An Exploration of User-Visible Errors in Web-based Applications to Improve Web-based Applications

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

Kinga Dobolyi

May 2010

© Copyright May 2010

Kinga Dobolyi

All rights reserved

Approvals

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science

Kinga Dobolyi

Approved:

Westley R. Weimer (Advisor)	William A. Wulf
Mary Lou Soffa (Chair)	Chad S. Dodson (Psychology)
John C. Knight	

Accepted by the School of Engineering and Applied Science:

James H. Aylor (Dean)

May 2010

Abstract

Web-based applications are one of the most widely used types of software and have become the backbone of the e-commerce and communications businesses. These applications are often mission-critical for many organizations, but generally suffer from low customer loyalty and approval. Although such concerns would normally motivate the need for highly-reliable and well-tested systems, web-based applications are subject to constraints in their development lifecycles that often preclude complete testing.

To address these constraints, this research explores user-visible web-based application errors in the context of web-based application error detection and classification. The main thesis of this work is that *user-visible web-based application errors have special properties that can be exploited to improve the current state of web application error detection, testing, and development*. This thesis is evaluated using seven specific falsifiable hypotheses. This research presents highly-precise, automated approaches to the testing of web-based applications that reduce the cost of such testing, making its adoption more feasible for developers. Additionally, a model of user-visible web application error severity is constructed, backed by a human study, to refute the current underlying assumption of error severity uniformity in defect seeding for this domain, as well as to propose software engineering guidelines to avoid high severity errors, and facilitate testing techniques in finding high-severity defects.

Studying error severities from the consumer perspective is a novel contribution to the web application testing field. This research approaches testing web-based applications by recognizing that errors in web applications can be successfully modeled using the tree-structured nature of XML/HTML output, that unrelated web applications fail in similar ways, and that these failures can be modeled according to their consumer-perceived severities, with the ultimate goal of improving the current state of web application testing and development.

The strategies presented in this dissertation have the potential to (1) increase the perceived return-on-investment for testing web-based applications, thereby improving their reliability, and (2)

decrease consumer loss due to errors and their perceived severity.

Acknowledgments

This has been a long journey, with many challenging stretches, and I would like to take this opportunity to thank all those involved in this voyage, starting from the final destination and working back to the point of departure.

I would like to begin by thanking my advisor, Wes Weimer, for his support over the past four years. In 2006, I was an incoming student with no research experience and a relatively weak computer science background, but I believe he saw potential in me despite these apparent shortcomings. There were countless lunches spent reviewing questions for the qualifying exams, as I had no background in theory, operating systems, or compilers. Wes remained patient while we essentially took off a year to catch up on this relevant background. Although these exams have been a great source of debate in our department, I'm thankful I was responsible for going through the process because I believe it changed me for the better; I learned there was nothing I could not master.

I'd also like to thank Wes for trusting me enough to allow me to explore my own research ideas the entire time we've worked together. It is incredibly rewarding to look back and have a significant body of work in which I am emotionally invested. While web applications may not have been an area Wes would have explored on his own, I'd like to thank him again for seeing potential in me and in this line of research. I hope that he's able to enjoy its success as much as I am.

In addition, Wes has been a great friend. Between research group movie nights, dinners, camping trips, and other activities, he made these four years here a pleasure for me. He was always available for emotional support, and I think he has a talent for choosing entertaining, intelligent, and supportive students — I'd like to thank Claire Le Goues, Pieter Hooimeijer, and Ray Buse for their friendship (as well as their comments). Although Krasi Kapitanova and Patrick Graydon were not technically part of our research group, I'd also like to thank them for their kindness and charm.

I am greatly indebted to John Knight for his inspiration, advice, and mentorship these past years. Furthermore, I'd like to thank the other members of my committee, Mary Lou Soffa, Bill Wulf, and

Chad Dodson for their support and insights.

Michelle Hugue (University of Maryland) has always been a shoulder to lean on and the source of encouragement these past years — thank you for showing me that I can accomplish whatever I set out to do.

Finally, I'd like to thank all my family and friends for standing by me — especially my brother David, my mother Hanna, and Linda Stevens. Most of all, I would like to thank my father, Zsolt, for his complete love and support. Without you, I would have never realized my dream, nor had the courage to go after it. Thank you for keeping me calm and reminding me always "amilyen olyan".

Contents

1 Introduction **1**

1.1 Challenges for Testing Web-based Applications 5

2 Background **11**

2.1 Web Applications 11

2.2 Software Testing 14

2.3 Summary 19

3 Testing Web-based Applications **20**

3.1 Defining Errors in Web-based Applications 20

3.2 Existing Approaches 22

3.3 Graphical User Interface Testing 29

3.4 Improving the Current State of the Art 30

4 Research Outline **31**

4.1 Improving Error Detection During Regression Testing 32

4.2 Focusing on Severe Errors to Improve Error Detection 34

4.3 Summary 36

5 Improving Error Detection During Regression Testing **38**

5.1 Challenges in Regression Testing Web-based Applications 38

5.2 Reducing the Cost of Regression Testing Web-based Applications 42

5.3 Validating the Assumptions of the Model 47

5.4 Selecting a Model for Differences in Web-based Applications 49

5.5 Evaluating the Oracle Comparator 54

5.6 Threats to Validity 59

5.7 Experimental Summary 60

5.8 Related Work to Error Detection in Web-based Applications 60
5.9 Summary 62

6 Automating Error Detection During Regression Testing 64
6.1 Challenges in Automatic Regression Testing of Web-based Applications 65
6.2 Fully Automating Regression Testing of Web-based Applications 65
6.3 Evaluating Automated Oracle Comparators 66
6.4 Training Data from Defect Seeding 73
6.5 Summary of Experiments 74
6.6 Threats to Validity 76
6.7 Related Work to Automated Error Detection in Web Applications 76
6.8 Summary 77

7 Modeling Consumer-Perceived Web Application Error Severities for Testing 78
7.1 Challenges with Consumer Retention 79
7.2 Strategies For Modeling Consumer Perceived Severity 80
7.3 Consumer-Perceived Error Severity Study 82
7.4 Modeling Consumer-Perceived Severities of Web Errors 89
7.5 Automatically Predicting Error Severity 95
7.6 Summary of Experiments 99
7.7 Related Work to Studying Web Errors and Severity 101
7.8 Summary 102

8 Addressing High Severity Errors in Web Application Testing 104
8.1 Strategies For Addressing High Severity Errors in Web Testing 105
8.2 Fault Severity Distribution by Application 106
8.3 Fault Features Related to Severities 107
8.4 Fault Causes Related to Severities 111
8.5 Test Case Reduction And Severity 114

8.6 Threats to Validity 118
8.7 Related Work to Web Failures, Severity, and Test Suites 119
8.8 Summary 120

9 Combining Error Detection During Regression Testing with Error Severity 122
9.1 Motivation 122
9.2 Feature Analysis 124
9.3 Severity of Missed Errors using SMART 128
9.4 Summary 131

10 Conclusions and Future Work 133
10.1 Improving Error Detection During Regression Testing 134
10.2 Automating Error Detection During Regression Testing 135
10.3 Modeling Consumer-Perceived Web Application Error Severities for Testing . . . 135
10.4 Addressing High Severity Errors in Web Application Testing 136
10.5 Combining Error Detection and Error Severity 137
10.6 Conclusions and Future Implications 137

A 139
A.1 Web Application Fault Severity Study 139
A.2 Web Application Fault Severity Survey 144
A.3 Dominant Technologies Used in Current Web Applications 146
A.4 Decision Tree For Consumer-perceived Fault Severity Prediction Model 151
A.5 Definition of Web-based Applications 154

Bibliography 155

List of Figures

1.1 A correct, expected rending of an example webpage. 2

1.2 An incorrect, unexpected rending of the example webpage. Note in particular that the formatting of the webpage is missing, making navigation difficult and likely confusing or upsetting users. 3

1.3 Three-tiered web application 7

1.4 Server-side dynamic content generation 8

2.1 An oracle-comparator 16

3.1 Web fault taxonomy by Marchetto *et al.*, reprinted from [69]. 21

5.1 An example alignment between two HTML trees 43

5.2 The benchmarks used in the experiments 48

5.3 The average values of features for test cases 49

5.4 The F_1-score, precision, and recall values for SMART on the entire dataset 51

5.5 Analysis of variance of the oracle comparator model 52

5.6 F_1-score, precision, and recall when trained and tested on individual projects, as well as all ten benchmarks 54

5.7 Simulated performance of SMART on 20232 test cases from multiple releases of two projects 56

6.1 The benchmarks used as **test data** for Experiment 1 67

6.2 F_1-score on each test benchmark using the Model, and other baseline comparators . 69

6.3 Recall on each test benchmark using the Model, and other baseline comparators . . 70

6.4 Precision on each test benchmark using the Model, and other baseline comparators 71

6.5 F_1-score for GCC-XML using the model with different numbers of test case output pairs from original-mutant versions of the source code 75

7.1 Severity scale for web application faults 83
7.2 Real-world web applications mined for faults 84
7.3 2-way analysis of variance of the response variable of fault severity as judged by human subjects 86
7.4 Average severity ratings from a total of 12,600 human judgments of 900 scenarios . 87
7.5 Kolmogorov-Smirnov test results for the dataset 89
7.6 Comparison of real-world and injected faults per application 90
7.7 Severity distribution of faults according to developer responses 91
7.8 Boolean surface features associated with web application faults 92
7.9 Average Spearman's Ranking Correlation Coefficient (SRCC) between each model and the average human over 100 held-out faults 93
7.10 Analysis of variance showing relative feature predictive power for finding high-severity faults 95
7.11 Cumulative severity over time when prioritizing faults 97
7.12 Performance of the automated model against the human-annotation-based model and other baselines 98
7.13 Analysis of variance showing feature predictive power for the automated model with no human annotation 99

8.1 The breakdown of severe faults for each benchmark application 106
8.2 Spearman's Ranking Correlation Coefficient (SRCC) between an application feature and the severity of its faults 107
8.3 Context-*independent* fault features. 108
8.4 Severity distribution as a function of context-*independent* fault characteristics. . . . 110
8.5 Common defect causes from the four hundred real-world faults in the human study 112
8.6 Fault severity as a function of underlying defect causes. 113
8.7 Fault severity uncovered via reduced test suites 117

9.1 Additional web application benchmarks. 123

9.2 Coefficients of significant ($p < 0.05$) feature values across all test benchmarks, plus the generic training dataset. 125
9.3 Severity of missed faults across the 3 PHP benchmarks. 130

A.1 The “current” page 141
A.2 The “next” page 142
A.3 The severity rating used in the human study 145
A.4 The survey used in the developer study 147

Chapter 1 Introduction

In the United States, 73% of the population used the Internet in 2008 [9], which contributed to the over $204 billion dollars in Internet retail sales in the same year [7]. While the global average for Internet usage was only 24% of the population by comparison [9], online business-to-business e-commerce[1] transactions total several trillions of dollars annually [5]. Therefore, there is a powerful economic incentive to produce and maintain high quality web-based applications.[2]

Although other types of software, such as operating systems, are also widely used and highly distributed, web-based applications face additional challenges in ensuring acceptability and maintaining a consumer base. Customer loyalty towards any particular website is notoriously low, and is primarily determined by the usability of the application [76]; unlike customers purchasing software such as Microsoft Windows, web consumers can easily switch providers without buying another product or installing another application. This challenge of customer allegiance is compounded by high availability and quality requirements: for example, one hour of downtime at Amazon.com has been estimated to cost the company $1.5 million dollars [81]. User-visible failures are endemic to top-performing web applications: several surveys have reported that about 70% of such sites are subject to user-visible failures, a majority of which could have been prevented through earlier detection [98]. For example, Figure 1.1 shows the correct representation of a webpage, while Figure 1.2 shows the same page with an example of a user-visible error: the website is missing some formatting components, making navigation difficult. One solution to maintaining a faithful consumer base and avoiding preventable monetary losses is to design web-based applications to meet high reliability, usability, security, and availability requirements [76], which translates into well-designed and well-tested software that is as free from error as possible.

Delivering high quality web-based applications has additional, domain-specific challenges be-

[1] The definition of business-to-business e-commerce includes all transactions of goods and services for which the order-taking process is completed via the Internet.

[2] see Section 2.1 for a formal definition of web applications and Section A.5 (in the Appendix) for a formal definition of web-based applications.

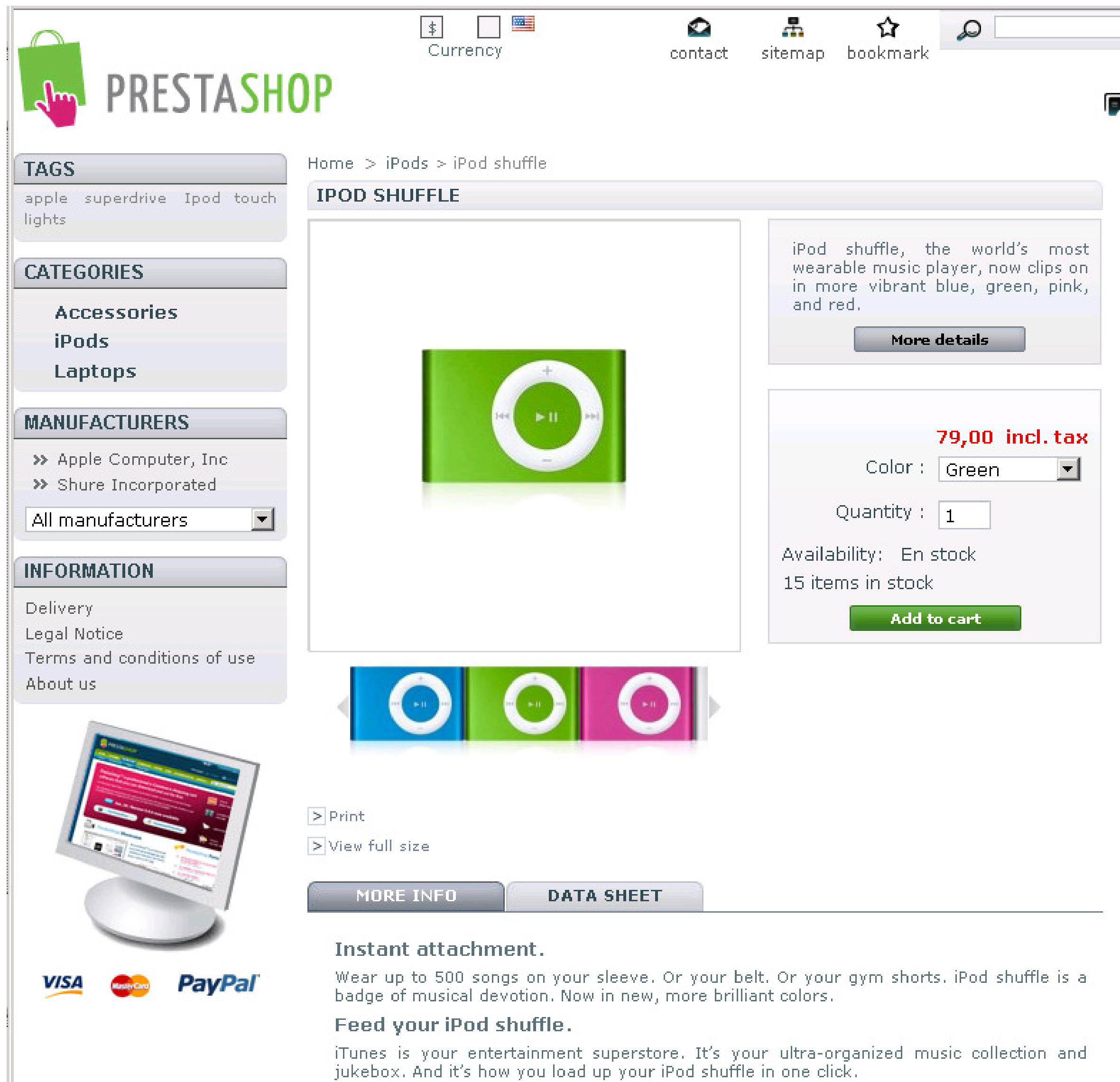

Figure 1.1: A correct, expected rending of an example webpage.

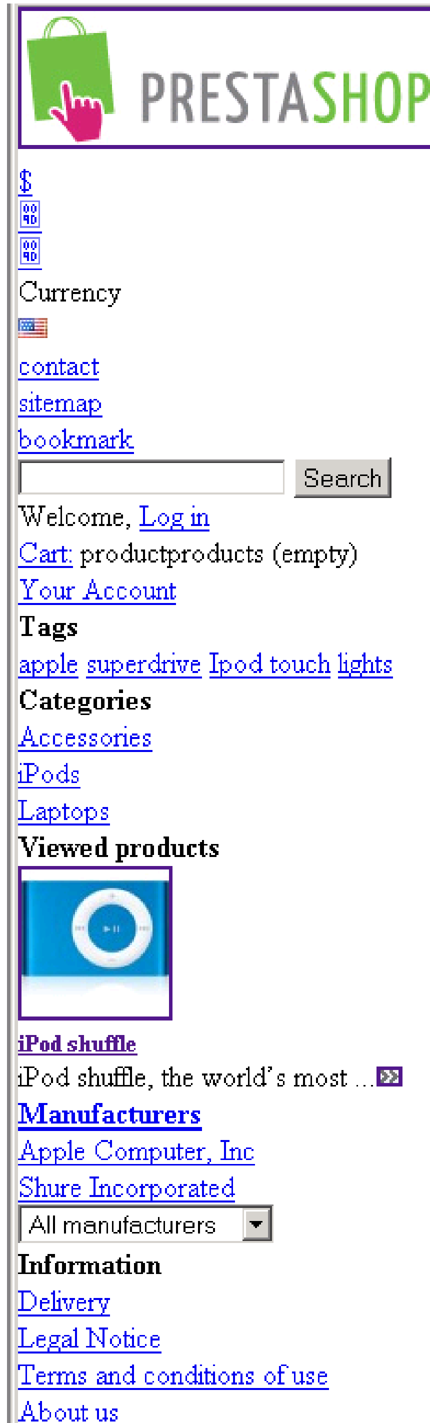

Figure 1.2: An incorrect, unexpected rending of the example webpage. Note in particular that the formatting of the webpage is missing, making navigation difficult and likely confusing or upsetting users.

yond those of consumer retention. Most web applications are developed without a formal process model [82]. Despite having strong quality requirements that would normally dictate the need for testing and stability, web applications have short delivery times, high developer turnover rates, and quickly evolving user needs that translate into an enormous pressure to change [86]. Web application developers often deliver the system without testing it [86]. The construction of web-based applications is subject to strenuous economic constraints revolving around time-to-market concerns in the face of rapid change.

Web-based applications are not fundamentally different from other software in terms of technologies used, however they deserve further attention due to three main characteristics:

- Web applications have become an integral part of the global economy, with Internet-based e-commerce projected to reach over one trillion dollars by 2010 [100], and therefore they are subject to unique and powerful economic considerations,
- Web-based applications provide a variety of services, but are commonly built as three-tiered architectures that output browser-readable code (see Figure 1.3), and consequently unrelated web-based applications often fail in similar ways, and
- Web-based applications are human-centric, implying not only a "consumer" use-case, but also defining the perceived acceptability of results through the eyes of the user.

While the economic urgency of delivering high-quality web-based applications is compounded by the lack of investment in formal processes and testing for this type of software, two insights offer hope of targeting development and testing strategies towards producing high-quality applications. First, as Figure 1.3 illustrates, although web applications are frequently complex, with opaque, loosely-coupled components, are composed in multiple programming languages, and maintain persistent session requirements, as this research demonstrates, they tend to fail in similar and predictable ways. This similarity is due to the fact that web-based applications render output in XML/HTML, where lower-level faults manifest themselves as user-visible output [81, 104]. Although web applications are often complicated amalgamations of various heterogeneous components, the

requirement that they produce HTML output corrals failures, even those from lower levels of the system, to the browser. This insight allows developers to centralize their testing strategies to this top level of the application.

Second, web applications are meant to be viewed by humans. While this implies that faults in the system will manifest themselves at the user level and drive away consumers, this human-centric quality of web applications can actually be viewed as an advantage. The acceptability of output becomes dependent on whether or not users were able to complete their tasks satisfactorily — a definition that encompasses a natural amount of leeway. Rather than viewing verification in absolute terms, developers that are subject to the extreme resource constraints web-based projects often entail may focus on reducing the number of high severity faults that will drive away consumers, as opposed to giving the same priority to all faults, regardless of their severity.

This research focuses on exploring user-visible errors in web-based applications to improve web-based application error detection, as well as studying the consumer-perceived severity of errors to guide web-based application design and testing. Before discussing various approaches towards more effective fault detection in Chapter 2, and in this domain in particular in Chapter 3, a general background of web-based application testing challenges is presented below.

1.1 Challenges for Testing Web-based Applications

Testing is a major component of any software engineering process meant to produce high quality applications. Maintenance activities consume 70% [31] to 90% [97] of the total life cycle cost of software, summing to over $70 billion per year in the United States [109], with regression testing accounting for as much as half of this cost [55, 90]. Despite the drive to retain consumers, testing of web-based applications is limited in current industrial practice due to a number of challenges:

- **Rate of Change.** The usage profile for any particular web-based application can quickly change, potentially undermining test suites written with certain use cases in mind [39]. Similarly, websites undergo maintenance faster than other applications [39]. Unlike other types of software, web-based applications are frequently patched in real-time in response to consumer

suggestions or complaints. Regression testing of web-based applications must be flexible enough to handle such small, incremental changes.

- **Resource Constraints.** Testing of web applications is often perceived as lacking a significant payoff [52]. This mindset is a consequence of short delivery times, the pressure to change, developer turnover, and evolving user needs [86, 119]. Given this human misconception of the value of testing, every effort to reduce the burden of testing for applications with such resource constraints must be made: applying automation to web testing methodologies increases their viability.

- **Dynamic Content Generation.** Unlike traditional client-server systems, client side functionality and content may be generated dynamically in web applications [119]. Figure 1.4 shows an example setup with server-side scripting, where the page source is tailored to individual users. User input, in the form of parameters passed in the URL, can change the control flow of the application — for example, pressing the back button may have unexpected consequences [61]. In a dynamically generated environment, the content of a page may be customized according to data in a persistent store, the server state, or session variables. Dynamic content may also be generated on the client side through actions within a webpage, such as mouse clicks. Validating dynamically-generated webpages is challenging because it often requires testing every possible execution path, and static analyses often have difficulty capturing the behavior of code generated on-the-fly by dynamic languages [26]. Hidden interfaces that are not exposed through static or dynamically generated pages cannot be tested by traditional testing techniques [48].

- **Persistent State and Concurrency.** Web-based applications frequently persist user information, and can be concurrently accessed by global clientele around the clock [36, 106]. Testing concurrent applications is known to be difficult [111]. A successful testing approach must account for the concurrent and continuous nature of user accesses as well as successfully model user interactions that update persistent state.

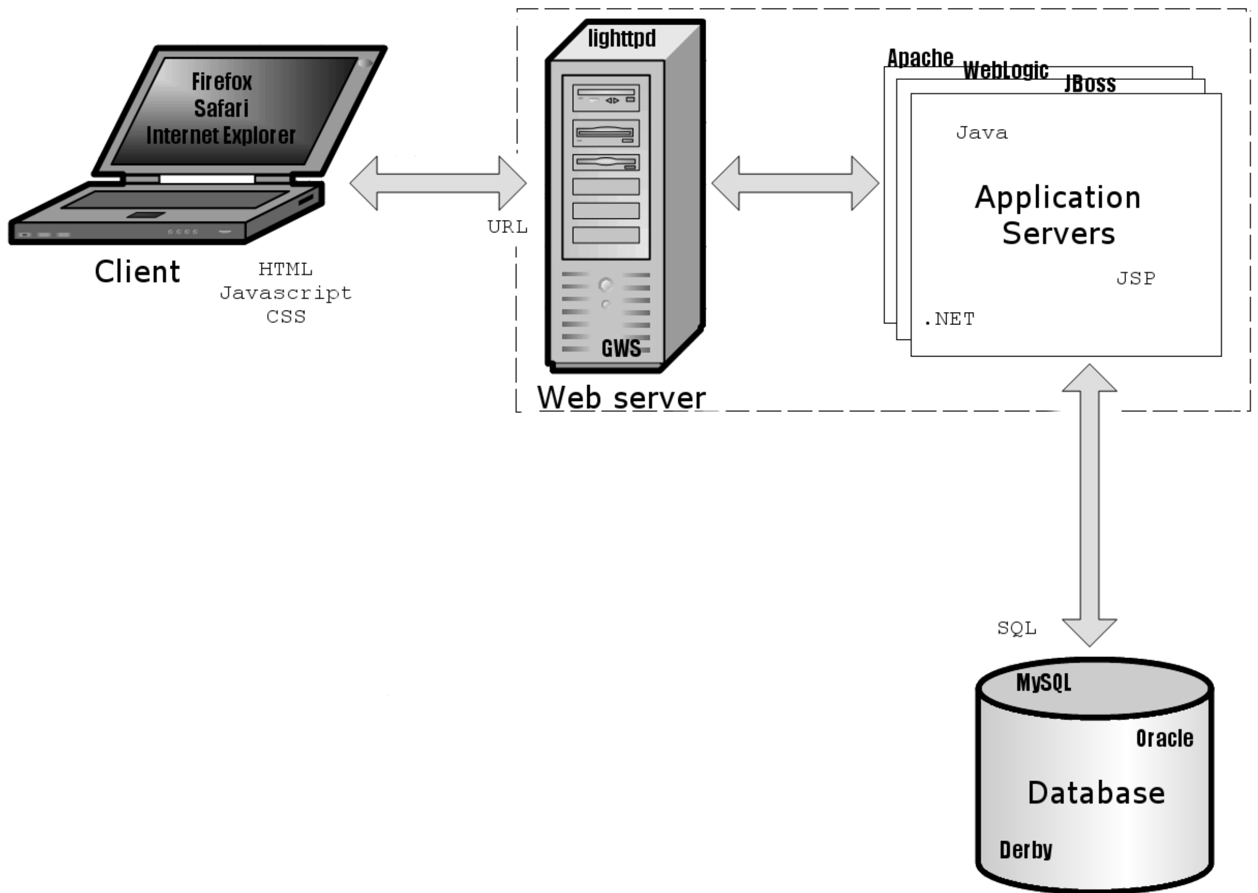

Figure 1.3: Three-tiered web application

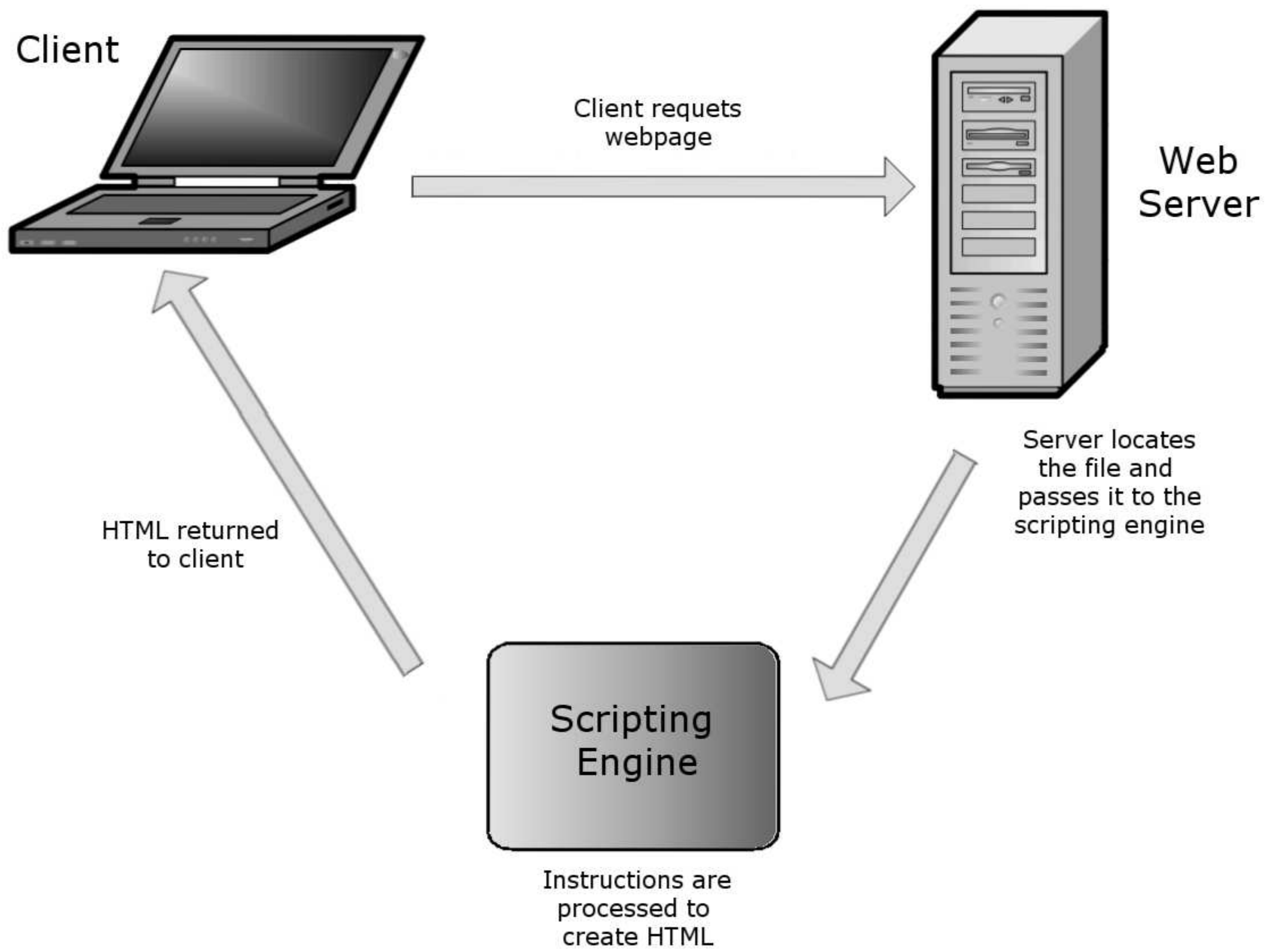

Figure 1.4: Server-side dynamic content generation. Adapted from `http://blog.search3w.com/dynamic-to-static/hello-world/`

Web-based applications are subject to unique challenges because of their development circumstances and the complexities of dynamic content generation. Although web applications have especially high reliability, usability, security, and availability requirements [76], time-to-market, other resource constraints, and the pressure-to-change often prevent these system from being tested [86]. In order for testing of web-based applications to be widely and successfully adopted, testing methodologies must be flexible, automatic, and able to handle their dynamic nature.

This research explores user-visible errors in web-based applications in the context of web-based application error detection. In doing so, the goal is to develop new techniques to reduce the cost of testing web-based applications as well as provide recommendations to make current testing techniques more cost-effective. The overall thesis of this work is that:

> *Web-based applications have special properties that can be harnessed to build tools and models that improve the current state of web application user-visible error detection, testing, and development.* (Thesis)

This main thesis is evaluated in terms of seven specific falsifiable hypotheses (H1) through (H7) that are detailed and experimentally tested in subsequent chapters.

This dissertation approaches the problem of error detection in web-based applications by recognizing that errors in web-based applications can be successfully modeled due to the tree-structured nature of XML/HTML output, and that unrelated web-based applications fail in similar ways. Additionally, by analyzing errors in web applications, this work defines a model of *consumer*-perceived severity. This model targets error detection and classification methodologies and evaluation techniques toward flagging and preventing high-severity errors to retain users in the face of low customer loyalty.

This work focuses on errors in web-based applications, in the context of web-based application testing. Although the term *testing* refers to both test case generation and test case execution, the research in this document will focus only on the latter, under the assumption that a test suite has already been generated. Because Chapter 5 and Chapter 6 also assume multiple versions of the same software, with the possibility for benign program evolutions, those chapters additionally focus on

regression testing in particular; the remainder of the dissertation speaks to testing in general.

The main contributions of this research are:

1. Reducing the cost of regression testing through improved error detection by capitalizing on the special structure of web-based application output to precisely identify user-visible errors.

2. Automating error detection during web-based application regression testing by relying on the discovery that unrelated web-based applications tend to fail in similar ways.

3. Formally grounding the current state of industrial practice by refuting fault injection as a standard for measuring web application test suite quality. This research assesses whether or not the assumption that all injected faults have the same non-trivial severity, and thus, the same benefit to developers, holds.

4. Understanding consumer-perceived severities of user-visible web application errors to build a model of consumer-perceived error severity.

5. Understanding how to avoid high-severity errors during web application design and development.

6. Reducing the cost of testing web applications by exposing high-severity errors through test case design, selection, and prioritization (test suite reduction).

The remainder of this dissertation is structured in the following manner. The next chapter introduces web-based applications and software testing in general, while Chapter 3 provides a background for the state-of-the-art in web application testing. Chapter 4 introduces and summarizes the contributions of this research. Chapter 5 discusses how to improve error detection during regression testing, and Chapter 6 expands upon the previous chapter by investigating automated ways to approach the same goal. Chapter 7 explores a model of consumer-perceived error severity to be used in testing, which Chapter 8 extends into concrete ways to address high severity errors in web application design and testing. Chapter 9 simultaneously explores automated error detection and error severity in the context of popular, real-world web-based applications. Chapter 10 summarizes the research presented in this document.

Chapter 2 Background

This chapter provides an overview of web applications (Section 2.1) as well as general software testing as related to this work (Section 2.2). Readers already familiar with the web domain and generic testing may proceed to Chapter 3 for a more detailed survey of testing as it relates of web-based applications in particular.

2.1 Web Applications

A *web application* is a type of software that is hosted on a server and can be accessed remotely through an Internet browser, usually by a human. The server is responsible for receiving requests from users, processing these requests, and then returning information to the user. Interaction with a web application is usually a three step process: 1) the user makes a request to the server through the browser, 2) the server processes the user request, and 3) the browser renders the response. These steps are detailed below.

2.1.1 The user makes a request to the server through the browser

In this first step, a user specifies the server as well as the information being requested by typing in a *URL* (Universal Resource Locator) into a browser such as Mozilla Firefox or Microsoft Internet Explorer. The browser is responsible for passing the request to the server through Hypertext Transfer Protocol (HTTP). A *request*, such as `http://www.mysite.com/register.php?id=John&age=32`, specifies the server name (`www.mysite.com`), the location of the requested information on the server (i.e., a file name — `register.php`), and other information such as the type of request (GET or POST). The browser may also collect information from the user through a web form, in the format of name-value pairs which it also passes to the server in the HTTP request. For example, the request above passes a variable named `id` with value *John* and a variable named `age` with value *32* to the server.

2.1.2 The server processes the user request

A new *user session* is started the first time a new user makes a request of a server. The server is responsible for managing the *session data*, such as identification information, associated with multiple and possibly simultaneous users. In the simplest interaction, the server locates a static request such as a file or image, to be sent back as a response to the user. These *static pages* are simply copied verbatim from an underlying filesystem or data store and do not depend on any information from the user or session state.

A user may also request a *dynamic page*, which is customized based on user provided data, session state, or other variables. During such a dynamic page request, the server is responsible for executing code that adjusts the output of the request based on whatever information was provided. For example, a user interacting with an online bookstore may request to see their purchase history over the past year. In this case the server will receive a request from the user which contains an account number or other information. The server can then use this account number to retrieve all purchases by that customer from a database, and generate a customized output page based on the user's purchase history. The response generated by the server is usually in the form of browser-renderable output in the Hyper Text Markup Language (HTML). The server then parcels the static or dynamic response into an HTTP response which it sends back to the browser.

Server-side application code may be written in various languages such as Java or PHP, using technologies such as JavaServer Pages (JSP) or Application Server Pages (ASP), and servers frequently have accesses to databases or other remote components. Sometimes the server encounters an error during the processing of the user request. For example, the server may be unable to find the requested resource, and respond with a "Page Not Found" or 404 error. Alternatively, the server-side application code may encounter an unexpected state, such as an empty object, and raise a `NullPointerException`. Components that the server interacts with, such as databases, may also be unable to provide desired information and can lead to additional errors. This work primarily focuses on studying such exceptions and error cases. This dissertation uses the terms error, fault, failure, defect, and bug interchangeably: Section 3.1 provides a formal definition of a failure in a web-based application. Although the source of the error may be in the web server level (responsible

for finding files), the application server level (responsible for generating dynamic content), or in external components (such as a database), the techniques discussed in this dissertation are applicable to all three locales as such errors generally manifest in user-visible output. Formally, this work examines user-visible errors in web-based applications. Other errors, such as an email not being sent by the server, are beyond the scope of this research.

2.1.3 The browser renders the response

Browsers are equipped to interpret HTML responses and render them visible to users. Various HTML tags and features can specify the format of text, multimedia, and functional components, although each browser may display equivalent HTML pages differently. Consequently, browser compatibility is a common concern when HTML does not render the same across different applications.

Browsers primarily render HTML code, although such responses frequently contain other information. Beyond images and multimedia objects, HTML files often embed client-side scripting components such as Javascript. *Client-side scripting* is used for simple tasks such as validating form inputs to save time, but cannot be used in all cases. For example, a user entering an email and password can be validated on the client side by checking whether either of these fields are non-empty — a requirement that is generic to all users, without having to pass a request to the server to do so. Authenticating the user, that is, ensuring the email and password match the expected values, however, cannot be accomplished on the client side because it is impossible for the browser to know what the correct email-password pair is without consulting the server (among other issues). A recent development in the web application domain is that of Asynchronous Javascript (AJAX), which is a client-side scripting language that allows for the retrieval of information from the server without reloading the web page. Testing the client-side functionality of web applications, as well as browser comparability concerns, are active research areas that are only explored in the consumer-perceived severities of potential errors of such a nature, and not in their detection or prevention, in this work.

2.2 Software Testing

Software testing is a broad term that classifies various techniques aimed at gaining confidence that the software implementation meets its specification. A *software specification* is a set of descriptions that indicates how the software implementation will meet the *software requirements*, which are the user-specified functional and non-functional objectives of the system to be designed. Examples of software testing activities include source code inspection, formal verification [113], and running test cases. The latter is referred to as *execution-based testing* [95], and is the focus of testing techniques in this work. Execution-based testing involves the generation, execution, and inspection of a group of individual test cases that are collectively referred to as a *test suite*. A test suite is a collection of *test case*s, where a test case verifies some desired property of the implementation.

2.2.1 Types of Testing

This section provides an overview of various types of testing approaches that are related to the research in this dissertation. For a thorough discussion of software testing, see for example Ammann and Offutt [22] or Mathur [70].

Functional and Non-Functional Testing

Both functional and non-functional testing goals are traceable to software requirements. *Functional testing* refers to verifying the implementation of requirements that specify the actions that a user or code can take. For example, demonstrating that pressing a button logs a user into a website, or that ordering a product from a vendor results in an email being sent to the consumer, are functional testing activities. By contrast, *non-functional testing* includes verifying issues such as scalability, availability, and security requirements. For example, guaranteeing that a server will be able to simultaneously process a predetermined number of user requests, or that adequate authentication policies exist, fall under non-functional testing. This work focuses on functional testing when proposing new approaches towards regression testing web-based applications.

Testing During the Software Lifecycle

Ideally, testing activities are an ongoing effort at the beginning of (or even before) source code implementation, and many fall into the following chronologically-ordered categories:

- **Unit Testing and Integration Testing**. *Unit testing* occurs before, during, or immediately after sections of code (such as classes or functions) have been written. Developers generate various tests that exercise the expected functionality of the code, corner cases, and individual components. By contrast, *integration testing* verifies that software components interface properly.
- **Regression Testing**. *Regression testing* is a supplementary approach to unit and integration testing that serves to ensure that changes to the code do not (re-)introduce defects. Regression tests are frequently executed after major changes to previously-working code have been dispatched, or as part of a nightly build or prerequisite to source code check-in in repositories.
- **Alpha and Beta Testing**. *Alpha* and *Beta* testing involve simulated or actual consumers exercising the source code in a realistic operational manner. This type of testing is generally applied when the software reaches a reasonable level of maturity with relatively few defects.

This research primarily focuses on regression testing of web-based applications under the assumption of a pre-existing test suite. Test suites can be run using all test cases they contain, which is known as a *retest-all* strategy. Re-running all test cases in a test suite may be an expensive task, consuming resources outside the range of what is available, especially in the web domain. In such cases, running a subset of the original test suite is referred to as a *test suite reduction* strategy; this issue is returned to in Section 8.5.

2.2.2 Oracles and Oracle Comparators

Inherent to almost all types of testing is the need for oracles, which are responsible for providing the correct, expected output of a test case. Formally, an *oracle* is a mechanism that produces an expected result and a *comparator* checks the actual result against the expected result [28]. Figure 2.1

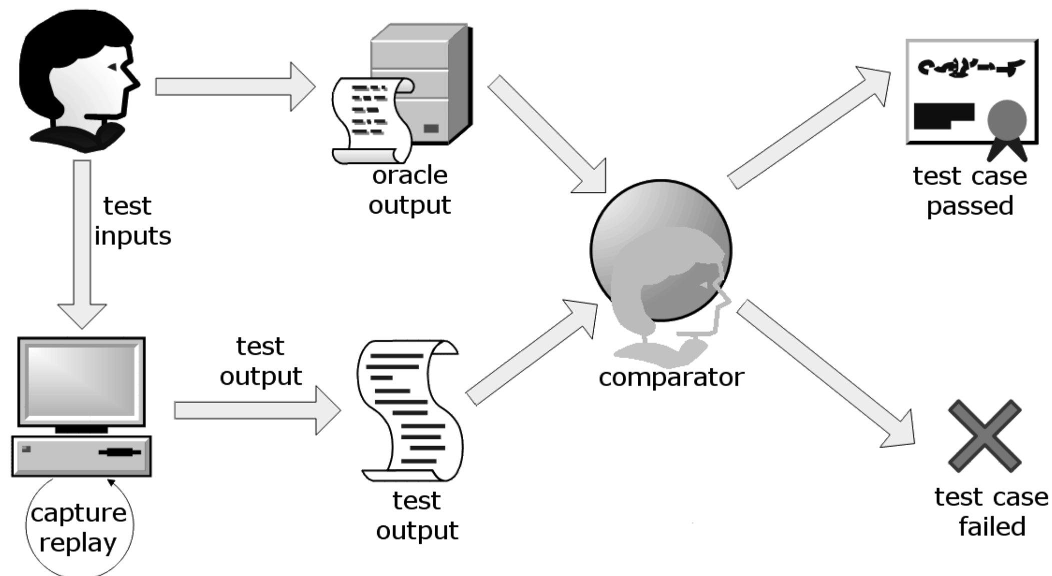

Figure 2.1: An oracle-comparator. A human (or in some cases software) provides test input to the system. The application is run on the test inputs and produces output. These test outputs are compared against oracle outputs (which must be specified in advance by a human or other software) using a comparator. The comparator may either be a developer manually examining output pairs, or it can be software. The comparator determines if the test case passed or failed.

diagrams the process of using an oracle comparator in testing. In the case of unit testing the oracle output may be manually specified. For other types of testing, and regression testing in particular, the oracle is commonly a previous, trusted version of the code.

Testing is often limited by the effort required to compare results between the oracle and test case outputs. The *oracle problem* [39] defines the need for comparison between obtained results and expected results. For many types of software, comparing the output of two versions of the application using textual differencing utilities such as the standard UNIX command `diff` is an effective means by which to identify defects in the code. Using `diff`-like tools for web-based applications, however, may return a large number of non-errors as requiring human inspection; this concern is further explored in the following chapter.

2.2.3 Test Suite Evaluation

Testing usually involves an oracle comparator that is responsible for correctly identifying passed test cases and test cases that reveal defects. For many types of software, and web applications in particular, however, determining whether or not a test case uncovers an actual fault may be difficult with an automated oracle comparator. Consequently, the performance of an oracle comparator can be described by the following four metrics:

- **True Positives** refer to actual defects in the source code uncovered by a test case, when the oracle comparator also flags the test case as requiring inspection.
- **False negatives** occur when a test case uncovers a fault in the code, but the oracle comparator labels the test case as passed. False negatives leave the potential to inadvertently ignore actual bugs.
- **True Negatives** are test cases that do not reveal any defects and that the oracle comparator also labels as not requiring human inspection. True negatives represent savings in terms of developer time by obviating the need to examine test case output manually.
- **False Positives** exist when a test case does not exhibit a fault, but the oracle comparator flags the output as requiring human inspection. False positives represent wasted developer effort, because a human examines such cases manually even though no actual defects exist.

The *precision* of a comparator is the number of true positives divided by the sum of true positives and false positives. The *recall* of a comparator is the number of true positives divided by the sum of true positives and false negatives. Ideally, an oracle comparator would be *perfectly precise* with a precision of 1.0, implying that there are no false positives. Ideally, an oracle comparator would also have *perfect recall*, with a recall of 1.0, implying that there are no false negatives. Precision can be trivially maximized by returning a single test case, while recall can similarly be maximized by returning all test cases. Combining the two measures by taking their harmonic mean results in the F_1-score. This metric gives equal weight to precision and recall. In practice recall, which penalizes missing real bugs, may be more important.

A *relatively-precise* or *reasonably-precise* oracle comparator seeks to minimize both the number of false positives and false negatives to below an acceptable threshold. In this work a *highly-precise* oracle comparator is defined as one where the ratio of the cost of examining true positive and false positive bug reports, compared to the cost of missing true negative bugs, is less than or equal to a previously published value of 0.023 [54], when comparing test case outputs between two program versions; for a formal definition of this notion of high precision see Section 5.5.2.

Although different oracle comparators may vary in their ability to correctly classify faults and non-faults, the burden of effective fault detection still lies with the test suite itself. Consequently, researchers and developers are frequently interested not only in measuring oracle comparator performance, but also in investigating the related domain of *test suite efficacy*. Unfortunately, defects in source code cannot be known a priori, which makes determining the effectiveness of a test suite's ability to reveal actual bugs challenging. Two widely-adopted complementary criteria are used to identify the efficacy of various test suites:

- **Code coverage** is a standard software engineering technique used to measure test suite efficacy. Each test case is mapped to some part of the code it exercises; coverage is often broken down into function, statement, branch, and other categories depending on the desired granularity. The goal is usually to maximize code coverage; for example, a test suite that exercises every statement in the code at least once is said to have full statement coverage. High code coverage is a desirable property due to the underlying assumption that executing code as part of a test suite will reveal defects.

- **Fault detection.** An orthogonal approach to code coverage is to directly measure the number of faults found through the use of a specific test suite. Because real-world faults are not known in advance (except when looking at older versions of a program), *fault-based testing* is used to introduce faults into the code meant to be uncovered by the test suite [28, 106]. A common approach for this is referred to as *fault seeding* or *fault injection*: faults can be manually inserted by individuals with programming expertise, or *mutation operators* can be used to automatically produce faulty versions of code, as in [24]. Examples of mutation

operators are deleting a line of code, replacing a logical `AND` with an `OR` in a line of code, and switching a + with a - operation.

Previous work has demonstrated that automatically-seeded faults using source code mutation are at least as difficult to find as naturally occurring ones for software in general [24, 56]. Whether or not manually seeded faults are equivalent to naturally occurring faults in the specific domain of web applications remains an open question.

Cost is also an important factor in determining test suite efficacy. This work presents a cost model where the quality of a testing methodology is defined as the product of the cost of an error and the number of such errors exposed by the test suite, divided by the cost of designing and running the test suite.

$$\frac{(\textit{cost of an error in terms of dollars lost}) \times (\textit{number of errors exposed})}{\textit{cost of running the test suite}}$$

Under this cost model a more *effective* test suite may ultimately discover fewer faults then a competitor. In reality, all testing is subject to this constraint, as an optimal test suite could theoretically uncover all faults by running the application on all possible inputs, an infeasible approach in practice. Given the large size of the input space, test suite reduction is one technique that aims to select test cases that are most likely to find bugs, or alternatively, to filter out test cases that are unlikely to find new bugs (such as duplicate tests).

2.3 Summary

This chapter provided a general background on web applications and software testing. Web applications involve passing information between a server and a client that may or may not be customized according to user data. Software testing is a means by which defects in the source code are uncovered. The following chapter explains the current state-of-the-art of applying general software testing techniques to web applications in particular.

Chapter 3 Testing Web-based Applications

This chapter presents an overview of the current state-of-the-art in web-based application testing technologies, as well as the criteria researchers use to evaluate competing approaches. Many standard software testing techniques have been adopted and adapted to this domain. Testing of web-based applications remains an open research area, however, due to their combination of high quality requirements and resource constraints. Most web-based application testing approaches either tackle the challenge of cost reduction through automation, or aim to provide guidelines or techniques to increase fault coverage in testing this type of software, where HTML code is often dynamically generated.

3.1 Defining Errors in Web-based Applications

Web-based applications present additional challenges in testing because the term "error" may have different meanings to different people. As an example, usability issues, such as the incapability of a customer to locate a `Login` link, may not be considered as errors in testing. Ma and Tian define a *web failure* as "the inability to obtain and deliver information, such as documents or computational results, requested by web users." [66]. It remains unclear whether usability (as opposed to correctness) issues are adequately considered in the automated testing processes of web applications.

Faults uncovered in testing can also be classified into different types, and some techniques are better at exposing certain types of faults [106]. Ostrand and Weyuker initially classified faults in terms of their fix-priorities [79], but later rejected that approach, concluding that using such severity measures was subjective and inaccurate [80, 106]. These *developer-perceived* fault severities are in contrast to *consumer-perceived* fault severities, which are presumably free of such biased and misleading judgments that are often due to internal politics within the development organization [80].

Fault taxonomies for web applications are in their infancy, in that only a few preliminary models exist. For web applications in particular, Guo and Sampath identify seven types of faults as an

Characteristics	Sub-Characteristics	Classes of Faults
A. Multi-tier architecture	1. client pages interpreted by browsers	f1. faults related to browser incompatibility f2. faults related to back button f3. faults related to the needed plugins
	2. server pages can dynamically generate client pages	f1. faults in the construction of dynamically built client-side pages f2. faults related to inputs of server-side pages
	3. use server-side components (e.g. JavaBeans)	f4. faults during file-system access f8. faults related to server environment (e.g., web server) f9. faults related to character encoding of the input data
	4. form and link are used to exchange data between components	f1. faults during form construction
	6. are database-based	f1. faults during database access or mangmt f4. faults on the information search process
	7. client-side page can be stored in proxies or browser cache	f4. wrong storage of information in cache
B. GUI	1. interfaces can by HTML-based	f1. faults related HTML interpretation by the browser f3. faults while manipulating DOM objects
	3. client pages can be organized on frames and framesets	f1. faults on frame synchronization f2. faults on frame loading
	5. interfaces need to be internationalized and multi-languages	f1. faults related to characters encoding f3. unintended jump among languages
C. Session-based	1. server components can use session objects	f2. faults in session synchronization f4. faults in persistence of session objects f5. faults while manipulating cookies
D. Hyperlinked structure	1. support hypertext and hyperlink	f2. faults related to web pages integrations f8. faults while build dynamic URL
	2. resources are accessed by URL	f1. faults due to the unreached resources f2. faults due to the not available resources
E. Protocols-based	1. can use the data encryption	f1. faults related to the use of encrypted communication
	2. can be used through proxies distributed on the net	f1. proxies do not support a given used protocols
F. Authentication	1. manage authorizations to allow the users to use/access resources	f2. fault during user authentication f3. faults in account management f6. faults in accessing/using resources without permission f8. faults in role management

Figure 3.1: Web fault taxonomy by Marchetto *et al.*, reprinted from [69].

initial step towards web fault classification [47]. Marchetto *et al.* validate a web fault taxonomy to be used towards fault seeding in [69]. Their fault categories are summarized in Figure 3.1, and are organized by characteristics of the fault that generally have to do with what level in the three-tiered architecture the fault occurred on or some of the underlying, specific web-based technologies (such as sessions). Faults are broken down into several categories and subcategories to aid developers when considering what types of defects to inject into their code. Despite their comprehensive nature, in these fault classifications [47, 69] there is no explicit concept or analysis of severity — while some categories of faults may, in general, produce more errors that would turn customers away (such as Authentication problems in Figure 3.1), this consideration is not explored. One of the main goals of the research presented in this document is to explicitly investigate the consumer-perceived severity of faults, which includes a breakdown of fault severity according to the source of the defect.

3.2 Existing Approaches

Several tools and techniques exist for testing web applications, but most of them focus on protocol conformance, load testing, broken link detection, HTML validation, and static analyses that do not address functional validation [39, 106]. These are low-cost approaches with a relatively high return on investment, in the sense that they can easily detect, without manual effort, some errors that are likely to drive away users. Static components of websites, such as links, HTML conformance, and spelling can be easily checked by automated spider-like tools that recursively follow all static links of the application, inspecting for errors [27].

3.2.1 Capture-Replay

Testing dynamic, functional components automatically is an active research area [26]. Validating the functionality of a web-based application can be challenging due in part to the manual effort required to generate, run, or verify test cases. For example, unit testing of web applications, using tools such as CACTUS [6], requires the developer to manually create test cases and oracles of ex-

pected output. Similarly, structural testing techniques require the construction of a model of UML or structural and behavioral test artifacts [58, 64, 85], which is usually carried out manually [106]. Other approaches towards functional validation are usually of a *capture-replay* nature [88], where interactions with the browser are recorded and then replayed during testing. In these cases, a developer manually records a set of test scenarios, possibly by interacting directly with the application, which can then be automatically rerun through the browser.

Although capture-replay is a highly portable testing approach [106], these types of tests can easily fail for trivial reasons. In what is known as the *fragile test problem*, replayed scenarios are subject to behavior, interface, data, and context sensitivities [73]. It is especially important to avoid these types of failures by maintaining flexibility of the test suite and verification strategies. Such sensitivities become apparent during an attempt to use an automated oracle comparator to compare test output of web-based applications (see Section 3.2.2).

User Session Based Testing

In *user session based testing*, user accesses are recorded by the server and replayed during testing [39, 101]. A user session is composed of a list of such `URL` requests, in order, starting when a request from a new IP address reaches the server and ending when the user leaves the website (or the session times out) [101]. Because `URL`s can contain form data as name-value pairs passed to the server, an important advantage of user-session based testing is the possibility of testing the dynamic (form) elements of the application without relying on the developer to provide inputs manually.

This specific type of capture-replay has the advantage that a small modification to the server can provide a large set of easily-collected data. Additionally, such input may be more representative of how users would interface with the website than interactions fabricated by in-house test creators. Elbaum *et al.* pioneered the study of leveraging user sessions as test cases to detect a high number of faults [39, 41]. The effectiveness of their approach increases as the number of recorded sessions grows. Replaying many user sessions may exceed the resource constraints typically placed on web application development. Consequently, optimizing the playback of recorded user sessions [102], reducing the size of these test suites [41, 93, 105], and statistically modeling user sessions to derive

test cases [94] are important steps to reduce the cost of replaying user sessions.

Although user session data is easy to capture and replay, one drawback of this approach is that it is meant to be applied during beta-testing phases or to supplement an existing test suite [39]. Like any testing methodology, a test suite composed of user sessions must still have a predefined set of correct outputs to be compared against. This oracle is often taken to be a test suite run on a previous, trusted version of the code, although without manual verification of all such test outputs, guaranteeing that no defects exist in these oracle files is impossible.

3.2.2 Oracles

Recent work [41,64,101,103,104] uses HTML output as oracles, because such data is easily visible and because lower-level faults typically manifest themselves as user-visible output [81,104]. Oracle comparators are frequently used for testing web applications, and in practice discrepancies are examined through human intervention [39,64,86,104]. For many types of software, using a textual `diff` is an effective method for differentiating between passed and failed test cases. Unfortunately, a `diff`-based comparator for web-based applications produces frequent false positives [104] which must be manually interpreted, and manual inspection is an expensive process. False positives are generated through the use of `diff` when the incremental nature of website updates (described in Section 1.1) may not change the appearance or functionality experienced by the user. That is, natural web-based application development may cause an imprecise comparator, such as `diff`, to flag an output pair for manual inspection when such inspection is not actually needed.

Consider the `diff` output from two TXT2HTML test case versions [3] (TXT2HTML is a tool that converts textual documents into HTML):

```
1  < <P>The same table could be indented.
2  < <TABLE border="1">
3  ---
4  > <p>The same table could be indented.</p>
5  > <table border="1" summary="">
```

A new attribute (`summary`) and tag (`</p>`) have been added, but these two pieces of code are

displayed the same way in most browsers. Assuming the first version is the oracle output and the second version is the corresponding test case output, a naïve `diff`-like comparator would return a false positive for the example above, unnecessarily requiring manual inspection.

Change Detection

Detecting changes between domain-specific documents is a frequent challenge is many applications. Change detection in web pages has been explored in the context of plagiarism detection [96] and web page update monitoring [25, 43, 62]. For example, users may want to monitor changes in stock prices, updates to a class webpage, or other pre-specified data through one of these approaches [1, 8, 25]. Such monitor and change detection tools are often faced with long runtimes, and recent work has focused on optimizing such algorithms to run faster [25]. Flesca and Masciari use three likeness measures to detect the percentage of similar words, measures of tree element positions, and similar attributes between two XML-based documents [43]. Such structure-aware analyses may be useful in designing highly-precise oracle-comparators, as long as the focus is shifted towards error detection. An ideal oracle comparator for web-based applications would be able to handle both the structural evolutions (as in DIFFX — see below) as well as updates to content (as in natural language tools) in order to specifically differentiate between defects and correct output, as opposed to pinpointing or summarizing updates. This dissertation presents such a highly-precise oracle comparator (see Section 2.2.3) that uses structural and semantic judgments to classify faults and non-faults.

In other types of software, differencing in tree-based documents (such as XML and abstract syntax trees) can be accomplished by a tool such as DIFFX [20], which characterizes the number of insertions, moves, and deletes required to convert one tree to the other as a minimum-cost edit script [20, 116]. Change detection for natural language text can be achieved through a bag-of-words model [60], standard `diff`, and other natural language approaches. Detecting changes between different source code versions is often accomplished through `diff` as well. Although recent work has explored using semantic graph differencing [83] and abstract syntax tree matching [75] for analyzing source code evolution, such approaches are not helpful in comparing XML and HTML

text outputs. Not only do they depend on the presence of source code constructs such as functions and variables, which are not present in generic HTML or XML, to make distinctions, but they are meant to summarize changes, rather than to decide whether or not an update signals an error.

Oracle Comparators for the Web

Traditional testing for programs with tree-structured output is particularly challenging [101] due to the number of false positives returned by a `diff`-like comparator [104]. Additionally, if such naïve comparators are employed, oracle output quickly becomes invalidated as the software evolves, as test cases are unable to pass the comparator due to minor updates. Instead, web-based applications would benefit from a highly-precise oracle comparator that is able to differentiate between unimportant syntactic differences and meaningful semantic ones. One approach is for developers to customize `diff`-like comparators for their specific applications (for example, filtering out mismatching timestamps), but these one-off tools must be manually configured for each project and potentially each test case — a human-intensive process that may not be amenable to the frequent nature of updates in the web domain.

Providing a reasonably-precise oracle comparator for web-based applications is an active area of research. Sprenkle *et al.* have focused on oracle comparators for testing web applications [101, 103, 104]. They use features derived from `diff`, web page content, and HTML structure, and refine these features into oracle comparators [104] based on HTML tags, unordered links, tag names, attributes, forms, the document, and content. Applying decision tree learning allowed them to target combinations of oracle comparators for a specific application, however this approach requires manual annotation [103].

3.2.3 Automation

Given the extraordinary resource constraints in web development environments (see Section 1.1), the automation of testing techniques has been a main focus of research in this domain. Automation can occur at any level of the testing life cycle, including test case generation, replay, and error detection. This dissertation focuses on automated error detection in web application testing though the

use of highly-precise oracle comparators (as described in Section 2.2.3) to verify the functionality of the website.

Automation of Test Case Generation

Two main approaches have been adopted for automatically generating test cases for web applications: harnessing user session data (as described in Section 3.2.1), and determining a list of valid URLs in the style of a normal user access through other analyses. One example of the latter is VERIWEB [27], where links on a website are visited and dynamic URLs are generated by exercising the forms present with pre-defined form inputs. Other methods are able to automatically generate test cases [86, 87] or repair session data [21] once a model of the system has been built. For scripting languages such as PHP, constraint-solving can be used to automatically generate form inputs that will crash the system [26, 117]. This dissertation explores the orthogonal area of automated test case comparison, which can be combined with any type of test case generation approach, including the automated ones discussed above.

Automated Test Case Replay

One advantage of testing web applications, as opposed to other types of software, is that the automation of test case replay is relatively straightforward if the test cases consist of URLs, as such tests can easily be replayed in a browser. For applications that persist data, care must be taken to ensure each run of the test suite is conducted on an appropriate version of the items in the database. With the exception of restoring database state between test suites, test case replay is otherwise easily accomplished in this domain without added infrastructure beyond the ability to make requests to the server. Experiments in this dissertation frequently use UNIX transfer utilities such as `wget` [12] and `curl` [13] to make such server requests as an effective way of replaying test cases automatically.

Automated Failure Detection

Relying on a human to manually verify all test case outputs places an unreasonably high burden on developers in the web application field. Instead, approaches such as `diff` and more sophis-

ticated oracle comparators seek to reduce the amount of test case output humans must manually examine. Automated failure detection in web application testing can be achieved though the use of reasonably-precise oracle comparators [101] (as described in Section 2.2.3) to verify the functionality of the website. Application-level failures in component-based services can also be detected automatically [34], although this approach is directed more at monitoring activities than testing. Validating large amounts of output or state remains a difficult problem and is the subject of ongoing research [53, 106]. One primary focus of this dissertation is such automated error detection through the use of a highly-precise oracle comparator. In contrast to other work that employs a *partially-automated* oracle comparator [106], Chapter 6 explores a *fully-automated* oracle comparator that does not depend upon manual annotations or configurations.

3.2.4 Test Suite Efficacy in Web-based Applications

Similar to the testing of other types of software, web-based application testing methodologies are frequently evaluated on some metric other than their ability to detect real-world faults in the current version of the application, as real-world faults cannot always be known in advance. Code coverage metrics are frequently used in web application testing [26, 39, 48, 64, 94, 101, 102, 103, 104, 105], although the average percentage of statement coverage falls well short of 100% (and is often closer to 60%) in many studies [26, 48, 94, 101, 102, 103, 104, 105]. As a complementary approach, fault injection through source code mutation (see Section 2.2.3) has also been used extensively in the web application testing community [26, 37, 39, 77, 101, 103, 104, 105].

The cost of developing and running a test suite is also an important consideration in the web-based application domain due to the frequent nature of updates and the rate at which this type of software is typically developed. Under the cost model of test suite efficacy, a smaller test suite is preferable to a larger one when its ability to detect defects is comparable. Traditional test suite reduction techniques such as Harrold, Gupta, and Soffa's reduction methodology [51] have been successfully applied to user-session based testing [41]. Other approaches focus on web applications characteristics in particular, such as data-flow [63] and finite state machine [23] analyses, and use case [35] and URL-based coverage [92]. Being able to reduce the size of the test suite

is an important consideration given the resource constraints which web applications are subject to. As Chapter 7 explores, however, it is important to perform both a quantitative and *qualitative* assessment of error detection preservation with such reduction approaches, in that not only the quantity, but also the severity, of faults be considered when comparing reduced test suites to their original counterparts.

3.3 Graphical User Interface Testing

Many similarities exist between Graphical User Interfaces (GUIs) and web applications — a browser-displayed webpage is a kind of GUI. Like a webpage, a GUI can be characterized in terms of its widgets[1] and their respective values. Xie and Memon define a GUI as a "hierarchical, graphical front-end to a software system that accepts input as user-generated and system-generated events, from a fixed set of events, and produces deterministic graphical output." [122]. Notably, they exclude web-user interfaces that have "synchronization and timing constraints among objects" and "GUIs that are tightly coupled with the back-end code, *e.g.*, ones whose content is created dynamically..." [122]. Therefore, this definition of GUIs excludes many web applications, and as a consequence, some GUI testing research is not applicable to testing web applications and vice-versa.

Like web applications, GUIs are difficult to test due to the exponential number of states the software can be in [121], as well as the manual effort required to develop test scripts and detect failures [32]. Similarly, they are often not tested at all, or are tested using capture-replay tools that capture either GUI widgets or mouse coordinates [72]. While advances in GUI testing technology may apply to the web application testing domain, the latter has its own additional challenges. Primarily, most GUIs lack a dynamically-generated HTML description. The availability of HTML as a standard description language for both content and presentation control implies that further analyses are possible on this output, and some GUI testing methodologies are not directly applicable. Web application content is very likely to by dynamically generated, while GUIs are relatively static by

[1] A widget is a visual element of a GUI that is used by a human to manipulate data. Windows, text boxes, and buttons are examples of widgets.

comparison. Additionally, customers using the web frequently have the option of easily switching providers, while GUI-based systems are often purchased and installed, making a direct comparison of consumer-perceived fault severity between the two types of software difficult. Faults are likely to manifest themselves in different ways; for example, web applications frequently fail and display stack traces in experiments in this work, while GUIs may be less likely to do so in the middle of normal GUI content. This dissertation focuses on web-based application user interfaces only. In future work, faults in GUI applications can be analyzed and some of the guidelines and techniques in the current work can potentially be extend to that domain.

3.4 Improving the Current State of the Art

Research in web-based application testing often focuses on reducing costs through (1) the automation of activities, and (2) more effective error exposure. By studying user-visible errors in web-based applications in the context of web-based application testing, one main goal of this dissertation is to further cut the costs of testing by modeling errors in web-based applications to identify them more accurately, as well as further automating the oracle comparator process. Specifically, this dissertation focuses on user-visible error detection, with the assumption of a provided test-suite with a retest-all strategy.

Additionally, this dissertation aims to make web testing more cost-effective by devising a model of user-visible error severity that will guide test case design, selection, and prioritization. This model of error severity has the additional benefits of refuting the underlying assumption that all faults are equally severe in fault-based testing [40, 108] for web applications, and offering software engineering techniques for high-severity fault avoidance to developers who do not have the resources to invest in testing. Unlike the severities explored by Ostrand and Weyuker [79, 80], these severities are not the developer-assigned severities to faults (such as found in bug reporting databases), but are instead based on human studies of consumer-perceived severities of real-world faults. Such human-driven results can be more indicative of true monetary losses and especially relevant in the web domain.

Chapter 4 Research Outline

This chapter provides an outline of the main contributions of this dissertation, focusing on user-visible errors in web-based applications in the context of web-based application error detection. Each main contribution is detailed in a subsequent chapter. Recall the overall thesis of this dissertation:

> *Web-based applications have special properties that can be harnessed to build tools and models that improve the current state of web application user-visible error detection, testing, and development.* (Thesis)

In this work, error detection is addressed in two main contexts: during regression testing of web-based applications, and through a focus on identifying severe errors. The main contributions are summarized below:

- This dissertation aims to reduce the cost of regression testing web-based applications by capitalizing on the special structure of web-based application output to precisely identify errors. **A highly-precise oracle comparator is constructed for XML/HTML to model errors and non-errors in this domain**. This oracle comparator is successful if it can significantly reduce the false positives associated with `diff` while having few or no false negatives. An additional success metric related to the cost of inspecting a bug report versus the cost of missing a bug is used to evaluate the cost effectiveness of this approach. Both human-assisted and fully-automated oracle comparators are proposed and evaluated.

- This research grounds the current state of industrial practice by refuting fault injection as a standard for measuring web application test suite quality. **A large scale human study demonstrates that the assumption that all injected faults have the same non-trivial severity, and thus, the same benefit to developers, is false in this domain**. The hypothesis that injected faults in web applications do not have the same consumer-perceived error sever-

ity is tested by verifying that these error severities do indeed vary on a statistically significant sample size of humans and injected faults.

- This dissertation also presents techniques to reduce the cost of testing web applications by exposing high-severity errors through test case design, selection, and prioritization (including test suite reduction). Specifically, **both human-assisted and fully automated models of consumer-perceived error severity for web application faults are constructed**. These models aid developers in test case prioritization. Such models are judged successful if they agree with human judgments of severity at least as often as humans agree with each other.
- This work provides insight on how to avoid high-severity errors during web application design and development. Specifically, **software engineering guidelines, based on the results of a large-scale human study, are provided to developers under the assumption that little or no resources are available to test web applications**. Such guidelines are considered useful if they correlate, in a statistically significant manner, with reduced error severities in real-world errors.

The remainder of this summary chapter presents seven specific falsifiable hypotheses used to evaluate this thesis (to be introduced in the following section).

4.1 Improving Error Detection During Regression Testing

The first main context of this work is regression testing of web-based applications. This dissertation details automated methods that reduce the number of false positives during regression testing web-based applications. False positives are a common problem when comparing expected HTML output to the actual output because differentiating between innocuous program changes and actual errors is difficult. For example, using a `diff`-like tool to compare HTML documents may incorrectly flag as suspicious a change to the height of an **`<img>`** tag, or variances in session cookie identifiers within a URL, because neither of these differences are faults. Reducing the number of test cases that are incorrectly labeled as requiring human attention saves developers time and effort, which

is especially important in the dynamic development environment of these types of applications (as detailed in Chapter 1). Similarly, these aggressive development conditions suggest that automation in this approach is necessary in this domain for any approach that is expected to be viable.

Chapter 5 details how the special structure of web-based applications can be used to provide a partially-automated, highly-precise oracle comparator for this domain that reduces the number of false positives associated with more naïve comparators, while missing few or no actual bugs. Because web-based applications mostly output in XML/HTML form, this user-visible layer of the application often corrals errors from lower levels not directly palpable to the user [81, 104]. This claim is formalized in Hypothesis (H1):

> *A highly-precise oracle comparator for web-based application testing can be constructed, based on specific knowledge of which structural and semantic features of output discrepancies are likely to be associated with errors, that reduces the number of non-errors flagged as requiring human inspection (i.e., the number of false positives) compared to off-the-shelf techniques such as* `diff` *and* `xmldiff` *while maintaining the ratio of the cost of examining a potential bug to the cost of missing an actual bug at or below a current state-of-the-art value of 0.023 [54].* (H1)

Chapter 6 expands upon such a partially-automated, highly-precise oracle comparator by providing full automation through a key insight about similarities between the special structure of web-based application output for unrelated applications. Not only do individual web-based applications lend their output to an analysis of structure and content that can yield more effective error labeling in regression testing, but, as this work demonstrates, unrelated web-based applications tend to both fail and evolve in predictable ways. This claim is formalized in Hypothesis (H2):

A highly-precise, fully-automatic *oracle comparator for web-based application testing can be constructed, based on pre-existing information from unrelated applications, that has fewer false positives than off-the-shelf techniques such as* `diff` *and* `xmldiff` *while maintaining the ratio of the cost of examining a potential bug to the cost of missing an actual bug at or below a current state-of-the-art value of 0.023 [54].* (H2)

This work shows that training a highly-precise oracle comparator with unrelated application data maintains gains in effective false positive reduction when compared with a human-assisted model. Consequently, both the partially- and fully-automated models significantly outperform more naïve approaches.

4.2 Focusing on Severe Errors to Improve Error Detection

The remainder of this work focuses on consumer-perceived error severity in web applications. Chapter 7 describes the results of a large-scale human study that forms the basis for a predictive model of consumer-perceived error severity in web applications. The current state of research in web application testing implicitly assumes that all injected faults are equally severe, and thus that raw fault detection counts can be used directly to evaluate test suite efficacy. The research presented in this work and human study data is used to refute this claim, formalized as Hypothesis (H3):

Faults injected into web applications, using an automated seeding process using mutation operators described in Section 2.2.3, or using manual fault seeding as in [106], vary in their underlying consumer-perceived severities. (H3)

This research also increases the utility of testing in this domain by allowing developers to focus on severe errors. Because testing may be perceived as having a low return on investment for web applications (as detailed in Chapter 1), a model of error severity is provided, based on human judgments, that allows developers to prioritize errors according to their likelihood of impacting consumer retention, thus encouraging web application developers to test more effectively. This

claim is formalized in Hypothesis (H4):

> *An automated model of consumer-perceived error severity can be constructed that agrees with human severity judgments at least as often as humans agree with each other, evaluated using the Spearman's Ranking Correlation Coefficient (SRCC) [44, 107].* (H4)

This model is more accurate than an average human at correctly labeling error severity. Both an annotation-based model, that relies on human input, as well as a fully automated model are presented. Such models can be used directly to prioritize errors for fixes, as well as in the following applications:

1. The error severity models provide insight on how to avoid high-severity errors during web application design and development, which are detailed in Chapter 8. As a concrete example, the severity of a user-visible error can be reduced by opting to present the defect in the form on a popup when possible, as opposed to presenting it on an unexpected error page. Therefore, this dissertation will explicitly explore the following Hypothesis (H5):

   > *There exists a statistically significant correlation (SRCC > 0.60 [44, 107]) between severe errors in web applications and various software engineering aspects of web application development.* (H5)

   Such an analysis is orthogonal to other work in this dissertation which seeks to reduce the cost of testing web-based applications directly.

2. The error severity models can be used to reduce the cost of testing web applications by exposing high-severity errors through test case design, selection, and prioritization (see test suite reduction, also in Chapter 8). This notion is formalized as Hypothesis (H6):

   > *There exist test suite reduction strategies that expose at least 90% of the severe errors found via corresponding retest-all approaches for web applications.* (H6)

   One goal of this work is to apply the consumer-perceived error severity model to various

test suite reduction techniques to explore the cost-to-error severity trade offs with different approaches.

Chapter 5 through Chapter 8 explore automated ways to reduce the cost of testing web-based applications, as well as provide a model of and guidelines for reducing the consumer-perceived severity of web application errors. Chapter 9 concludes the research contributions of this dissertation by combining these two approaches and investigating the severity distributions of errors detected using the automated oracle comparator presented in Chapter 6 on a set of popular web applications. Formally, Hypothesis (H7) is introduced:

At most 1% of the false negatives produced by the proposed highly-precise, fully-automated oracle comparator correspond to severe errors. (H7)

The goal of such a study is to measure the automated oracle-comparator's performance on a dataset that makes heavy use of non-determinism, thereby potentially increasing the relative number of false positives associated with an oracle comparator. In addition, the consumer-perceived severity of those errors missed by the automated oracle comparator presented in Chapter 6 is evaluated.

4.3 Summary

The main thesis of this dissertation is that user-visible web-based application errors have special properties that can be exploited to improve the current state of web application error detection, testing and development. This conjecture can be broken down into seven main sub-hypotheses:

- that recognizing errors in web-based applications can be successfully modeled due to the tree-structured nature of XML/HTML output (see (H1)),
- that unrelated web-based applications fail in similar ways (see (H2)),
- that not all failures are equally severe from a consumer perspective (see (H3)),
- that these failures can be modeled according to their consumer-perceived severities (see (H4)),

- that severe errors correspond to specific software engineering aspects during web application development (see (H5)),
- that test suites can be reduced in size while preserving severe error exposure (see (H6)),
- and that automated tools to detect failures rarely miss severe errors (see (H7)).

By improving upon error detection, this research provides efficient, automated approaches to the testing of web-based applications that reduce the cost of this activity, making its adoption more feasible for developers. Additionally, constructing a model of web application error severity invalidates the current underlying assumption of error severity uniformity in fault seeding, helps developers focus their effort through error prioritization, guides software engineering to avoid high severity errors, and assists testing techniques in finding high-severity errors. Studying error severities from the consumer perspective is a novel contribution to the web application testing field.

Chapter 5 Improving Error Detection During Regression Testing

This chapter explores ways to improve error detection during regression testing web-based applications. Specifically, a highly-precise oracle comparator, called SMART, is presented, that makes use of the special structure of web-based application output to reduce the number of non-errors labeled for inspection by more naïve comparator approaches, while minimizing the number of missed true errors. This contribution will be used to evaluate Hypothesis (H1) from the previous chapter, that:

> *a highly precise oracle comparator for web-based application testing can be constructed, based on specific knowledge of which structural and semantic features of output discrepancies are likely to be associated with errors, that reduces the number of non-errors flagged as requiring human inspection (i.e., the number of false positives) compared to off-the-shelf techniques such as* `diff` *and* `xmldiff` *while maintaining the ratio of the cost of examining a potential bug to the cost of missing an actual bug at or below a current state-of-the-art value of 0.023 [54].* (H1)

By recognizing that user-visible errors in web-based applications can be successfully modeled in this way, such an oracle comparator can be constructed that provides significant savings to developers over `diff`-like approaches.

5.1 Challenges in Regression Testing Web-based Applications

Regression testing is frequently limited by the effort required to compare results between two versions of program output. Formally, testing involves an *oracle* mechanism (as described earlier in Section 2.2.2), that produces an expected result and a *comparator* that checks the actual result against the expected result [28]. In practice, the oracle is commonly taken to be the output of a pre-

vious, trusted version of the code on the same input and the comparator is a simple `diff` of the two outputs. Any difference implies that the test case should be inspected by developers; this commonly suggests an error in the new version, but may also indicate a discrepancy in the oracle output (e.g., the correct output may legitimately change as the program gains new functionality). Unfortunately, traditional regression testing is particularly burdensome for web-based applications (e.g., [101]) because using `diff` as the oracle comparator produces too many false positives (see Section 3.2.2).

Consider the following example from a GCC-XML test case; GCC-XML is an output extension to the GCC compiler [2]:

```
1  <?xml version="1.0"?>
2  <GCC_XML>
3    <Namespace id="_1" name="::" members="_3 _4 "/>
4    <Namespace id="_2" name="std" context="_1" members=""/>
5    <Function id="_3" name="bad" returns="_5" context="_1" location="f0:9">
6      <Argument name="src" type="_5"/>
7      <Argument name="o5" type="_5"/>
8      <Argument name="w5" type="_5"/>
9    </Function>
10   <Function id="_4" name="good" returns="_5" context="_1" location="f0:3"
       >
11     <Argument name="src" type="_5"/>
12     <Argument name="o5" type="_5"/>
13     <Argument name="w5" type="_5"/>
14   </Function>
15   <FundamentalType id="_5" name="unsigned int"/>
16   <File id="f0" name="/home/eas2h/gcc-4.3.1/gcc/testsuite/gcc.c-torture/
       unsorted/bx.c"/>
17 </GCC_XML>
```

A newer version of the program is run on this same test case with the following output:

```
1  <?xml version="1.0"?>
2  <GCC_XML>
3    <Namespace id="_1" name="::" members="_3 _4 " mangled="_Z2::"/>
```

```
4     <Namespace id="_2" name="std" context="_1" members="" mangled="_Z3std"
          />
5     <Function id="_3" name="bad" returns="_5" context="_1" mangled="
          _Z3badjjj" location="f0:9" file="f0" line="9" endline="11">
6       <Argument name="src" type="_5"/>
7       <Argument name="o5" type="_5"/>
8       <Argument name="w5" type="_5"/>
9     </Function>
10    <Function id="_4" name="good" returns="_5" context="_1" mangled="
          _Z4goodjjj" location="f0:3" file="f0" line="3" endline="5">
11      <Argument name="src" type="_5"/>
12      <Argument name="o5" type="_5"/>
13      <Argument name="w5" type="_5"/>
14    </Function>
15    <FundamentalType id="_5" name="unsigned int"/>
16    <File id="f0" name="/home/eas2h/gcc-4.3.1/gcc/testsuite/gcc.c-torture/
          unsorted/bx.c"/>
17  </GCC_XML>
```

Some elements, such as the `<File>` element (on line 16), are exactly the same in both outputs. Other elements, however, have different attribute values: when these two test cases outputs are compared using `diff`, the following differences are returned (the text above the dashed line was generated by the older application, while the rest is output from the newer version):

```
1  >    <Namespace id="_1" name="::" members="_3 _4 "/>
2  >    <Namespace id="_2" name="std" context="_1" members=""/>
3  >    <Function id="_3" name="bad" returns="_5" context="_1" location="f0:9
        ">
4  ---
5  <    <Namespace id="_1" name="::" members="_3 _4 " mangled="_Z2::"/>
6  <    <Namespace id="_2" name="std" context="_1" members="" mangled="_Z3std
        "/>
7  <    <Function id="_3" name="bad" returns="_5" context="_1" mangled="
        _Z3badjjj" location="f0:9" file="f0" line="9" endline="11">
8  10c10
```

```
9  >    <Function id="_4" name="good" returns="_5" context="_1" location="f0
        :3">
10 ---
11 <    <Function id="_4" name="good" returns="_5" context="_1" mangled="
        _Z4goodjjj" location="f0:3" file="f0" line="3" endline="5">
```

Notice that in the older version, the **<Function>** element on line 3 of the `diff` output contains the same XML attributes and attribute values as the matching **<Function>** element of the newer version on line 7. However, the newer version also contains four additional attributes: **mangled**, **file**, **line**, and **endline**. These additional attributes are new functionality added in the later version of GCC-XML, and should be unit tested (see Section 2.2). In terms of regression testing, on the other hand, using standard `diff` to compare the two results would flag each of the matching **<Namespace>** and **<Function>** pairs between the two versions above, even though all of the old functionality remains the same (i.e., each element in the new version contains the same attributes and attribute values as in the old). In this case such added functionality would be flagged as a false positive by a naïve comparator like `diff`. By contrast, SMART, the oracle comparator presented in this chapter, avoids labeling this test case as a potential bug by modeling errors and non-errors in a web-based application.

The above example, as well as the example in Section 3.2.2, demonstrate that using `diff` to compare outputs for regression testing is often not appropriate for XML or HTML applications because of the potential for frequent false positives. Similarly, small formatting changes in HTML files, changes to boilerplate natural language text, or rearrangement of elements may be flagged by a `diff` comparison, even though the higher-level human interpretation of the output remains unaffected. An oracle comparator specialized for web-based applications should avoid returning such benign differences, while maintaining the ability to detect actual faults. Hard-coding any such rules, however, not only requires manual effort, but leaves such a comparator vulnerable to future program evolutions that impact such policies. SMART operates by learning features of faulty and correct test case output, rather than relying on approaches that involve manual implementation of a static set of rules.

Although HTML-specific comparison tools, such as HTMLMATCH [14] (which displays only textual changes between two files), and generic GUI-based tools that compare source code files for changes (such as WINMERGE [19] and EXAMDIFF [11]) can ignore more changes between pairs of files than `diff`, such approaches are meant to provide a visual summary of differences, and are unable to classify changes as errors or non-errors. For example, HTMLMATCH would highlight a change to a timestamp in the text of an HTML file as a meaningful change, when in most cases such a difference would be ignored by developers. Similarly, although EXAMDIFF is able to ignore whitespace and case when comparing program source code, it is unable to decide whether or not changes to the file merit human inspection or not. By contrast, SMART is able to distinguish between changes that are likely to indicate an error and those that are normal program evolutions.

5.2 Reducing the Cost of Regression Testing Web-based Applications

SMART reduces the cost of testing web-based applications by modeling differences in pairs of XML/HTML output, saving developers effort over more naïve approaches. In particular, insights from structural differencing algorithms (e.g., [20]) as well as semantic features are combined (e.g., [101]) into a semantic distance model for test case output. This distance metric then forms the heart of SMART, a highly-precise oracle comparator, where a regression test should be inspected if the new output's distance from the oracle output exceeds a certain cutoff.

SMART classifies test case output based on structural and semantic features of tree-structured documents. Although some are complicated, most features are quite simple, such as counting the number of inserted elements when converting one tree into the other. As a concrete example, consider the HTML, **`<b><u>text</u></b>`**, renders identically to **`<u><b>text</b></u>`**, even though the order of the bold and underline tags has been reversed.

5.2.1 Tree Alignment

Web-based applications produce HTML and XML output with tree-like properties. To recognize such features, SMART first aligns such output trees by matching up nodes with similar elements.

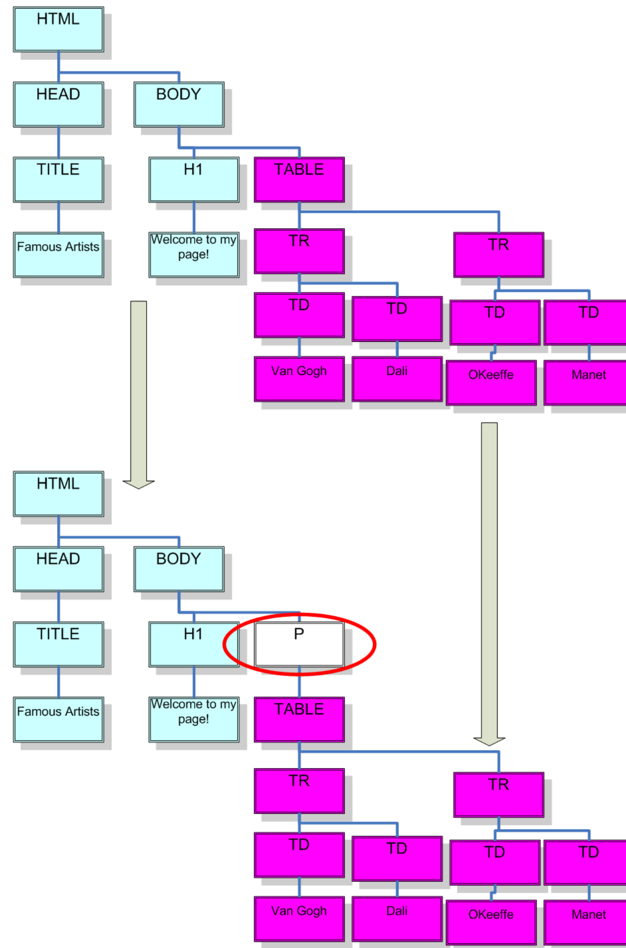

Figure 5.1: An example alignment between two HTML trees.

An alignment is a partial mapping between the nodes of one tree and the nodes of the other. For example, Figure 5.1 shows two HTML trees that, when aligned, only differ by the added **<P>** tag (circled in Figure 5.1). The shaded elements in both trees represent equivalent subtrees in both files, with arrows showing the mapping from a subtree in one tree to the subtree in the other. To see why this alignment is necessary, consider these two HTML fragments:

```
1  <u><b>textA</u></b>  <i><u><b>textB</b></u></i>
2  <i><b><u>textB</u></b></i>
```

Fragment alignment is determined before features can be counted: if the **textB** subtree of #2 is aligned with the **textA** subtree of #1, an inversion between the **<u>** and **<b>** tags can be counted.

However, if the `textB` subtree of #2 is aligned with the `textB` subtree of #1, inversions can be counted between the `<u>`, `<b>` and `<i>` tags. Consequently, the desired alignment is one that minimizes the number of changes that describe the difference between two documents. The DIFF-X [20] algorithm, with a quadratic runtime for calculating structural differences between XML documents, is adapted to compute alignments on general tree-structured data (including HTML files). By contrast, subgraph isomorphism, the problem where one must determine whether one given graph is a subgraph of another, is known to be NP-complete. Rather than using such a general and expensive approach for aligning XML/HTML trees, the DIFF-X algorithm was selected due to its quadratic runtime. DIFF-X generates a minimal edit script that describes the changes to transform one tree into another; the primitives for the edit script comprise both the node and subtree edit operations that are natural in this domain. Matching pairs of elements between the newer and older trees allows SMART to identify local features derived from element pairs, as well as global features, such as the addition of natural language text across elements in the document.

5.2.2 Modeling Structural Differences Between Pairs of Web Output

Once XML/HTML trees have been aligned, SMART can calculate feature values, based on the differences between aligned nodes. This section describes features based on the tree structure of the test case output that signal interesting changes that merit human inspection. Taken together, these tree-based features are meant to flag a wide assortment of differences identified between two XML or HTML files, based on the tree structure itself.

The DIFF-X **Algorithm.** Three of these features are taken from a variant of the DIFF-X [20] algorithm that was adapted to work on arbitrary tree-structured inputs rather than just XML. The algorithm computes the number of moves, inserts and deletes required to transform the first input tree into the second. It does this via bottom-up exact tree matching combined with top-down isolated tree fragment mapping; this amalgamated approach provides a high quality characterization of the relationship between the two input trees.

It is likely that moves, inserts, and especially deletes frequently correlate with bugs, and that the size of the change indicates the severity of the error. For example, a failure that results in a stack

trace being printed will involve a deletion of a large amount of data and an insertion of the trace itself. Considering moves instead of delete-insert pairs reduces the size of the changes between two trees.

Inversions. Similarly, it is likely that inverted elements in XML or HTML do not indicate high-level semantic errors. Two related types of inversions are considered. In both cases, a preorder traversal of all nodes in both of the document trees is performed. All text nodes are removed to consider only structural inversions. The two traversals are then sorted, and the longest common subsequence [68] between them is calculated. The longest common subsequence provides a specialized mapping between the two documents. All nodes not in the common subsequence are removed, and the lists are unsorted, returning the remaining nodes to their original relative orders. Finally, lists are compared element-wise and each difference, a structural inversion, is counted.

Grouped Changes. In addition to detecting changes to individual tree elements, a separate feature is considered when a set of elements that form a contiguous subtree are changed, as a group. The size of the grouped change in terms of the number of elements involved is measured, under the hypothesis that larger changes are more likely to indicate an error such as a stack trace, as opposed to smaller changes to natural language text on a single line, which are unlikely to warrant attention. Such clustered edits are more likely to deserve inspection, often because they contain missing components or lengthy exception reports. Grouped changes are also reported both as a boolean feature and as a weighted value corresponding to the size of the changes in the document. The boolean feature is meant to capture the presence or absence of any grouped change, while the weighted version provides a finer-grained analysis when necessary.

Depth of Changes. The relative depth of any edit operation within a tree is noted, under the assumption that changes closer to the root may be more likely to signal large semantic differences and thus more likely to merit human inspection.

Changes to Only Text Nodes. In many changes the differences between two outputs will be limited to text nodes while the tree structure remains unchanged. Documents with such text-only differences are less likely to contain semantic errors and thus likely to not be inspected as commonly. This feature is an important distinction from `diff`, which will flag all textual changes

as potential errors.

Order of Children. Two aligned nodes that are otherwise similar but have the order of their children changed are noted. Changes in the order of children (as opposed to changes in the order of attributes) may not indicate high-level semantic errors and thus could not be inspected. This feature is complementary to moves being associated with errors in test case output.

5.2.3 Modeling Human-Judgment Differences Between Pairs of Web Output

In addition to tree-based features, SMART also attempts to detect changes a human would discern between two rendered versions of output files. These features are specific to HTML and attempt to identify differences that a human observer would notice on a web browser; the goal is to approximate human judgments about differences in HTML outputs.

Text and Multimedia Ratios. Natural language and images play an important role in the human interpretation of a webpage [84]. The ratio of displayed text between two versions is measured. Similarly, the ratio of text to multimedia objects is calculated for each individual version, and then compared across both versions for a final ratio. Replacing a small amount of text with an image, such as replacing a textual link with a button, is not a large semantic difference. On the other hand, changing many words in a small document may merit inspection. Similarly, the ratio of new words to the number of old words in the file is recorded, under the hypothesis that large changes in natural language text are more likely to signal errors.

Error Keywords. Web-based applications often exhibit similar failure modes. Beyond the standard error messages displayed by web servers (such as 404 errors described in Section 2.1.2), many other violations are tied to the underlying languages, and can be reasonably predicted by a textual search of the document for error keywords, such as "exception" (see Chapter 7 and Chapter 8). Searching for natural language text to signal page errors has been previously explored in [27]. Output pairs containing error keywords in the newer version, but not in the older, are likely to merit human inspection.

Changes to Input Elements. Input elements, such as buttons and forms, represent the primary interface between consumers and the application. Changes to these input elements are noted under

the hypothesis that a missing button or form indicates a significant loss of functionality and likely requires examination.

Changed or Missing Attribute Values. When two aligned elements contain the same attribute but have different attribute values, an error may exist, potentially signaling the need for human inspection. Consider this example:

```
1 < <Type id="_8" name="int"/>
2 ---
3 > <Type id="_8" name="unsigned int"/>
```

If the two `<Type>` elements on lines 1 and 3 are aligned then the change from `"int"` to `"unsigned int"` represents a meaningful change. Note that this is different from an instance where the second `<Type>` has a new attribute that the original does not. Changed attributes may or may not be significant; consider the semantic difference between an update to a `height` attribute of an image as opposed to the mistyping of an `action` attribute of a form element. Removing attributes, however, is generally likely to merit human inspection.

5.3 Validating the Assumptions of the Model

The previous section presented structural and semantic features hypothesized to correlate with faults or non-faults in test case output. The goal of this section is to determine whether it is reasonable to use these tree-structured features to detect test case outputs that merit human inspection. In doing so, such an oracle comparator should at least out-perform other, more naïve approaches. As a baseline, the use of `diff` as an oracle comparator is verified to generate many false positives. Similarly, trends in feature values for faults versus non-faults are a prerequisite for the success of this approach.

5.3.1 Validation Setup

Ten open-source benchmarks that produced either XML or HTML output, totaling 473,000 lines of code, were used to evaluate oracle comparators in this section. Benchmarks were selected from an

Benchmark	Versions	LOC	Description	Test cases	Test cases to Inspect
HTMLTIDY	Jul'05 Oct'05	38K	W3C HTML validation	2402	25
LIBXML2	v2.3.5 v2.3.10	84K	XML parser	441	0
GCC-XML	Nov'05 Nov'07	20K	XML output for GCC	4111	875
CODE2WEB	v1.0 v1.1	23K	pretty printer	3	3
DOCBOOK	v1.72 v1.74	182K	document creation	7	5
FREEMARKER	v2.3.11 v2.3.13	69K	template engine	42	1
JSPPP	v0.5a v0.5.1a	10K	pretty printer	25	0
TEXT2HTML	v2.23 v2.51	6K	text converter	23	6
TXT2TAGS	v2.3 v2.4	26K	text converter	94	4
UMT	v0.8 v0.98	15K	UML transformations	6	0
Total		473K		7154	919

Figure 5.2: The benchmarks used in the experiments. The "Test cases" column gives the number of regression tests used for that project; the "Test cases to Inspect" column gives the number of those tests for which manual inspection indicated a possible bug.

assortment of domains, considering only benchmarks for which multiple versions were available and for which a set of test cases was available. Including multiple versions of the same project tests SMART's ability to ignore natural program evolutions. Figure 5.2 summarizes the programs used.

For each benchmark, the test case output generated by the two versions of the benchmark indicated was manually inspected. This inspection marked the output as "definitely not a bug" or "possibly a bug, merits human inspection", conservatively erring on the side of requiring human inspection. This initial experiment involved 7154 pairs of test case output, of which 919 were labeled as requiring inspection.

5.3.2 Validation Results

Of the 7154 test cases considered, `diff` flagged 4969, compared to the 919 flagged as potential errors by manual annotation; using `diff` as a comparator yielded four times as much wasted effort as potentially-required effort.

Features generally take on different values for differences computed by `diff` that merit human inspection and for differences that do not. Figure 5.3 shows the average normalized features' values for those two cases; the spread of values suggest that it should be possible to distinguish those cases

Feature	Average – No Inspect	Average – Inspect
Text Ratio	0.7996	0.9636
Grouped Boolean	0.0007	0.9767
Text Only	0.9946	0.0179
Grouped Change	0.0002	0.1301
Children Order	0.0010	0.1769
Inversions	0.0010	0.0016
Depth	0.0007	0.0172
DIFF-X-delete	0.0007	0.1203
DIFF-X-insert	0.0041	0.0109
Error Keywords	0.0000	0.0096
New Text	0.6197	0.9624
New Functionality	0.0000	0.0038
Missing Attribute	0.0047	0.1580
DIFF-X-move	0.0004	0.0507
Seen Elements	0.0000	0.0014
Changed Attribute	0.5244	0.9546

Figure 5.3: The average values of features for test cases flagged by `diff` that (1) do not merit manual inspection and (2) do merit manual inspection, as determined by human annotators. Each feature is individually normalized to 1.0.

using features outlined in Section 5.2.2. For example, the normalized feature value of text only changes is 0.9946 for test case output that need not be inspected and 0.0179 for test case output that should be inspected. Thus, the use of feature values to distinguish faulty from non-faulty XML/HTML test case output is demonstrated as a reasonable basis upon which to build an oracle comparator model.

5.4 Selecting a Model for Differences in Web-based Applications

Having shown features take on significantly different values for potentially erroneous versus correct output, it is possible to phrase the feasibility of using a highly-precise oracle comparator using these features as an information retrieval task. SMART creates a linear regression model based on those features and selects an optimal cutoff to form a binary classifier. Linear regression was chosen as a classifier due to its simplicity and the ease of which analyses can be performed. Although more sophisticated classification algorithms may have been able to more accurately classify faults and

non-faults, this linear regression approach yielded close to optimal results in the benchmarks used. The performance of such a classifier can then be measured in terms of the number of false positives and negatives returned for the benchmarks in Section 5.3.1.

5.4.1 Cross Validation

Linear regression is a type of supervised learning[1] algorithm; a potential threat to the validity of the results in this section is over-fitting by testing and training on the same data. To circumvent this, 10-fold cross validation [57] was employed. Test cases were randomly assigned from the data set into ten equally-sized groups. Each group was reserved once for testing, and the remaining nine groups were used to train the model; thereby never training and testing on the same data. Next the cross validation results were averaged and compared to the results of the same model when trained and tested on the entire data set. If the two outcomes were not significantly different, it can be reasoned that testing and training on the same data did not introduce a bias.

5.4.2 Model Selection

Although Section 5.3.1 demonstrated that features take on different values for test cases that contain faults and those that do not, such observations should not be hard-coded into an oracle comparator that is meant to be robust. Instead, SMART, the highly-precise oracle comparator proposed in this work, uses machine learning to classify pairs of tree-structured outputs based on whether a human should inspect them or not. To test the validity and feasibility of such an approach, the following experiment was conducted on the dataset from Figure 5.2:

1. The cross-validation steps (Section 5.4.1) are first performed. On each fold, a linear model is trained as if the response variable (i.e., the boolean human annotation of whether a human should inspect that output or not) were continuous in the range [0,1].

2. The real-valued model outputs are turned into a binary classifier by comparing against a cutoff. A linear search is utilized to find a model cutoff. Depending on how the result

[1]The term learning is used in this dissertation to refer to a technique for using training data to set the parameters of a model [49].

Comparator	F_1-score	Precision	Recall
SMART	0.9931	0.9972	0.9890
SMART w/ cross-validation	0.9935	0.9951	0.9920
`diff`	0.3004	0.1767	1.0000
`xmldiff`	0.2406	0.1368	1.0000
fair coin toss	0.2045	0.1286	0.4984
biased coin toss	0.2268	0.1300	0.8868

Figure 5.4: The F_1-score, precision, and recall values for SMART on the entire dataset. Results for `diff`, `xmldiff`, and random approaches are given as baselines; `diff` represents current industrial practice.

of applying the linear model compares to the cutoff, the oracle comparator reports that the outputs need be or need not be inspected. Those cutoff values are chosen that yield the highest F_1-score for each validation step (see Section 2.2.3).

3. After cross-validation, the model is trained on the entire data set. The best model cutoff, to maximize the F_1-score, is once again selected.

Figure 5.4 shows the precision, recall, and F_1-score values for the dataset from Section 5.3.1. As a point of comparison, the predictive power of `diff`, `xmldiff` [4], coin toss, and biased coin toss were computed as baseline values. The fair coin returns "no" with even probability. The biased coin returns "no" with probability equal to the actual underlying distribution for this dataset: $(7154-919)/7154$. Note that the biased coin toss cannot generally be implemented in the field since it relies on knowing the distribution of right answers in advance. Despite this, SMART has clear advantages in predictive power over `diff`, `xmldiff`, and random or biased chance; yielding three times `diff`'s F_1-score. `xmldiff` is an off-the-shelf `diff`-like tool for XML and HTML [4], and is able to ignore features such as whitespace, namespaces, and case in text elements when comparing two XML/HTML files. `xmldiff` was, however, a worse comparator than basic `diff` because it was unable to parse several files due to natural language text (usually warnings such as those generated by HTMLTIDY) at the top or elsewhere in the file, producing even more false positives. `diff` and `xmldiff` were chosen instead of the *Struct* oracle comparator of Sprenkle *et*

Feature	Coefficient	F	p
Text Only	- 0.288	168970	< 0.001
DIFF-X-move	+ 0.002	150840	< 0.001
DIFF-X-delete	+ 0.029	46062	< 0.001
Grouped Boolean	+ 0.714	7804	< 0.001
DIFF-X-insert	+ 0.029	4761	< 0.001
Grouped Change	- 0.012	465	< 0.001
Children Order	- 0.002	317	< 0.001
Inversions	+ 0.001	246	0.020
Missing Attribute	- 0.048	121	< 0.001
Error Keywords	+ 0.174	115	< 0.001
Depth	- 0.000	21	< 0.001
Text Ratios	- 0.007	18	< 0.001
Input Elements	- 0.019	5	0.03

Figure 5.5: Analysis of variance of the oracle comparator model. A + in the 'Coefficient' column means high values of that feature correlate with test cases outputs that should be inspected. The higher the value in the 'F' column, the more the feature affects the model. The 'p' column gives the significance level of F; features with no significant main effect ($p > 0.05$) are not shown.

al. [101] to avoid false negatives.

Little to no bias was revealed by cross-validation. The absolute difference in F_1-score between the model and its corresponding averaged cross validation steps was 0.0004. This shows that results obtained using the corresponding model trained on the entire data set were never significantly different from the averaged results from each set of cross validation steps. Consequently, it seems as though web-based applications tend to fail and evolve in predictable ways; cross-validation is able to demonstrate that regardless of the training and testing data subsets, SMART was able to accurately classify potential faults and non-faults. In the following section, this observation is investigated more deeply by an analysis of the contribution of various features to SMART's performance.

In this section, relative feature importance is evaluated, including which features correlate with output that should be inspected. Figure 5.5 shows the results of a per-feature analysis of variance [74] on the model using the entire dataset. F denotes the F-ratio, which is close to 1 if the feature does not affect the model; conceptually F represents the square root of variance explained by that feature over variance not explained. The p column denotes the significance level of F (i.e., the probability that the feature does not affect the model). The table lists only those features with

a significant main effect: the standard association of statistical significance when $p \leq 0.05$ [42] is adopted. The *Coefficient* column indicates the how strongly each feature correlated with either faults (a positive value) or non-faults (a negative value).

The most significant feature was whether or not the change involved only low-level text. This is the key distinction between the feature-based oracle comparator in this chapter for tree-structured output and the state-of-the-art for normal textual output: in normal practice, changes to the text of the output indicate regression errors. However, text-only changes have a strong negative effect: test case outputs that differ only in small amounts of non-structural text do *not* merit human inspection. This is one of the key reasons SMART is able to outperform `diff`.

The DIFF-X-move feature was frequently correlated with test case errors. It may seem counter-intuitive that moves, as opposed to insertions or deletions, would indicate a need for human inspection. In practice, however, tree-structured moves show up as a side-effect of other large changes; the introduction or deletion of one element often involves a move of its neighbors. Rather than relying on only insertions or deletions, moves are able to capture both of these types of changes as a move almost always occurs in conjunction with any of the other two edits. Despite the high F-ratio of the DIFF-X-move feature, its model coefficient was an order of magnitude smaller than those of insert or delete. Although moves were most frequently associated with errors, SMART requires that other features also had to be present in order for the test case output to merit inspection.

The boolean feature that indicates the presence or absence of clustered changes was also highly correlated with errors. Variations in the sizes of the grouped changes are not as salient as their existence. Grouped changes were more important than DIFF-X-inserts, which may have been scattered across the output.

Some features were less powerful than originally hypothesized. For example, the presence of error keywords did not effect the model as much as the features listed above. This issue is further explored in Chapter 9.

Benchmark	Comparator	F_1	Precision	Recall
HTMLTIDY	SMART	1.000	1.000	1.000
	`diff`	0.048	0.025	1.000
	`xmldiff`	0.021	0.010	1.000
GCC-XML	SMART	0.999	1.000	0.999
	`diff`	0.352	0.213	1.000
	`xmldiff`	0.352	0.213	1.000
All ten (global)	SMART	0.993	0.997	0.989
	`diff`	0.300	0.177	1.000
	`xmldiff`	0.241	0.138	1.000

Figure 5.6: F_1-score, precision, and recall when trained and tested on individual projects, as well as all ten benchmarks. Results for `diff` and `xmldiff` are presented as baselines.

5.5 Evaluating the Oracle Comparator

As the previous section demonstrated, a highly-precise oracle comparator can be constructed through a linear regression approach that uses surface features of XML/HTML output to determine whether or not an output pair should be examined. Such features are useful in distinguishing correct from erroneous output because their values vary for these two classes of results, as shown in Section 5.3.1. Such an oracle comparator provides near-perfect precision and recall for the dataset in the previous section, and significantly outperforms `diff` and other baselines. In this section, this highly-precise oracle comparator, SMART, is further evaluated, on a per-project basis, as well as in a effort saving scenario.

5.5.1 Per-Project Oracle Comparators

Section 5.4.2 presented an oracle comparator that was tested and trained on XML/HTML output from ten benchmarks. In this section, the same oracle comparator is evaluated when testing and training on data from a single project; training data from each benchmark application is used to tailor multiple oracle comparators corresponding to each individual project. The performance of these per-project models is compared to the global model (the oracle comparator built from the entire dataset) in Figure 5.6.

Experimental Procedure

The same experimental setup was used for the per-project classifiers as that described in Section 5.3.1. In total 6513 test case output pairs for GCC-XML and HTMLTIDY were employed, because those two benchmarks were large enough to feasibly admit for individual study. Figure 5.6 shows the average F_1-score, precision and recall values when individual oracle comparators were trained and tested on each program separately. As in Section 5.4.1, cross validation revealed little to no bias for the per-project classifiers.

Results

For HTMLTIDY, SMART obtained perfect performance, with no false positives or false negatives. The precision score for this project is thus an order of magnitude better than that of `diff`: SMART presents only 25 test case outputs to developers compared to the 960 flagged by `diff`.

For GCC-XML near-perfect recall (0.999) and perfect precision were achieved; 874 test cases were flagged by SMART for human inspection, compared to `diff`'s 4100, with the exception of one test case that did merit human inspection that the oracle comparator failed to flag.

Project-specific feature weights contributed to strong per-project performance. For example, the DIFF-X-delete and DIFF-X-insert features were equally important for the HTMLTIDY project, but not across all benchmarks. As another example, DIFF-X-insert and error keywords were significantly associated with errors in HTMLTIDY but not at all in GCC-XML. Although an effective model can be trained using the data from a single project, a more effective model can be achieved by using per-project information. However, the effects of various features across projects remained relatively constant; this observation will be explored further in the next chapter.

5.5.2 Measuring Effort Saved

Intuitively, using SMART for either of the projects alone would seem advantageous. When the same global model is applied to all ten benchmarks, however, recall suffers: some actual regression test errors are not flagged for human inspection. In such cases the oracle comparator is a net win if it

Benchmark	Release	Test Cases	Should Inspect	True Positive		False Positives		False Negatives		Ratio
				SMART	diff	SMART	diff	SMART	diff	
HTMLTIDY	2nd	2402	12	5	12	78	781	7	0	0.0099
	3rd	2402	48	48	48	0	782	0	0	0
	4th	2402	254	109	254	1	574	145	0	0.2019
	5th	2402	48	48	48	0	775	0	0	0
	6th	2402	20	19	20	1	774	1	0	0.0013
GCC-XML	2nd	4111	662	658	662	16	2258	4	0	0.0018
	3rd	4111	544	544	544	0	2577	0	0	0
total		20232	1588	1431	1588	96	8521	157	0	0.0183

Figure 5.7: Simulated performance of SMART on 20232 test cases from multiple releases of two projects. The 'Test Cases' column gives the total number of regression tests per release. The 'Should Inspect' column counts the number of those tests that manual annotation indicated should be inspected (i.e., might indicate a bug). The 'Inspected' column gives the number of tests that SMART and `diff` flag for inspection. The 'False Positives' and 'False Negatives' columns measure accuracy, and the 'Ratio' column indicates the value of *LookCost*/*MissCost* above which SMART becomes profitable (lower values are better).

costs more to triage, inspect and resolve the smaller number of test outputs than it does to miss a few test cases that merit human inspection.

Although SMART has near-perfect precision and recall in the experiments presented so far, the model will generally not be tested and trained on the same data, nor will humans be willing to annotate 90% of their test case output as in the cross validation steps. In this section, a more realistic scenario is explored, where developers are responsible for training SMART on 20% of the output from each run of the test suite, which they manually annotate, and testing on the remaining 80% percent. Therefore, the hypothetical benefit of SMART is evaluated in terms of developer effort saved, when used to determine if humans should inspect regression test output over multiple revisions to software projects. A situation in which a development organization uses this technique on all regression tests between successive releases of the same project is considered. It is assumed that humans manually inspect a small percentage of the test case output flagged by `diff` — 20% in this experiment — and then train the oracle comparator on that information (as in Section 5.3.1), using it to guide the inspection of the remaining test cases. Subsequent releases of the same project retain training information from previous releases, as well as incorporate the false positive or true

positive results of any test case that the tool deemed to require manual inspection.

Experimental Procedure

Two benchmarks, GCC-XML and HTMLTIDY, had three or more released versions available. For each successive version considered, test case differences were manually annotated to determine if a human should inspect them (see Section 5.3.1). The dataset for this simulation thus includes 20232 regression test output pairs spanning seven software releases for two projects. Note that this is a slightly different setup than that of Figure 5.2 — for example, while there were 25 test cases to inspect for the version of HTMLTIDY used in Figure 5.4, here five different releases of HTMLTIDY are used that have correspondingly different numbers of test outputs that should be inspected (12–254).

The number of test cases flagged for manual inspection is measured (i.e., the 20% used for initial training as well as the true positives and false positives produced by the oracle comparator) as well as the number of false negative test cases that should have been flagged for inspection (i.e., that indicated potential bugs found via regression testing) but were not. Each of these carries an associated software engineering cost.

The amount of effort saved by developers can be estimated when using this oracle comparator, by defining a cost of looking (*LookCost*) at a test case and a cost of missing (*MissCost*) for each test case that should have been flagged but was not. A useful investment occurs when the cost of using the oracle comparator:

$$(\mathit{TruePos} + \mathit{FalsePos}) \times \mathit{LookCost} + \mathit{FalseNeg} \times \mathit{MissCost}$$

is less than the cost of $|\texttt{diff}| \times \mathit{LookCost}$. That is, this approach saves effort when the cost of looking at the test cases flagged by `diff` but not by this technique exceeds the cost of missing any relevant test cases not reported. The condition under which this technique is profitable is then:

$$\frac{\mathit{LookCost}}{\mathit{MissCost}} > \frac{-\mathit{FalseNeg}}{\mathit{TruePos} + \mathit{FalsePos} - |\texttt{diff}|}$$

It is assumed *LookCost* $\ll$ *MissCost* [118], so the goal is for this ratio to be as small as possible.

Results

Figure 5.7 shows the results of this experiment. For example, when applying the oracle comparator to the last release of HTMLTIDY, the ratio above which it becomes profitable is about 1/1000; if the cost of missing a potentially useful regression test report is less than or equal to 1000 times the cost of triaging and inspecting a test case, developer effort is saved. A ratio of 0 indicates that there are no false negatives, and in such cases this approach always outperforms `diff`, regardless of *LookCost* or *MissCost*. Figure 5.7 also shows the number of test cases that a developer would need to examine when using `diff`.

SMART's performance generally improves on subsequent releases, and it totally avoids false negatives in one instance for both benchmarks. It is at its worst when there is a large relative number of regression test errors (e.g., for a rushed release that fails to retain required functionality). For the fourth release of HTMLTIDY, the number of test cases that should be inspected is an order-of-magnitude higher than usual. A cautious development organization might use this tool only when the manual annotation of 20% of the test case outputs shows a historically reasonable number of regression test errors.

Previous work on bug report triage has used a *LookCost* to *MissCost* ratio of 0.023 as a metric for success for an analysis that required 30 days to operate [54], and that ratio is adopted as a baseline here. The typical performance of the oracle comparator, which includes the cost of the 20% manual annotation burden and would take 1.3 hours on average per release, is 0.0183 — a 20% improvement over that figure. If the HTMLTIDY outlier mentioned above is excluded, the ratio is 0.0015; the utility of previous tools is exceeded by an order of magnitude and this approach requires an order of magnitude less time.

In general, *LookCost*, the time to compare the results of a test suite to the oracle, is typically a few minutes for each test case [78]. *MissCost* varies by application domain: it can be low for applications where servers can easily update deployed software, but is often high for web applications with high quality-of-service requirements [120]. An IBM publication provides concrete examples

of industrial values for these costs: based on a 2008 report [118], *LookCost* is $25 and *MissCost* is $450 (the cost of a defect "during the QA/testing phase"). With those cost figures, using SMART reduces the costs associated with regression testing over all releases shown in Figure 5.7 by 48% ($131730 vs. $252725). Even if *MissCost* doubles to $1000, this technique still reduces the costs by 22% ($195175 vs. $252725).

5.6 Threats to Validity

Although SMART outperforms `diff` by over a factor of three, it is possible that these results do not generalize to industry practice for various reasons. For example, the benchmarks used in these experiments may not be representative of other projects. To mitigate this threat, the two large benchmarks (HTMLTIDY and GCC-XML) were selected from different domains, and the global dataset was supplemented with other, smaller benchmarks to increase the diversity of the data used for testing. It is possible that some results are more indicative for the two larger benchmarks than the smaller ones, however; Chapter 6 and Chapter 9 evaluate SMART on more test applications. Even if the benchmarks are representative, it is possible to overfit the model to the data; cross-validation steps in Section 5.4.1 suggests that is not the case.

Similarly, the results may not generalize if the oracle comparators are not used as developers would in practice: the regression test output of versions of HTMLTIDY and GCC-XML that were several months apart were examined, but in practice some organizations may perform these tests more frequently, such as during a nightly build. Because the model's performance depends on the relative frequency of bugs between release versions, and not the number of bugs in general, this approach should still be able to save developer effort, even if regression tests are run more frequently.

Finally, human annotations may not have accurately flagged potential errors in regression test output. To avoid missing actual bugs, these annotations were conservative: only test outputs as not meriting inspection were annotated if humans were highly certain they did *not* indicate an error. Thus non-errors may have been annotated as errors, which translates into less of an opportunity

to outperform `diff`, but does not impact the correctness of the approach. In addition, because annotators were also responsible for suggesting some features for the model, it is possible that bias exists in the annotations themselves, although care was taken to avoid this situation and baseline annotations were computed at least twice on each output pair to guarantee a minimum level of consistency. Section 9.3 presents an experiment where SMART is tested on a set of manually-injected known faults, as opposed to the *potential* faults used in this chapter, which avoids such a complication of humans mis-labeling non-faults as faults.

5.7 Experimental Summary

In this chapter syntactic and structural features have been used to build a model that classifies which regression test case outputs merit human inspection based on a machine learning approach. On 7154 test cases from 10 projects, this highly-precise oracle comparator obtains a precision of 0.9972, a recall of 0.9890, and an F_1-score of 0.9931, more than three times better than `diff`'s F_1-score of 0.3004. Although these are very strong machine learning results, the technique was further tested in a simulated deployment involving 20232 test cases and multiple releases of two projects. In that scenario there were 8425 fewer false positives than `diff`, and development effort is saved when the ratio of the cost of inspecting a test case to the cost of missing a relevant report is over 0.0183; numbers in that range correspond to savings for typical industrial practice.

5.8 Related Work to Error Detection in Web-based Applications

There is currently no industry standard for comparing pairs of XML/HTML documents beyond that of `diff` used in capture-replay contexts and user-session based testing [41]. Developers have the option of customizing `diff`-like comparators for their target applications, such as using regular expressions to filter out conflicting dates, but these tools must be manually configured for each application and potentially each test case, and may not be robust as the website evolves.

Sprenkle *et al.* focus on reasonably-precise oracle comparators for testing web applications [101, 103, 104] with HTML output. Building on a capture-replay testing framework for user

session data [101], they investigate features based on `diff`, content, and structure. They refine these features into oracle comparators [104] based on HTML tags, unordered links, tag names, attributes, forms, the document, and content. They then investigate applying decision tree learning to identify the best combination of oracle comparators for specific applications [103]. SMART also combines machine learning and oracle comparators, but additionally includes features and experiments that are not HTML-specific and can be applied to any tree-structured data. Finally, they validate their approach by measuring their oracle comparators' abilities to reveal seeded defects in a single version of an application (i.e., measuring differences between the clean application and a fault-seeded one). By contrast, the experiments in this chapter train and test on data between *different versions* of the same application. The approach detailed in this work aims to not flag common and benign program evolutions, in contrast to the setting of Sprenkle *et al.* [103], where a application with deterministic output would yield no false positives with a `diff` comparator.

Many testing methodologies use oracle comparators that require manual intervention in the presence of discrepancies [39, 64, 86, 104]. For example, user session data can be used as both input and also test cases [39, 101], but there must be a way to compare obtained results with expected results. Lucca *et al.* address web application testing with an object-oriented web application model [64]. They outline a comparator which automatically compares the actual results against the expected values of the test execution. SMART can be thought of as a working instantiation of such a design, extending the notion to structural differences.

Sneed explores a case study on testing a web application system for the Austrian Chamber of Commerce [99]. A capture-replay tool was used to record the dialog tests, and XML documents produced by the server were compared at the element level: if the elements did not match, the test failed. The feature-based comparator in this chapter also compares XML documents, but does not necessarily rely on exact element matching, and thus reports fewer false positives.

Meszaros describes experiences using capture-replay scripts for agile regression testing [73]. The focus is on "fragile" test cases where a "robot user" fails for seemingly trivial reasons. Two parts of the fragile test problem are interface sensitivity, where "seemingly minor changes to the interface can cause tests to fail even though a human user would say the test should still pass", and

context sensitivity, such as the changing of the date and time of the test. The oracle comparator in this chapter solves the fragile test problem for many test cases, allowing such tests to be used in later versions.

Binkley [29, 30] as well as Vokolos and Frankl [114] approach regression testing by characterizing the semantic differences between two versions of program *source code* using program slicing. By doing so, only program differences between two versions needed to be tested and the total number of test cases that need to be executed between versions were reduced. The approach in this chapter focuses on the semantic differences between two versions of the program *output*; the technique is orthogonal to theirs, and the oracle comparator described in this work can be used in a retest-all framework, or in conjunction with their approach, in a setting in which some regression tests have been skipped due to source code similarity.

5.9 Summary

This chapter presented SMART, a highly-precise oracle comparator, for reducing the cost of regression testing by using syntactic and structural features to decide whether or not test case output merits human inspection. In domains with tree-structured output, such as HTML, XML or abstract syntax trees, traditional `diff`-based comparisons yield too many false alarms. A number of features were suggested that can be used to distinguish potential errors from harmless functionality additions or rendering changes. For example, changes to the text of HTML output are negatively correlated with the presence of potential errors, while tree-structured differences, such as moving a subtree from one part of the output to another, are positively correlated with potential errors.

SMART was evaluated both as a model and as a cost-saving technique. As a model evaluated on 7154 test case pairs from 10 projects, a precision of 0.9972, a recall of 0.9890 and an F_1-score of 0.9931 was obtained, over three times as good as the standard `diff` F_1-score of 0.3004. These strong machine learning results were complemented with a simulated deployment involving 20232 test cases. SMART had only 96 false positives — 8425 fewer than `diff` — and saved development effort when the ratio of the cost of inspecting a test case to the cost of missing a relevant report

was over 0.0183, a range both corresponding to a savings for typical industrial practice and also 20% better than previously-published results. Web applications and XML-processing middleware applications are becoming increasingly important; SMART presents a first step toward an oracle comparator that makes regression testing for them attractive.

The main thesis of this dissertation is that user-visible web-based application errors have special properties that can be exploited to improve the current state of web application error detection, testing and development. This chapter demonstrated that errors in web-based applications can be modeled using the tree-structure nature of XML/HTML output, and that such a model reduces the costs associated with regression testing in this domain. Highly-precise oracle comparators were trained on manually-annotated data from the application being tested. The next chapter will explore using training data for such oracle comparators from unrelated web-based applications.

Chapter 6 Automating Error Detection During Regression Testing

The previous chapter introduced a partially-automated highly-precise oracle comparator that showed significant savings over more naïve `diff`-like approaches when comparing test case output for error detection, but required training the oracle comparator on a set of manually-annotated output from the application being tested. This chapter will focus on expanding the concepts from the previous chapter, by outlining techniques for *fully automating*[1] such an oracle comparator. Specifically, Hypothesis (H2), that

> *a highly-precise,* fully-automatic *oracle comparator for web-based application testing can be constructed, based on pre-existing information from unrelated applications, that has fewer false positives than off-the-shelf techniques such as* `diff` *and* `xmldiff` *while maintaining the ratio of the cost of examining a potential bug to the cost of missing an actual bug at or below a current state-of-the-art value of 0.023 [54],* (H2)

is tested. The main thesis of this dissertation is that user-visible web-based application errors have special properties that can be used to improve the current state of web application error detection, testing and development. Under such an observation, this chapter shows how exploiting similarities across seemingly unrelated web-based applications obviates the need to provide manually-annotated training data to an oracle comparator that relies on machine learning.

[1] Although a human will still have to inspect test case pairs labeled as errors with the approach in this chapter, the phrase *fully-automatic* is used throughout this dissertation to contrast with mechanical processes that involve much more human activity. Specifically, a fully-automatic oracle comparator in this work is defined as one that has the following properties: 1) it does not require the user to manually annotate output to serve as training data, 2) it does not require customizing the tool by training it on manually seeded faults, 3) it does not require the user to specify parts of HTML output files to ignore (as when manually customizing a `diff`-like tool), and 4) it does not require the manual configurations as in [103]

6.1 Challenges in Automatic Regression Testing of Web-based Applications

Despite the ubiquitous use of web applications, most are not developed according to a formal process model [82]. As Chapter 1 discussed, web applications are subject to high levels of complexity and pressure to change. Regression testing is an established approach for increasing the reliability of applications in the face of recurring updates. Unfortunately, a general want of resources [51, 71, 115], combined with the additional complexities of web applications [86], makes automation a necessity if regression testing is to be adopted in the web-based application domain. Although automated replay of existing web-based application test suites is relatively straightforward, regression testing is constrained by the effort required to compare test results between two program versions (see Section 3.2.2).

6.2 Fully Automating Regression Testing of Web-based Applications

This chapter explores the idea that other web-based application output, unrelated to the application-at-test, can be used to train an oracle comparator to recognize error situations. Though the corpus of training data is not related to the application-at-test, the types of and manifestations of faults in such applications are often similar in nature, making it possible to build a general predictive fault model. By shipping this set of training data to the oracle comparator described in the previous chapter, the comparator process can be automated by not relying on any additional form of manual annotation (as in Chapter 5) or manual fault seeding (as in [103]).

Such an approach focuses not only on the similar ways unrelated web applications fail, but also on the equally important ways in which they tend to benignly evolve. Ignoring harmless program evolutions is central to reducing the number of false positives associated with any oracle comparator in this domain. Conversely, correctly modeling truly erroneous output allows an oracle comparator to minimize false negatives.

6.3 Evaluating Automated Oracle Comparators

This section experimentally tests the hypothesis that web-based application similarities can be exploited to aid in the automation of various aspects of testing web-based applications. Specifically, training a model on one set of data, unrelated to the application-at-test, results in successfully testing the oracle comparator, SMART, on a separate web-based application. The underlying similarities between these two sets of training and testing output make this approach feasible.

6.3.1 Experimental Setup

Since the approach for modeling output differences involves supervised learning, SMART is first trained before testing. Recall from the previous chapter that the model takes a weighted sum of feature values for a pair of test case outputs and indicates that they should be inspected if the sum exceeds a certain cutoff; the weights and the cutoff are determined on a per-project basis using linear regression. In this chapter, training the model requires a set of test case output pairs with known labels (i.e., annotations indicating whether the pair should be inspected or not). One option is to annotate a subset of test cases from the application-at-test to be used as training data (as in the previous chapter), but this practice has the disadvantage of requiring human effort. Instead, pairs of test case output from *unrelated*, publicly-available applications are used as training data for the model. This has the advantage of not requiring new manual annotations of test case output, with the potential drawback of not being as effective as training data tailored to the application-at-test. Experimental results in Section 6.3.2 show, however, that very high levels of accuracy can be achieved using this approach, due to the underlying similarities between web-based applications and the ways in which they fail.

The corpus of training data for the experiments in this chapter are the same pairs of annotated test case output from Section 5.3.1, summarized in Figure 5.2. As an option, a developer using SMART may also add their own test cases from previous projects to the set of training data, assuming that they have already annotated those output pairs, but in general any additional developer annotations are not required.

Benchmark	Versions	LOC	Description	Test cases	Test cases to Inspect
HTMLTIDY	Jul'05 Oct'05	38K	W3C HTML validation	2402	25
GCC-XML	Nov'05 Nov'07	20K	XML output for GCC	4111	875
VQWIKI	2.8-beta 2.8-RC1	39K	wiki web application	135	34
CLICK	1.5-RC2 1.5-RC3	11K	JEE web application	80	7
Total		108K		6728	941

Figure 6.1: The benchmarks used as **test data** for Experiment 1. The "Test cases" column gives the number of regression tests used; the "Test cases to Inspect" column counts those tests for which the manual inspection indicated a possible bug. When testing on HTMLTIDY or GCC-XML, it is removed from the training set.

This automated version of SMART was evaluated by examining its performance in terms of false positives and false negatives. Four benchmarks, shown in Figure 6.1, were selected to serve as test data, while the applications from the previous chapter in Figure 5.2 were reserved as training data. Although ten benchmarks were used as the training corpus, only two of them (HTMLTIDY and GCC-XML) had enough test case output pairs that were labeled as faults (given by the "Test Cases to Inspect" column) to serve as *testing* (as opposed to training) subjects. Two open source web applications (CLICK and VQWIKI) were also chosen to supplement test benchmarks in a "worst-case scenario" fashion: none of the training benchmarks are considered typical web applications, so successful performance on them further supports the hypothesis of inherent web-based application similarities.

VQWIKI [18] is wiki server software that can be used out-of-the-box as a web application. CLICK [10] is a Java Enterprise Edition web application framework that ships with a sample web application demonstrating the framework's features. The other two testing benchmarks, HTMLTIDY and GCC-XML, are open-source XML-based applications that are also a part of the training benchmarks. For these two applications, each benchmark's respective test case outputs were removed from the corpus of training data, making it impossible to test and train on the same data. Therefore, the training data for CLICK and VQWIKI were the test case output pairs from the ten benchmarks in Figure 5.2, while GCC-XML and HTMLTIDY were trained with the nine remaining benchmarks from Figure 5.2, when the test cases for each respective application were removed from the training

data set. In total the model was tested on 6728 test case pairs, 941 of which were labeled as errors by manual inspection (see Figure 6.1).

6.3.2 Experimental Results

This section presents empirical measurements of SMART's predictive power at detecting faults between test case output pairs (analogous to Section 5.5). Figure 6.2 shows the model's F_1-score values for each test benchmark. A score of 1 indicates perfect performance. Also included are F_1-score values for unbiased and biased coin toss, standard `diff`, and `xmldiff` [4], an off-the-shelf `diff`-like comparator for XML and HTML (see Section 5.4.2). The unbiased coin toss returns "inspect" with a probability of 0.5, while the biased coin toss returns "inspect" with the dataset's actual underlying ratio: $(6728 - 941)/6728$ (note that it is not possible to know this ratio *a priori* in the field).

SMART is anywhere from over 2.5 to almost 50 times as good as `diff` at correctly labeling test case outputs, with similar improvements over `xmldiff`. For the two web applications, the oracle comparator achieves perfect precision and recall — an optimal result. The scores for the large XML benchmark, HTMLTIDY, are also close to perfect (an F_1-score of 0.98). Overall, using test case output pairs from unrelated web-based applications to train a model to predict errors in the application-at-test is a successful approach. The underlying similarities between web-based applications in general make this possible. An analysis of variance [74] revealed that features associated with text-only changes were strongly negatively associated with errors in most benchmarks. By employing an available model and training set combination such as this, developers would be able to significantly reduce the number of false positive test case output pairs they must inspect, without requiring annotations or additional human effort to train the model.

Figure 6.4 shows SMART's precision scores for each benchmark, as well as baseline comparators, highlighting the model's predictive power over `diff`-like comparators. Figure 6.3 presents the model's recall scores, where it is challenged by `diff` in that the latter will always be able to return all true positive errors. For the two web applications (VQWIKI and CLICK), the oracle comparator is equally as good as `diff` at returning error cases, while for HTMLTIDY its score is competitive.

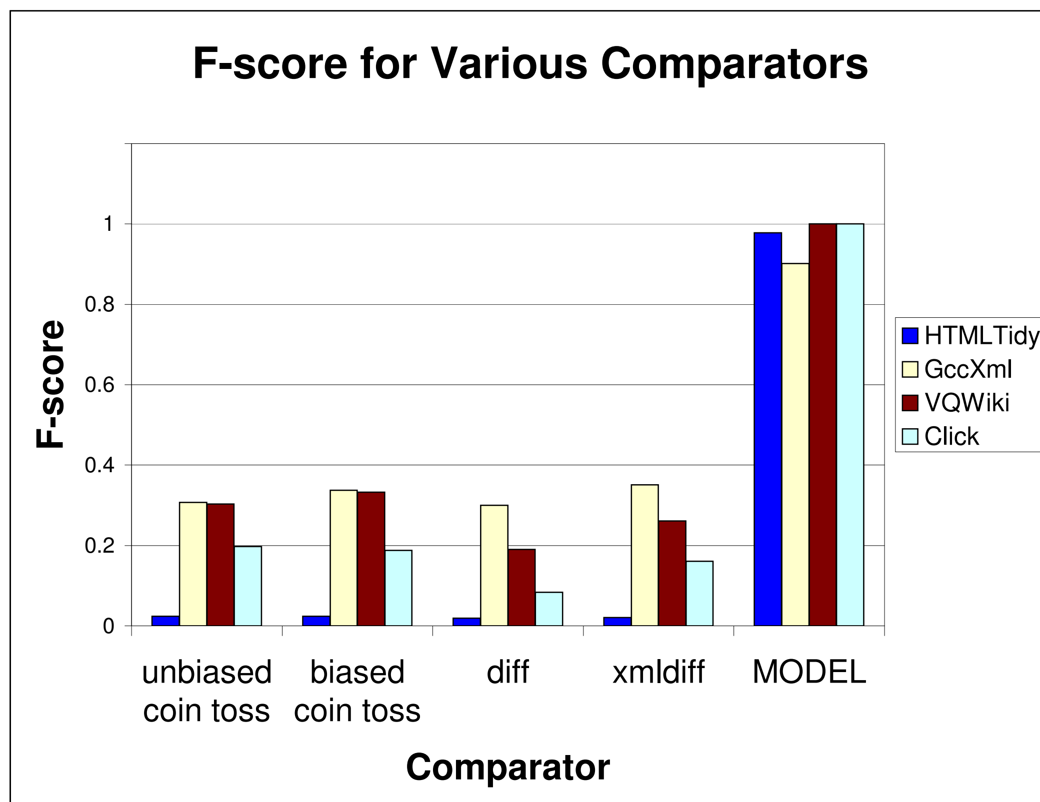

Figure 6.2: F_1-score on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK) using the Model, and other baseline comparators. 1.0 is a perfect score: no false positives or false negatives.

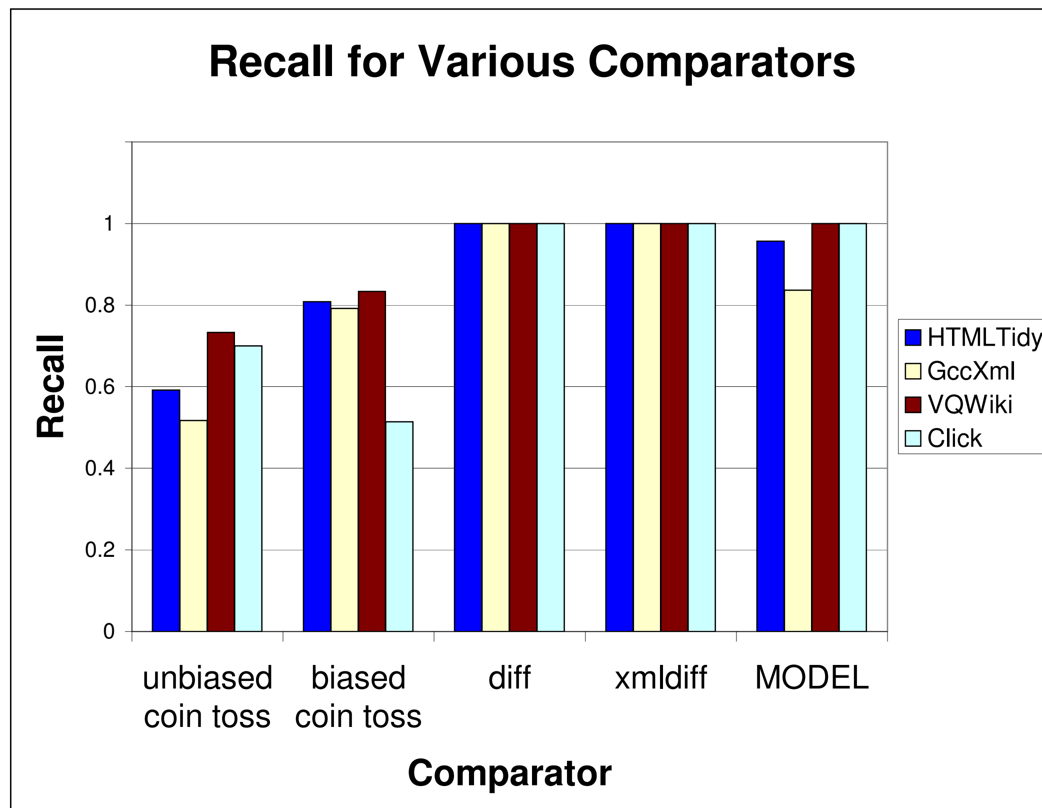

Figure 6.3: Recall on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK) using the Model, and other baseline comparators.

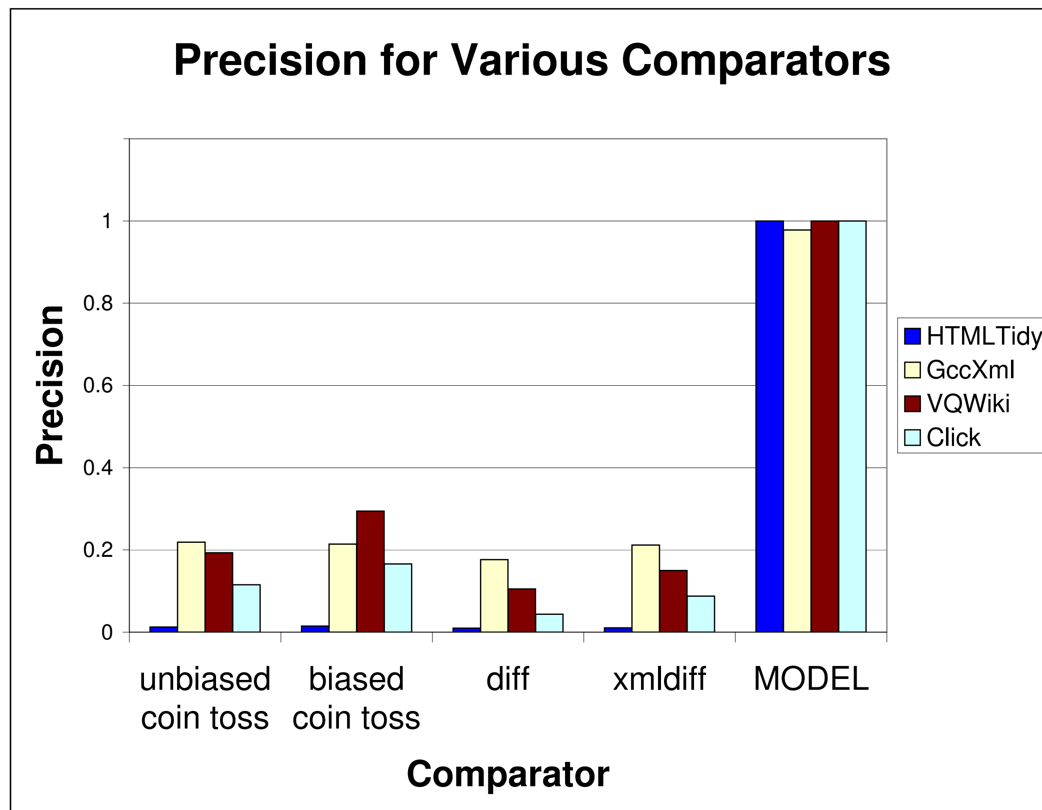

Figure 6.4: Precision on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK) using the Model, and other baseline comparators.

The savings using this approach can be estimated by defining the cost of looking at a test case (*LookCost*) and the cost of missing a bug (*MissCost*). Recall from the previous chapter that the approach is advantageous when its associated cost:

$$(\mathit{TruePos} + \mathit{FalsePos}) \times \mathit{LookCost} + \mathit{FalseNeg} \times \mathit{MissCost}$$

is less than the cost of current industrial practice of $|\texttt{diff}| \times \mathit{LookCost}$. Developers will save effort when the cost of examining false positives flagged by `diff`, but not SMART, is greater than the cost of missing any relevant test cases with the oracle comparator:

$$\frac{\mathit{LookCost}}{\mathit{MissCost}} > \frac{-\mathit{FalseNeg}}{\mathit{TruePos} + \mathit{FalsePos} - |\texttt{diff}|}$$

It is assumed $\mathit{LookCost} \ll \mathit{MissCost}$ [118], so the goal is for this ratio to be as small as possible. For the two web applications, the perfect F_1-scores imply savings are always produced with respect to `diff`: 75% and 96% of the test case pairs reported as errors by `diff` were false positives for CLICK and VQWIKI respectively, and SMART eliminates the need to check any of these, while simultaneously correctly flagging all potential errors. For HTMLTIDY, savings over `diff` occur if the ratio of *LookCost* to *MissCost* is at least 0.0004 (in other words, if the cost of missing a bug is no more than 2500 times the cost of looking at a report). SMART's performance is significantly better than the 0.0015 ratio of partially-automated work in the previous chapter, due in part to not incurring a penalty for the *LookCost* associated with the manual annotation of the training data in the previous chapter.

While the F_1-score for the other XML benchmark, GCC-XML, was three times better than that of `diff`, its recall score of 0.84 implies that the oracle comparator may be missing a significant number of actual errors. An analysis of variance revealed that GCC-XML relied heavily on deletions being positively associated with errors, while in GCC-XML's training data the opposite was the case. Rather than recommend that developers return to using a `diff`-like comparator to avoid missing bugs, or employ other methods where manual annotation is required, they can continue to apply SMART with one modification: they can extend the training data with test case output

pairs between unmodified source code executions and fault-injected source code executions. A cautious development organization might randomly spot-check 10% of the results predicted by the technique. Such a spot-check would still involve less effort than a standard `diff` comparator, and is likely a one-time cost to evaluate the compatibility of SMART's readily-available training data set with that of the output in the application-at-test. If the results are insufficiently accurate, the test suite can be augmented by defect seeding, as described in the next subsection.

6.4 Training Data from Defect Seeding

This section details using defect seeding to generate additional training data for GCC-XML. Defect seeding offers the benefit of annotation-free training data generation, while still tailoring the training data to the current application under test, thereby saving developers from manually training an oracle comparator such as SMART.

The relatively low recall value for GCC-XML in Section 6.3.2 suggests that GCC-XML may exhibit some errors that are different from the instances of errors in the general training data set provided to SMART. Given that GCC-XML is an XML transformer for compiled C++ files, and the rest of the training applications mostly transform text or XML/HTML documents, this is not surprising. Developers can, however, automatically tailor the training set to their application as needed using defect seeding.

The basic approach is to automatically seed the source code of the application with defects [56] and run the resulting mutated program on its existing regression test suite. Any difference in the output can be attributed to the injected fault, and that output pair can be added to the training data with the label "should inspect". The process is repeated until a sufficient number of training instances have been generated. Using defect seeding or mutation to simulate errors in test case output for web-based applications has previously been explored [59,103]. While automatically generating, compiling and running mutants can be CPU-intensive, manual intervention is not required.

Defect seeding was implemented for GCC-XML with a subset of mutation operators described by Ellims *et al.* [24]. Examples of mutation operators include deleting a line of code, replacing a

statement with a return, or changing a binary operator, such as swapping `AND` for `OR`. To generate a mutant version of GCC-XML, a single line of source code was randomly chosen from all of the source code files for that project, and a mutation was applied to it. For each mutant version of the program, only a single line was mutated. Each mutant version of the source code was compiled separately, followed by re-running the test suite, recording as erroneous any cases where the output from the mutant source code differed from that of the original output. The overall process was quite rapid: using single-line seeded faults obtained 11,000 usable erroneous output pairs within 90 minutes on a 3 GHz Intel Xeon computer.

Figure 6.5 shows the F_1-scores, averaged over 1000 trials, when adding between 0 and 5 defect-seeded output pairs to the set of training data, and then running SMART for GCC-XML. Selecting 0 mutants is provided as a baseline. The large margin of error when adding only one mutant output pair implies that performance depends on selecting the most useful mutant outputs to include as a part of the training data set. However, selecting any mutant output is always better than selecting none. It is possible to dramatically affect the model's predictive power by adding a single mutant; for the case of GCC-XML, there were only 44 errors in the training data set, and adding one more to such a small number can significantly change the results. For training data sets that contain more errors, it is likely that more mutants will be required, although this chapter demonstrates that it is quite simple to automatically generate these defects.

In addition, no significant performance gains are witnessed beyond adding 5 mutant output pairs, at which point the F_1-score was an essentially-perfect 0.999. Very little application-specific training data (5 labeled output pairs) are needed to bring even the worst performing benchmark up to almost-perfect performance; even this application-specific data can be obtained automatically.

6.5 Summary of Experiments

Inherent web site similarities are a promising way to reduce the burden of human effort in regression testing for web-based applications. Section 6.3 demonstrated that using test case output pairs from *unrelated* web-based applications to train a model to predict errors in output in the application-at-

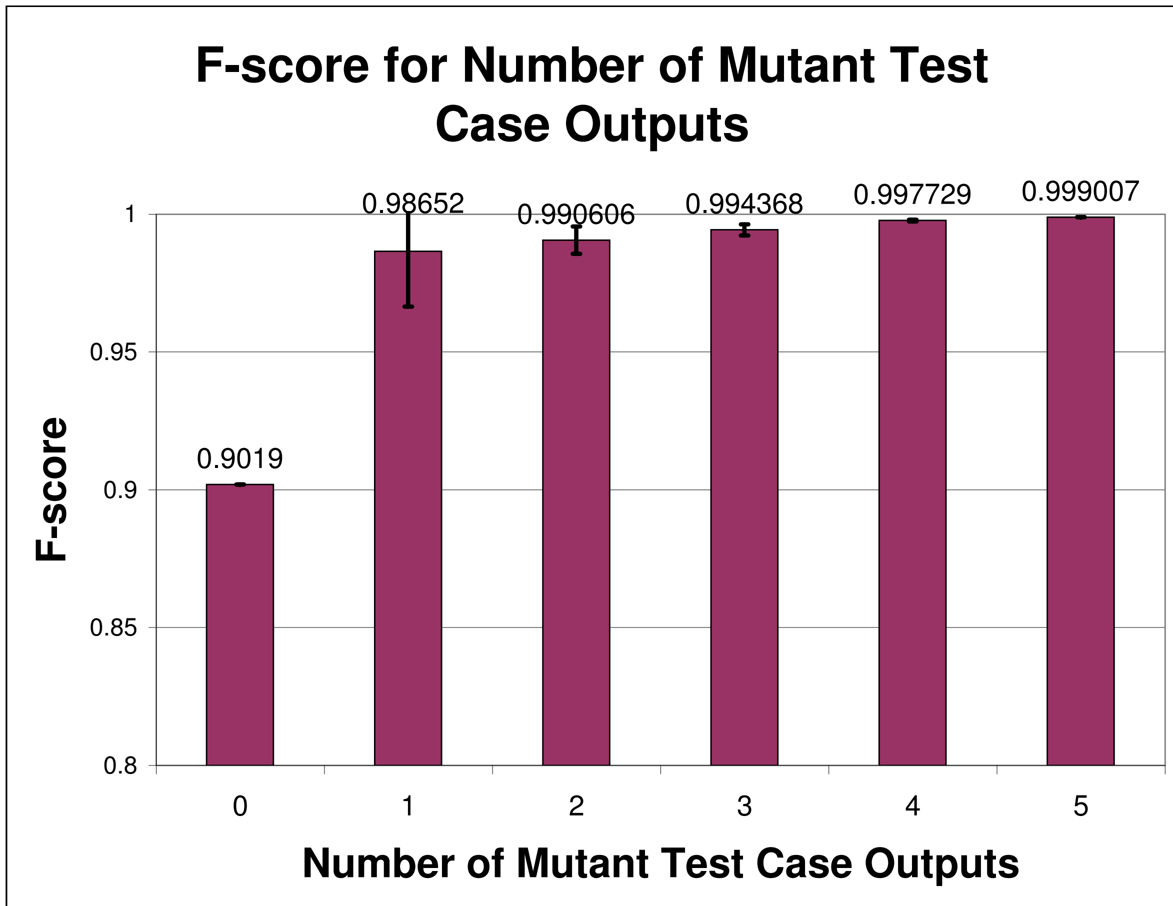

Figure 6.5: F_1-score for GCC-XML using the model with different numbers of test case output pairs from original-mutant versions of the source code. The "0" column indicates no mutant test outputs were used as part of the training data. Each bar represents the average of 1000 random trails; error bars indicate the standard deviation.

test is a viable strategy, achieving perfect recall and precision for the two web application benchmarks, and close to perfect (0.98 and 0.99) F_1-scores for the two XML-based applications. To obtain the F_1-score of 0.999 for GCC-XML, the training data was augmented with five automatically generated outputs obtained via defect seeding. In all cases the oracle comparator, SMART, outperforms a `diff`-like comparator by a factor between 2.5 and 50 times, thereby significantly reducing the number of false positives, and thus the developer cost, with respect to `diff`.

6.6 Threats to Validity

Although significant savings in the amount of effort required to automate parts of the regression testing process are shown in this chapter, it is possible that the benchmarks selected to test on were not indicative of other applications. To mitigate this threat, open-source benchmarks were chosen rather than toy applications, selected from a variety of domains. The combined benchmarks are over seven times larger than the combined benchmarks of the most closely related previous work [104] in terms of lines of code, with over twice as many total test cases.

In cases where the technique does not work as well as desired, defect-seeding results suggest that largely-automatic improvement is possible. Adding mutant test case outputs to the set of training data for the automated oracle comparator can help to tailor the model to the application-at-test, and the low number of mutants required implies that it may even be possible to provide a very small (≤ 5) set of error instances to tailor the tool to a specific application. Because any difference between the expected output and defect-seeded source code output is considered a fault, however, developers should take care not to include false positives, such as cases where timestamps are different, in the training data set, unless they label them as non-bugs.

It may also be possible that there are certain web applications for which this approach does poorly, despite defect seeding, because the specification of the application-at-test has unusual properties. For example, consider a Wiki application where the formatting and content of displayed natural language text *is* important. If fault seeding is unable to provide suitable defects on which to train the model to recognize small changes in natural language text as errors, this approach is unlikely to yield savings over `diff`. Such unusual cases are left for future work.

6.7 Related Work to Automated Error Detection in Web Applications

To the best of my knowledge, using SMART with a pre-existing corpus of training data is the first fully-automated approach towards providing a highly-precise oracle comparator for this domain. For a review of partially-automated approaches in this area, as well as other work related to fault detection in web-based applications, see Section 5.8.

6.8 Summary

Testing web-based applications is often overlooked due to a lack of time and resources, despite their high reliability requirements. Although automating test suite replay is relatively simple, comparing test results with expected output remains a challenge for this domain. This chapter presented a new technique that takes advantage of inherent similarities between web-based applications to automate parts of the regression testing process. Using a `diff`-like comparator for web-based output yields a significant number of false positives that must be manually inspected: instead, a fully automated highly-precise oracle comparator is offered that is based on a model trained on data from unrelated web-based applications. This technique was evaluated on 6728 test case pairs, and was found to outperform `diff` anywhere from 2.5 to 50 times, achieving perfect precision and recall half the time, and very close to perfect precision and recall otherwise.

The main thesis of this dissertation is that user-visible web-based application errors have special properties that can be used to improve the current state of web application error detection, testing and development. This chapter explored one component of such an observation: how exploiting similarities across seemingly unrelated web-based applications, makes providing manually-annotated training data unnecessary, thereby yielding a fully-automated highly-precise oracle comparator.

Chapter 7 Modeling Consumer-Perceived Web Application Error Severities for Testing

The previous two chapters provided means for automating parts of the regression testing process for web-based applications by relying on the structure of XML/HTML output as well as similarities in the way web-based applications fail. The following chapters will focus on further increasing the return on investment for testing in this domain by focusing on severe errors, recognizing that consumer-perceived severity is an important characteristic of errors in web applications. This chapter begins by probing the concept of consumer-perceived error severity in web applications. Specifically, Hypothesis (H3), that

faults injected into web applications, using an automated seeding process using mutation operators described in Section 2.2.3, or using manual fault seeding as in [106], vary in their underlying consumer-perceived severities, (H3)

is established as a baseline. The rest of this chapter then recognizes that web application testing is plagued by a perceived low return on investment (see Chapter 1), and explicitly evaluates Hypothesis (H4), that

an automated model of consumer-perceived error severity can be constructed that agrees with human severity judgments at least as often as humans agree with each other, evaluated using the Spearman's Ranking Correlation Coefficient (SRCC) [44, 107]. (H4)

Such a consumer-perceived error severity model could allow developers to prioritize user-visible errors according to their likelihood of impacting consumer retention, thereby encouraging testing these applications, by increasing the perceived return on investment of testing activities.

7.1 Challenges with Consumer Retention

Despite the growing usage of web applications, extreme resource constraints during their development frequently leave them inadequately tested. Additionally, customer loyalty to any particular website remains notoriously low, and is primarily determined by the usability of the application [76]. Despite this issue, consumer satisfaction and retention are rarely formally addressed during the development and testing of web applications.

This chapter focuses on targeting development and testing strategies toward consumer retention, making them more attractive to developers. Two insights suggest that it is possible to do so. First, although web applications are frequently complex, composed of multiple components, and written in various programming languages, they tend to fail in similar ways. The previous chapter demonstrated that such similarities in failures are highly predictable. These similarities stem from the fact that web applications render output in HTML, where lower-level faults frequently manifest themselves as user-visible output [81, 104]. This insight allows developers to focus their testing strategies on this top level of the application to capture a broad class of faults.

Second, web applications are meant to be viewed by a human user. While this implies that faults will often be user-visible, this human-centric quality of web applications can be exploited by defining the acceptability of output as whether or not users are able to complete their tasks satisfactorily. Rather than viewing verification in absolute terms, developers may focus on reducing high severity faults that may cause consumers to abandon the application.

Consumer-perceived error severities have not been thoroughly studied in the context of web applications, even though this domain is highly human-interaction centric. Although intuitions abound (e.g., some may believe that small typographical errors are less likely to upset consumers than incorrect shopping cart totals, or that missing banner ads are less severe than entirely-blank pages), because there are no concrete, evidence-based guidelines in making such judgments, developers may currently be unsure or unaware of how to focus testing methodologies towards fixing high-severity errors first. In general, the extent to which various sorts of errors will drive away consumers remains unclear.

Consider, for example, users of a service who are in the process of updating stored personal profile data. When they attempt to save their changes through the web application, their changes are confirmed, although they receive a small warning message at the top of the screen regarding a seemingly unrelated issue. Some users may not notice the warning, others may ignore it, and some with a technical background may interpret the warning to be harmless. It is also possible, however, that users may be left uneasy with respect to the persistence of their profile changes in the presence of any warning message on such a confirmation screen. This relationship between visible web application errors and their consumer perception has not been explored in web application development and testing.

An orthogonal problem to unknown consumer perception of web application error severity is that of correctly identifying the severity of a particular fault, despite human judgment. In other words, we often not only fail to consider consumer-perceived severities of web application faults, but even when we do, it is unclear how to assign a fault an accurate consumer-perceived severity judgment. Even the concept of fault severity is not always straightforward. Developer-perceived fault severities are frequently recorded during the testing and maintenance phases of software development in bug repositories, but these judgments have been found to not represent true severities and may instead factor in other variables, such as the politics behind labeling a bug with a certain severity rating [80]. Instead, this work focuses on consumer-perceived severities, rather than developer judgments, in an effort to more precisely identify those faults that are likely to drive consumers away.

7.2 Strategies For Modeling Consumer Perceived Severity

Relying on a single human observer to judge the severity of a particular error, especially if that person is a developer of the application-at-test, may not necessarily lead to accurate assessments of the impact of such defects on consumer retention. Consequently, a formal model of fault severity is desired, that strongly agrees with *average* human judgments, as opposed to the opinion of any single individual. Such a formal model of fault severity can then be used to guide developers in fault

prioritization, or to more effectively evaluate competing testing approaches in the web application domain. This chapter builds a model of consumer-perceived severities from a large-scale human study; surface and semantic features related to HTML rendering can be used to model consumer-perceived fault severities, based on a survey containing 12,600 datapoints from 386 humans. The formal model agrees with the average human 84% of the time, and more than the humans agree with each other. In addition, many of the features used in the model can be detected automatically, and this chapter shows that an automated judgment of consumer-perceived fault severity can be made with only a 1% drop in accuracy when compared to the non-automated version. Such a model can increase the return-on-investment for web application testing, since it can help developers to focus on bugs that consumers care about.

As a baseline, this chapter verifies that faults have varying severity levels, before attempting to build such a model of consumer-perceived severity. Although this point may seem obvious, much of the current research in testing web applications uses fault detection as an orthogonal approach to code coverage in evaluating testing approaches, and assumes that all faults are of the same severity [26, 37, 39, 77, 101, 103, 104, 105]. As a result, competing testing approaches may be incorrectly evaluated in terms of efficacy if only the number, and not the severity, of uncovered faults is measured. Fault-based testing is used to introduce faults into the code meant to be uncovered by the test suite [28, 106]. Such fault seeding can be achieved in two ways: faults can be manually inserted by individuals with programming expertise, or mutation operators can be used to automatically produce faulty versions of code. While this work does not directly address the difficulty of uncovering seeded faults, it does show that naïvely-injected faults do not have a uniform consumer-perceived severity, suggesting that standard fault seeding metrics are misleading for web application test suite evaluation. In other words, the assumption that all seeded faults have the same severity, and thus that a test suite that finds more faults is necessarily better, is not always true when considering consumer-rated severity.

Having established that web application faults are not all equally severe, this work models consumer-perceived fault severity by analyzing surface features of both the browser-displayed GUI as well as HTML output to arrive at a predictive model. The goal is to provide a model that agrees

with average consumer-perceived severities more often than consumers agree with each other, as a single individual human judgment may not be accurate. Access to such a predictive model can therefore provide developers with an objective assessment of the severity of a web application fault with higher accuracy than asking an arbitrarily chosen human observer to make the same judgment. Ultimately, developers could use such a model to focus their development and testing efforts on the elimination of high severity faults. Prioritizing faults according to their severity can increase the perception of return-on-investment for testing by saving developer effort and increasing consumer retention by focusing on severe faults first.

While other approaches have explored fault taxonomies for web application faults, to the best of my knowledge this work is the first to address these faults in the context of consumer-perceived severity. In addition, this chapter provides an *automated* model that, when given a correct version of HTML output and a faulty version, predicts the severity of the fault with higher accuracy than that of human judgments.

7.3 Consumer-Perceived Error Severity Study

This section examines the consumer-perceived severity of real-world and seeded faults to show that they have varying levels of severity, thereby refuting the underlying assumption in fault-based testing of web applications that a test suite that detects more faults than its competitors will necessarily detect more bugs that may drive away consumers, supporting Hypothesis (H3). Establishing that faults have varying severities is important to be able to more accurately assess the fault detection capabilities of competing testing approaches, and is a prerequisite to the consumer-perceived fault severity model in this chapter.

7.3.1 Setup and Definitions

The hypothesis that web application faults have varying severity levels was tested by designing a human study where subjects rate their perceived severity of various potential web application faults as if they were consumers of the web application under examination. Four hundred real-world

Description	Severity Rating
I did not notice any fault	0
I noticed a fault, but I would return to this website again	1
I noticed a fault, but I would probably return to this website again	2
I noticed a fault, and I would not return to this website again	3
I noticed a fault, and I would file a complaint	4

Figure 7.1: Severity scale for web application faults.

faults were obtained for evaluation by the users in the study. These faults were presented to human subjects who were asked to rate the severity of each fault on the 5-point scale in Figure 7.1. Severity was intentionally left formally undefined.

Each potential fault was presented as a scenario triple:

- the *current* webpage
- a *scenario description* of the task the user is trying to accomplish in the scenario, and the action taken
- the *next* webpage (which may or may not contain a fault)

This before-and-after scenario view is necessary because web application faults depend on context and the use of web applications is inherently dynamic. For example, a scenario may begin with a login screen for a website, and include the description that the user has just entered a valid username and password and is going to click the *login* button. The human participants are then shown the *next* page, as if, according to the scenario narrative, they had clicked the button. Human subjects were presented screenshots of both the *current* and *next* page, and were allowed to toggle freely between them before deciding on a severity rating.[1]

The four hundred real-world faults originated from the bug report databases of 17 open-source benchmarks summarized in Figure 7.2. Faults were randomly but systematically selected from these

[1]The written survey instructions and an example of the *current* and *next* screenshots are available in the Appendix in Section A.1.

Name	Language	Description	Faults
Prestashop*	PHP	e-commerce	30
Dokuwiki*	PHP	wiki	30
Dokeos	PHP	e-learning	22
Click*	Java	JEE webapp framework	3
VQwiki*	Java	wiki	6
OpenRealty*	PHP	real estate listing management	30
OpenGoo	PHP	web office	30
Zomplog	PHP	blog	30
Aef	PHP	forum	30
Bitweaver	PHP	content mgmt framework	30
ASPgallery	ASP.NET	gallery	30
YetAnother Forum	ASP.NET	forum	30
ScrewTurn	ASP.NET	wiki	30
Mojo	ASP.NET	content mgmt system	30
Zen Cart	PHP	e-commerce	30
Gallery	PHP	gallery	30
other	-	-	9

Figure 7.2: Real-world web applications mined for faults. All applications were sources for human-reported faults taken from defect report repositories, as well as non-faults taken from indicative usage. An asterisk indicates an application used as a source for manually- and automatically-injected faults.

repositories by starting from the newest repository entry and working backwards until 15 faults had been successfully reproduced, and then repeating this process from the oldest fault moving forward (four of the benchmarks yielded fewer than 30 viable faults). The goal of selecting faults in this manner, as opposed to choosing faults randomly, was to replicate faults encountered in both immature and more established web applications. The description of the fault in the repository was used to obtain or replicate a screenshot and the HTML code of the *current* and *next* pages, along with a *scenario description*. Screenshots included in bug repository defect reports were used unchanged as the *next* image if applicable (e.g., given sufficient image resolution). To capture as many classes of faults as possible, written scenario descriptions were relayed to study participants to provide supplemental bug report detail that may not be obvious simply by examining a *current* image. For example, subjects were instructed that they had permissions to access a given item when such context was necessary for the fault to be recognized. Similarly, when making a purchase through a shopping cart participants were told to imagine they had successfully completed checkout steps 1–3 before showing them the fault in step 4 related to information entered on step 1 on the *current* screen. In addition to these 400 real-world faults, 100 non-faults (i.e., indicative, fault-free behavior) were obtained from all of the benchmarks.

Coupled with the 400 real-world faults and 100 non-faults, the study employed 200 manually-injected faults and 200 automatically-injected faults equally distributed among those benchmarks in Figure 7.2 denoted by an asterisk, plus one other PHP-based web forum called VANILLA. Manual injection was accomplished by instructing three graduate students with programming experience to insert one fault into the source code at a time, and then re-running a test suite according to the methodology in Sprenkle *et al.* [101]. Automatic source code mutation was similarly used to generate mutant versions with one inserted fault per test suite run. Mutants were generated using the mutation methodology in Section 6.4.

The 400 real-world faults were combined with the 400 injected faults and the 100 non-faults to yield a corpus of 900 scenarios. These scenarios were then randomly assigned into groups of 50. Each survey participant rated a group of 50 scenarios without knowing whether any particular scenario was a real-world fault, an injected fault, or a non-fault. Users were instructed to use

Factor	F value	p value
Rating	592	< 0.001
Human Rater	5	< 0.001

Figure 7.3: 2-way analysis of variance of fault severity as judged by human subjects.

their real-life experience with web applications to decide on a severity for each potential fault they viewed. Users were advised to assume that any perceived fault would be fixed upon a subsequent return to the webpage, but that this fix would occur at an unknown point between the present time and up to one year in the future; vendors may take over one hundred days to patch even serious problems [110, p.13].

Over 380 anonymous subjects, a majority from the undergraduate populations at the Universities of Virginia and Maryland, participated in the study. Approximately half of the subjects were first year students, with the rest being second year or higher. Subjects were compensated with either $5 upfront or the chance to participate in prize drawings of up to $150. Completing the survey took 20 minutes on average. To ensure that human voters were not subject to training effects (where subjects may have been influenced in their latter severity ratings by having already rated some number of faults), the variance of votes on the first 25 faults in each set was compared to the variance of the rest of the 50 faults subjects viewed. These values differed by 0.02 on a 5-point scale, respectively, indicating that training effects are unlikely in this experiment and there was no need to repeat the measurement for the same subjects in a different fault presentation order.

To ensure the number of subjects yielded statistically significant results, a 2-way analysis of variance [74] was conducted to separate the variance due to differences in true fault severity from the variance due to differences in human judgments, in the actual votes made by subjects. The results of this 2-way ANOVA are summarized in Figure 7.3. The F value of the *Rating*, which corresponds to the square root of the variance explained by that feature over variance not explained, was 100 times that of the F value of *Human Rater*s, indicating that the contribution of the ratings themselves, as opposed to who made the rating (i.e. the *Human Rater*), was much more significant with respect to the fault severities recorded. This implies that the number of subjects was enough

Fault Type	Low ≤ 1	Medium > 1 and ≤ 2	Medium-High > 2 and < 2.5	Severe ≥ 2.5
Real-world	23%	30%	19%	28%
Automatic-injected	25%	25%	27%	23%
Manual-injected	23%	28%	27%	22%
Non-fault	92%	7%	0%	1%

Figure 7.4: Average severity ratings from a total of 12,600 human judgments of 900 scenarios.

to generate statistically significant results, as any particular human responsible for making a rating had little impact on the fault severities. Additionally, inter-annotator agreement was measured using Pearson's rho [89], with a value of 0.70 and p value of 0.03 for this dataset, demonstrating relatively strong inter-annotator agreement and supporting the conclusion that the number of subjects viewing each fault yielded statistically significant results.

7.3.2 Study Results

Over 12,600 severity scores were recorded from 386 humans, with at least 12 votes per fault. Figure 7.4 presents the distribution of severities across real-world, manually-injected, automatically-injected, and non-faults. All faults, whether real or injected, do not have the same severity, refuting the underlying assumption that each detected fault in fault injection-based testing is equally important [26, 37, 39, 77, 101, 103, 104, 105]. This implies that a test suite that uncovers more faults may not necessarily produce an application with higher customer retention, because the underlying severity of the discovered faults are unknown. Web applications may be more vulnerable to severe faults than some other types of software, due to their direct contribution to consumer loss as defined in the severity rating scale in Section 7.3.1. A *severe fault* is defined to be one with an average human severity rating of 2.5 or higher, as these are most likely to result in lost consumers, based on Figure 7.1.

In addition, Figure 7.4 reveals the underlying distribution of fault severities for manually and automatically-injected faults in the experiment. Seeded faults had almost equal numbers of faults in each severity category, indicating that fault-injection based techniques, overall, are a reasonable way of measuring test suite quality in that they do test faults with varying severities, although fault injec-

tion alone without a severity analysis cannot be used to compare two test suites if one is interested in characterizing consumer retention. Given that injected faults have varying severities, Section 7.5 details an automated model that can correctly predict consumer-perceived fault severities with high precision, and could be combined with fault-injection based testing to provide a more accurate representation of test suite efficacy. Figure 7.5 presents the results of a Kolmogorov-Smirnov analysis [33] of this dataset, which attempts to determine if two datasets differ significantly. No statistically significant differences were observed between real-world, automatically-injected, and manually-injected fault severity distributions, corresponding to the results in Figure 7.4. As expected, the distributions of real-world, automatically-injected, and manually-injected faults *were* significantly different than those of non-faults.

Figure 7.6 compares the severity distributions of real-world faults to these manually- and automatically-injected defects broken down by each application that had both injected faults and real-world faults, with the exception of CLICK which only had three real-world faults. In three of the four benchmarks the automatically and manually seeded faults both failed to generate as many severe faults as the real-world faults for the same application, although the severity distribution of the real-world faults in this study is not necessarily that of web applications in general. For example, low-severity faults are likely under-represented in the bug reporting databases from which the scenarios were drawn.

In an attempt to characterize the relative distribution of fault severities in the real world, ten web application developers were surveyed.[2] Figure 7.7 presents the results of the developer survey. In an effort to secure anonymity, participants were allowed to take the survey without providing the lines of code (or any other identifying information) of their current project. The largest benchmark (denoted with an asterisk) was described as similar to Microsoft Hotmail, Yahoo Mail and Windows Vista's Weather Gadget, with 250-300 million customers daily. Even in this limited survey, high-severity faults were a relatively small percentage of faults overall, similar to other studies of fault severity in industrial settings [79]. Although this survey may suffer from the problems of developer self-reporting previously mentioned in [80], it is presented as supplemental evidence that real-world

[2]The written survey instructions are available in the Appendix in Section A.2.

Group 1	Group 2	D value	p value
Real-world	Automatically-injected	0.0691	0.524
Real-world	Manually-injected	0.0969	0.170
Automatically-injected	Manually-injected	0.0616	0.839
Real-world	Non-fault	0.7211	< 0.001
Automatically-injected	Non-fault	0.7154	< 0.001
Manually-injected	Non-fault	0.7083	< 0.001

Figure 7.5: Kolmogorov-Smirnov test results for the dataset.

faults vary in their severity.

7.4 Modeling Consumer-Perceived Severities of Web Errors

In the previous section, web application faults, whether real-world or seeded, were shown to differ in their severity as perceived by consumers. This section shows that a model of consumer-perceived severity can be successfully built that agrees with average human judgments more often than humans agree with each other. First, a proof-of-concept human-assisted model is presented, that is then extended to an automated version, and supporting Hypothesis (H4). Web applications are sensitive to the consumer perception of their reliability. Such a model can be used by developers of web applications to prioritize faults they plan to fix, or by researchers to compare various testing methodologies, in an effort to minimize consumer loss.

7.4.1 Modeling Error Severity

Although web applications themselves vary widely in their presentation and functionality, web application *errors* do have common features (see Chapter 6). For example, a stack trace may be displayed to users for a number of reasons, across a number of platforms, in unrelated web applications. Such errors in deeper levels of the application are commonly corralled into user-visible HTML [81, 104]. Based on this insight, seventeen boolean surface features of web application browser output are presented that may be indicative of faults. These features, summarized in Figure 7.8, can accurately model consumer-perceived fault severity: given the presence or absence

Benchmark	≤ 1	> 1 and ≤ 2	> 2 and < 2.5	≥ 2.5
PRESTASHOP real-world	37%	27%	17%	20%
PRESTASHOP automatically-injected	24%	45%	24%	6%
PRESTASHOP manually-injected	30%	33%	21%	15%
OPENREALTY real-world	41%	21%	31%	7%
OPENREALTY automatically-injected	21%	24%	18%	36%
OPENREALTY manually-injected	45%	24%	27%	3%
DOKUWIKI real-world	22%	22%	11%	44%
DOKUWIKI automatically-injected	36%	27%	24%	12%
DOKUWIKI manually-injected	12%	27%	21%	39%
VQWIKI real-world	0%	33%	33%	33%
VQWIKI automatically-injected	21%	24%	31%	24%
VQWIKI manually-injected	36%	32%	14%	18%

Figure 7.6: Comparison of real-world and injected faults per application. Reported as a percentage of total faults for each group.

LOC	Low	Medium	Medium-High	Severe
200K	57%	12%	8%	23%
2,000K*	38%	40%	15%	7%
n/a	90%	5%	3%	2%
20K	90%	10%	0%	0%
n/a	95%	3%	1%	1%
1K	75%	12%	12%	1%
n/a	85%	10%	0%	5%
$>$ 100K	78%	14%	2%	6%
n/a	86%	10%	0%	3%
1K	90%	10%	0%	0%
Average	42%	36%	14%	8%

Figure 7.7: Severity distribution of faults according to developer responses. *Average* refers to the average distributions for projects where lines of code (*LOC*) were provided.

of these seventeen features, the model produces a severity judgment that agrees strongly with the average human judgment. These features are orthogonal to those presented in Section 5.2.2.

A fault may lead to a combination or constellation of features. For example, a small warning message on a page that does not interfere with its main functionality may be considered both an *Error Message* and *Cosmetic* according to Figure 7.8. No faults from the human study were labeled as having more than six features. Figure 7.8 also lists the percent of real-world faults in the human study that exhibited each feature in the *% of Faults* column.

The severity model relies on human annotators to label each fault with its respective features. Twelve graduate students, with an average of twelve, four, and twelve, years of programming, web programming, and web usage experience respectively, were recruited as annotators. The model calculates the consumer-perceived severity associated with a web application fault via a manually-constructed decision tree that was engineered through an analysis of 300 of the 400 real-world faults described in Section 7.3.1, chosen at random. The remaining 100 faults were held out to serve as testing data.

For example, a fault labeled as having an *Error Message* and an *Arithmetic Calculation Error* would be predicted to have high severity. An example of such a situation is a miscalculation of shipped quantities in a shopping cart with a seemingly unjustified associated warning message to

Feature	Description	% of Faults
Arithmetic Calculation Error	Generally for shopping-cart based applications, any error in calculating the amount paid, shipping, taxes, discount applied, quantities ordered, etc.	3
Blank Page	An empty page containing no information or text.	2
404 Error	An error experienced when the URL is not found; the words "404" or "not found" must appear somewhere on the page.	3
Cosmetic	An error that does not affect the functionality of the website, such as a typo, small formatting issues, bits of visible HTML code, etc.	24
Language Error	An inability to encode or correctly convert characters between languages, often resulting in incorrect characters on the page.	2
CSS Error	An error in loading the stylesheet between the *current* and *next* pages.	≤ 1
Code on the Screen	Any error that results in non-HTML, non-SQL program code appearing on screen, including any error referring to a line number.	24
Error Message / Other Error	Either any error message, or any error that cannot be classified in any other category.	52
Form Error	Missing, malformed, or extra buttons, form fields, drop-down menus, etc, including incorrectly validating forms.	7
Missing Information	Any part of a webpage that is missing, not including images.	13
Wrong Page / No Redirect	An unexpected page is loaded.	12
Authentication	Any errors that occur during login.	6
Permission	Any errors occurring with respect to user permissions in an application, such as access being incorrectly denied to a user.	4
Session	An unexpected session timeout or other session-related issues.	1
Search	Errors occurring during searching, such as incorrectly printing out results.	2
Database	Any errors associated with accessing or querying a database, including visible SQL code being displayed.	9
Failed Upload	An error during the upload of an item.	5
Missing Image	A missing image.	3

Figure 7.8: Boolean surface features associated with web application faults. These features form the basis of the formal model of human-perceived fault severity.

Model	SRCC	Accuracy	Severe Missed
Manual Decision Tree	0.84	84%	1/30
Individual human (avg)	0.70	59%	8/30
Always Average Rating	0.51	58%	30/30
Always Median Rating	0.51	59%	30/30
C4.5 Decision Tree	0.76	85%	5/30

Figure 7.9: Average Spearman's Ranking Correlation Coefficient (SRCC) between each model and the average human over 100 held-out faults. A correlation of 1 indicates perfect agreement, whereas a correlation of 0 indicates no correlation. An SRCC score of more than 0.5 is considered to have moderate to strong correlation for a human study [44]. The model agrees with the average human judgment more strongly than humans agree with the average human judgment. An 'Accurate' prediction differs with the average human value by less than 0.75.

the user. Similarly, perceiving a fault as *Cosmetic* would result in an assignment of low to moderate severity, even when combined with other features. The complete decision tree is available in the Appendix in Section A.4; it includes 29 conditional judgments. Its behavior can be summarized by noting that *Arithmetic Calculation Error*s, *Errors Message / Other Error*s, *Authentication* and *Permission* issues, *Code on the Screen*, and loading the incorrect page or no page at all are associated with more consumer dissatisfaction than other fault features.

7.4.2 Human-annotated Model Performance

The predictive model was tested on the remaining 100 real-world faults excluded from the training dataset. The model's performance was compared to that of subjects from the human study (see Section 7.3.1). A number of baseline approaches, such as always predicting the average fault severity, always predicting the median fault severity, and a C4.5 decision tree [49] derived automatically from the same training set of human annotations used to construct the manual decision tree model, were also included.

The goal of the model is to agree with the average human severity rating for this group of faults more often than the humans themselves agree. Therefore, the Spearman's Ranking Correlation Coefficient (SRCC) [107] was measured between each technique and the average human severity rating for each fault, averaged across all 100 held-out testing faults. Figure 7.9 presents the results.

Although there is a strong positive correlation between individual human judgments of error severity and the average perceived severity of a particular fault, the predictive model outperforms all other baselines with an SRCC of 0.84, including the average performance of individual humans. The average standard deviation of human judgments on a single fault was 0.95, almost an entire point on the 5 point scale; a single human opinion of fault severity is therefore inherently unreliable. As a concrete and typical example, in one moderate fault, four respondents said they would return to the website, seven would probably return, while two would not. In the two most severe faults included in the study, while 22 users would file a complaint or not return to the website, three respondents reported that they probably *would* visit the website again. Figure 7.4, which includes the consumer-perceived severity of *non*-faults, demonstrates that humans are not always accurate at judging the average severity: they sometimes perceive even non-faults as faults. Relying on any one human observer, such as a developer involved in making the web application, is therefore not necessarily a reliable way to infer actual severities of faults in the spirit of prioritizing them for resolution. Developer-perceived severities have the additional confounds related to inaccurate judgments detailed in [79, 80].

Figure 7.9 lists the percentage of accurate severity judgments for all models across the test set of 100 faults, as well as the number of severe faults missed. The manual decision tree model, using the seventeen features in Figure 7.8 accurately predicted the severity of 84% of the faults in the sample, where an accurate prediction is defined as lying within 0.75 of the goal severity score, a round cutoff that is less than both the standard deviation and variance. Missing severe faults is likely more dangerous than assigning high severity to non-severe faults, as the former may translate into lost consumers, while the latter only increases the number of faults a developer may have to examine. Overall, the model missed only one high-severity fault, and in cases where the prediction was not within 0.75, it generally predicated the fault as too severe, rather than not severe enough. Notably, the model outperforms an average human in terms of severe fault identification, implying that it is more reliable to use the model to assign fault severities to web application faults than to rely on any one arbitrary individual.

The predictive power of the features was also investigated by performing an analysis of vari-

Feature	Correlation	F	p value
Code on the Screen	+	19.47	< 0.001
Cosmetic	-	13.23	< 0.001
Database	+	12.36	< 0.001
Authentication	+	6.99	0.01
Functional Display	-	6.00	0.01
Code Error	+	4.40	0.03

Figure 7.10: Analysis of variance showing relative feature predictive power for finding high-severity faults. F denotes the F-ratio, which is the square root of variance explained by that feature over variance not explained. Higher F values affect the model more. The last column denotes the significance level of F (i.e., the probability that the feature does not affect the model); values below 0.05 are significant.

ance [74] when predicting *high-severity* faults, shown in Figure 7.10 (only features with a significant main effect are listed). Web application faults that were labeled by humans as having any type of code on the screen (e.g., *Code on the Screen* and *Database* errors) were most significantly correlated with high-severity faults. Faults labeled as *Cosmetic* were very unlikely to be considered severe, and faults during authentication and those that displayed error messages were judged as more likely to result in consumer loss than other types of faults.

7.5 Automatically Predicting Error Severity

Having established that it is possible to model web application error severities with greater precision than an average individual human would, this section shows that it is possible to automate such a model, and thus not rely on human annotation of fault surface features, without a large sacrifice in accuracy. An automated model is now described that examines HTML output to assign a fault severity to a faulty web page when compared with the expected oracle output. The model is evaluated not only by comparing its precision to that of the human-based model of Section 7.4.2, but also with an experiment to show the potential savings developers may experience when using the model to prioritize faults.

7.5.1 Experimental Setup

The automated model relies on the same decision tree architecture as the human-dependent model, but approximates the seventeen features' labels (see Figure 7.8) automatically by examining HTML code. For example, the *Wrong Page* feature is set if the only HTML elements shared between two pages are `<html>`, `<head>`, and `<body>` elements, while a *Missing Image* occurs if the `<img> src` attribute has changed or is missing between the two webpage versions. The model first builds a mapping between the correct, expected HTML output and the faulty HTML using the DIFF-X [20] algorithm (presented in Section 5.2.2), and then examines the unmapped nodes between the two webpages in order to determine which features apply. This use of features based on a mapping between before-and-after web page HTML is similar in spirit to approaches that reduce the cost of regression testing for web-based applications in previous chapters.

Because the automated model cannot mimic many context-dependent human judgments, the decision tree from Section 7.4.1 is modified to focus more on those features that are likely to be correctly labeled. For example, while a human may distinguish between an incorrect page being loaded and a missing image, the model will frequently flag both the wrong page and missing image attributes in such an instance as the incorrect HTML input happens to be missing the image by virtue of it being the wrong page.

To measure the potential savings associated with the automated model, consider a scenario where a developer has limited resources to fix known faults between releases of the application, and therefore wishes to prioritize those faults by severity. It is assumed that focusing on high-severity faults, instead of low and medium severity faults, will result in greater consumer retention. However, addressing each fault requires developer effort. Savings are therefore measured in terms of the number of non-severe faults the automated model can correctly flag which do not need to be addressed, while simultaneously avoiding failing to flag faults that are severe and thus need preferential attention. The same 100 held-out real-world faults from Section 7.4.2 are used as the test dataset so as to compare the automated model's performance to that of the annotation-based model of the previous section.

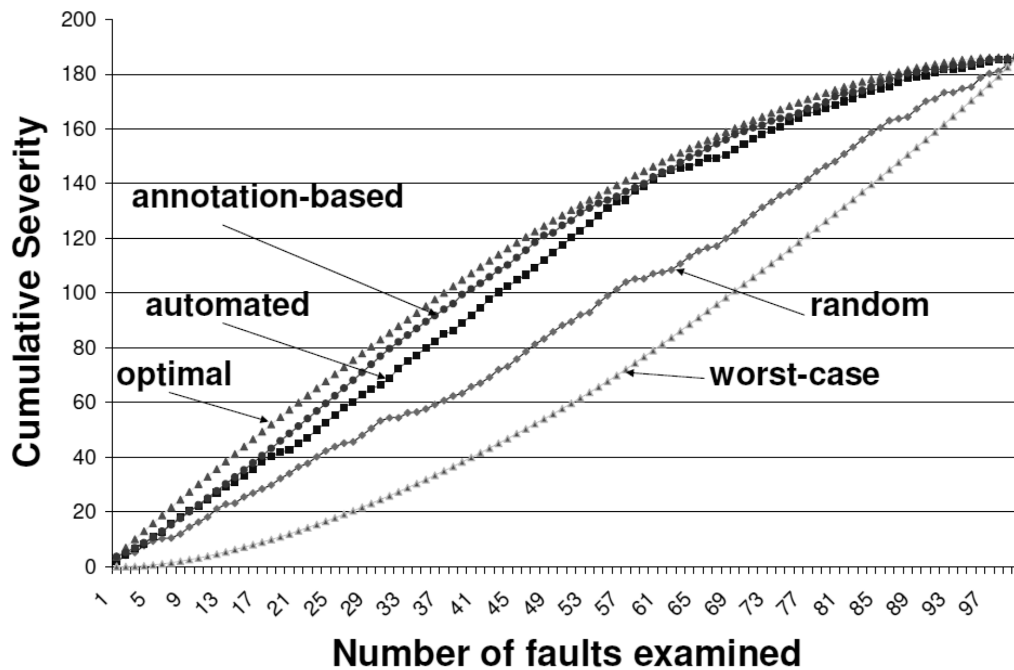

Figure 7.11: Cumulative severity over time when prioritizing faults by the automated and human-annotated models versus random priority, optimal priority, and worst-case priority.

7.5.2 Automated Model Performance

Figure 7.12 summarizes the automated model's performance on the dataset of 100 real-world faults. The automated model has a Spearman Ranking Correlation Coefficient of 0.78 with average human judgments, placing it between the manual model (0.84) and human themselves (0.70) in terms of agreement with the norm. While it drops in accuracy, in terms of correctly labeling fault severities, by 1% when compared to the human-annotated model, the SRCC scores are comparable, and exceed that of humans on average. The automated model missed no severe faults, and it is able to identify 39 out of 70 non-severe faults to be assigned a lower developer priority. While the human-annotated model correctly finds 61 such faults, note that it requires initial investment of humans manually examining each webpage output. The automated model is conservative in that it will generally label any fault with a medium-high or high severity as severe and requiring human attention. Consequently, it is able to achieve high accuracy with this approach, estimating true severities to within 0.75 (although the savings decrease by about a third, due to situations where

Model	Accuracy	Severe Missed	Non-Severe Correct
Automated Model	83%	0/30	39/70
Annotation-based Model	84%	1/30	61/70
Individual Human (avg)	59%	8/30	53/70
Always Average Rating	58%	30/30	70/70
Always Median Rating	59%	30/30	70/70
C4.5 Decision Tree	85%	5/30	65/70

Figure 7.12: Performance of the automated model against the human-annotation-based model and other baselines. *Accuracy* is the percentage of faults predicted within 0.75 of the average human rating. *Severe Missed* refers to the number of severe faults incorrectly labeled, *Non-Severe Correct* to the number of non-severe faults correctly labeled.

a medium-severity fault is predicted as severe and requiring developer attention, while still falling within the 0.75 cutoff). Figure 7.11 presents the cumulative severity over time when prioritizing the 100 faults in the dataset using the automated model and human-annotated model versus using random prioritization, optimal prioritization, and worst-case prioritization. Both the automated and manually-annotated models significantly outperform random prioritization, and closely approximate optimal prioritization.

This experiment demonstrates that the automated model has high accuracy with respect to finding severe faults, and can save developers resources by allowing them to prioritize the faults they will fix by severity. As explained in Section 7.3.2, the dataset has a relatively high proportion of severe faults (28%). It is likely that web applications experience severe faults as a small percentage of their total fault distribution, and therefore, the automated model may deliver even higher savings in industrial contexts where there are more non-severe faults than in the experiments in this chapter.

Additionally, the predictive power of the features was investigated by performing an analysis of variance [74] when predicting *high-severity* faults with the automated model. Figure 7.13 lists the significant features (cf. Figure 7.10). Again, web application faults that had *Code on the Screen* were significantly correlated with high severity faults. Similarly, *Cosmetic* faults were still unlikely to be considered severe.

Feature	Correlation	F	p value
Cosmetic	-	30.51	< 0.001
Functional Display	+	27.12	0.01
Code on the Screen	+	22.83	< 0.001
Code Error	+	5.32	0.02
Wrong Page	-	5.31	0.02

Figure 7.13: Analysis of variance showing feature predictive power for the automated model with no human annotation. F denotes the F-ratio, which is the square root of variance explained by that feature over variance not explained. Higher F values affect the model more. The last column denotes the significance level of F (i.e., the probability that the feature does not affect the model); values below 0.05 are significant.

7.6 Summary of Experiments

Consumer-perceived error severities have not been thoroughly studied in the context of web application testing. In Section 7.3.1, a human study was conducted on 400 real-world faults, 400 injected faults, and 100 non-faults to demonstrate that faults have varying severity distributions with an average standard deviation of 0.95. The results refuted the underlying assumption in fault injection that all faults are equally severe (a standard deviation of 0.0) and supported Hypothesis (H3).

Having established that faults have varying severities, Section 7.4.1 presented a predictive model of fault severity, that correlates strongly with consumer-judged severity, agreeing with humans more often than they agree with themselves. Because this initial model relies on human annotators to label surface features of faults, in Section 7.5 an automated fault severity predictor was introduced that operates without manual effort. Both the human-annotation- based and automated fault severity predictors have comparable levels of agreement in terms of labeling fault severity to that of humans, with SRCC scores of 0.84, 0.78, and 0.70 respectively, supporting Hypothesis (H4). Because the human-annotation-based model is more precise in terms of predicting fault severity than an average human, developers can use this model to replace human judgments of fault severity when prioritizing test cases, and in situations where such resources are unavailable, the automated model be used with a minimal loss of accuracy.

In a hypothetical defect report prioritization scenario, the automated model and annotation-based models we were able to correctly identify (and thus free developers from focusing on) 39

and 61 out of 70 non-severe defect reports, and thus 39% and 61% of all faults could be correctly de-prioritized. In industry application these savings could increase to 51% and 80%, respectively, due to the lower prevalence of severe faults in practice as reported in Section 7.3.2.

7.6.1 Threats to Validity

Although the annotation-based model outperforms humans at predicting consumer-perceived fault severity, and shows significant savings in terms of prioritizing severe and non-severe faults, it is possible that the mined faults are not indicative of web application faults in general. To mitigate this threat, a large number of benchmarks were chosen from varied domains and using heterogeneous languages. Other work [69] has examined a similar number of real-world faults to construct a web fault taxonomy, and many of the faults in the dataset were in the same equivalence classes. In addition, it is possible that the human study participants are not indicative of average consumers. For example, the population of undergraduate students may attach a different severity to shopping cart monetary miscalculations than would average consumers. Conversely, the feature annotation requires specific expertise (e.g., to distinguish between general *Error Message*s and *Database* ones) to form the basis of the model; to mitigate this threat a dozen experienced graduate students were used.

Simulating or capturing user experiences using screenshots with the *current-description-next* scenario idiom may also have been inaccurate. Although either screenshots submitted with bug reports, or carefully constructed screenshots given the description of the fault in the bug repository were used, it is possible that faults may not have been exactly replicated. Constructed screenshots erred on the side of conservatism, only introducing the error exactly as described and in the context of the application; for example, knowing to color an error message in red was not obvious unless such instructions were found in the text of a bug report. There are also classes of errors such a scenario cannot capture, such as an email not being delivered, or security vulnerabilities; neither the survey nor the models can speak to such errors. Such non-visible faults, however, are likely to compose a small percentage of faults overall: in a sample of over 200 real-world faults mined from the bug repositories of the benchmarks from this chapter, using the same mining methodology as

for the 400 real-world faults, 90% percent of the reported defects were user-visible.

In an effort to characterize the stability of the models in this chapter, the Appendix includes Section A.3 which grounds the dominant technologies in the current web development environment, and specifically the benchmarks used, with respect to the models. As these technologies evolve and change, it is possible that the models may need to be updated, although most of the features in Figure 7.8 are generic and not tied to any particular implementation.

Finally, the automated model assumes an existing, correct HTML output, which may not be readily available when a bug is reported to a developer through a repository. For developers fixing bugs by examining bug reports, given the overhead of reading a bug report, using the annotation-based model delivers high accuracy in terms of fault severity assigned with very little additional effort. Using the automated model is most appropriate in regression testing settings where the oracle output is immediately available.

7.7 Related Work to Studying Web Errors and Severity

Causes of failures in web applications have been examined by Pertet and Narasimhan [81]. Software failures, operator error, hardware and environmental failures, and security violations were identified as four failure categories, with significant causes including system overload, resource exhaustion, complex fault recovery routines, and system complexity. Although manifestations of failure, such as partial or total site unavailability, system exceptions, incorrect results, data loss, and performance slowdown, were identified, no attempt was made to assign severities to failures. By contrast, the models in this chapter assign consumer-perceived severities to web application faults.

Strecker and Memon examine the relationship between faults, test suites, and fault detection for GUIs [108]. They also note that it is a common research practice to assume all faults are equally severe, and that determining what a truly representative fault set looks like is difficult. Elbaum *et al.* [40] try to account for factors such as fault severity as a metric in test suite construction, but they give no guidelines for measuring fault severity besides the time required to locate or correct a fault, lost business, or damage to persons or property — figures which are difficult to calculate in general

or in advance. Ma and Tian present a defect classification framework to analyze web errors and identify problematic areas in the context of reliability improvement [67]. Their research relies on web server logs to extract information, rather than studying browser output. Although they mention defect severity as a classification attribute, like Elbaum *et al.* they provide no guidelines for how to measure this feature. The work in this chapter provides annotation-based and automated models to accurately assign consumer-perceived severities to web application faults.

Ostrand and Weyuker examine faults distributions in large industrial software systems, under which fault severities were assigned according to fix priority [79]. In follow-up work [80], they discovered that such developer-reported severities were highly subjective, and often inconsistent, inaccurate, or motivated by political considerations. Ultimately, they rejected using such a developer-reported fault severity measure in their fault localization predictor due to these concerns. The work presented here relies on consumer-perceived fault severities to avoid many of the problems associated with self-reported developer assignments.

Zhou and Leung analyze object-oriented design metrics for predicting fault severity. They discovered that features such as methods per class, coupling between object classes, and lack of cohesion in methods were statistically significant across fault severity in their exploratory study. The models in this chapter relies on HTML output, rather then web application source code, to predict fault severity. In addition, their study [123] may suffer from subjectivity problems in reporting fault severity [80], which can be minimized by relying on consumers to judge fault severity rather than developers.

7.8 Summary

Web applications are often untested due to extreme resource constraints during their development [86]. Providing a consumer-perceived error severity model can allow developers to prioritize errors according to their likelihood of impacting consumer retention, and thus encourage web-application developers to test more effectively. Obtaining 12,600 human judgments of 400 real-world faults, 400 injected faults, and 100 non-faults led to the discovery that relying on a single

human observer to judge the severity of a particular fault is inherently unreliable. Consequently, this chapter presented two models of error severity that outperform humans in terms of accurately predicting the average severity of web application errors. The first model relies on human annotations of error surface features, and successfully identified 87% of non-severe faults to be assigned low priority in the experiments. A fully automated model was also presented, that can obviate examining 55% of such faults. Both models are significantly better than humans at flagging severe faults for examination, and can therefore replace or augment humans when assigning fix priorities to faults encountered in web application development and testing.

The main thesis of this dissertation is that user-visible web-based application errors have special properties that can be used to improve the current state of web application error detection, testing and development. This chapter showed how the consumer-perceived severity of an error can be considered during test case prioritization, saving significant effort while retaining consumers. Having a successful predictive model of consumer-perceived error severity can also lead to the exploration of ways to map software engineering practices, testing techniques, and current technologies to high-severity errors, which is explored in the following chapter. Specifically, Section 8.5 explores applying the automated model of error severity introduced in this chapter to test suite reduction approaches for web applications.

Chapter 8 Addressing High Severity Errors in Web Application Testing

The previous chapter hypothesized that a model of error severity can be used to guide testing techniques towards prioritizing high severity errors, thereby increasing the perceived return on investment for testing in the web domain. This chapter expands upon those results by applying the error severity model in techniques to reduce the introduction of high severity errors during application development, as well as further reducing the cost of testing web applications by focusing on revealing high severity errors during test case design, selection, and prioritization.

Although web applications are highly human-centric, consumer-perceived error severity has not been systematically approached as a metric for selecting development and testing strategies in this domain. This chapter will provide concrete guidelines to increase the perceived return-on-investment of testing, and show that even coarse-grained, automated test cases can detect high severity errors. Specifically, Hypothesis (H5), that

> *there exists a statistically significant correlation (SRCC > 0.60 [44, 107]) between severe errors in web applications and various software engineering aspects of web application development,* (H5)

is evaluated. Developers can also mitigate consumer perceptions of error severity by presenting errors using specific idioms that minimize disruptions to application interaction. Finally, the trade offs between various user-session-based test suite reduction approaches for web applications and the severity of uncovered faults will be examined. Hypothesis (H6), that

> *there exist test suite reduction strategies that expose at least 90% of the severe errors found via corresponding retest-all approaches for web applications,* (H6)

is also evaluated in this chapter; even modest testing approaches are able to provide significant predicted gains in consumer satisfaction by focusing on flagging and preventing high severity errors.

8.1 Strategies For Addressing High Severity Errors in Web Testing

The study of consumer-perceived high-severity faults in web applications can be used to direct development and testing strategies towards their reduction in the context of such demanding development circumstances. In the human study from Section 7.3, several observations can be made about severe faults in web applications, which translate into implementable software engineering guidelines. In addition, the distribution of consumer-perceived fault severities is demonstrated to be *independent* of the type of application. Neither the application type (e.g., shopping cart or online forum), nor, to a lesser degree, the technologies used (e.g., PHP or ASP.net) are predictors of high or low severity faults. Similarly, certain fault *features*, distinct from the context of the fault, are shown to be related to the consumer-perceived severity of these faults. Given the same defect in the source code, the presentation of the fault to the user (e.g., via a stack trace or a human-generated error message) influences the likelihood that the consumer will return to the website in the future. Finally, the *kinds* of defects that are associated with faults of varying consumer-perceived severity levels are explored. For example, many severe faults are due to configuration errors that are easily detected by running a simple test suite, while less severe faults may require changes to specific lines of application code, and can be more difficult to detect.

This chapter further discusses such observations and supplements them with concrete suggestions that developers can implement. These contributions provide the foundation for guidelines that assume few resources will be allocated to testing. In situations where web applications are already being tested, the trade offs between various test case reduction (see Chapter 2) methodology costs and the severities of the faults these techniques uncover in the context of user-session-based testing are studied. One goal is to provide developers guidance towards choosing an optimal testing approach when performing automated testing. This chapter examines the trade offs between various test suite reduction methodologies [105] in terms of the consumer-perceived fault severities revealed by each approach.

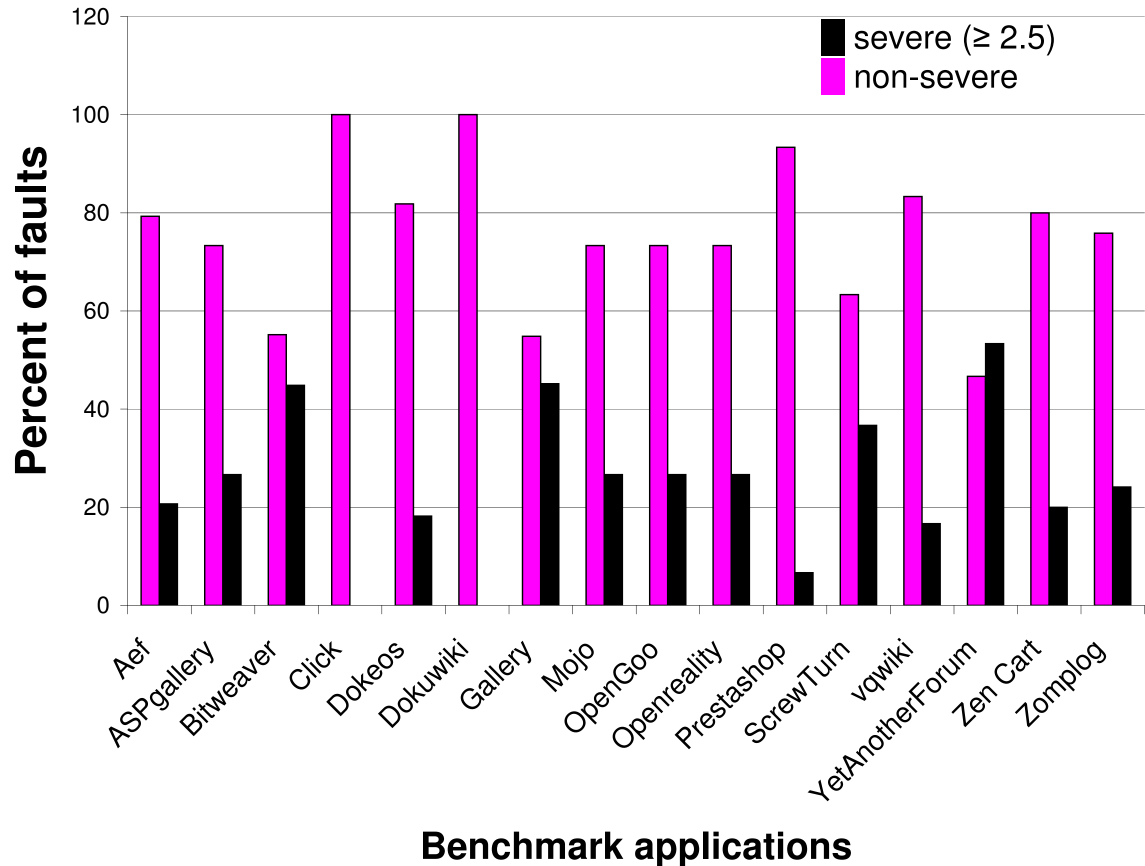

Figure 8.1: The breakdown of severe faults for each benchmark application. Each application is individually normalized to 100%. There is no apparent pattern between the type of application and its fault severity distribution (see also Figure 8.2).

8.2 Fault Severity Distribution by Application

The study of 400 real-world faults in the previous chapter revealed an approximately even distribution of low, medium, medium-high, and severe faults (labeled with average ratings of ≤ 1, > 1 and ≤ 2, > 2 and < 2.5, and ≥ 2.5 respectively, see Figure 7.1) within each of the benchmarks. Although lower-severity faults are likely underrepresented in this population, because they are more frequently not reported or recorded in the bug repositories faults were mined from, the relatively high number of severe faults that were witnessed provides a large data set of severe faults to study. Recall that severe faults are most likely to translate into consumer losses (see Figure 7.1).

Feature	SRCC with high severity
is written in PHP	0.16
is written in ASP.net	0.32
is a Gallery	0.38
is a Wiki	0.35
is a Forum	0.34
is a Content Mgmt. System	0.30
is E-commerce	0.22

Figure 8.2: Spearman's Ranking Correlation Coefficient (SRCC) between an application feature and the severity of its faults. An SRCC of 0.3–0.5 is considered weak, while 0–0.3 indicates little to no correlation [44].

First, correlations between high severity faults and either the type or the web application (e.g., a shopping cart versus a forum), or the underlying technologies involved (e.g., PHP or ASP.net) were analyzed. Figure 8.1 presents the distribution of fault severities across the seventeen benchmarks from Figure 7.2, normalized to 100% for each application. Figure 8.2 shows the Spearman's Ranking Correlation [44] between various features, such as the programming language used or application type, and consumer-perceived fault severity. Overall, the application features all hover between no correlation and very weak correlation with high severity faults. While a limited argument could be made that ASP.net and image gallery applications are slightly more likely to have high severity faults, the results generally refute the hypothesis that certain application types have higher user-perceived fault severities. For example, among benchmarks in the dataset with at least 30 faults, both the benchmark with the highest number of severe faults (ZEN) and the lowest number of severe faults (PRESTASHOP) were e-commerce, shopping-cart based applications. Similarly, the choice of development language and infrastructure in these benchmarks did not strongly correlate with fault severity.

8.3 Fault Features Related to Severities

The primary goal of this chapter is to provide empirically-backed recommendations to help developers produce reliable web applications under the assumption of limited testing resources. This

Feature	Description
Same Page	The error is visible within the same page and application (imagine a website with frames); the title, menu, and/or sidebars stay the same
New Page	A page is loaded that does not look like other pages in the application; examples are blank pages or server-generated error messages
Generic Error Message	A human-readable wrapper around an exception, which frequently provides no useful information about the problem
Popup	The error resulted or was displayed in a popup
Server	The error was a standard server-generated complaint, such as an HTTP 404 or 500 error
stack trace	A stack trace or other visible part of non-HTML code
Other Error Message	Text exists on the page indicating there was an error (as opposed to a missing image or other "silent" fault)

Figure 8.3: Context-*independent* fault features.

section demonstrates that faults of varying severities can be classified according to both contextual and context-independent characteristics.

Contextual features are defined to be visual stimuli or use-case based characteristics exposed to the user as part of the fault manifestation. Contextual features are tied to the underlying origin of the defect in the source code. Examples of such visual stimuli include stack traces, missing images, and small cosmetic errors. Use-case based features associated with faults include authentication, permission, or upload scenarios. The previous chapter used such contextual features to assign severities to faults in web applications (see Figure 7.8).

Context-independent features, by contrast, can be viewed independently of the actual context of the fault. Given the same source-level defect, the multiple ways the fault may be presented to users are the various context-independent fault manifestations. Context-independent features, summarized in Figure 8.3, include displaying the fault on the same page as the current page (in a "frame" style where the header, footer, and side menu bars are preserved), loading a new page that is visually different from the normal theme of the web application, wrapping the fault in a generic

or customized human-readable error message, popups, server-generated error messages (such as HTTP "404 not found" errors — see Section 2.1.2), or displaying a stack trace.

Contextual fault characteristics can highlight possibilities for focusing testing on certain parts of the source code. Context-independent features, by contrast, can be translated into opportunities to decrease the perceived severity of faults, regardless of their origin.

In terms of contextual features (from Figure 7.8), a deeper analysis of the human study data from Section 7.3 reveals that errors presented with a stack trace were viewed as most severe, followed by database errors with visible SQL code, authentication problems, and then error messages in general. Cosmetic errors, such as a typographic mistake, that do not affect the usability of the website were perceived as having the lowest severity, followed by form errors such as extra buttons. Although minor faults, such as typos, formatting issues, and form field problems are often easier to trace in the source code, in that there can be few to no business logic defects involved with these kinds of faults, finding such trivial errors can actually be potentially more challenging when using traditional testing approaches in a web application environment. Imagine a test suite with oracle output exists for an old version of a web application. When the application undergoes development, the HTML output will most likely change, and it becomes difficult to distinguish between faulty output and harmless program evolutions, especially with automated tools (see Section 5.1). By contrast, severe errors that present with a stack trace or errors messages on the screen are more easily identified, due to the relative larger difference between expected and actual output. These results highlight the benefits of using even simple, automated testing approaches that search for error keywords [27] in that keywords are likely to quickly and reliably identify high-severity faults. The issue of fault causes is further discussed in Section 8.4.

Modifying the context-independent features of consumer-visible faults is an orthogonal approach to testing strategies in the arena of consumer retention. Although the visual presentation is not the only deciding factor with respect to consumer-perceived fault severity, various fault presentations are associated with different severity levels. When programming defensively, developers often have the option of managing the way faults are presented to users. Consider the case of the inability to upload an image; the set of real-world faults used in the human study includes thirteen

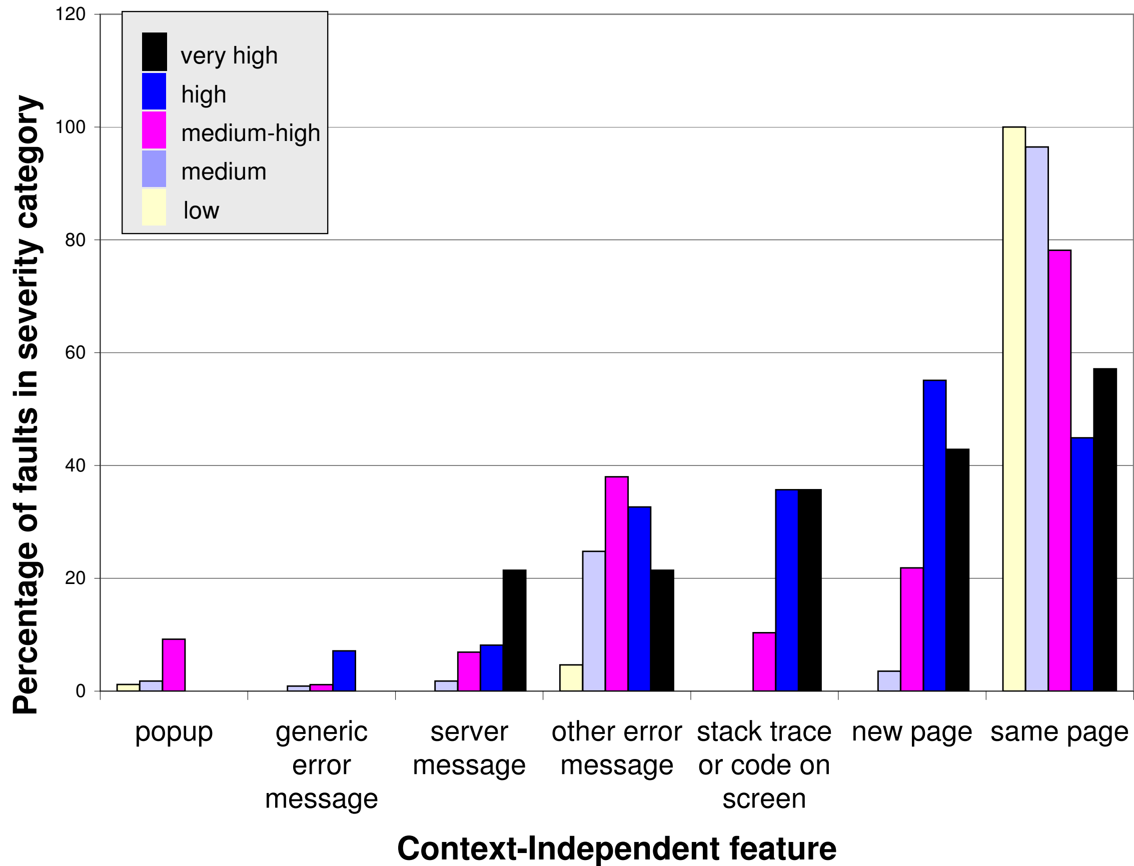

Figure 8.4: Severity distribution as a function of context-*independent* fault characteristics.

such instances. In every case where the fault was judged as severe (5 out of 13 instances), the faulty page was displayed with either a stack trace or a server generated error message, in a new page (i.e., a webpage with a different header, footer, sidebar menus, and layout than the original website template). When the same fault was judged as medium-high severity (4 out of 13 times) a stack trace was present, but in half of those pages the page layout did not change. When the fault was judged with only medium severity (the remaining 4 out of 13 times), the page layout remained the same, and a stack trace was usually absent. This example demonstrates how developers can reduce the perceived severity of such faults by preventing the appearance of the page from changing and wrapping the fault in an error message displayed on the same page.

Figure 8.4 presents the visual, context-independent characteristics of the real world faults in the

study, broken into five fault severity groups (very high faults have severities ≥ 3). Each feature is shown with the corresponding percentage of total faults of all severity ratings that fell into that category. For example, about 10% of medium high severity faults were presented as popups. Because a fault can occur either on the *same page*, or on a *new page*, these two features are mutually exclusive and therefore each severity category will total to 100% between these two items; this is not the case for the remaining features as not all faults had to exhibit any of these conditions. In general, faults that occur in the *same page*, as opposed to loading a new and visually discordant page, are much more likely to be judged as lower severity. The converse is also true: faults with *new page*s are associated with increasing severity ratings. The analysis of the study revealed that the worst way to present faults to consumers is in the form of a stack trace; wrapping the fault in an error message was associated with a lower severity. Server-generated error messages were also poorly received. Error messages that were displayed as popup windows were regarded as the least upsetting to consumers. Developers conscious of the impact on consumer-perceived severity of various fault presentations can therefore choose options such as maintaining the same page appearance and relying on popups that are associated with higher consumer satisfaction.

8.4 Fault Causes Related to Severities

In this section, the causes of errors are analyzed, independently of their visual or use-case context, to associate faults of varying severities with different components in the code or environment. The goal is to provide developers with ways to target web application design and testing to reduce the frequency of high severity faults by focusing on the potential causes of defects. In the analysis of the human study data, nine recurring causes of defects were identified, summarized in Figure 8.5. Recall that all defects in the study were taken from real-world bug reports filed in bug databases (see Section 7.3.1).

Figure 8.6 shows the fault severity distributions associated with the causes from Figure 8.5. Severe faults are frequently associated with unhandled NULL objects, missing files or incorrect upgrades, database issues, and incorrect configurations. The lower the severity of a fault, the more

Cause	Description	SRCC with high severity
Database	An error in the database configuration or structure	0.66
SQL	A buggy SQL query that lead to an exception	0.69
NULL	An empty code or database object which lead to an exception	0.69
Source Code	An error due to incorrect logic in the source code	0.18
Config	Configuration settings were inconsistent	0.68
Component	A third party component was incompatible or caused an error	0.62
Upgrade	A file was missing, or a recent upgrade caused an error	0.63
Permission	The operating system failed to allocate resources or open files	0.68
Server	Incorrectly configured server	0.68

Figure 8.5: Common defect causes from the four hundred real-world faults in the human study. The SRCC column gives the Spearman correlation between faults having that cause and high severity (≥ 2.5). An SRCC above 0.5 is considered moderate to strong correlation [44].

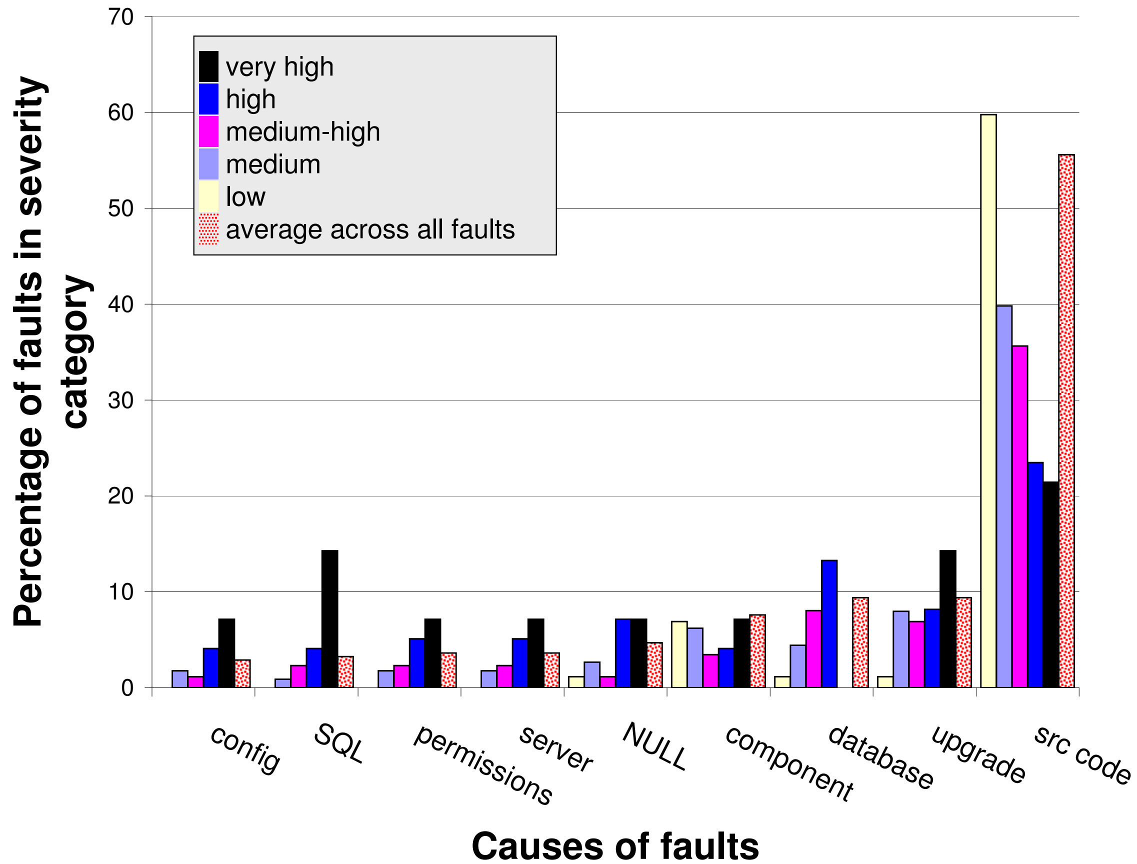

Figure 8.6: Fault severity as a function of underlying defect causes.

likely it was due to erroneous logic in the application source code not associated with the database or NULL objects. These results are consistent with those of Figure 7.10, in that errors that frequently manifest as stack traces are due to unhandled exceptions, and database access problems are associated with displaying SQL code on the screen. As explained in Section 8.3, severe faults that are due to exceptions or configuration issues are often easier to detect with even a coarse-grained, automated test suite, because the difference between the expected output and actual application output is more dramatic.

Finding logic errors in program source code poses a greater challenge, as it is less likely that any individual test case will exercise any single line of code, and the difference between the oracle and actual test output may be difficult to recognize as a fault instead of a harmless program evolution

(see Section 5.1). Once again, even a simple, naïve test suite that uses keywords to detect faulty output can be used with a high return on investment when seeking high severity faults. Approaches that orthogonally address the prevention of unexpected NULL objects [38] and database testing rigs [34] can be used in conjunction with such modest testing techniques to successfully prevent or flag proportionally more severe faults.

8.5 Test Case Reduction And Severity

The previous sections tied severe faults to different types of visual stimuli, use cases, and defect causes, supporting Hypothesis (H5). Even a simple test suite which looks for error keywords or large differences between expected and actual test case output may thus capture a large percentage of high severity faults. This section presents further empirically-backed recommendations, assuming that an organization has some resources to invest in more rigorous testing approaches, but still would like to minimize the resources required.

Consider a scenario in which user-session-based testing (see Section 3.2.1), a kind of capture-replay technique that is largely automatic, is currently implemented as a company's testing methodology. This approach works by recording user accesses through the server and replaying them during testing [39, 101]. Although a server needs only small modifications to log such sessions, this type of testing has the drawback that large volumes of session data are captured. Replaying all recorded user sessions is often infeasible, and *test suite reduction* with user-session-based testing has been studied extensively [41, 51, 65, 93, 105]. The goal of test suite reduction is to select a subset of all user sessions to replay that will reveal the largest number of faults during testing. While the trade offs between fault detection and test case size have been explored for web applications [93, 105], the severity of the faults uncovered by various reduction techniques has yet to be addressed. In particular, this section will present empirical data to support Hypothesis (H6).

8.5.1 Reduction — Experimental Setup

To measure the severities uncovered by various test suite reduction techniques, 90 faults were manually seeded in three PHP applications in Figure 8.7 denoted by an asterisk (an online store, a forum, and a real estate website). The faults were equally distributed among the applications and were seeded according to the methodology of Sprenkle *et al.* [105]. One hundred and fifty user sessions were then collected from volunteers asked to interact with each application in a typical manner. A test case is defined here to be all the `URL`s in one user session, and each test case was run on a faulty version of a benchmark with one fault injected at a time. All three web applications had database components, the states of which were saved at the beginning of each user session so that when a test case was replayed, it operated on the same relative state.

Various user-session test suite reduction strategies explored by Sprenkle *et al.* [105] were selected to be implemented: *retest-all*, *Harrold-Gupta-Soffa*, and *Concept*. The *retest-all* strategy does not reduce the size of the test suite and serves as a baseline for fault detection. *Harrold-Gupta-Soffa* is a general technique [51] that uses a heuristic which selects a subset of the original test suite by approximating the optimized reduced set (a NP-complete problem). The algorithm chooses test cases from the original test suite one at a time, always choosing the test case that will cover the most untouched `URL`s next. *Concept* is Sprenkle *et al.*'s orthogonal approach that builds a concept lattice where each node is a test case that inherits the attributes from all previous nodes. The lattice therefore constructs a partial ordering between all test cases, based on the `URL`s each test case covers. The parent nodes of the bottom of the lattice represent the minimal set of test cases that will cover all `URL`s using this approach. Two readily-available tools were adopted, `concepts` [45] and `RAISE` [46], to implement the *Concept* and *Harrold-Gupta-Soffa* approaches, respectively.

Both methodologies need to associate requirements with each test case, where a requirement is some desired coverage property. The experimental setup in [105] was followed, taking a test case to be user session composed of `URL`s in a specific order, and its requirements to be the respective `URL`s each user session exercises. Because `URL`s frequently contain form data as name-value pairs, Sprenkle *et al.* [91, 105] was followed by examining the `URL`s independent of these values. For example, `http://example.com/order.php?sku=12&id=11` and `http://example.com/order.php?id=15`

are considered the same URL and therefore the same requirement, because this approach ignores all name-value pairs after the ? in the URL. Conversely, the OPENREALTY benchmark used the same PHP page for almost all requests, and specified actions via arguments such as `do=Preview`. For this benchmark the value of this action name-value pair only was considered in mapping the URL to a requirement.

The fault detection ability of each test suite was tested by cloning each benchmark into 30 versions with one seeded fault each, and running each test suite on each cloned version. The faulty output and expected output were collected for each faulty version of source code by customizing a `diff`-like tool to ignore data such as timestamps and session tokens that would be different between even correct versions of output. Comparing the HTML output of web applications in regression testing is a known problem, due to the inability to distinguish between erroneous output and harmless program evolutions, and has been addressed through partially- and fully-automated tools (see Section 5.1). Although SMART, the highly-precise oracle comparator presented in Chapter 5 and Chapter 6 could have been applied to detect erroneous output, using a customized `diff`-like comparator eliminated false positives and false negatives that may have occurred with such a model.

After collecting the faulty and expected versions of output, the consumer-perceived fault severity of each defect was then measured: the automated formal model derived from the user study in the previous chapter was used to accurately predict the severity between a faulty HTML output and an oracle output.

8.5.2 Reduction — Experimental Results

Figure 8.7 shows the number of test cases (user sessions) for each test reduction methodology in this experiment, and the number of uncovered faults of varying severities. Because the severity of a fault depends on the concrete manifestation of the fault in HTML, rather than source code, the same fault may materialize with different severities on different URLs. For example, a fault in a line of code that accesses database values may display a stack trace when exercised from certain URLs, but may only show a warning in others. Therefore, Figure 8.7 shows the number faults in each severity category when considering the average severity rating for any HTML manifestation of a particular

Method/Benchmark	Test Cases	Low	Medium	Medium-High	High	Total
retest-all PRESTASHOP	50	0	3	24	3	30
HGS PRESTASHOP	8	0	3	24	3	30
Concept PRESTASHOP	27	0	3	24	3	30
retest-all OPENREALTY	50	1	3	1	23	28
HGS OPENREALTY	15	1	3	1	20	25
Concept OPENREALTY	40	1	3	1	23	28
retest-all VANILLA	50	5	22	2	1	30
HGS VANILLA	4	5	22	2	1	30
Concept VANILLA	9	5	22	2	1	30

Figure 8.7: Fault severity uncovered via reduced test suites. The "Method" is either *retest-all* (the baseline), *HGS* [51] or *Concept* [105]. The "Test Cases" column counts the number of test cases in the reduced suited produced by that methodology (out of 50). The "Low" through "High" columns count the number of faults exposed in each such severity level. The "Total" column gives the total number of faults across all severities exposed by each technique on each benchmark application.

fault.

Previous work has shown both the *Concept* [105] and *Harrold-Gupta-Soffa* [51] test suite reduction methodologies to be highly effective at maintaining the fault exposure properties of the original *retest-all* baseline — when all faults are treated equally. This experiments shows that these techniques are also effective when fault severity is taken into account. Reduced test suites were able to match the same number of exposed faults in cases where the faults were generally of medium-high (PRESTASHOP) and medium (VANILLA) severity. For the benchmark that happened to be seeded with mostly severe faults (OPENREALTY), both test suite reduction approaches had comparable results to the baseline. Although this property of effective fault exposure may not generalize beyond these benchmarks and more experimental work is required, this experimental setup seeks to reproduce the results of existing test suite reduction techniques with fault severity in mind; test suite reduction of user-session based test suites is an effective way to maintain high levels of severe fault exposure while reducing costs in this study.

Test suite reduction techniques may also want to take advantage of various properties of severe fault localizations in web applications. Figure 7.10 showed that both database and authentication errors were highly correlated with severe faults. Consequently, test cases as URLs that exercise parts of the authentication process may be selected to be assigned special priority in test suite reduction approaches in this domain. Similarly, test cases that are known to interact with the database can also be considered more likely to reveal severe faults.

8.6 Threats to Validity

Because the results detailed in this chapter depend on the human study and fault severity model of the previous chapter, the threats to validity from the previous chapter are also inherited here. For example, it is possible that the conclusions drawn from this dataset do not disseminate to web applications in general, although an effort was made to select open-source real-world benchmarks from a wide variety of domains and technologies.

It is also possible that because fault localization and manifestation conclusions were drawn

from bug repositories, that the underlying cause of, or actual visual representation of, a fault may be inaccurate or incomplete. To lessen this danger, fault causes were conservatively assigned; for example, either the bug description or the actual HTML rendering had to provide reasonable indication that a NULL object was being mishandled, for the fault to be classified as due to NULL in Figure 8.6. It is conversely plausible that other faults were also due to NULL issues that were not so labeled because not enough information was provided to make this judgment. Although exact knowledge of fault localization may change the actual data values in Figure 8.6, it is unlikely that the overall trends witnessed in this section would be significantly impacted; there is no reason to believe that missing labels (such as not knowing something was due to a NULL issue) would not be relatively evenly distributed along all severity categories, thereby leaving the overall trend the same.

8.7 Related Work to Web Failures, Severity, and Test Suites

The prevalence of failures in web application has been widely studied. A 2005 survey identified 89% of online customers encounter problems when completing online transactions [50, 81]. User-visible failures are common in top-performing web applications: about 70% of top-performing sites are subject to user-visible failures within the first fifteen minutes of testing [112], a majority of which could have been prevented through earlier detection [98].

A number of preliminary web fault taxonomies have been proposed. Guo and Sampath identify seven types of faults as an initial step towards web fault classification [47]. Marchetto *et al.* validate a web fault taxonomy to be used towards fault seeding [69], using fault characteristics such as level in the three-tiered architecture the fault occurred on or some of the underlying, specific web-based technologies (such as sessions). In these fault classifications [47, 69] there is no formal concept or analysis of severity; some categories of faults may produce more errors that would turn customers away, but this consideration is not explored. This chapter also divides up faults according to fault localization, but is able to associate consumer-perceived severities with different sources of faults.

Di Lucca *et al.* have also explored reduction of user-session test suites in [65], where they mea-

sure coverage of URLs as well as Built Client Pages, which are dynamically built pages generated from user input and application state data. Rather than only considering URLs as coverage criteria, they rely on static and dynamic analysis of the web application to generate a reduced test suite smaller than those of *Concept* [105] for their benchmarks. This chapter focused on *Concept*'s and *Harrold-Gupta-Soffa*'s [51] reduction techniques because they rely on simple analysis of URLs in collected use cases; in future work this study could be extended to reduction techniques such as Di Lucca *et al.* have explored.

8.8 Summary

Formal testing of web applications is frequently a casualty of the extreme resource constraints of their development environments [86]. At the same time, web applications are highly human-centric, and consumer retention is known to be low [76]. This chapter analyzed the results of a study of the consumer-perceived severity of 400 real-world web application faults (from Section 7.3.1) with the explicit goal of providing concrete software engineering guidelines to increase the perceived return-on-investment of even naïve testing approaches. Given the same defect in the source code, different visual presentations of the fault to consumers were found to have different impacts on their perceived severity. Controlling fault presentation by opting for pop-ups and error messages over stack traces and changing page layout is a simple and effective way to reduce the consumer-perceived severity of faults. In studying the causes of various types of faults, the utility of coarse-grained test suites that only detect relatively large differences in HTML output or rely on error keywords were revealed as effective ways of capturing many high severity faults, encouraging all developers to invest in at least some minimal testing infrastructure. Finally, the trade offs between the costs of various user-session-based test suites for web applications were examined, finding that reduced test suites were effective at maintaining fault exposure properties across all fault severity levels.

The main thesis of this dissertation is that user-visible web-based application errors have special properties that can be used to improve the current state of web application error detection, testing

and development. This chapter explored the hypothesis that consumer-perceived error severity can be used to increase the perceived lack of return on investment within the demanding development environment, by focusing on retaining consumers through the prevention of severe errors; even simple testing approaches are able to achieve significant gains in consumer satisfaction. Having presented an automated model of consumer-perceived error severity in Chapter 7, this chapter explored practical applications of such a model in terms of software engineering guidelines as well as test suite comparison. The following chapter will similarly explore applying the error severity model in an applied setting by measuring the error severity exposure properties of SMART, the oracle comparator from Chapter 6, on popular web applications.

Chapter 9 Combining Error Detection During Regression Testing with Error Severity

The previous four chapters explored techniques for reducing the costs associated with regression testing web-based applications through automated oracle comparators that rely on both the structure of HTML/XML output as well as the predictable way in which web applications evolve and fail, and the consumer-perceived severity of user-visible errors in this domain. This chapter explores the trade offs between using such an automated oracle comparator and the severity of the errors uncovered and overlooked by such an approach. Specifically, Hypothesis (H7), that

at most 1% of the false negatives produced by the proposed highly-precise, fully-automated oracle comparator correspond to severe errors, (H7)

is tested on a set of popular, open-source web applications. Automated oracle comparators remain a cost-effective approach for regression testing under these circumstances, despite the heavy use of non-deterministic output (such as session cookie identifiers) by applications. This chapter shows that the oracle comparator in Chapter 6 will primarily fail to flag non-severe errors.

9.1 Motivation

Chapter 6 presented a fully automated approach using SMART, a highly-precise oracle comparator, to reduce the number of false positives associated with regression testing web-based applications while minimizing or eliminating false negatives. This chapter presents SMART's performance on three popular, open-source PHP web applications. Although the model's performance on two web applications was evaluated in Section 6.3.2, CLICK and VQWIKI, the goal of this chapter is to investigate SMART's performance on additional and potentially more typical and challenging benchmarks. For this purpose the model's performance was evaluated on three open-source popular web applications summarized in Figure 9.1.

Benchmark	Versions	LOC	Description	Training Faults	Testing Faults	Test Suite
PRESTASHOP	v1.1.0.55	155K	e-commerce (shopping cart)	0	431	83
OPENREALTY	v2.5.6	185K	real estate listing management	4506	62	33
VANILLA	v1.1.5a	35K	web forum	895	462	48
Total		375K		5401	955	164

Figure 9.1: Additional web application benchmarks.

The first benchmark, PRESTASHOP, is an e-commerce application with over 24,000 companies deploying their own instances of the product worldwide [16]. Besides featuring authentication and database properties, PRESTASHOP is interesting from a regression testing perspective because some of its customer-facing pages include a featured product that might change between page views. OPENREALTY is an online real estate listing management application with over ten thousand registered members in their development forum [15]. VANILLA is a standards-compliant, multi-lingual, theme-able, pluggable discussion forum for the web with “over 300,000 businesses, brands, and fans” [17]. All three PHP applications make use of session cookies that result in additional non-deterministic output that would be flagged by a naïve comparator such as `diff`.

In an effort to provide a crisper analysis of SMART’s fault revealing properties for these popular web applications, manual injection of source code faults (see Section 2.2.3) for each benchmark is employed, rather than running the test suite on two different versions as in previous sections. Although this setup no longer provides an opportunity to ignore natural program evolutions, this experimental design was chosen to know with certainty whether or not the faults that should be flagged are actual errors. This is in contrast to the *potential* faults SMART sought to flag in previous chapters. Furthermore, the heavy use of non-deterministic output in these benchmarks provides a challenge similar to that of ignoring harmless program evolutions in the previous experiments.

The *Testing Faults* column of Figure 9.1 gives the number of manually injected faults SMART should detect. Because these are known faults, rather than potential faults as flagged by a human annotator, the experiments in this section are expected to have more false negatives. Therefore, this section characterizes the *consumer perceived severity* of correctly flagged and missed faults.

If the severity of missed faults is not high, using SMART under the automated approach is still a useful investment to developers, even if some bugs are missed, given the resource constraints of development in this domain. Specifically, the use of such a highly-precise oracle comparator in this artificial setting, when seeking to identify injected faults in one source code version, is considered successful if on average 99% of the severe faults are correctly identified (see Hypothesis (H7)), and if on average at least 70% of non-faults are correctly labeled as such.

9.2 Feature Analysis

Before evaluating the performance of SMART in an experimental setup with known faults, to measure the consumer-perceived severity of defects that the oracle comparator fails to flag, the hypotheses about feature associations with faults and non-faults presented in Section 5.2.2 are first explored. Figure 9.2 presents the results of analysis of variance [74] experiments conducted on all of the test applications from Chapter 6 plus the three PHP benchmarks from this chapter added in. Each value represents the coefficient of the feature in the model — higher values are more strongly associated with errors, for features whose p-value is less than 0.05. The F values are not shown, and for the training dataset, only those features are shown which overlap with at least one significant feature from one of the testing benchmarks.

Although several analyses of variance for feature significance were conveyed in the dissertation thus far, this section examines the results qualitatively in detail, tying in feature importance to concrete software features remains to be explored. A deeper analysis of delete, move, text-only, and error keywords features is conducted in this section, beginning with their significance in the representative PHP benchmarks, and extending the analysis to all test applications visited in this work.

9.2.1 Moves and Deletes

Of all the features SMART employs, DIFF-X-move and DIFF-X-delete were the only two shared by every test benchmark, as seen in Figure 9.2. In addition, the F-value (not shown) for at least one

Benchmark	move	insert	delete	text only	group	child	missing attr	inversion	new text	depth	error keywords	text ratio	group binary	functionality
PRESTASHOP	+0.084	-0.004	+0.009	-0.368		-0.008	-0.515							
OPENREALTY	+0.008		+0.036	-0.388								+0.576		
VANILLA	-0.002	+0.023	+0.024	-0.326	-0.025	-0.003								
HTMLTIDY	+0.001	+0.050	-0.025	-0.919	-0.121			-0.061	-0.010		+0.220	+0.004	+0.553	+0.630
GCC-XML	+0.000		+0.005	-0.163	-0.013			+0.000		-0.011			+0.835	
CLICK	+0.000	+0.000	+0.000	+0.000	+0.000	+0.000		+0.000		+0.000	+1.000	+0.000		
VQWIKI	+0.088	+0.000	-0.000											
TRAINING	+0.002	+0.029	+0.029	-0.288	-0.012	-0.002	-0.048	+0.001		-0.000	+0.174	-0.007	+0.714	-0.019

Figure 9.2: Coefficients of significant ($p < 0.05$) feature values across all test benchmarks, plus the generic training dataset.

of these features is always either the highest or second-highest for each specific benchmark model, indicating that these features significantly affected the respective model.

Given the importance of DIFF-X-move and DIFF-X-delete in the oracle comparator models, the goal is to characterize the nature of the faults these features tended to predict, as they were generally indicative of faults across most benchmarks. To do so, the HTML output for test cases in the dataset that were labeled as having high DIFF-X-move and DIFF-X-delete values were manually examined, as well as those with low DIFF-X-move and DIFF-X-delete values. For the three PHP web applications, high DIFF-X-delete values indicated large chunks of expected output were missing. For example, in VANILLA an output pair with a high DIFF-X-delete feature value typically corresponded to output in which all forum comments are gone, or in which a search returned no results. In OPENREALTY, high DIFF-X-delete values were associated with missing a large loan calculator form, while in PRESTASHOP, they indicated blank pages that contained a single error message such as "Error: install directory is missing". Similarly, high DIFF-X-delete values in VQWIKI and CLICK, the other two web applications, were instances of pages not being found or missing the entire body of data. By contrast, low DIFF-X-delete values were associated with less severe errors such as missing links, small parts of pages, or small bits of functionality like a calendar Javascript.

Like DIFF-X-delete, high DIFF-X-move values were also generally associated with faults. For the three PHP benchmarks, high DIFF-X-move scores indicated authentication failures, missing entire forms, or other high severity faults with explicit error messages. For the other benchmarks, high DIFF-X-move values revealed the same faults as those with DIFF-X-delete scores. Low DIFF-X-move values are less indicative of lower severity errors than low DIFF-X-delete scores. For example, low DIFF-X-move scores were found when large parts of the webpage are missing, as well as for small amounts of missing data such as parts of pages or incorrectly calculated data items.

Overall high DIFF-X-delete and DIFF-X-move values are generally indicative of severe faults in web-based applications, as they correlate with large amounts of missing data. The severity of a user-visible fault can frequently be predicted by the size of the DIFF-X-delete value.

9.2.2 Text-only differences

The third feature examined in depth is when the difference between two HTML outputs can be qualified as only changes to the natural language text within the documents. As Section 5.2.2 hypothesized, such a feature is likely to play a large part in SMART's ability to outperform a `diff`-like comparator. More than any other feature, text-only changes significantly impacted the negative performance of SMART in cases of false positives and false negatives when its respective F-value was high (PRESTASHOP, OPENREALTY, VANILLA, GCC-XML, and HTMLTIDY).

For example, text-only changes in PRESTASHOP correlate negatively with faults, but were not helpful at reducing false positives. For PRESTASHOP, a rotating "featured product" display caused even the "clean" (non-fault) output pairs to include text-only changes. Text-only changes in OPENREALTY and VANILLA had the opposite effect on SMART's performance: rather than failing to rule out false positives, in the case of OPENREALTY text-only changes were responsible for every false negative. For example, faults, such as incorrectly calculating the number of comments on a forum, appeared to SMART as simple text-only changes. In VANILLA, all other false negatives were due to SMART ignoring changed HTML attribute values, the default behavior to avoid flagging changes to image height and other non-errors. For example, in VANILLA an input field's `name` attribute was mistyped. In future work, SMART's performance when flagging attribute changes for HTML files as well as XML output can be explored.

9.2.3 Error Keywords

Finally, the issue of error keywords and their ability to predict faults in web-based application output is revisited here. Manual inspection of output pairs rated as severe revealed that they frequently contain error keywords. For example, a common severe fault renders as an otherwise-blank page containing only a server-generated error message. Despite this, the overall predictive power (F-value, not shown) of this feature was generally low, with the exception of CLICK where error keywords are able to perfectly predict actual faults. The main reason is that none of these real-world web applications ever displayed a stack trace in test case output, and instead wrapped their visible errors in more human-friendly formats; this behavior is in contrast to the experience with web ap-

plication faults in general (see Chapter 8). From a consumer perspective, it could be possible to design web applications that fail elegantly without stack traces or upsetting error messages and yet still rely on a sophisticated tool such as SMART that can nevertheless detect such failures without relying on a search for any particular keywords in the document.

9.3 Severity of Missed Errors using SMART

Having studied feature significance across the benchmarks from Chapter 6 as well as the three new PHP benchmarks presented in this chapter, the issue of fault severity with respect to faults missed by SMART is now explored. Recall that for the three PHP benchmarks in Figure 9.1, manual injection of source code defects, as opposed to using two different versions of each benchmark, yielded a set of *Testing Faults* to be revealed by the oracle comparator. This section measures the consumer-perceived severity of the subset of these known faults that SMART may miss. Specifically, Hypothesis (H7), that at most 1% of the false negatives produced by the highly-precise, fully-automated oracle comparator correspond to severe faults, is empirically tested.

9.3.1 Experimental Setup

Recall from Chapter 6 that SMART uses a pre-existing corpus of data (in Figure 5.2) to train the oracle comparator. In Section 6.4, this set of training data was augmented with automatically injected faults to improve performance. This same setup is adopted here; in addition to the training dataset of Figure 5.2, Figure 9.1 indicates the number of automatically injected faults used as training data for each benchmark in the *Training Faults* column. Because of the heavy reliance on non-deterministic output for these PHP benchmarks, the training data set was also supplemented with a pair of "clean" runs of the test suite where no faults were injected but where non-deterministic output still existed, and labeled these test outputs as non-faults. Figure 9.1 indicates the number of test cases in such a clean run in the *Test Suite* column. The training data for OPENREALTY and VANILLA were augmented with such non-fault and injected-fault information; for comparison, PRESTASHOP was not.

To test SMART's performance on the PHP benchmarks, manual fault injection was utilized according to the methodology of Sprenkle *et al.* [105] to obtain the *Testing Faults* in Figure 9.1. Another pair of "clean" runs were also included in the testing dataset to measure SMART's ability to reduce false negatives. It may be more difficult to reduce false positives in web applications which make the heaviest use of non-deterministic output. Fault severity was provided by the automated model presented in Section 7.5 which calculates consumer-perceived (as opposed to developer-perceived) fault severity at least as accurately as humans do, on average.

9.3.2 Experimental Results

Figure 9.3 presents the results of a severity analysis of the faults missed by SMART, broken down into severe (≥ 2.5) and non-severe (medium (> 2 and < 2.5), low (> 1 and ≤ 2), and very low (≤ 1)) categories, as in Section 7.3.1. These severity ratings correspond to varying levels of human actions based upon a fault seen; for example, severe faults occur when the user would either 1) file a complaint, 2) not return to the website, or 3) probably return to the website. Conversely, a rating of very low indicates that no fault was noticed by the consumer. Severity is considered because SMART may fail to report some defects, but not all defects are equally important to users. The *Weighted Found* column refers to the percent of total faults correctly identified by SMART when assigning weights ranging from 1–4 for severity in increasing order. PRESTASHOP, OPENREALTY, and VANILLA missed 1%, 9%, and 0% of the severe faults in their testing data respectively, indicating that overall the percentage of severe faults missed is extremely low. Out of 532 severe faults considered, SMART missed only 7.

The last column of Figure 9.3 shows the number of non-faults that were classified correctly by SMART. Using the terminology of Section 5.5.2, this corresponds to the amount of effort saved compared to `diff` (i.e., the factor multiplied by *LookCost*). For example, for VANILLA, SMART reduces developer inspection costs by 97% without missing any severe faults. For PRESTASHOP, SMART reduces inspection costs by 47% while missing only 1% of severe faults. Precision, in terms of correctly identifying non-faults, was found to be inversely proportional to recall (the number of actual faults found).

Benchmark	Goal Severe Faults	Goal Medium Faults	Goal Low Faults	Goal Very Low Faults	Miss Severe Faults	Miss Medium Faults	Miss Low Faults	Miss Very Low Faults	Weighted % Found Faults	% Found Non-Faults (savings)
PRESTASHOP	302	45	57	27	3	32	17	26	98%	47%
OPENREALTY	44	1	1	16	4	1	0	10	85%	100%
VANILLA	186	183	10	83	0	5	8	0	89%	97%

Figure 9.3: Severity of missed faults across the 3 PHP benchmarks.

This experiment demonstrated that even under circumstances where the number of severe faults to be detected is artificially high, SMART correctly flags 99% of such known defects. At the same time, SMART correctly labels over 70% of non-faults, on average, as not requiring human inspection, which is a significant improvement over a `diff`-like tool that would label all output as containing errors due to the non-deterministic nature of these benchmarks. In VANILLA, all severe faults were correctly detected, while 97% of the non-faulty output was labeled as such. These empirical results were achieved without requiring any manual annotation or customization of the oracle comparator, making the adoption of this highly effective approach almost effortless in industrial settings.

9.4 Summary

The main thesis of this dissertation is that user-visible web-based application errors have special properties that can be used to improve the current state of web application error detection, testing and development. This chapter explored the trade offs when using a fully-automated highly-precise oracle comparator, SMART, that relies on the structure of XML/HTML output as well as similarities between unrelated web-based applications, and the consumer-perceived severities of known errors missed by such an approach. Using a set of popular, open-source web applications that make heavy use of non-deterministic HTML output, SMART was shown to save developers from examining 50–100% of the non-faults in regression test output, while correctly flagging most actual errors. In cases where faults were not flagged by the oracle comparator, SMART missed 1% of severe faults on average. In addition, the precision with which non-faults were correctly labeled as not requiring human attention was found to be inversely proportional to the accuracy with which actual faults are appropriately labeled.

Overall, using this fully-automated oracle comparator was found to be an effective approach for reducing the cost of regression testing real-world, popular websites due to the special properties of web applications. Specifically, unrelated web-based applications fail and evolve in similar ways, making it possible to train such an oracle comparator automatically; the oracle comparator is

effective at classifying faulty and non-faulty output by relying on features of XML/HTML output. Appreciating the impact of consumer-perceived error severity in this domain can allow such an automated oracle comparator to be readily adopted, as the severity of errors missed by this approach can be measured and was found to generally be low. Consequently, the thesis that user-visible web-based application errors have special properties that can be used to improve the current state of error detection in this domain was supported in this chapter.

Chapter 10 Conclusions and Future Work

Although web-based applications are involved in transactions totaling several trillion dollars annually, designing and testing them remains a challenge due to resource constraints during their development and the additional complexities involved in generating dynamic content. Such pressures frequently leave web applications untested, despite their high reliability requirements: testing is perceived to have a low return-on-investment in this domain. This research explored errors in web-based applications in the context of web-based application error detection. The main thesis is that user-visible web-based application errors have special properties that can be exploited to improve the current state of web application error detection, testing and development. Two main contexts were explored: 1) error detection during the regression testing of web-based applications, and 2) focusing on severe errors in design and testing. Within these contexts, seven hypotheses were tested:

- that recognizing that errors in web-based applications can be successfully modeled due to the tree-structured nature of XML/HTML output (see (H1)),
- that unrelated web-based applications fail in similar ways (see (H2)),
- that not all failures are equally severe from a consumer perspective (see (H3)),
- that these failures can be modeled according to their consumer-perceived severities (see (H4)),
- that severe errors correspond to specific software engineering aspects during web application development (see (H5)),
- that test suites can be reduced in size while preserving severe error exposure (see (H6)),
- and that automated tools to detect errors rarely miss severe errors (see (H7)).

Specifically, this work demonstrated how the special structure of web-based applications can be harnessed to provide a partially-automated highly-precise oracle comparator for the web-based application testing domain. Such a comparator is an important contribution to this field because more naïve approaches are associated with a high number of false positives, and resource constraints are extreme. By relying on underlying similarities between the evolution and failure of unrelated web-based applications, this work extended such a comparator to be fully automatic. Having provided a means for reducing the effort required to test web-based applications, a model of consumer-perceived error severity was then explored. The model, derived from the results of a large-scale human study, was shown to be more accurate than average humans at judging error severity, even under complete automation. Developers have the opportunity to save effort as well as retain more consumers when such a model is used to prioritize errors for fixes, understand how to avoid high severity errors during web application development, and safely employ test suite reduction techniques without sacrificing revealing high-severity errors. This dissertation is further summarized below.

10.1 Improving Error Detection During Regression Testing

Chapter 5 presented a partially-automated highly-precise oracle comparator for reducing the cost of regression testing by using syntactic and structural features to decide whether or not test case output merits human inspection. In the web-based testing domain, traditional `diff`-based comparators are prone to a high rate of false positives. Due to the special structure of web-based application output, a number of features were suggested that can be used to distinguish potential errors from harmless functionality additions or rendering changes. The highly-precise oracle comparator, SMART, was evaluated as a model and as a cost-saving technique. As a model evaluated on 7154 test case pairs from 10 projects, SMART obtained a precision of 0.9972, a recall of 0.9890 and an F_1-score of 0.9931, which is over three times as good as the standard `diff` F_1-score of 0.3004. These strong machine learning results are complemented with a simulated deployment involving 20232 test cases; SMART had 1% of the false positives of `diff` — and saves development in typical

industrial practice as well as doing 20% better than previously-published results, thereby supporting Hypothesis (H1).

10.2 Automating Error Detection During Regression Testing

Chapter 6 built on the results in Chapter 5 by providing a fully-automated highly-precise oracle comparator for regression testing web-based applications. By taking advantage of the inherent underlying similarities in the way in which web-based applications evolve and fail, a fully automated comparator was introduced that outperformed `diff` anywhere from 2.5 to 50 times, achieving perfect precision and recall half the time, and very close to perfect precision and recall otherwise, supporting Hypothesis (H2). Such automation was achieved by relying on pre-existing training data from unrelated web-based applications to train a comparator for the application-at-test; in situations where such a comparator missed too many actual errors, source code mutation could be used as an automated means by which to supplement the training data set with application-specific output, thereby improving the oracle comparator's performance.

10.3 Modeling Consumer-Perceived Web Application Error Severities for Testing

Chapter 7 provided a consumer-perceived error severity model that allows developers to prioritize errors according to their likelihood of impacting consumer retention, thereby encouraging more effective testing. A human study was presented with over 12,600 human judgments of 400 real-world faults, 400 injected faults, and 100 non-faults, leading to the discovery that relying on a single human observer to judge the severity of a particular fault is inherently unreliable. Specifically, Hypothesis (H3) was supported by the observation that not all web application errors have the same severity level. Building on such a baseline, two models of error severity were then presented, each outperforming humans in terms of accurately predicting the average severity of web application errors. The first model relies on human annotations of error surface features, and suc-

cessfully identified 87% of non-severe errors to be assigned low priority in the experiments. A fully automated model was also presented, that can obviate examining 55% of such errors. Both models are significantly better than humans at flagging severe errors for examination, and can therefore replace or augment humans when assigning fix priorities to errors encountered in web application development and testing, supporting Hypothesis (H4).

10.4 Addressing High Severity Errors in Web Application Testing

Chapter 8 further analyzed the results of a study of the consumer-perceived severity of 400 real-world web application faults with the explicit goal of providing concrete guidelines to increase the perceived return-on-investment of even naive testing approaches. This chapter showed that the distribution of consumer-perceived fault severities is *independent* of the type or language of the application, making it theoretically possible to deliver high quality software for any use obviating constraints on the underlying technologies used. Similarly, given the same defect in the source code, different visual presentations of the fault to consumers have different impacts on their perceived severity. Controlling fault presentation by opting for pop-ups and error messages over stack traces and changing page layout is a simple and effective way to reduce the consumer-perceived severity of faults. In studying the causes of various types of faults, the utility of coarse-grained test suites that only detect relatively large differences in HTML output or rely on error keywords were revealed as effective ways of capturing many high severity faults, encouraging all developers to invest in at least some minimal testing infrastructure. Taken together, these observations support Hypothesis (H5), as various software engineering aspects of web application development correlate with severe faults. Finally, the trade offs between the costs of various user-session-based test suites for web applications were examined, finding that reduced test suites were effective at maintaining fault exposure properties across all fault severity levels, supporting Hypothesis (H6).

10.5 Combining Error Detection and Error Severity

Chapter 9 sought to tie together the insights in fault detection during automated regression testing and consumer-perceived fault severity in web applications. The trade offs when using a fully-automated highly-precise oracle comparator, SMART, that relies on the structure of XML/HTML output as well as similarities between unrelated web-based applications, and the consumer-perceived severities of known errors missed by such an approach was explored. Using SMART was found to be effective at reducing the cost of regression testing real-world, popular web applications due to their special properties, supporting Hypothesis (H7). Specifically, unrelated web-based applications fail and evolve in similar ways, making it possible to train SMART automatically; the oracle comparator is effective at classifying faulty and correct output by relying on features of XML/HTML output. Appreciating the impact of consumer-perceived error severity in this domain can allow such an automated oracle comparator to be readily adopted, as the severity of faults missed by this approach can be measured and was found to generally be low.

10.6 Conclusions and Future Implications

This dissertation focused on the special properties of web-based applications that can be used to improve development, testing, and error detection in this domain. Given the promising results in Chapter 9, I hope one implication of this work is that developers will readily use the models within this document to invest in testing web-based applications, as testing is frequently overlooked in this domain. To the best of my knowledge, the oracle comparator presented in Chapter 6 and Chapter 9 is the first fully-automated approach towards comparing test case outputs in web-based applications. Because there are few upfront costs in adopting such an approach, and as Chapter 9 demonstrated, the oracle comparator was highly effective for real-world web applications that rely heavily on non-deterministic output.

The research in this dissertation was conducted under the assumed context that few or no resources are devoted to testing web applications. Chapter 7 presented a model of consumer-perceived error severity that was used in Chapter 8 to provide concrete software engineering guidelines to de-

velopers in the web application domain. My aspiration is that such guidelines can be easily and effectively incorporated in industrial settings, as they can help prevent the loss of consumers by reducing the severity of faults in web applications. Recognizing that testing is an expensive and time-consuming process, I also anticipate that the manual and automated models of consumer-perceived web error severity will see industrial applications, specifically when prioritizing defects for fixes. To the best of my knowledge, the work in this document is the first to formally study consumer-perceived faults in web applications, and to provide both human-assisted and automated models for their classification.

My hope is that the strategies presented in this dissertation will lead to (1) an increase in the perceived return-on-investment for testing web-based applications, thereby encouraging testing and consequently improving their reliability, as well as (2) a decrease in consumer loss due to errors and their perceived severity. Having provided first steps towards cost-cognizant means by which to achieve these two goals, I believe that future research can focus on the prevention of errors in web-based applications, in addition to their detection and severity classification.

APPENDIX

Appendix A

This Appendix provides supplemental material to various studies and experiments contained in this dissertation. In particular, Section A.1 and Section A.2 contain information about the two user studies conducted in Chapter 7. Section A.1.2 through Section A.1.3 are an exact replication of the instructions presented to human subjects for the consumer study in Section 7.3.1. Similarly, Section A.2 duplicates exactly the information presented to developers in the small survey in Figure 7.7 in the same chapter. Section A.3 grounds the dominant technologies used in the models in Chapter 7. Finally, Section A.4 presents the decision tree the manually-annotated model in Section 7.4.1 uses to determine the consumer-perceived severity of user-visible faults. Section A.5 provides a formal definition of web-based applications.

A.1 Web Application Fault Severity Study

A.1.1 Participants and Subject Data

There were no prerequisites or special skills participants were required to have, except that they had previously used the Internet (through a browser). There were no age, sex, or other restrictions on volunteers, although a majority of people taking this survey were undergraduate students at the Universities of Virginia and Maryland. It is possible that the results are biased towards younger people, although these same individuals may use the net more frequently, especially when making purchases online.

A five level rating scale is used by participants to rate the severity of faults they see, shown in Figure A.3. It is possible that users may not agree that filing a complaint has a higher severity (4) than not returning to the website (3), although the implied scale of low severity to high severity is meant to prevent such interpretations.

A.1.2 About

It has been estimated that 40 to 70 percent of web applications exhibit user-visible errors. In some instances, these faults can be so severe that customers are unable to complete their activities on a website and companies end up losing business as a result. Web applications are unique in their requirements for high quality (as customer loyalty is low), and the speed at which they are developed. Consequently, testing would be especially important for websites, but is often overlooked due to a perceived low return on investment.

In this study, we will be examining the user (or customer) perceived severity of various errors encountered during normal website activities. Our goal is to be able to characterize the nature of different severities of web application faults, as well as get an idea for the underlying distribution of the different severity levels.

If you have any questions please feel free to contact me (Kinga Dobolyi) at dobolyi@virginia.edu

A.1.3 Instructions for Rating Websites

Subject Matter

You will be asked to examine pairs of website screenshots in order to identify and rank the severity of webpages that exhibit faults. The websites you will be looking at are based off of real-world web applications, although the faults you will see are simulations.

You will be shown 50 website pair screenshots. Some of these screenshots will not have any faults, but many of them will. If you correctly identify all of the actual faults in your set of 50 trials, you will be entered in a drawing for a $50 Amazon.com gift certificate.

We will not ask you for your name, and will not record any identifying information. Data obtained in this study will be used to identify a taxonomy or model of web applications faults. We anticipate including an evaluation of this tool in an upcoming publication.

Completing this survey is completely voluntary. If you do choose to participate, you will be asked to rate the severity of a set of 50 website screenshot pairs on a 0 – 4 scale. No special

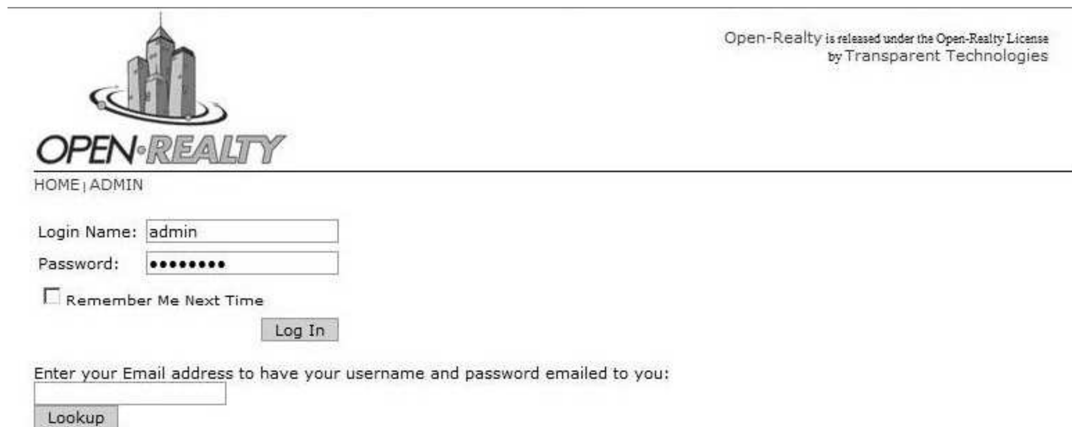

Figure A.1: The “current” page

knowledge or experience is required for participation. Most people complete the program in about 15 minutes, but there is no time limit.

Example Trial

You will be shown a pair of website screenshots.

- The first page corresponds to the “current” page in the browser. You will see a small explanation of what you, the user, are trying to do on the current page - note that you will be unable to actually click anything on the website, because it is only a screen capture. For example, you may see a login screen with a username and password entered, and you will be told that you want to log in to the application, and to pretend that you clicked the *Log In* button. Figure A.1 is an example of such a “current” page.
- The second page corresponds to the “next” page in the browser - that is, what would appear if you took the action described on the “current” page. For example, for the login page scenario described above, the “next” page would be a screen capture of the welcome page of the website you would see after you have successfully logged in. Figure A.2 is an example of such a “next” page.
- You will then be asked to determine whether or not you think there is a fault on the “next” page, based on what you saw and were instructed to pretend to do on the “current” page. If

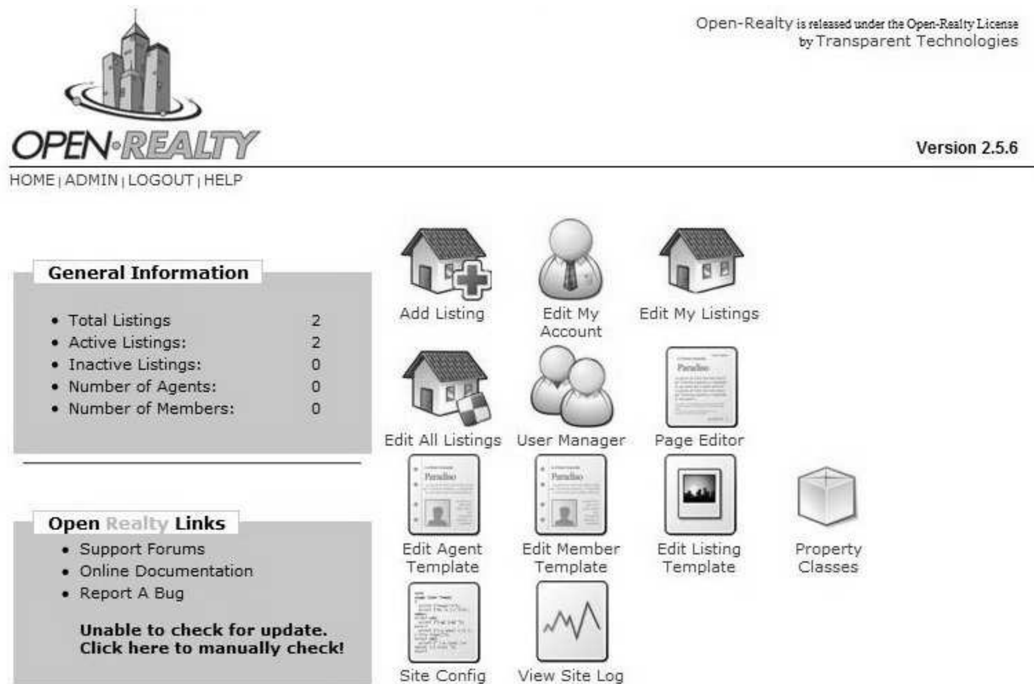

Figure A.2: The “next” page

you believe there is a fault, you will be asked to rate the severity of that fault as we define in Figure A.3.

Things to Keep in Mind

Please consider the following items as you are completing the study:

- The “current” pages you will see are not intended to contain faults. If you do notice a fault on the “current” page, please DO NOT consider that a fault for the purposes of our experiment. Only rate the faults that you see on the “next” pages.
- Please do not make any assumptions about the distribution of faulty versus non-faulty “next” pages you will see. While you will see some faulty pages and some non-faulty pages, the frequency of faulty pages you will be shown may not correspond to your experience in your daily life.

- When you do notice a fault on the "next" page, in making your decision of which severity rating to assign it to, assume that the fault will eventually be corrected, but you do not know when. For example, if the fault is that clicking on a button returns a blank page, you should assume that at some point in the future when you click on that button it will return the correct page. You do not know, however, when that will be — it may be the next time you click the button (if this were a real application), or it may not be fixed for 1 year.

- You will have access to this set of instructions as a help link while you are completing the experiment, which will open in a separate pop-up window.

- If you want, you can skip a set of screen captures for any reason. However, you can't go back.

Web Application Fault Severity Study

After you have read the instructions above and are ready to start, click below.

Launch Web Application Fault Severity Study

Reward

To encourage participation, we offer a financial reward for participation. You will be asked to select from the following two options when you start the study:

- We will give out $5 to anyone who completes the study until money runs out

- We will enter you in a drawing to win a $100 gift certificate to Amazon.com

These rewards are in addition to the $50 Amazon.com gift certificate drawing you can qualify for if you correctly find all faults in the web application screen captures you will be presented with.

Upon completion of the severity rating, you will receive an 8 character completion code. Bring this code to the following address any time to receive your reward: Olsson 219 (Westley Weimer's Office) 151 Engineer's Way Charlottesville, VA 22903

In order to receive your Amazon.com prizes (if you win the drawings), we will need to be able to contact you by email. You will therefore have the option of providing your email address before the study begins, which will only be associated with your completion code. If you do not wish to provide your email, you may still complete the study and still collect the $5 reward (when applicable) in person.

FAQ

How long does it take? We designed the experiment to take about 15 minutes. However, there is no time limit.

How do I know if the web page has a fault? We are asking you to use your previous web browsing experience to determine whether or not the web page screen captures you will see have faults.

Where did these web pages come from? Various open source projects.

A.2 Web Application Fault Severity Survey

The following survey is part of a study on the severity of web application faults and failures at the University of Virginia department of computer science. Our goal is to estimate the distribution the severity of faults in real web application development environments. In doing so, we will be able to design testing techniques and methodologies that target high-severity faults. Please read the instructions below and complete the survey to the best of your ability; your participation is entirely voluntary. We do not record your name, company, or any other information that could identify your submission, therefore, the data we collect remains anonymous.

We are offering a drawing for a $25 Amazon.com gift certificate for survey participants. If you would like to participate in this drawing, you may provide us with your email address to notify you if you are the winner, though this step is optional.

Thank you in advance, Laura Dobolyi

PhD Graduate Student University of Virginia dobolyi@virginia.edu

Severity Level	Description
0	The fault was not noticeable by customers/users on the website/application
1	Customers/users would return this website/application again
2	Customers/users probably would return this website/application again
3	Customers/users would not return to this website/application again
4	Customers/users would complain about this website/application

Figure A.3: The severity rating used in the human study

A.2.1 Instructions

Our goal in conducting this survey is to measure the distribution of fault severity in real world web application development environments. To do so, we ask you to asses the level of severity of faults you have encountered during your web application development and provide us with either the actual or relative distribution of those faults, according to the ranking in the table in Figure A.3:

An example of an actual distribution of faults would be to report out of 323 faults encountered, 56 were level 0, 79 were level 1, 60 were level 2, 84 were level 3, and 44 were level 4.

An example of a relative distribution of faults would be to report that 17% of faults were level 0, 24% of faults were level 1, 19% were level 2, 26% were level 3, and 14% were level 4.

Note that the previous two distributions are examples and are not meant to imply any kind of specific distribution that you should report.

In determining the distribution of faults your company has encountered during development and product maintenance, please report both bugs found during testing by developers as well as bugs reported by customers during or after deployment. We are interested in measuring these faults together and do not make the distinction between the two when collecting statistics on fault severity.

In addition, please use the following guidelines when selecting which faults to include in the fault severity rankings of this survey:

- Include bugs from the entire time of the product development lifecycle once testing has be-

gun. In other words, do not report faults that occurred only in the last year; instead, please report all faults encountered during the testing and product deployment/maintenance (when applicable).

- Include all and only user-visible faults. A user visible fault is a bug that exists on the website itself, though it may originate from any level of the application. For example, a database error may produce incorrect results, return wrong or missing information, or show an error message or crash dump on the website itself, which a customer/user is exposed to - in this case because the user can see this error on the website, it should be recorded in the survey. Other errors such as broken or missing links or images may be found in faulty HTML code and should also be reported. An example of an error that is NOT user visible and should NOT be reported is a missing or broken logfile that is only used by developers to debug the system.
- Duplicate faults (such as 5 users reporting the same error) should be reported only once.

Enter Your Results

Please use the form in Figure A.4 to report the distribution of faults you encountered using the guidelines above. If you are reporting a relative distribution using percentages, report the percentages in the column “Number of Faults (or percentage)”. Please consistently use either actual number or percentages.

A.3 Dominant Technologies Used in Current Web Applications

Chapter 7 through Chapter 9 make use of a web application fault severity model that relies on surface features of web application output to decide on a consumer-perceived fault severity rating for a particular defect. Because the manually-annotated version of the model uses a textual description of fault features provided to humans, and is therefore relatively easily updated to reflect changes in underlying web application technologies, this section focuses on the impact of such changes on the automated severity model which compares HTML output. Arguably, the latter model is more sensitive to technological advances and the impact of such evolution will be explored here.

Severity Level	Description	Number of Faults (or percentage)
0	The fault was not noticeable by customers/users on the website/application	
1	Customers/users would return this website/application again	
2	Customers/users probably would return this website/application again	
3	Customers/users would not return to this website/application again	
4	Customers/users would complain about this website/application	
OPTIONAL DATA		
Project duration from start:		optional months
Project testing duration:		optional months
Project deployment duration:		optional months
Project approximate size:		optional lines of code or specify units: optional
Project description:		optional
Your email (used only to notify you if you won the gift certificate drawing):		optional
Submit Results		

Figure A.4: The survey used in the developer study

The following features are unlikely to become obsolete as long as HTML tags for basic functionality (such as images and forms) do not change, although both the manual and automated models in Chapter 7 can be updated to reflect the current tags should such changes occur:

- **Arithmetic Calculation Error**s are differences in numeric output and do not depend on any particular implementation.
- **Missing Information** and **Missing Image**s are identified in the automated model in situations where HTML tags are missing between versions; consequently, there is no reliance on any particular underlying technologies.
- **Blank Page**s and **Wrong Page**s are easily characterized by missing information (see above) and missing any meaningful HTML tags besides `<HTML>`, `<HEAD>`, and `<BODY>`.
- **Form Error**s are identified by comparing the instances of various HTML tags associated with such functional items across two HTML outputs. Should HTML standards change, the model can be updated to reflect additional types of functional HTML elements identified by new HTML tags.
- **Cosmetic** and **Language Error**s indicate only small, non-meaningful changes between output and are not calculated based on any particular implementation.
- **Code on the Screen** and **Error Message / Other Error**s are calculated by searching for generic code constructs such as equal signs and braces, as well as keywords such as "error", "warning" and "line", that exist in one HTML output file and not the other. Because these keywords and constructs do not depend on any particular language, they are likely to be robust in the face of technological updates.
- **Failed Upload**s and **Search** errors occur when various keywords, such as "upload" or "results" occur on one HTML output but not the other. These keywords are predicted to occur in natural language messages to web application users, rather than as a consequence of any particular underlying implementation, and are therefore unlikely to change as web application technologies evolve.

While the other features used by the models in Chapter 7 are also generic, they do assume some underlying implementation or technologies. In particular,

- **CSS Error**s are identified in the automated model by counting and comparing the `<LINK>` tags and their `href` attributes. Such an implementation is not sensitive to changes in Cascading Stylesheet technology, as it simply looks for instances in HTML output where these stylesheets are imported.

- **404 Error**s are calculated by counting the number of times the keywords "not found" or "404" occur on one HTML output but not the other; such keywords can be updated to reflect new server implementations.

- **Session** errors occur when various keywords, such as "expired" or "log in" occur in one HTML output and not another. Evolutions in session management may affect the automated model if they notify users of session expiration using different terminology, although in these instances the model can be supplemented with additional keywords to examine.

- **Database** errors are mined by comparing HTML output and looking for the presence of SQL keywords in one output and not the other. As database technologies evolve, the model may have to be updated to rely on new constructs in SQL or other database languages.

- **Permission** and **Authentication** features are identified by comparing the number of occurrences of natural language keywords such as "access" and "password" between two HTML outputs. Such keywords are meant to capture the human-readable warning messages generated by web applications when logins are unsuccessful or access to various parts of websites are denied, rather than focusing on any particular software implementations of authentication.

Although these features are still relatively universal, the technologies they are built upon are explicitly documented below, particularly within the benchmark applications used to derive the severity models.

A.3.1 Cascading Stylesheets

Cascading Stylesheets (CSS) are a way to control the presentation of various HTML forms to the user, without directly embedding such controls into the HTML output directly. Instead, CSS are frequently imported, or otherwise isolated from the rest of the HTML output so that such presentation control elements are separated from the rest of the document. Although the actual CSS implementation and standards are not important for the purposes of this work, the models in Chapter 7 do assume that these stylesheets are directly imported into HTML output.

A.3.2 Web Application Server Implementation

The models in Chapter 7 attempt to identify missing pages, in part, by looking for a particular standard HTTP response code known as a 404 or not found error message. These errors occur when a client was able to establish communication with the server, but the server was unable to find the requested resource for the client. It is also possible to customize these server-generated error pages within the web server into more human-friendly formats that may display links, search forms, or other potentially helpful information. Although this work assumes that such a standard HTTP code and response is unlikely to change, because web servers can customize such error messages, other features in Chapter 7 are able to identify instances where the incorrect or unexpected web page is loaded.

A.3.3 Sessions and User Authentication

Because HTTP is a stateless protocol, client and servers rely on user sessions to manage the stateful properties of the web application user experience. In these situations an HTTP session cookie is frequently used as a means to identify client sessions to the server; the cookie has a unique identifier that is passed between the client and server to identify the session.

In many instances, a client-server interaction is built around concepts of restricted access and privacy. For example, in an online bookstore, all users are free to search the shop to browse for books and session cookies can be used to manage the different search results per user request.

These same cookies can also be used to control access to restricted areas of the website, such as user accounts, by requiring that users log in with some identifying information (such as a username and password) to gain access to their account. The session cookie then becomes responsible for managing this authenticated access between the client and the server. Such session cookies that are used to deal with authenticated users are frequently set to expire after a certain amount of inactivity to prevent situations where the user forgets to log out and leaves his or her public computer accessible to others. The models in this work are limited in the context of user sessions and authentication in that they assume users will use usernames and passwords to validate themselves through a website, generating a session token, and these sessions frequently expire and ask the user to log back in or display an error message.

A.3.4 Databases

State is frequently managed in web applications through the use of databases, as HTTP is a stateless protocol. Almost all of the benchmarks used to build the models in Chapter 7 rely on database access for storing stateful data. Although the actual database technology is transparent in this work, it is assumed that Structured Query Language (SQL) is used in conjunction with relational database management systems. Consequently, the automated severity model from Section 7.5 expects basic SQL keywords, such as `SELECT` and `UPDATE`, to be present in error messages when SQL code is dumped on the screen.

A.4 Decision Tree For Consumer-perceived Fault Severity Prediction Model

```
1 double score = -1;
2
3 if (wrongPage.compareTo("on") == 0 && missingInfo.compareTo("on") == 0)
4     score = 2.5;
5 else if(permission.compareTo("on") == 0 && session.compareTo("on") != 0)
6     score = 2.5;
```

```
 7  else if(missingPhoto.compareTo("on") == 0
 8          && codeError.compareTo("on") != 0){
 9      if(cosmetic.compareTo("on") == 0)
10              score = 0.5;
11      else
12              score = 2.5;
13  }
14  else if(mathCalc.compareTo("on") == 0 && codeError.compareTo("on") == 0)
15      score = 2.5;
16  else if(functionalDisplay.compareTo("on") == 0
17          && codeError.compareTo("on") == 0)
18      score = 2.5;
19  else if(database.compareTo("on") == 0)
20      score = 2.5;
21  else if(codeError.compareTo("on") == 0)
22      score = 2.5;
23  else if(codeDump.compareTo("on") == 0)
24      score = 2.5;
25  else if(authentication.compareTo("on") == 0){
26      if(permission.compareTo("on") == 0)
27          score = 3;
28      else
29          score = 2.5;
30  }
31  else if(error404.compareTo("on") == 0)
32      score = 2.5;
33  else if(wrongPage.compareTo("on") == 0)
34      score = 2;
35  else if(session.compareTo("on") == 0)
36      score = 2;
37  else if(search.compareTo("on") == 0 && cosmetic.compareTo("on") != 0)
38      score = 2;
39  else if(missingInfo.compareTo("on") == 0
40          && cosmetic.compareTo("on") != 0)
```

```
41      score = 2;
42  else if(upload.compareTo("on") == 0)
43      score = 1.5;
44  else if(missingPhoto.compareTo("on") == 0
45          && codeError.compareTo("on") == 0)
46      score = 1.5;
47  else if(mathCalc.compareTo("on") == 0 && cosmetic.compareTo("on") != 0)
48      score = 1.5;
49  else if(functionalDisplay.compareTo("on") == 0
50          && cosmetic.compareTo("on") != 0)
51      score = 1.5;
52  else if(css.compareTo("on") == 0)
53      score = 1.5;
54  else if(mathCalc.compareTo("on") == 0 && cosmetic.compareTo("on") == 0)
55      score = 1;
56  else if(language.compareTo("on") == 0)
57      score = 1;
58  else if(functionalDisplay.compareTo("on") == 0
59          && cosmetic.compareTo("on") == 0)
60      score = 1;
61  else if(database.compareTo("on") == 0 && cosmetic.compareTo("on") == 0)
62      score = 0.5;
63  else if(missingInfo.compareTo("on") == 0
64          && cosmetic.compareTo("on") == 0)
65      score = 0.5;
66  else if(search.compareTo("on") == 0 && cosmetic.compareTo("on") == 0)
67      score = 0.5;
68  else    //if no fault was seen
69      score = 0.5;
70
71  if (cosmetic.compareTo("on") == 0 && score > 0.5)
72      score = 1;
```

A.5 Definition of Web-based Applications

The terms web-based application and web application are frequently used interchangeably in the web community. For the purposes of this document, a web-based application is different from a web application in that web-based applications may output XML code that does not necessarily end up rendered by a browser. For example, web services frequently communicate through XML, and such XML output is passed between separate components rather than displayed directly to a user.

Bibliography

[1] Copernic tracker home page. http://www.copernic.com/en/products/tracker/index.htm, 2006.

[2] GCC-XML. http://www.gccxml.org/HTML/Index.html, 2008.

[3] txt2html - text to HTML converter. http://txt2html.sourceforge.net/, 2008.

[4] A7soft jexamxml is a java based command line xml diff tool for comparing and merging xml documents. http://www.a7soft.com/jexamxml.html, 2009.

[5] Gartner group forecasts b2b e-commerce explosion. http://www.crn.com/it-channel/18833281, 2009.

[6] Jakarta cactus. http://jakarta.apache.org/cactus/, 2009.

[7] Online sales to climb despite struggling economy according to shop.org/forrester research study. http://www.shop.org/c/journal_articles/view_article_content?groupId=1&articleId=702&version=1.0, 2009.

[8] Website-watcher home page. http://www.aignes.com, 2009.

[9] World internet usage statistics news and world population stats. http://www.internetworldstats.com/stats.htm, 2009.

[10] Apache click. http://click.apache.org/, 2010.

[11] Examdiff - the freeware visual file compare tool. http://www.prestosoft.com/edp_examdiff.asp, 2010.

[12] Gnu wget. http://www.gnu.org/software/wget/, 2010.

[13] Man page for curl (linux section 1) - the unix and linux forums. http://www.unix.com/man-page/Linux/1/curl/, 2010.

[14] More information on html match, the best html comparison tool for web developers. `http://www.htmlmatch.com/hminfo.htm`, 2010.

[15] Open-realty. `http://www.open-realty.org/`, 2010.

[16] Prestashop free open source e-commerce software for web 2.0. `http://www.prestashop.com/`, 2010.

[17] Vanilla - free, open-source forum software. `http://vanillaforums.org/`, 2010.

[18] Vqwiki.org. `http://www.vqwiki.org/`, 2010.

[19] Winmerge. `http://winmerge.org/`, 2010.

[20] Raihan Al-Ekram, Archana Adma, and Olga Baysal. diffX: an algorithm to detect changes in multi-version XML documents. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11, 2005.

[21] Nadia Alshahwan and Mark Harman. Automated session data repair for web application regression testing. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 298–307, 2008.

[22] Paul Ammann and Jeff Offutt. *Introduction To Software Testing*. Cambridge University Press, New York, NY, USA, 2008.

[23] Annelise Andrews, Jeff Offutt, and Roger Alexander. Testing web applications by modeling with fsms. In *Software Systems and Modeling*, volume 4, pages 326–345, April 2005.

[24] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.

[25] Hassan Artail and Michel Abi-Aad. An enhanced web page change detection approach based on limiting similarity computations to elements of same type. In *Journal of Intelligent Information Systems*, volume 32, pages 1–21, February 2009.

[26] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272, 2008.

[27] Michael Benedikt, Juliana Freire, and Patrice Godefroid. Veriweb: Automatically testing dynamic web sites. In *World Wide Web Conference*, May 2002.

[28] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Booch, Jacobson and Rumbaugh publishers, 1999.

[29] David Binkley. Using semantic differencing to reduce the cost of regression testing. In *International Conference on Software Maintenance*, pages 41–50, 1992.

[30] David Binkley. Semantics guided regression test cost reduction. *IEEE Transactions on Software Engineering*, 23(8):498–516, 1997.

[31] Barry Boehm and Victor Basili. Software defect reduction. *IEEE Computer Innovative Technology for Computer Professions*, 34(1):135–137, January 2001.

[32] Penelope A. Brooks and Atif M. Memon. Automated gui testing guided by usage profiles. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342, 2007.

[33] I. M. Chakravarti, R. G. Laha, and J. Roy. *Handbook of Methods of Applied Statistics*, volume I. John Wiley and Sons, USE, 1967.

[34] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[35] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. A technique for reducing user session data sets in web application testing. In *Proceedings of the Eighth IEEE International Symposium on Web Site Evolution*, pages 7–13, 2006.

[36] G.A. Di Lucca. Testing web-based applications: the state of the art and future trends. In *Computer Software and Applications Conference*, pages 65–69, July 2005.

[37] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 411–420, 2005.

[38] Kinga Dobolyi and Westley Weimer. Changing java's semantics for handling null pointer exceptions. In *International Symposium on Software Reliability Engineering*, pages 47–56, November 2008.

[39] Sebastian Elbaum, Srikanth Karre, and Gregg Rothermel. Improving web application testing with user session data. In *International Conference on Software Engineering*, pages 49–59, 2003.

[40] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, 2001.

[41] Sebastian Elbaum, Gregg Rothermel, Srikanth Karre, and Marc Fisher II. Leveraging user-session data to support web application testing. *IEEE Transactions on Software Engineering*, 31:187–202, 2005.

[42] Ronald A. Fisher. *Statistical methods for research workers*. Oliver and Boyd, 1970.

[43] S. Flesca and E. Masciari. Efficient and effective web change detection. *Data Knowledge Engineering*, 46(2):203–224, 2003.

[44] Lawrence L. Giventer. *Statistical Analysis for Public Administration*. Jones and Bartlett Publishers, 2007.

[45] Google Code. Colibri-java. In `http://code.google.com/p/colibri-java/`, October 2009.

[46] Google Code. raise: Reduce and prioritize suites. In `http://code.google.com/p/raise/`, October 2009.

[47] Yuepu Guo and Sreedevi Sampath. Web application fault classification - an exploratory study. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 303–305, 2008.

[48] William G. J. Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 145–154, 2007.

[49] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11, 2009.

[50] Harris Interactive. A study about online transactions, prepared for TeaLeaf Technology Inc., October 2005.

[51] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, 1993.

[52] Edward Hieatt and Robert Mee. Going faster: Testing the web application. *IEEE Software*, 19(2):60–65, 2002.

[53] Douglas Hoffman. A taxonomy for test oracles. *Quality Week*, 1998.

[54] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Automated software engineering*, pages 34–43, 2007.

[55] G. M. Kapfhammer. Software testing. In *The Computer Science Handbook*, chapter 105, 2004.

[56] John C. Knight and Paul E. Ammann. An experimental evaluation of simple methods for seeding program errors. In *Proceedings of the 8th international conference on Software engineering*, pages 337–342, 1985.

[57] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[58] David Chenho Kung, Chien-Hung Liu, and Pei Hsia. An object-oriented web test model for testing web applications. In *24th International Computer Software and Applications Conference*, pages 537–542, 2000.

[59] Suet Chun Lee and Jeff Offutt. Generating test cases for xml-based web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, page 200, 2001.

[60] David D. Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. pages 4–15. Springer Verlag, 1998.

[61] Daniel R. Licata and Shriram Krishnamurthi. Verifying interactive web programs. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 164–173, 2004.

[62] Seung Jin Lim and Yiu-Kai Ng. An automated change detection algorithm for html documents based on semantic hierarchies. In *Proceedings of the 17th International Conference on Data Engineering*, pages 303–312. IEEE Computer Society, 2001.

[63] Chien-Hung Liu, David C. Kung, Pei Hsia, and Chih-Tung Hsu. Object-based data flow testing of web applications. In *Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, page 7, 2000.

[64] G. Di Lucca, A. Fasolino, F. Faralli, and U. de Carlini. Testing web applications. *International Conference on Software Maintenance*, page 310, 2002.

[65] G.A. Di Lucca, A. R. Fasolino, and P. Tramontana. A technique for reducing user session data sets in web application testing. *Web Site Evolution, IEEE International Workshop on*, 0:7–13, 2006.

[66] Li Ma and Jeff Tian. Analyzing errors and referral pairs to characterize common problems and improve web reliability. In *International conference on web engineering, Oviedo , Spain*, 2003.

[67] Li Ma and Jeff Tian. Web error classification and analysis for reliability improvement. *Journal of Systems and Software*, 80(6):795–804, 2007.

[68] David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.

[69] Alessandro Marchetto, Filippo Ricca, and Paolo Tonella. Empirical validation of a web fault taxonomy and its usage for fault seeding. In *Proceedings of the 2007 9th IEEE International Workshop on Web Site Evolution*, pages 31–38, Washington, DC, USA, 2007. IEEE Computer Society.

[70] Aditya P. Mathur. *Foundations of Software Testing*. Addison-Wesley Professional, 2008.

[71] Atif Memon, Ishan Banerjee, Nada Hashmi, and Adithya Nagarajan. DART: A framework for regression testing "nightly/daily builds" of GUI applications. In *International Conference on Software Maintenance*, 2003.

[72] Atif M. Memon and Qing Xie. Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Transactions on Software Engineering*, 31(10):884–896, 2005.

[73] Gerard Meszaros. Agile regression testing using record & playback. In *Object-oriented programming, systems, languages, and applications*, pages 353–360, 2003.

[74] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[75] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM.

[76] J. Offutt. Quality attributes of web software applications. *Software, IEEE*, 19(2):25–32, Mar/Apr 2002.

[77] Jeff Offutt, Ye Wu, Xiaochen Du, and Hong Huang. Bypass testing of web applications. *Software Reliability Engineering, International Symposium on*, 0:187–197, 2004.

[78] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, 1998.

[79] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, New York, NY, USA, 2002. ACM.

[80] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, 2004.

[81] Soila Pertet and Priya Narasimhan. Causes of failure in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University Parallel Data Lab, December 2005.

[82] Roger S. Pressman. What a tangled web we weave. *IEEE Software*, 17(1):18–21, 2000.

[83] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 188–197, Washington, DC, USA, 2004. IEEE Computer Society.

[84] C. Ranganathan and Shobha Ganapathy. Key dimensions of business-to-consumer web sites. *Information and Management*, 39(6):457–465, 2002.

[85] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, 2001.

[86] Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Annals of Software Engineering*, 14(1-4):93–114, 2002.

[87] Filippo Ricca and Paolo Tonella. Anomaly detection in web applications: A review of already conducted case studies. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 385–394, 2005.

[88] Filippo Ricca and Paolo Tonella. Web testing: a roadmap for the empirical research. In *Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, pages 63–70, 2005.

[89] J. L. Rodgers and A. W. Nicewander. Thirteen ways to look at the correlation coefficient. *The American Statistician*, 42(1):59–66, 1988.

[90] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[91] Sreedevi Sampath, Valentin Mihaylov, Amie Souter, and Lori Pollock. A scalable approach to user-session based testing of web applications through concept analysis. In *International Conference on Automated Software Engineering*, pages 132–141, 2004.

[92] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, and Lori Pollock. Integrating customized test requirements with traditional requirements in web application testing. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 23–32, 2006.

[93] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, Lori Pollock, and Amie Souter Greenwald. Applying concept analysis to user-session-based testing of web applications. *IEEE Transactions on Software Engineering*, 33(10):643–658, 2007.

[94] Jessica Sant, Amie Souter, and Lloyd Greenwald. An exploration of statistical models for automated test case generation. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[95] Stephen R. Schach. Testing: principles and practice. *ACM Computing Surveys*, 28(1):277–279, 1996.

[96] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 76–85. ACM Press, 2003.

[97] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. 2003.

[98] Luis Moura Silva. Comparing error detection techniques for web applications: An experimental study. In *Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*, pages 144–151, 2008.

[99] Harry M. Sneed. Testing a web application. In *Workshop on Web Site Evolution*, pages 3–10, 2004.

[100] Fawzy Soliman and Mohamed A. Youssef. Internet-based e-commerce and its impact on manufacturing and business operations. In *Industrial Management & Data Systems*, pages 546–552. MCB UP Ltd, 2003.

[101] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. In *Automated Software Engineering*, pages 253–262, 2005.

[102] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. A case study of automatically creating test suites from web application field data. In *Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 1–9, 2006.

[103] Sara Sprenkle, Emily Hill, and Lori Pollock. Learning effective oracle comparator combinations for web applications. In *International Conference on Quality Software*, pages 372–379, 2007.

[104] Sara Sprenkle, Lori Pollock, Holly Esquivel, Barbara Hazelwood, and Stacey Ecott. Automated oracle comparators for testing web applications. In *International Symposium on Reliability Engineering*, pages 117–126, 2007.

[105] Sara Sprenkle, Sreedevi Sampath, Emily Gibson, Lori Pollock, and Amie Souter. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *International Conference on Software Maintenance*, pages 587–596, 2005.

[106] Sara E. Sprenkle. *Strategies for automatically exposing faults in web applications*. PhD thesis, 2007.

[107] S. E. Stemler. A comparison of consensus, consistency, and measurement approaches to estimating interrater reliability. *Practical Assessment, Research and Evaluation*, 9, 2004.

[108] J. Strecker and A.M. Memon. Relationships between test suites, faults, and fault detection in gui testing. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 12–21, April 2008.

[109] Jeff Sutherland. Business objects in corporate information systems. *ACM Computing Surveys*, 27(2):274–276, 1995.

[110] Symantec. Internet security threat report. In *http://eval.symantec.com/mktginfo/enterprise/white_papers/ent-whitepaper_symantec_internet_security_threat_report_x_09_2006.en-us.pdf*, September 2006.

[111] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *Software Engineering, IEEE Transactions on*, 18(3):206–215, Mar 1992.

[112] TeaLeaf Technology Inc. Open for business? Real availability is focused on users, not applications. October 2003.

[113] Axel van Lamsweerde. Formal specification: a roadmap. In *ICSE - Future of SE Track*, pages 147–159, 2000.

[114] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Reliability, quality and safety of software-intensive systems*, pages 3–21, 1997.

[115] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

[116] Y. Wang, D.J. DeWitt, and J.-Y. Cai. X-diff: an effective change detection algorithm for xml documents. In *International Conference on Data Engineering*, pages 519–530, March 2003.

[117] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260, 2008.

[118] Leigh Williamson. IBM Rational software analyzer: Beyond source code. In *Rational Software Developer Conference*. http://www-07.ibm.com/in/events/rsdc2008/presentation2.html, June 2008.

[119] Ye Wu and Jeff Offutt. Modeling and testing web-based applications. Technical Report ISE-TR-02-08, 2002.

[120] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques of maintaining evolving component-based software. *International Conference on Software Maintenance*, page 236, 2000.

[121] Qing Xie and Atif M. Memon. Model-based testing of community-driven open-source gui applications. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 145–154, 2006.

[122] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16(1):4, 2007.

[123] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10):771–789, 2006.