a variable then compares the resulting variable type with the expected type. The method "*as-*

sertEquals" performs this comparison and asserts whether the expected type matches the type retrieved.

5. Unvalidated buttons. Including a button validation strategy can be helpful in revealing these faults. In our case, for example, this strategy should check if the cart is empty or not. If it is empty then the checkout button should not redirect to the "Checkout" page.
6. Latency faults. These types of faults can be revealed by testing different/identical functionalities provided in a page in the same test case rather than testing one function in each test case. This can be performed by selecting a number of desired books in the same category and adding them to the cart. This makes the automated test case pause for some time (might be seconds or more, depending on the latency). This pause between tasks related to the execution of one test case allows the tester to be aware of any issue when adding multiple books in one session.

The testing procedure is performed semi-automatically – the test itself is conducted automatically. However, uncovering a latency fault requires the presence of the tester to monitor the execution of the test case to check the latency between the performed tasks. This might be performed automatically by setting a specific time to wait and if the task was not completed by the time specified then there is a latency fault. This can be achieved by utilizing the functionalities provided in PHPUnit and Selenium.

The results showed that WB2 discovered all the faults that WB1 revealed. Therefore, it is recommended to use WB2 when test requirements consider textual differences, circular links and a variety of input values. Focusing on

using WB2 is both time and cost efficient. The reason for this conclusion is that the two techniques are similar in concept, the only difference is that WB2 provides more detailed, in-depth testing and considers a wide variety of input selections. However, if test requirements ignore textual differences, circular links and input fields that need to be tested by the "boundary values" and "each condition/all conditions" testing strategies, then WB1 is sufficient.

3.1.3 US1 testing technique

The US1 test suite size was 20, representing the total number of sessions conducted by the participants. Running the test suite resulted in 57 faults being detected from a total of 69.

The number of undetected faults was 12. Comparing the revealed faults to the list of faults, the majority of defect types that were revealed were related to issues with circular links, broken links when navigating between pages, invalid database requests (requests that return incorrect results) and invalid input values. The invalid inputs include special characters, letters in numeric fields, numbers in a field that requires only letters, null values and some invalid boundary values. In addition, US1 was able to reveal latency faults that WB1 and WB2 could not reveal. This is because US1 has no constraints on the users when testing the functionalities of the web pages in one test case (session). In other words, users can test more than one functionality (either the same or different) in the same session. Consequently, this reveals the time it takes to execute each task. However, in WB1 and WB2 if there are multiple occurrences of the same button that executes the

"Add to cart" function, then only one of these buttons is tested randomly. One example of such a situation is that some users added different books from the same category "Sports" during the same session, that uncovered a latency defect by the tester noticing how long it took to accomplish the task. The presence of the tester reporting the timing of each test case has aided in the process of uncovering this latency fault. It can be challenging to automate the whole process because the test cases are written manually and there is no available function or tool that evaluates the time spent by each task to be completed.

To automate the procedure of detecting a latency fault, a suggestion for tool development is discussed next. The tool should accept a web site URL to allow the tester to open, view and perform tasks in a web page. It should also include a stopwatch to measure the elapsed time spent by each task upon completion and a field that accepts a predefined completion time for each task provided by the tester. Finally, a function is required to compare the completion time of each task (using the stopwatch) with the time that was set by the tester. If the task completion time is less than or equal to the predefined time, then there is no latency fault. Otherwise, there is a latency fault. Moreover, additional faults were revealed by US1 but not by WB1 or WB2 such as a change to a variable type from hidden to integer. This fault was revealed due to the presence of the tester either by checking the Apache access log for the input fields attribute that appears with the URL or when replaying the sessions with the ability to view the web pages during the test. Human intervention was necessary when applying this technique as it is implemented in a semi-automated manner where the tester writes the code of the test suite to be executed by the automated tool and then monitors the

replayed sessions.

The faults that were not revealed represent a combination of invalid input values (special characters, letters, numbers and null values) that users did not provide as an input during their use of the web application. Furthermore, cascading faults and another faults that were related to invalid boundary values were also not detected.

3.1.4 US2 testing technique

The US2 test suite size was 20 and it revealed 54 faults that were similar to the types of fault that were detected by US1. These faults were related to some invalid values provided in input fields (special characters, letters and numbers), broken links, invalid circular links, latency in loading of a page and some invalid boundary and null values.

The number of undetected faults was 15. These were related to some invalid boundary values, invalid values and some faulty links. It is worth noting that the number and types of fault detected depend on the user behaviour when navigating the web pages and when providing input values.

To our surprise, the US2 technique revealed less faults than US1. Looking through this, it seems to be caused by the fact that all of the requests that follow request r_i in session U_a were discarded when mixing the requests of two sessions. As can be seen in Figure 3.1, the requests that follow r_3 in session U_1 were discarded when combined with the requests from session U_8 to

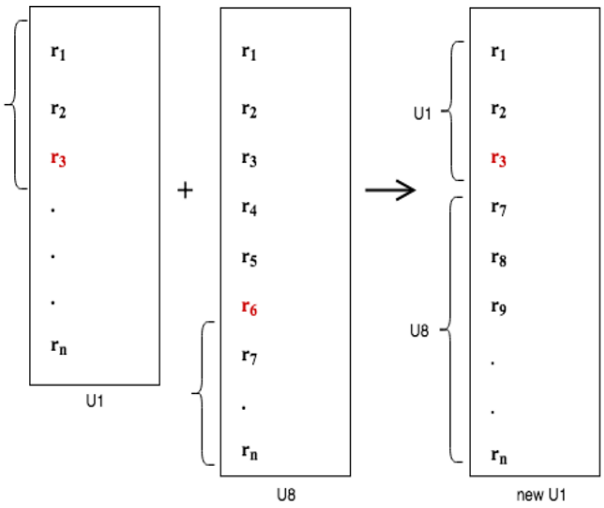


Figure 3.1: US2 mixing session requests.

make the new session U_1. The discarded requests could have revealed some of the missing faults that US2 could not discover.

Session-based (US1 and US2) testing techniques summary

US1 detected faults: The strength of this technique is in detecting issues related to invalid circular links, navigation links and latency faults.

US1 unique faults: US1 was able to discover six unique faults that the white-box techniques (WB1 and WB2) could not reveal. These faults are as follows:

1. Two broken links (redirecting from the "About us" to the "History" and

the "Cooking" pages).

2. Latency fault. This involved adding more than one book in the same session. This was not discovered by WB2 because it tested one valid case of adding a book and another invalid case of adding an invalid quantity of a book. Each test case was performed in isolation.
3. The dropdown menu (Javascript code) was not clickable to navigate from the "History" page to other pages.
4. Input field type. A type that relates to a hidden field was changed to be an integer when it should be hidden.
5. Logic fault.

Faults that US1 could not reveal: Based on our work, the following categories of faults cannot be detected by the US1 technique.

1. Some invalid boundary values and invalid input values (special characters, letters values in fields that are restricted to numeric/numeric values in fields that are restricted to letters).
2. Unvalidated buttons that redirect to a page/link when a prerequisite step should be performed before proceeding. An example is redirecting to the "Checkout" page in the case of an empty cart.
3. Cascading faults.

Enhancements to the US1 technique to reveal undetected faults Here, we provide some suggested improvements to the US1 technique to address the undetected faults.

1. Some invalid boundary values and invalid input values. These types of fault can be revealed by providing a general list of the invalid input types to consider when testing the various input fields. A document that illustrates the list of input fields and which invalid types to consider when testing a particular field can be helpful to guide users in the testing process. Suppose that there are two input fields to test in a web page. The first input field is *first name* (accepts letters only not less than three and not longer than 10), the second is *phone number* (accepts numbers only of length 10). The document should include the following invalid cases when testing these input fields. For evaluating the invalid cases of *first name*, the participant can examine any number, any special characters, and letters of length less than three and more than 10. For evaluating the invalid cases of *phone number*, consider any combination of letters, any special characters, and any numeric values less/more than 10.
2. Unvalidated buttons. Including a button validation strategy can be helpful in revealing such faults. In our case, for example, this strategy can check if the cart is empty or not. If it is empty then the "Checkout" button should not redirect to the "Checkout" page.
3. Cascading faults. With these types of fault, it seems to be difficult to discover in general not because of a limitation in the testing technique or the automated tools but because of the nature of such faults.

Insights about the US1 and US2 techniques are summarized in the following points:

1. The number of detected faults depends primarily on the input values provided by users during their use of the web application. So, if users did not exercise the input fields with invalid data that reveals faults, then the number of detected faults could be low. To overcome this issue, providing users with a clear and detailed description of the various types of input fields and the invalid values that need to be considered when testing those input fields would achieve a higher rate of fault detection.
2. Normal user behaviour which represents the input of valid data -that users insert in their normal operation- could be beneficial in revealing certain types of faults. Normal user operation should not be considered to be problematic. It could be in some cases when the types of faults require invalid data to reveal it -which is sometimes provided in abnormal users behaviour -. Suppose for example, there is an input field that accepts numbers only. A user acts normally and adds any number in the input field, he then receives an error message that informs him that the input is invalid when it is not. This normal behaviour revealed a fault by using valid data. Therefore, these types of faults could never be revealed by invalid data "abnormal" users behaviour.
3. It was also noticed that some requests were not tested in session U_a in the US2 sessions generation procedure. This is due to all of the requests that were ignored and as a result not being added to the test case. These requests are located after request r_i in session U_1 as illustrated previously in Figure 3.1. This could be an issue if the discarded requests had values that were not available in any other test case that could

reveal specific types of defects. One solution to this can be achieved by creating two new sessions then test each separately. First, combine the requests located above request r_i with all of the requests that are located below request r_j. Second, combine the requests located below request r_i with all of the requests that are located below request r_j. This might help in discovering all of the faults that US2 missed in the originally proposed procedure.

Clearly, the suggested solution will result in an increase in the test suite size, and the corresponding cost and effort when conducting the expanded methodology.

4. US2 requires a lot of effort and it is time consuming to generate new sessions from the sessions pool. On top of that, it revealed less faults than US1 and the types of discovered faults were not different than what US1 revealed.
5. US2 could be beneficial in capturing conflicts in users request. For example, having one available book to purchase while two users are simultaneously trying to add it to their carts. However, this could not be measured as the web application was not developed to capture such a situation.
6. The procedure of selecting a random request r_i in session U_a can be done automatically with the help of a tool rather than manually. The tool should accept a file that has all of the requests related to one session then select a random line. This process could eliminate any bias when selecting the random request and adds more formalism to it.

The procedure of creating the test cases in US2 has been shown by Elbaum et al. in [1] and [26]. In [1], there was no clear indication on what to do with

the last session U_b that remained unused after applying the US2 procedure. In the results, they showed that the test suite was one session smaller when compared to the US1 test suite size. In [26], however, the procedure of creating US2 test cases stated that if there is a session with request r_i selected and no viable request r_j, then the session is directly used as in US1. So, this makes the test suite size of US2 match the number of directly used sessions in US1.

Applying the two procedures (replaying the unused session and not using it at all) showed that, with regards to fault detection, adding the unused session had no effect in terms of finding additional faults. The reason for this is that the session itself (the session that was left) had three different selected r_j requests which were added to several previous sessions when performing the US2 procedure. Selecting a similar request r_j that matches request r_i indicates that all of the requests that follow the selected request r_j are combined with the request r_i. This demonstrates that all of the requests in this remaining session were used and all of the faults that were there were reported by the previously generated test cases (sessions).

US2 seems to be a powerful technique in revealing the conflict between different requests made by different users. That is the reason behind combining requests to examine the behaviour of the application in such a situation. An example of this behaviour is if there are two requests from two different users trying to purchase the last copy of a book at the same time. The US2 procedure reveals how the web application behaves in such a situation and whether it reacts as expected. However, this situation was not evaluated in the designed web application. It is one of the features to be considered for

future work. The US2 procedure could be used if there was a need to measure the conflicting requests from multiple users simultaneously, taking into account the suggested enhancement. However, as mentioned earlier, this will introduce higher cost and overhead to the system if applied. If there are no such situations to evaluate, then the US1 technique is sufficient.

3.1.5 HYB (WB2 + US1) technique

The test suite size for the HYB technique was 99 test cases. The total number of faults detected was 51 faults. Combining the two techniques showed that the total number of revealed faults was less than when compared to each of the techniques individually.

The types of faults that were uncovered were a mixture of some invalid values, broken links, latency in loading of a page, buttons with faulty behaviour and logic faults. The number of undetected faults was 18. The types of the undiscovered faults were the same as the revealed faults. The reason behind this is because either the URL request for the web page was not examined (because it was not recorded in the Apache access log as a request from users) or users did not provide such input to uncover such faults.
The HYB technique utilizes the translated URLs of the test requirements in the WB2 technique. In other words, it uses the URLs that are provided by the WB2 and US1 techniques. Figure 3.2 consists of two URLs: first, a test requirement transformed to a URL and its empty (name-value) attributes in the WB2 technique. The second is, a matching URL from the Apache access log related to a user session in US1.

- **Test case requirement for WB2:**

online_shopping/shopSc.php?id= &quantity= &name= &price= &add=

- **URL and (name-value) pair constructed from Apache access log:**

GET /online_shopping/shopSc.php?id=25 &quantity=1 &name=The+Sea & price=23 &add=Add+to+cart

Figure 3.2: URL extracted from the WB2 technique and matched with equivalent URL from US1 sessions.

These URLs illustrate a test case of the "Science" category page when purchasing a book. The second URL that was extracted from the Apache access log represented the purchasing of a book titled "The Sea" with an id equal to 25 and a quantity of 1. If this user session URL did not exist, then the test requirement URL constructed by the WB2 technique will be discarded. This occurs because the attributes of the first URL that represent the WB2 test requirement will have no values to be filled by the user session URL. As a result, this test requirement of WB2 will not be tested and faults related to this page will not be recorded.

Hybrid testing technique summary It was noticed that this technique did not provide a clear delineation among the types of detected and undetected faults. Moreover, using the US1 and WB2 techniques separately is recommended as each technique has a unique strategy in revealing specific types of faults. Using them in the HYB approach seems to remove the unique properties of these individual techniques.

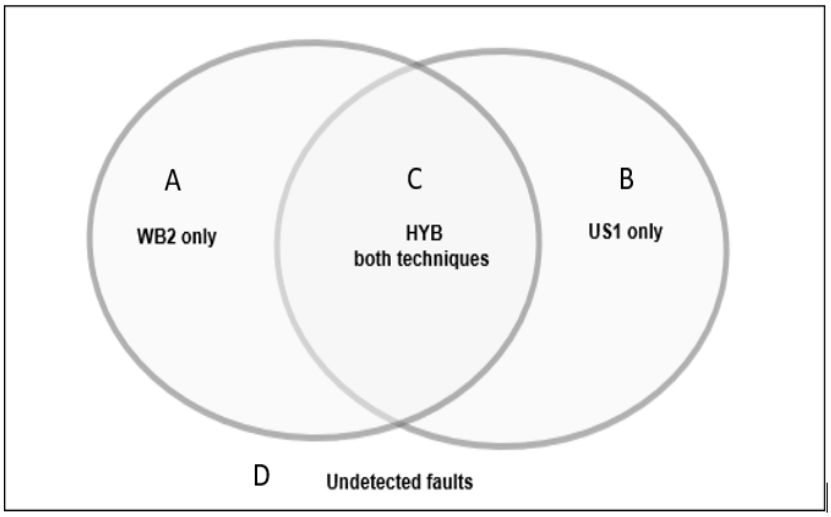


Figure 3.3: Fault detection capability of the HYB technique when compared to the US1 and WB2 techniques.

Figure 3.3 shows a detailed comparison of the fault detection capabilities of the HYB technique when compared to the US1 and WB2 techniques.

- *Region A* represents the faults detected by the WB2 technique only. The HYB technique failed to detect any of the faults that were only revealed by the WB2 technique. This is because these faults were related to input fields values, whereas HYB relies on US1 for values to fill the attributes in WB2 test requirements. In addition, not all of the test requirements (URLs) of WB2 could be covered because the user sessions exercised some of the test requirements.
- *Region B* represents the faults detected by the US1 technique only. The HYB technique failed to reveal all of the faults that were detected by US1 only. The majority of these faults were related to circular links, navigation links and a change in the type of an input field. These

faults were not part of the WB2 test requirements that HYB uses in its procedure.

- *Region C* represents the faults uncovered by both techniques (WB2 and US1).

  The HYB technique was able to reveal some of the faults uncovered by both techniques. The faults that were not revealed include some URL links for faulty pages that were not recorded in the Apache access log by US1 sessions. So, these URLs could not be matched with an equivalent WB2 test requirement.

- *Region D* represents the undetected faults by both techniques.

  The HYB technique failed to detect the faults that both techniques could not reveal. As mentioned earlier, HYB relies on the test requirements of WB2 and the matched URL requests of US1 logged in the Apache access log. If either of the two techniques could not detect these types of faults, then HYB could also not detect it.

3.2 Testing techniques metrics

The effectiveness of each testing technique can be reported through the following metrics: test suite coverage, PHP code coverage and fault detection. For each technique, the result of performing the test suite is summarized in Tables 3.1, 3.2 and 3.3, respectively. Table 3.1 reports the coverage percentage of the total number of executed lines corresponding to each test suite. Table 3.2 shows the total number of PHP lines executed by each technique and Table 3.3 provides the number of faults detected by each technique. More

details regarding these metrics are provided in the following sections.

3.2.1 Test suite coverage metric

Table 3.1 lists each testing technique, the test suite size (number of test cases), the lines of code corresponding to each test suite and the coverage percentage obtained for each test suite. It can be observed that the WB1 technique had the least code coverage of 84.76% when compared to the other testing techniques. The WB2 technique performed slightly better with a coverage of 89.55%.

Table 3.1: A comparison between the testing techniques in terms of test suite code coverage.

Testing technique	Test suite size	Lines of code	Test suite coverage (line coverage%)
WB1	47	715	84.76%
WB2	98	1646	89.55%
US1	20	1209	98.76%
US2	20	1173	99.32%
HYB (WB2 + US1)	99	1255	99.04%

On the other hand, the user session-based techniques (US1 and US2) achieved higher test suite coverages of 98.76% and 99.32%, respectively. The HYB technique achieved a coverage of 99.04% which is higher than WB1, WB2 and US1 but less than US2.

3.2.2 Code coverage metric

Table 3.2 provides the PHP code coverage of each technique. This coverage has been achieved by executing the test suite related to each testing technique on the web application. The code coverage for the techniques is similar, with WB1 achieving the highest coverage of 68.73%.

Table 3.2: A comparison between the testing techniques in terms of PHP code coverage.

Testing technique	PHP code coverage (PHP line %)
WB1	68.73%
WB2	55.56%
US1	57.88%
US2	55.81%
HYB (WB2 + US1)	57.36%

One of the differences in the metrics when comparing the test suite coverage with the code coverage is that the test suite metric was measured automatically by PHPUnit. On the other hand, the PHP code coverage was

measured semi-automatically by xDebug. Upon the completion of each test suite, the tester had to extract the results of the coverage data and report the number of executed lines manually, then calculate the code coverage result. The coverage percentage, however, does not guarantee better detection of faults. It provides an indicator of how many and which lines of code have been executed by each technique. The test suite code coverage provides an overview of the number of lines related to each test suite that were executed when running the test suite itself. In other words, it shows which lines of the test cases have been executed. The code coverage, however, shows the number of lines related to the web application pages that were executed when running the test suite.

3.2.3 Fault detection metric

Table 3.3 lists each testing technique, its test suite size, number of faults detected by each technique and number of unique faults that each technique was able to reveal. The WB2 technique outperformed the other techniques in revealing 61 faults out of 69.

On the other hand, the WB1 technique detected the lowest number of faults. The WB2 and US1 techniques were able to reveal some unique faults that the other techniques could not reveal. A detailed list of these unique faults was discussed in the previous sections.

Table 3.3: A comparison between the testing techniques in terms of fault detection.

Testing technique	Test suite size	# of detected faults	# of unique detected faults
WB1	47	28	-
WB2	98	61	9
US1	20	57	6
US2	20	54	-
HYB (WB2 + US1)	99	51	-

3.3 Scalability

In this section an overview regarding the testing techniques usage in large scale web applications is discussed.

Ricca & Tonella testing techniques Evaluating Ricca & Tonella testing techniques revealed that the WB2 technique has achieved high testing coverage and fault detection rates. This is because it performs a detailed and in-depth testing approach. Although the testing technique is beneficial in detecting faults, it introduces a large test suite size if applied on a large scale web application. The test suite size increases dramatically when the web application size increases. Also, the cost of generating various input values by a tester to fill the input fields will increase when the test suite size increases.

User session testing techniques The US1 technique achieved a higher fault detection rate when compared to the US2 technique. The US1 technique can be used on a large scale web application because the size of its test suite will depend on the number of user sessions performed. One great advantage that the US1 technique presents is the generation of input values. These values will be provided by the users of the web application while performing the testing procedure. This eliminates the need for a tester to provide such input data used to exercise the input fields. Although this is an advantage, it is important to know that detecting faults will depend on the quality of input values provided by users and types of faults found in the application.

3.4 Threats to validity

One of the limitations of this study is related to the process of seeding faults in the web application under consideration. This procedure was essential in the evaluation of the testing techniques. Natural faults could have been a better approach rather than seeded faults. However, finding a web application with many functionalities written in PHP with naturally occurring faults was not a viable possibility.

Another limitation is related to the session-based testing techniques (US1 and US2). User involvement is critical in the evaluation of such techniques. During the execution of the sessions, users were provided with some guidance on how to navigate the pages of the web application and what to include in the input fields (valid or invalid data) in an effort to achieve as realistic user sessions as possible. However, there are clearly potential issues with this

testing environment.

Moreover, a limitation related to the generated UML in the white-box techniques is presented. In the absence of the ReWeb and TestWeb tools that were provided by Ricca & Tonella in [2] to automatically generate the UML model, the UML model used in this study was manually constructed. The threat was reduced by carefully following the guidelines on how to generate the UML model in [3].

Chapter 4

Future Work

We would like to examine the effectiveness of using the Selenium IDE tool with the US1 technique when conducting the tests. Improvements could be expected if users are able to perform the sessions using the Selenium IDE tool. After the user completes the session, the tester will export the session in the form of PHPUnit code to be executed later. This will eliminate the need for manually writing the code.

The US2 technique was suggested because it could be useful in revealing the faults caused by conflict requests made by different users simultaneously. This situation will be considered for future work when implementing the web application.

Moreover, a new approach to merge the requests that belong to different sessions will be evaluated when implementing the US2 technique. This

approach might consider adding the neglected requests that followed the request r_i which was observed earlier in the proposed US2 procedure.

Chapter 5

Conclusion

Effective testing of web applications requires the performance of systematic approaches that are properly conducted to help in the process of detecting faults. Evaluating the testing techniques in our work revealed that the WB2 and US1 techniques appear to be the most effective compared to other techniques in terms of fault detection.

The WB2 technique outperformed the WB1, US1, US2 and HYB approaches in terms of fault detection. Although the WB2 technique was able to reveal the highest number of faults, it has introduced a larger test suite size compared to the WB1, US1 and US2 techniques. Also, WB2 requires the manual generation of input values to fill the input fields to be able to conduct the tests.

As for the US1 technique, it has provided a smaller test suite size than WB1, WB2 and HYB with a fault detection rate less than the rate achieved by WB2 but higher than the WB1, US2 and HYB techniques. The values used in exercising the input fields were obtained from the user sessions. These user sessions can provide useful testing values if users are guided on which (valid and invalid) input types to consider when testing the web application.

It was observed that with US1, it could be more appropriate to use the Selenium IDE tool to avoid manually writing the test suite as PHPUnit and Selenium code.

US1 and WB2 techniques have both proved to operate better when used separately than being used in a hybrid form. Each technique has its unique strategy in revealing different types of faults. However, using these techniques in the HYB approach seems to remove the unique properties of each.

Using the automated tools PHPUnit and Selenium has helped in automating the conduct of the tests and reduced the time and effort required.

Bibliography

[1] Elbaum, S., Karre, S., & Rothermel, G. (2003, May). Improving web application testing with user session data. In Proceedings of the 25th International Conference on Software Engineering (pp. 49-59). IEEE Computer Society.

[2] Ricca, F., & Tonella, P. (2001, July). Analysis and testing of web applications. In Proceedings of the 23rd international conference on Software engineering (pp. 25-34). IEEE Computer Society.

[3] Ricca, F., & Tonella, P. (2001). Understanding and restructuring Web sites with ReWeb. IEEE MultiMedia, 8(2), 40-51.

[4] Stevens, J. (2017, August 18). Internet Statistics & Facts (Including Mobile) for 2017. Retrieved December 11, 2017, from https://hostingfacts.com/internet-facts-stats-2016/.

[5] Programming Language Usage. (2017). [online] Available at: https://trends.builtwith.com/framework/programming-language [Accessed 11 December 2017].

[6] Benedikt, M., Freire, J., & Godefroid, P. (2002). VeriWeb: Automatically testing dynamic web sites. In In Proceedings of 11th International World Wide Web Conference (WWW2002).

[7] Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., & Ernst, M. D. (2010). Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Transactions on Software Engineering, 36(4), 474-494.

[8] Bruns, A., Kornstadt, A., & Wichmann, D. (2009). Web application tests with selenium. IEEE software, 26(5).

[9] Girgis, M. R., Mahmoud, T. M., Abdullatif, B. A., & Zaki, A. M. (2014). An Automated Web Application Testing System. International Journal of Computer Applications, 99(7), 37-44.

[10] Selenium Browser Automation. [online] Available at: http://www.seleniumhq.org/ [Accessed 11 December 2017].

[11] Mendes, E., & Mosley, N. (2006). Web engineering. Berlin: Springer.

[12] Welcome to PHPUnit!. [online] Available at: https://phpunit.de/ [Accessed 11 December 2017].

[13] de Jesus, F. R., de Vasconcelos, L. G., & Baldochi, L. A. (2015, April). Leveraging task-based data to support functional testing of web applications. In Proceedings of the 30th Annual ACM Symposium on Applied Computing (pp. 783-790). ACM.

[14] Vasconcelos, L. G., & Baldochi Jr, L. A. (2012). Usatasker: A task definition tool for supporting the usability evaluation of web applications. In Proc. of the IADIS Internat. Conf. WWW/Internet (pp. 307-314).

[15] Sampath, S., Greenwald, A. S. & Pollock, L. (2004). Towards Defining and Exploiting Similarities in Web Application Use Cases through User Session Analysis.

[16] Milani Fard, A., Mirzaaghaei, M., & Mesbah, A. (2014, September). Leveraging existing tests in automated test generation for web applications. In Proceedings of the 29th ACM/IEEE international conference on Automated software engineering (pp. 67-78). ACM.

[17] IBM. (2016, January 01). Rational Functional Tester. [online] Available at: http://www-03.ibm.com/software/products/en/functional [Accessed 11 December 2017].

[18] PHP. [online] Available at: https://www.w3schools.com/php/ [Accessed 16 December 2017].

[19] xDebug.org. (2017). xDebug - Debugger and Profiler Tool for PHP. [online] Available at: https://xdebug.org/ [Accessed 18 December 2017].

[20] Di Lucca, G. A., Fasolino, A. R., Faralli, F., & De Carlini, U. (2002). Testing web applications. In Software Maintenance, 2002. Proceedings. International Conference on (pp. 310-319). IEEE.

[21] Sampath, S., Sprenkle, S., Gibson, E., Pollock, L., & Greenwald, A. S. (2007). Applying concept analysis to user-session-based testing of web applications. IEEE Transactions on Software Engineering, 33(10).

[22] Liu, C. H., Kung, D. C., & Hsia, P. (2000). Object-based data flow testing of web applications. In Quality Software, 2000. Proceedings. First Asia-Pacific Conference on (pp. 7-16). IEEE.

[23] Periyasamy, J. Software Testing Tutorials - Manual and Automation Questions Answers. [online] Available at: http://jobsandnewstoday.blogspot.ca/2013/07/boundary-value-analysis-with-examples.html/ [Accessed 25 December 2017].

[24] CodeCeption. [online] Available at:http://codeception.com/quickstart [Accessed 30 December 2017].

[25] SimpleTest. [online] Available at:http://www.simpletest.org/ [Accessed 30 December 2017].

[26] Elbaum, S., Rothermel, G., Karre, S., & Ii, M. F. (2005). Leveraging user-session data to support Web application testing. IEEE Transactions on Software Engineering, 31(3), 187-202. doi:10.1109/tse.2005.36.

[27] Beizer, B. (1990). Software testing techniques (2nd ed.), Van Nostrand Reinhold Co., New York, NY.

Appendices

Appendix A

List of seeded faults

In Table A we provide an explicit list of the types of faults that were seeded in the web application used in the study.

Table A.1: Faults seeded in the web application.

Page name	**Fault description**
The "History" page	1. The processing of the "History" page was done in the "Children" page (changing the value of the form action to be performed in the "Children" page instead of the "History" page). 2. The type of the variable *price* was changed to a different type which allowed the manipulation of its value. 3. The Java script dropdown menu was disabled to make links unclickable to prevent navigation between pages.

	4. The logic for validating the acceptable values of the variable *quantity* disallowed the acceptance of any value if the session/cookie was not set. That resulted in an error message that appeared each time a user tried to add a book.
The "Children" page	The form action value was changed to redirect to a page that did not exist instead of the "ShopC.php" page that processed the (add/delete) books request.
The "Business" page	The delete functionality was disabled to prevent the deletion of books from the cart.
The "Science" page	The form action value was changed to redirect to the "Children" page if the user session was set and to the "History" page otherwise.
The "Cooking" page	The value of the *price* variable was not displayed.
The "Sport" page	1. The wrong category of books was shown due to a fault in the retrieved table from the database. 2. Refresh interval took too long to process requests which resulted in a latency fault. 3. The *quantity* variable accepted more than the allowed value. 4. The variables *name* and *price* of books were not displayed.
The "Check future books" page	1. The "Business" and "Home & Garden" options in the dropdown menu displayed invalid results.

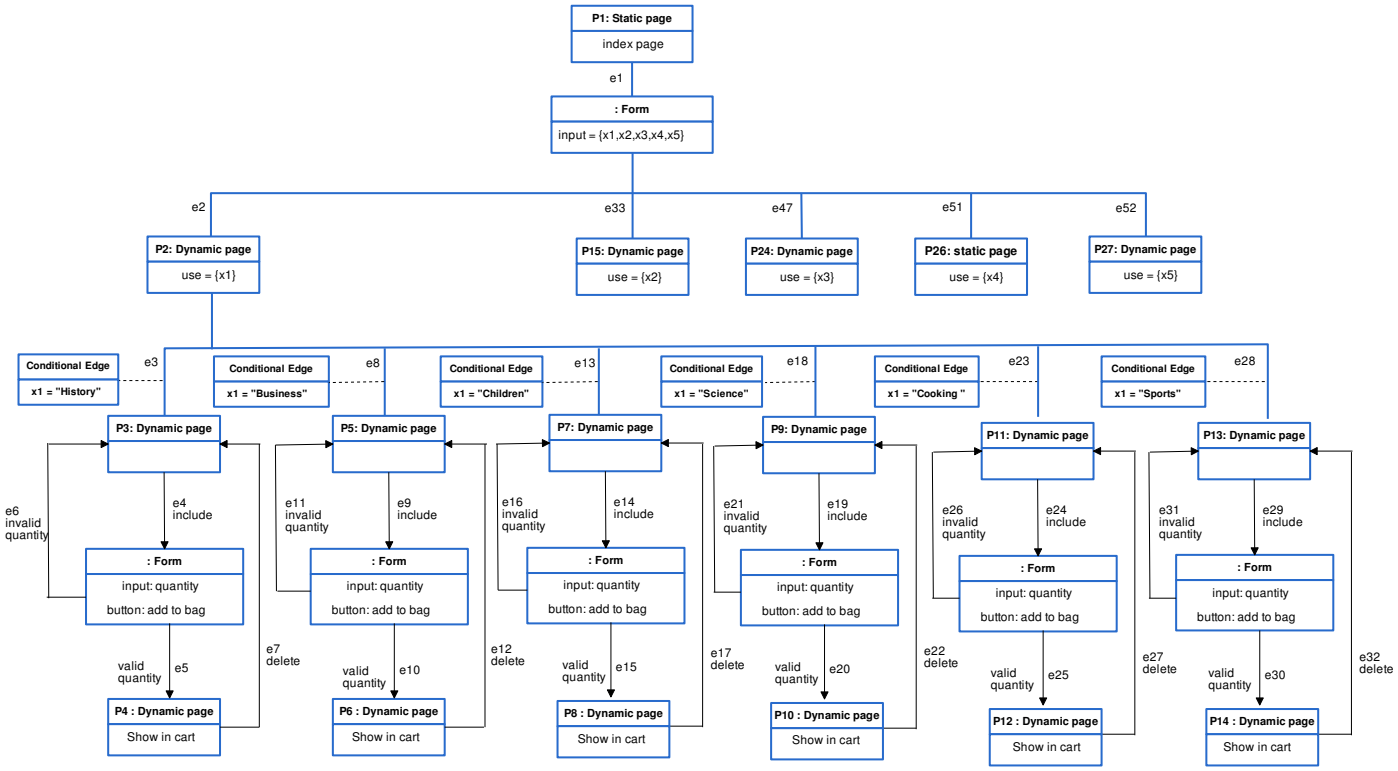
	2. Changing the array name for the variables *quantity* and *ISBN* resulted in the invalid retrieval of results from the database. 3. The "Back" button was not redirecting the user to the previously visited page rather it was constructed to go back two pages from the previously visited page.
The "Contact form" page	1. The variable *Phone* accepted values that may contain letters, special characters, null and have invalid length. 2. The variable *Subject* accepted values that contained numbers and null values. 3. The variable *User name* accepted values that may contain numbers, special characters and null values. 4. An invalid port was provided for sending emails.
The "My cart" page	The button redirected to the "Checkout" page if the cart was empty before performing any validation.
The "Checkout" page	1. The *First name* and *Last name* variables accepted numbers, special characters and invalid (min, max) length of values. 2. The *Address* variable accepted special characters and invalid (min, max) length of values. 3. The *City* variable accepted numbers and special characters. 4. The *Phone* variable accepted letters, special characters, null and invalid (min, max) length of values. 5. The *Card number* variable accepted letters, special characters and invalid (min, max) length of values.

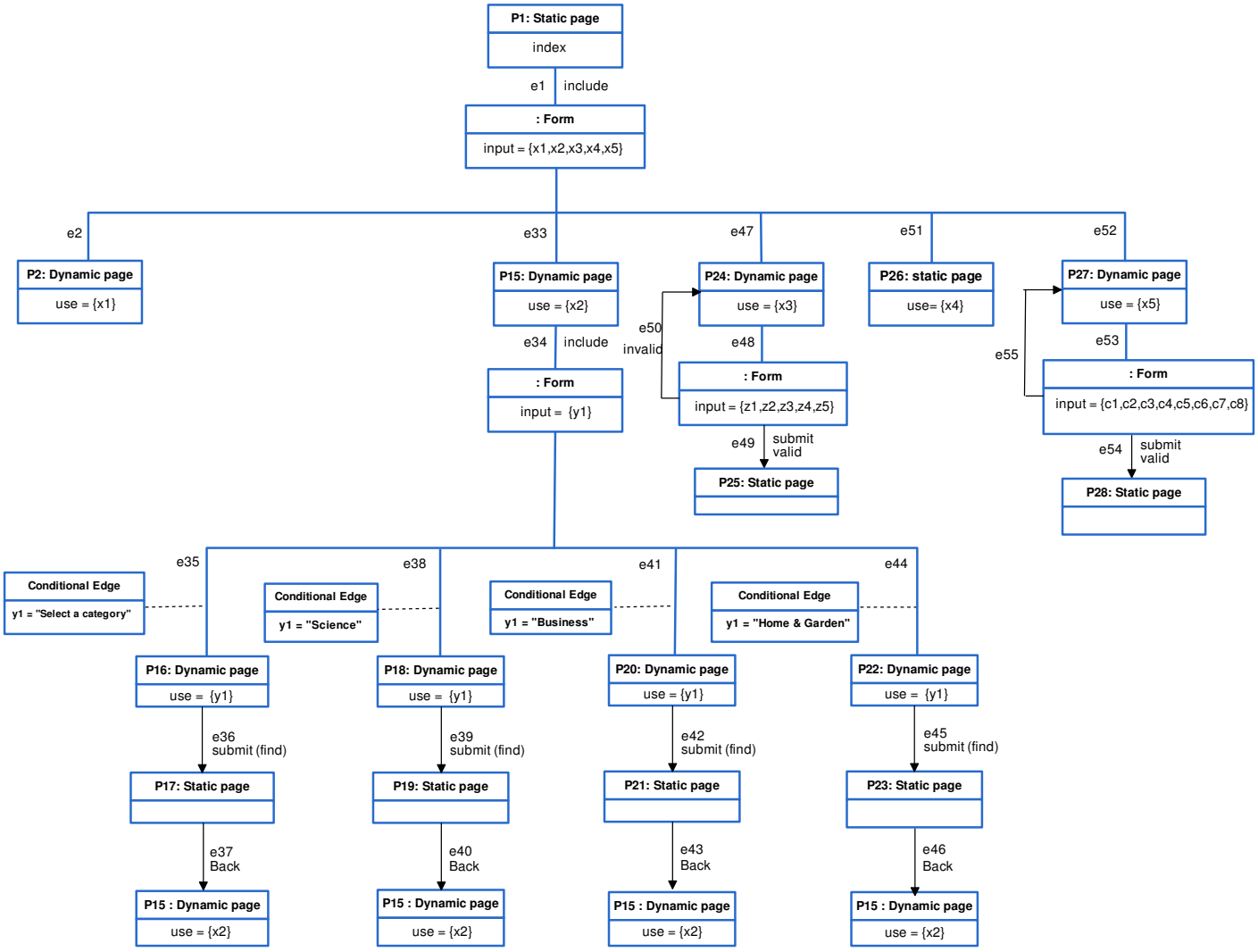
	6. The *CVC* variable accepted letters, special characters, null and invalid (min, max) length of values. 7. The *Expiration date* variable accepted letters, special characters, null and invalid (min, max) length of values.
The "About us" page	The links for the "Cooking" and the "History" pages redirected to a page that did not exist.

Appendix B

Web application UML model

In this appendix, we provide the complete UML model constructed for the web application used in this study.


P1: Static page
index page
e1
: Form
input = {x1,x2,x3,x4,x5}
e2
e33
e47
e51
e52
P2: Dynamic page
use = {x1}
P15: Dynamic page
use = {x2}
P24: Dynamic page
use = {x3}
P26: static page
use = {x4}
P27: Dynamic page
use = {x5}
Conditional Edge
x1 = "History"
e3
Conditional Edge
x1 = "Business"
e8
Conditional Edge
x1 = "Children"
e13
Conditional Edge
x1 = "Science"
e18
Conditional Edge
x1 = "Cooking "
e23
Conditional Edge
x1 = "Sports"
e28
P3: Dynamic page
P5: Dynamic page
P7: Dynamic page
P9: Dynamic page
P11: Dynamic page
P13: Dynamic page
e6
invalid
quantity
e4
include
: Form
input: quantity
button: add to bag
valid
quantity
e5
P4 : Dynamic page
Show in cart
e7
delete
e11
invalid
quantity
e9
include
e10
P6 : Dynamic page
e12
delete
e16
invalid
quantity
e14
include
e15
P8 : Dynamic page
e21
invalid
quantity
e19
include
e20
P10 : Dynamic page
e17
delete
e26
invalid
quantity
e24
include
e25
P12 : Dynamic page
e22
delete
e31
invalid
quantity
e29
include
e30
P14 : Dynamic page
e27
delete
e32
delete


P1: Static page
index
e1
include
: Form
input = {x1,x2,x3,x4,x5}
e2
e33
e47
e51
e52
P2: Dynamic page
use = {x1}
P15: Dynamic page
use = {x2}
P24: Dynamic page
use = {x3}
P26: static page
use= {x4}
P27: Dynamic page
use = {x5}
e50
invalid
e34
include
e48
e53
e55
: Form
input = {y1}
: Form
input = {z1,z2,z3,z4,z5}
: Form
input = {c1,c2,c3,c4,c5,c6,c7,c8}
e49
submit
valid
e54
submit
valid
P25: Static page
P28: Static page
e35
e38
e41
e44
Conditional Edge
y1 = "Select a category"
Conditional Edge
y1 = "Science"
Conditional Edge
y1 = "Business"
Conditional Edge
y1 = "Home & Garden"
P16: Dynamic page
use = {y1}
P18: Dynamic page
use = {y1}
P20: Dynamic page
use = {y1}
P22: Dynamic page
use = {y1}
e36
submit (find)
e39
submit (find)
e42
submit (find)
e45
submit (find)
P17: Static page
P19: Static page
P21: Static page
P23: Static page
e37
Back
e40
Back
e43
Back
e46
Back
P15 : Dynamic page
use = {x2}
P15 : Dynamic page
use = {x2}
P15 : Dynamic page
use = {x2}
P15 : Dynamic page
use = {x2}

Appendix C

User session source code

Source code written in PHPUnit and Selenium representing a sample of a user session is provided here.

Listing C.1: A sample of a user session.

```
1  public function test_user()
2  { $driver->get("http://localhost:8888/online_shopping/Faulty/index.
   ↪ php");
3  $driver->findElement(WebDriverBy::XPath("//button[contains(.,'Shop
   ↪ available books')]"))->click();
4  $driver->findElement(WebDriverBy::XPath("//a[@href='Cooking.php']"))
   ↪ ->click();
5  $driver->findElement(WebDriverBy::XPath("(//input[@name='add'])[4]")
   ↪ )->click();
6  $driver->findElement(WebDriverBy::XPath("(//input[@name='add'])[3]")
   ↪ )->click();
7  $driver->findElement(WebDriverBy::XPath("(//input[@name='add'])[5]")
   ↪ )->click();
8  $driver->findElement(WebDriverBy::XPath("//a[@href='MyCart.php']"))
   ↪ ->click();
9  $driver->findElement(WebDriverBy::XPath("//a[@href='Book_Search.php
   ↪ ']"))->click();
10 $driver->findElement(WebDriverBy::XPath("//button[contains(.,'Shop
   ↪ available books')]"))->click();
11 $driver->findElement(WebDriverBy::XPath("//a[@href='Science.php']"))
   ↪ ->click();
12 $driver->findElement(WebDriverBy::XPath("(//input[@name='add'])[4]")
   ↪ )->click();
13 $driver->findElement(WebDriverBy::XPath("//button[contains(.,'Shop
   ↪ available books')]"))->click();
14 $driver->findElement(WebDriverBy::XPath("//a[@href='Science.php']"))
   ↪ ->click();
15 $driver->findElement(WebDriverBy::XPath("(//input[@name='add'])[2]")
```

```
    ↪ )->click();
16  $driver->findElement(WebDriverBy::XPath("//a[@href='Book_Search.php
    ↪ ']"))->click();
17  $driver->findElement(WebDriverBy::XPath("//button[contains(.,'Shop
    ↪ available books')]"))->click();
18  $driver->findElement(WebDriverBy::XPath("//a[@href='Science.php']"))
    ↪ ->click();
19  $driver->findElement(WebDriverBy::XPath("(//input[@name='add'])[5]")
    ↪ )->click();
20  $driver->findElement(WebDriverBy::cssSelector('span.text-danger'))->
    ↪ click();
21  $driver->switchTo()->alert()->dismiss();
22  $driver->findElement(WebDriverBy::XPath("//button[contains(.,'Shop
    ↪ available books')]"))->click();
23  $driver->findElement(WebDriverBy::XPath("//a[@href='Business.php']")
    ↪ )->click();
24  $driver->findElement(WebDriverBy::XPath("(//input[@name='add'])[5]")
    ↪ )->click();
25  $driver->findElement(WebDriverBy::XPath("//a[@href='MyCart.php']"))
    ↪ ->click();
26  $driver->findElement(WebDriverBy::name('checkout'))->click();
27  $driver->findElement(WebDriverBy::XPath("(//input[@name='fname'])"))
    ↪ ->sendKeys('Fern');
28  $driver->findElement(WebDriverBy::XPath("(//input[@name='lname'])"))
    ↪ ->sendKeys('Marti');
29  $driver->findElement(WebDriverBy::XPath("(//input[@name='address'])"
    ↪ ))->sendKeys('111 main street');
30  $driver->findElement(WebDriverBy::XPath("(//input[@name='city'])"))
```

```
    ↪ ->sendKeys('Hamilton');
31  $driver->findElement(WebDriverBy::XPath("(//input[@name='ph1'])"))->
    ↪ sendKeys(811);
32  $driver->findElement(WebDriverBy::XPath("(//input[@name='ph2'])"))->
    ↪ sendKeys(4717511);
33  $driver->findElement(WebDriverBy::XPath("(//input[@name='cardnum'])"
    ↪ ))->sendKeys('4234561254368799D');
34  $driver->findElement(WebDriverBy::XPath("(//input[@name='CVC'])"))->
    ↪ sendKeys(567);
35  $driver->findElement(WebDriverBy::XPath("(//input[@name='Exp1'])"))
    ↪ ->sendKeys(88);
36  $driver->findElement(WebDriverBy::XPath("(//input[@name='Exp2'])"))
    ↪ ->sendKeys(22);
37  $driver->findElement(WebDriverBy::name('checkout'))->click(); }
```