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

A Dissertation Proposal

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 2009

Thesis Proposal Committee

Mary Lou Soffa (chair)

Westley Weimer (advisor)

John C. Knight

Willaim A. Wulf

Chad S. Dodson (Psychology)

June 2009

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 they 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 further constraints in their development lifecycles that often preclude complete testing.

To address these constraints, this research will explore user-visible web-based application errors in the context of web-based application fault detection and classification. The main thesis of this research is that *web-based application errors have special properties that can be exploited to improve the current state of web application fault detection, testing, and development*. This proposed research will result in 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, I propose to construct a model of user-visible web application fault severity, backed by a human study, to validate or refute the current underlying assumption of fault severity uniformity in defect seeding for this domain, propose software engineering guidelines to avoid high severity faults, and facilitate testing techniques in find high-severity faults.

Studying fault severities from the customer perspective is a novel contribution to the web application testing field. This research will approach testing web-based applications by recognizing that errors in web applications can be successfully modeled due to 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 customer-perceived severities, with the ultimate goal of improving the current state of web application testing and development.

Contents

Chapter 1 Introduction

1.1 Motivation

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 [8]. While the global average for Internet usage is only 24% of the population by comparison [9], online business-to-business e-commerce[1] transactions total several trillions of dollars annually [6]. Therefore, there is a powerful economic incentive to produce and maintain high quality web-based applications[2].

Although many types of software, such as operating systems, are also widely used and highly distributed, web-based applications have additional challenges in ensuring acceptability and maintaining a customer base. Customer loyalty towards any particular website is notoriously low, and is primarily determined by the usability of the application [43]; unlike customers purchasing software such as Microsoft Windows, web customers can easily switch providers without having to buy another product or install 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 [47]. 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 [56].

Delivering high quality web-based applications has its own additional challenges. Most web applications are developed without a formal process model [48]. Despite having high 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 [51]. Web application developers often deliver the system without testing it [51].

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: (1) Web-based applications form the backbone of the e-commerce and communication businesses, and therefore they are subject to unique and powerful economic considerations, (2) Web-based applications provide a variety of services, but are commonly built as three-tiered architectures that output browser-readable code (see Figure 6.1), and consequently unrelated web-based applications often fail in similar ways, and (3) Web-based applications are human-centric, implying not only a "customer" use-case, but also defining the perceived acceptability of results through the eyes of the user.

[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 the Appendix for a definition of web-based applications.

1.2 Web-based Applications

While the economic urgency of delivering high-quality web-based applications is only 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 6.1 illustrates, although web applications are frequently complex, with opaque, loosely-coupled components, are composed in multiple programming languages, and maintain persistent session requirements, as my research will show, they tend to fail in similar and predictable ways. I hypothesize that 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 [47, 60]. 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.

Second, web applications are meant to be viewed by a human user. While this implies that faults in the system will manifest themselves at the user level and drive away customers, I claim that this human-centric quality of web applications should 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 project often entail may focus on reducing the number of high severity faults that will drive away customers.

1.3 Challenges for Testing Web-based Applications

Testing is a major component of any software engineering process meant to produce high quality applications. Despite the drive to retain customers, 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 [23]. Similarly, websites undergo maintenance faster than other applications [23]. Unlike other types of software, web-based applications are frequently patched in real-time in response to customer 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 [29]. This mindset is a consequence of short delivery times, the pressure to change, developer turnover, and evolving user needs [51, 67]. 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 [67]. The content of a page may be customized according to data in a persistent store, the server state, or session variables. Validating dynamically-generated webpages is difficult because it often requires

testing every possible execution path, and static analyses have difficulty capturing the behavior of code generated on-the-fly by dynamic languages [15].

1.4 Errors in the Context of Web-based Application Testing

In order for testing of web-based applications to be made widely and successfully adopted, testing methodologies must be flexible, automatic, and able to handle their dynamic nature. This proposed research will explore errors in web-based applications in the context of web-based application fault detection. In doing so, my 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. As my thesis, *I hypothesize that web-based applications have special properties that can be harnessed to build tools and models that improve the current state of web application fault detection, testing, and development*. I approach the problem of fault 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 to define a model of severity, I seek to target fault detection and classification methodologies and evaluation techniques toward detecting high-severity faults to retain users in the face of low customer loyalty.

The contributions of this research will be: (1) tools and algorithms to further automate fault detection during the testing of web-based applications, making the efficient adoption of such testing techniques more feasible for developers, and (2) a model of web application fault severity to guide software engineering and testing techniques to avoid and find high-severity faults, respectively.

Chapter 2 Background

2.1 Testing Web-based Applications

This section 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. 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 code is often dynamically generated.

2.2 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 [23, 62]. 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. Unit testing of web applications using tools such as CACTUS [7] require the developer to manually create test cases and oracles of expected output. Similarly, structural testing techniques require the construction of a model [35, 39, 50], which is usually carried out manually [62]. 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 [17]. Testing the dynamic, functional components automatically is an active research area [16]. Tools that do approach functional validation are usually of a capture-replay nature [52], 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.

2.2.1 Oracles

Inherent to 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 [18]. Figure 6.3 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. Recent work [32, 39, 57, 59, 60] uses HTML output as oracles, because such data is easily visible and because lower-level faults typically manifest themselves as user-visible output [47, 60]. Oracle comparators are frequently used for testing web applications, and in practice discrepancies are examined through human intervention [23, 39, 51, 60].

Testing is often limited by the effort required to compare results between the oracle and test case outputs. 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 [60] which must be manually interpreted. Manual inspection is an expensive process, however, and the incremental nature of website updates described in Section 1.3 often may not change the appearance or functionality experienced by the user.

Change Detection

Detecting changes between domain-specific documents is a frequent challenge is many applications. For example, differences in tree-based documents (such as XML and abstract syntax trees) can be accomplished by a tool such as DIFFX [10], which characterizes the number of insertions, moves, and deletes required to convert one tree to the other as a minimum-cost edit script [10, 66]. Change detection for natural language text can be achieved through a bag-of-words model, 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 [49] and abstract syntax tree matching [42] 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.

Change detection in web pages has been explored in the context of plagarism detection [55] and web page update monitoring [25, 37, 13]. 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 [13, 3, 1]. Flesca and Masciari use three similarity measures to detect the percentage of similar words, measures of tree element positions, and similar attributes between two XML-based documents [25]. Such structure-aware analyses may be useful in designing reasonably precise oracle-comparators, as

long as the focus is shifted towards error detection. An ideal comparator for web-based applications would be able to handle both the structural evolutions (such as DIFFX) as well as updates to content (such as natural language tools) in order to specifically differentiate between defects and correct output, as opposed to pinpointing or summarizing updates.

Oracle Comparators for the Web

Traditional testing for programs with tree-structured output is particularly challenging [57] due to the number of false positives returned by a `diff`-like comparator [60]. 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 reasonably *precise 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 comparator for web-based applications is an active area of research. Sprenkle *et al.* have focused on oracle comparators for testing web applications [57,59,60]. They use features derived from `diff`, web page content, and HTML structure, and refine these features into oracle comparators [60] 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 [59].

2.2.2 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 failure detection. This work will focus on automated failure detection in web application testing though the use of reasonably precise comparators [57] (as described in Section 2.2.1) to verify the functionality of the website. Application-level failures in component-based services can also be detected automatically [20], 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 [30, 62].

2.2.3 Measuring Test Suite Efficacy

Similar to the testing of other types of software, web-based application testing methodologies must be 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 *a priori*. Two widely-adopted complementary criterion are used to identify the efficacy of various test suites:

- **Code coverage** is a standard software engineering technique used to measure test suite efficacy. Code coverage metrics are frequently used in web application testing [16,23,27,39,54,

57, 58, 59, 60, 61], although the average percentage of statement coverage falls well short of 100% (and is often closer to 60%) in many studies [16, 27, 54, 57, 58, 59, 60, 61].

- **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 [16, 22, 23, 44, 57, 59, 60, 61]. 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 [18,62]. There are two main options for this so-called *fault seeding*: faults can be manually inserted by individuals with programming expertise, or mutation operators can be used to automatically produce faulty versions of code. It is hypothesized that automatically-seeded faults using source code mutation are at least as difficult to find as naturally occurring ones for software in general [12, 33]. Whether or not manually seeded faults are equivalent to naturally occurring faults in web applications remains an open question.

Cost is also an important factor in determining test suite efficacy, especially when considering the resource constraints web development is subject to (see Chapter 1). In this cost model, 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. Under the cost model a more *effective* test suite may ultimately discover fewer faults then a competitor. 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). Traditional test reduction techniques such as Harrold, Gupta, and Soffa's reduction methodology [28] have been successfully applied to user-session based testing [32]. Other approaches focus on web applications characteristics in particular, such as data-flow [38], finite state machine [11] analyses, use case coverage [21] and URL-based coverage [53].

2.2.4 Defining Errors in Web-based Applications

Web-based applications present additional challenges in testing because the term "fault" may have different meanings to different people. As an example, usability issues, such as the inability of a customer to locate a `Login` link, may not be considered as faults 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." [40]. 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 [62]. Ostrand and Weyuker initially classified faults in terms of their fix-priorities [45], but later rejected that approach, concluding that using such severity measures was subjective and inaccurate [46, 62].

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 initial step towards web fault classification [26]. Marchetto *et al.* validate a web fault taxonomy to be used towards fault seeding in [41]. Their fault categories are summarized in Figure 6.4, 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). In these fault classifications [26, 41] 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, this consideration is not explored.

2.3 Graphical User Interface Testing

Many similarities exist between Graphical User Interfaces (GUIs) and web applications — a brower-displayed webpage is a kind of GUI. Like a webpage, a GUI can be characterized in terms of its widgets 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." [69]. 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..." [69].

Like web applications, GUIs are difficult to test due to the exponential number of states the software can be in [68], as well as the manual effort required to develop test scripts and detect failures [19]. Similarly, they are often not tested at all, or are tested using capture-replay tools that capture either GUI widgets or mouse coordinates [2]. 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 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 customer-perceived fault severity between the two types of software difficult, and faults are likely to manifest themselves in different ways (for example, web applications frequently fail and display stack traces, while GUIs are less likely to do so in the middle of normal GUI content). This research will focus on web-based application user interfaces only. In future work, I would like to analyze faults in GUI applications and potentially extend some of the guidelines and techniques in the current proposed work to that domain.

2.4 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 precise error exposure. By studying errors in web-based applications in the context of web-based application testing, my goal 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, my research will focus on fault detection, with the assumption of a provided test-suite with a retest-all strategy.

Additionally, I propose to make web testing more cost-effective by devising a model of fault severity that will guide test case design, selection, and prioritization. This model of fault severity will have the additional benefits of validating or refuting the underlying assumption that all faults are equally severe in fault-based testing [24,63] 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 [45, 46], 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 customer-perceived severities of real-world faults. I claim such human-driven results would be more indicative of true monetary losses and especially relevant in the web domain.

Chapter 3 Goals and Approaches

This research explores errors in web-based applications in the context of web-based application fault detection. My main hypothesis is that web-based application errors have special properties that can be exploited to improve the current state of web application fault detection, testing and development. This chapter details the goals of my research and the approaches and steps I will take to carry it out.

3.1 Goals

The main goals for this research are:

1. Improve fault detection during regression testing web-based applications to reduce the cost of this activity by capitalizing on the special structure of web-based application output to precisely identify errors.

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

3. Understand customer-perceived severities of web application errors.

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

5. Understand how to avoid high-severity faults during web application design and development.

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

By improving upon fault detection, this proposed research will result 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, I aim to construct a model of web application fault severity to validate the current underlying assumption of fault severity uniformity in fault seeding, guide software engineering to avoid high severity faults, and assist testing techniques in find high-severity faults.

Studying fault severities from the customer perspective is a novel contribution to the web application testing field. This research will approach the web-based application testing challenge by recognizing that errors in web-based applications can be successfully modeled due to the tree-structured nature of XML/HTML output, that unrelated web-based applications fail in similar ways,

and that these failures can be modeled according to their customer-perceived severities. Figure 6.5 summarizes the proposed outline.

3.2 Research Steps

This section details the major steps to achieve the goals above.

3.2.1 Step 1: Construct a reasonably precise oracle-comparator that uses the tree-structured nature of XML/HTML output and other features.

In Step 1, I propose to focus on reducing the cost of current regression testing techniques for web-based applications by focusing on fault detection. Regression testing programs with tree-structured output is traditionally challenging [57] due to the number of false positives returned by naïve `diff`-like comparators [60]. Comparators that are not robust enough to handle the incremental, and often non-functional, evolutions of web applications further compound the problem by invalidating old oracle outputs.

I propose to construct a reasonably *precise oracle comparator* that reduces the number of false positives associated with traditional regression testing output comparison approaches for web-based applications without sacrificing true positives[1]. To do so, I target the tree-structured nature of XML/HTML output and build a comparator that examines these two output trees. This approach will classify test case output based on structural and semantic features of tree-structured documents. A semantic distance metric that is based on the weighted sum of individual features will decide whether or not an output pair needs to be examined by a human. I propose to use linear regression to learn the feature weights and identify a global cutoff for each benchmark application. The idea behind this approach is to model web-based application errors on a per-project basis through feature analysis; once I have modeled the signature of an erroneous output in a specific application, I will to use the model to differentiate between correct and faulty output.

3.2.2 Step 2: Harness the similar way in which web applications fail to avoid the need for human annotations in training a reasonably precise oracle-comparator.

Although Step 1 aims to reduce the effort required to verify regression test outputs, the approach is not entirely automated. In my preliminary work, a small amount (20%) of test cases output must be manually annotated in each iteration to train the model. In this step, I propose to employ the inherent similarities between unrelated web-based applications to train a model for a reasonably precise comparator in an automatic manner.

I will annotate pairs of oracle-testcase output from a set of benchmark applications to use as training data for a model of web-based application errors as in Step 1. I will then use this model as a comparator for separate, unrelated applications. This step is possible because of the predictable way in which unrelated web-based applications often fail; I will explicitly test this hypothesis by recording what features are shared between different applications' faults and evolutions. While this is a reasonable general approach, it is possible that there are target test applications that do not

[1] The word reasonably is defined as an F-score of 0.9 or better.

exhibit faults in a manner similar enough to my corpus of training data to apply this technique as-is. In such cases, I propose to use fault injection through source code mutation to generate oracle-fault pairs of output, that I can then apply to the training data set and customize my comparator to the application at test, all the while avoiding manual annotations. Using fault seeding to simulate errors in test case output for web-based applications has previously been explored in [36, 59].

3.2.3 Step 3: Conduct a human study of real-world fault severity to identify a model of fault severity.

Customer-perceived fault severities have not been studied in the context of web applications, even though this domain is highly human-interaction centric. While fault severities are frequently recorded during the testing and maintenance phases of software development in bug repositories, 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 [46]. Due to the business-oriented nature of web applications, it is less likely that customers will report faults in bug repositories — instead, they are more likely to contact the website's company directly. For example, Amazon.com does not have a customer-accessible bug repository and instead offers customers correspondence through email or phone [5].

This research will attempt to build a model or taxonomy of customer-perceived fault severities through the use of real-world faults (from open-source web applications) in a human study.[2] In the human study, subjects will be asked to view pairs of website screenshots corresponding to the *current-next* page idiom, and identify the severity of faults encountered on the *next* page. For the initial study, real-world faults will be collected from technical forums of open-source benchmark web applications. Human subjects will be asked to categorize faults according to how likely they are to drive away a customer. Once the different levels of fault severities are populated with real-world errors, I will examine the faults in each category to determine commonalities that can be used to create a model of severity based on features of the fault. For example, faults in purchasing a product from an online vendor, such as a shopping cart not updating or a payment not being processed, are likely to be much more distressing for customers than a simple typo in a product description. I will also capture the number and characteristics of each different type of fault. Although code synthesis is rising in popularity, the scope of this step will be limited to general faults in hand-crafted web applications, with the aim of extending my model to synthesized code in later work. Additionally, I propose to discover more about how web errors are developed and reported in industrial development. Although the human study is likely to give a good estimate of this distribution, I will also survey web application developers that are currently working on web applications for this information. In essence, this survey will ask developers to report how many faults of each different severity level they encountered during the entire lifetime of their current project.

3.2.4 Step 4: Compare the severities of real-world faults to seeded faults using human data.

After creating a model of the severity of real-world faults in Step 3, I propose to validate or refute the underlying assumption that fault seeding is an accurate way to measure test suite efficacy. While

[2] I have obtained UVA IRB approval for all human studies described in this proposal (IRB SBS 2009009200, March 12 2009).

fault seeding assumes that all faults have the same severity [24, 63], this assumption may be dangerous for web applications if the seeded faults happen to be of low severity. By contrast, seeding only high-severity faults is not necessarily a disadvantage. To measure the severity levels of seeded faults, the human subject study from Step 3 will include seeded faults mixed in with the real-world faults. Half of these seeded faults will be manually generated, and the other half obtained from automatic source code mutation. Subjects will not know if they are rating a real-world fault or a seeded one during the experiment.

The severity ratings for faults will be broken down per benchmark, and analyzed to see if:

- the severities of seeded errors have uniform distributions, or
- the severity distribution of seeded errors matches the distribution of real-world errors, according to the results of the survey from Step 3.

In cases where the same benchmark application was used with both real-world and seeded faults, the distributions will be compared directly.

3.2.5 Step 5: Identify underlying technologies and methodologies that correlate with high-severity faults.

Testing web applications is sometimes perceived as lacking a payoff [29] and developers often forgo it altogether [51]. Because it is unlikely that the economic conditions surrounding web application development will change in the near future, providing developers with guidelines to build better systems in the absence of testing remains an important consideration. While advances in reducing the cost of testing increase the likelihood of testing approaches being adopted, offering alternatives to achieve high quality systems with less of a reliance on testing is an orthogonal approach, and the two are not mutually exclusive.

Based on the model of web application error severities derived in Step 3, I propose to further analyze high severity errors in an attempt to tie them to underlying code, programming languages, components, or software engineering practices. To do so, I plan to use error features available from the technical forums of these open source benchmarks for the real-world errors in Step 3, combined with surface features of the errors themselves, and map these features into my severity categories. Although not all bugs reports will provide specifics on how the error was discovered or patched, often the screenshots of each error can offer valuable information, such as a stack trace, which can then be pieced together into a narrative of why the error occurred. As I am examining errors in bug repositories, I also propose to measure the percentage of faults reported that are user-visible, as these are the types of faults my work is able to address. In this step I will also ground the dominant technologies in the current web development environment to characterize the stability of the model I am building.

3.2.6 Step 6: Identify testing techniques to maximize return on investment by targeting high-severity faults.

Returning to the example of a shopping cart error versus a misspelled word, my proposed fault model from Step 3 will be able to identify the severities associated with each of these types of faults. I thus suggest to make recommendations on how to find higher-severity defects during testing. For example, higher priority may be given to test cases that exercise the business logic of

the shopping cart. Although this example is a natural conclusion, it is an important one, as other metrics used in test case selection, such as code coverage, may not give high priority to the shopping cart business logic code. As another example, it is unknown how the typical *white screen of death* (WSOD) exhibited by faulty web applications affects customer perception of the website overall. Such errors have varied causes — for example, the server may be overloaded, or if the page was written in PHP, a simple syntax error in the code can prevent any information from being displayed. If such occurrences are found to drive away customers and the application is using PHP, it may be advisable to re-run all test cases executing the modified PHP files and use program slicing to determine which subset of test cases should then be executed [65].

Applying a model of fault severity to testing introduces a new metric for the (web application) test suite reduction research community. There are two options to associate test cases with the severities of faults they are likely to expose:

- either the user patterns (or use cases) of the test suite [21] will have to be analyzed and assigned severity ratings, or
- severity ratings will have to be associated with parts of the code and then the code must be mapped to exercising test cases.

Automatic analysis of user session data and `URL`s as test cases is inherently easier than automatic analysis of dynamic-code-generating web application source code. In the former the `URL`s *are* the test cases; therefore, an analysis to reduce the test suite size can target these items directly. For the latter, in order to reduce the test suite through metrics that depend on the characteristics of the source code, there must be a way to associate which test suite exercises which piece of code. The *Tarantula* fault localization algorithm [14] can be applied to this problem to associate test cases with the parts of code they execute. Which approach I will use depends on whether or not fault severity can be determined by examining the `URL`, or if different severities are more associated with certain parts of the source code (such as database accesses, authentication, business logic, etc).

3.2.7 Experimental evaluation

- **Steps 1 and 2:** To evaluate my reasonably precise comparator I will use the *recall* and *precision* metrics from the domain of information retrieval. My comparator will be successful if it is able to minimize the number of errors it fails to identify in addition to minimizing the number of non-errors it mistakenly reports. For Step 1 I will also conduct a longitudinal study across multiple released versions of the same benchmark to approximate the cost savings over a naïve `diff`-like comparator my technique offers.
- **Step 3:** The accuracy of the fault severity model will be evaluated by training and testing on separate subsets of human subjects. Information from the training set will be used to construct a model that classifies faults into severity categories. The fault severity model will be successful if it can correctly identify most high severity faults in the held-out testing set. Since not all humans agree on fault severities, my predictive model will be successful if it is able to agree with humans about as often as they agree with each other on average. The distribution of fault severities in the human study will be compared to the distribution of faults collected from web application developer surveys.

- **Step 4:** The assumption that using fault seeding to measure test suite efficacy in web applications will either be validated or refuted based on whether or not seeded faults exhibit at least as many high severity variants as those in the real world. In addition, the distributions of seeded fault severities versus real-world fault severities obtained from the survey in Step 3 will be compared for similarity using standard statistical approaches.

- **Steps 5:** In this step I propose to recommend software engineering guidelines to reduce high-severity faults in the absence of testing. Because the measure of customer-perceived severity is a subjective one, this implies the need for several data points. Although it would theoretically be possible to compare industrial web applications developed with competing methodologies (one intended to minimize faults of high severity, the other serving as a control), it is not feasible to obtain enough benchmarks within the scope of this dissertation for statistically significant results. Instead, I will attempt to survey developers and ask them to rate their adherence to my guidelines and their observed error rates under the assumption that I will be able to find enough volunteers. I will thus be able to determine which of my guidelines are most effective at reducing high severity faults. In the meantime, being able to identify commonalities between faults of a specific severity serves as a proof-of-concept for the derived guidelines.

- **Step 6:** Fault severity as a metric for test suite reduction can be compared to other approaches in this area by quantifying the number and severity of faults exposed by each test reduction technique. My approach of applying an error severity model to web application test suite reduction will be successful if I am able to reveal more high-severity faults per benchmark application than comparable existing techniques.

Chapter 4 Preliminary Work

This section will describe preliminary research conducted in studying web-based application errors in the context of web-based application fault detection. Experiments in Steps 1 and 2 will show that a feature-based analysis of web-based application errors can be applied to fault detection during regression testing these systems. Step 2 will also show that web-based application errors have commonalities that span across project bounds. Steps 3 – 6 will continue to present analogies across web applications to develop a model of customer-perceived web-based application error severities. Although work remains to be done for Steps 3 though 6, this chapter will demonstrate that a careful study of errors and their detection in web-based applications can reduce the costs associated with testing these systems.

4.1 Step 1: Construct a reasonably precise oracle-comparator using tree-structured XML/HTML output and other features.

The goal of this step is to reduce the cost of regression testing web-based applications by exploiting the special structure of web-based application output to precisely identify errors. I hypothesize that errors in these systems have quantifiable features that can be used to derive a model of errors in a specific application. To do so, I built a reasonably precise comparator for each target application

that reduces the number of false positives associated with naive `diff`-like approaches. The next section describes how I built my reasonably precise comparator to model errors through structural and semantic features of the pairs of oracle-testcase output. Section 4.1.2 describes my experimental setup and results.

4.1.1 Comparing pairs of documents

My approach classifies test case output based on structural and semantic features of tree-structured documents. To do so, I parse the XML/HTML output of both the oracle[1] and test case to align these input trees by matching up nodes with similar elements. My goal is to find the minimal number of changes required to align the two documents, and to do so, I adapt the DIFFX [10] algorithm for calculating structural differences between XML documents. I then calculate the value of 22 features for each pair of trees. My features fall into two main categories: those that measure differences in the tree structure of the document, and those that emulate human judgment of interesting differences between pairs of XML/HTML output. Features may be correlated positively or negatively with test output errors, depending on the target application. Most of my features are relatively simple, and I summarize the most important ones in Figure 6.8. For each pair of oracle-testcase output, each feature is assigned a numeric weight that measures its relative importance. Whenever the weighted sum of all feature values for a pair of oracle-testcase output exceeds a certain cutoff value, my model decides that the output is worth examining by a human. The weights and cutoff value are learned empirically; I return to this issue when discussing my experimental setup below.

4.1.2 Experimental Setup and Results

I evaluated my reasonably precise comparator on ten open source benchmarks from an assortment of domains, summarized in Figure 6.6. For each benchmark, I manually inspected outputs for each version; the older version was assumed to be the oracle output and the newer version the test output. I marked each output pair as "definitely not a bug" or "possibly a bug, merits human inspection". I conservatively erred on the side of requiring human inspection. My initial experiments involve 7154 pairs of test case output, where 919 were labeled as requiring inspection.

I then evaluated my reasonably precise-comparator as an information retrieval task by creating a linear regression model from my feature values and identifying an optimal cutoff to form a binary classifier. Because I test and train on the same data, I used 10-fold cross validation [34] to detect and rule out any bias introduced by doing so. I then use precision and recall to evaluate my precise-comparator's effectiveness at correctly labeling pairs of test case output. *Precision* can be trivially maximized by returning a single test case, while *recall* can similarly be maximized by returning all test cases. I avoid these scenarios by combining the two measures and taking their harmonic mean. The result is the F_1*-score*.

Figure 4.1 shows my precision, recall, and F_1-score values for my dataset, as well as `diff`, `xmldiff` [4], coin toss, and biased coin toss as baseline values. The biased coin toss returns "no" with probability equal to the actual underlying distribution for this dataset: $(7154-919)/7154$. My precise-comparator is three times as effective as `diff`, and is overall quite powerful with its F_1-score being close to perfect. Cross validation revealed that there was little to no bias from overfitting (a delta of 0.0004).

[1] The oracle output is output from a previous, trusted version of the code.

Comparator	F_1-score	Precision	Recall
precise-comparator	0.9931	0.9972	0.9890
precise-comparator 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 4.1: The F_1-score, precision, and recall values for my reasonably precise-comparator on my entire dataset. Results for `diff`, `xmldiff`, and random approaches are given as baselines; `diff` represents current industrial practice.

I also analyzed which features influenced my comparator the most through an analysis of variance (see Figure 4.2). My most powerful feature was whether or not the changes between the pairs of output involve only natural language text — this feature is strongly negatively correlated with errors, and explains my significant advantage over a `diff`-like comparator. In contrast, the DIFFX-move feature was frequently correlated with test case errors, as these changes show up as a side-effect of other large changes such as the introduction or deletion of one element often moves neighbors. Despite the high F-ratio of the DIFFX-move feature, its model coefficient was an order of magnitude smaller than those of insert or delete, which implies that other features also had to be present in order for the test case output to merit inspection.

My analysis of variance relies on three assumptions: (1) that my samples are independent, (2) that the underlying distribution of the features is normal, and (3) the variances of each feature are similar. I will explicitly test assumptions 2 and 3 by conducting an Anderson-Darling normality test and explicitly measuring the variance of each feature, respectively. If the underlying distribution is not normal, a different ANOVA will be employed (such as the KruskalWallis test), while any feature whose variance deviates from the norm will be discarded from the ANOVA analysis.

I also conducted a longitudinal study to measure the hypothetical amount of effort that could be saved when my reasonably precise-comparator is applied in an industrial setting. I considered the situation where an organization uses my reasonably precise-comparator on all successive product releases, and I assume that humans manually annotate a small percentage of test case output (20%) flagged by `diff` for each version, using this as training data for the comparator. Subsequent releases of the project retain training information from previous releases, and incorporate the false positive or true positive results of any test case that my tool deemed to require manual inspection.

The amount of effort saved by developers using my reasonably precise-comparator is measured 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 in my reasonably precise-comparator occurs when the cost of looking at the false positives flagged by `diff`, but not my approach, exceeds the cost of any missed test cases:

$$(TruePos + FalsePos) \times LookCost + FalseNeg \times MissCost$$

is less than the cost of $|\texttt{diff}| \times LookCost$. Therefore, I are profitable when:

Feature	Coefficient	F	p
Text Only	- 0.217	179000	0
DIFFX-move	+ 0.003	170000	0
DIFFX-delete	+ 0.017	52700	0
Grouped Boolean	+ 0.792	9070	0
DIFFX-insert	+ 0.019	862	0
Error Keywords	+ 0.510	410	0
Input Elements	+ 0.118	184	0
Depth	- 0.001	128	0
Missing Attribute	- 0.045	116	0
Children Order	- 0.000	77	0
Grouped Change	- 0.078	62	0
Text/Multimedia	+ 0.009	19	0
Inversions	- 0.000	6	0.02
Text Ratios	- 0.001	6	0.02

Figure 4.2: Analysis of variance of my 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 ($p \geq 0.05$) are not shown.

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

I assume $\mathit{LookCost} \ll \mathit{MissCost}$, so I would like this ratio to be as small as possible (see Figure 6.9). For example, when applying my technique to the last release of HTMLTIDY, my approach is profitable if the ratio is about 1/1000 — that is, if the cost of missing a potentially useful regression test report is no greater than 1000 times the cost of triaging and inspecting a test case I am able to save developers effort. A ratio of 0 is optimal with respect to false negatives and is always an improvement over `diff`. My reasonably precise-comparator generally improves on subsequent releases, sometimes completely avoiding false negatives. My model is at its worst, however, when there is a large relative increase in errors between two versions (see the fourth release of HTMLTIDY)— such a situation can exist during a rushed release that breaks existing code.

Previous work on bug report triage has used a *LookCost* to *MissCost* ratio of 0.023 as a metric for success [31]. My average performance (0.0183) is a 20% improvement over that figure, and when I exclude the HTMLTIDY outlier mentioned above I achieve a ratio of 0.0015, exceeding the utility of previous tools by an order of magnitude.

4.2 Step 2: Exploit similarities in web application failures to avoid human annotations when training a reasonably precise oracle-comparator.

The goal of this step is to further automate regression testing of web-based applications by relying on the predictable and similar ways in which they fail to train a reasonably precise oracle-

comparator with out the need for manual annotation. Existing reasonably precise-comparators for web applications typically have average F-measures of up to 0.91, in terms of finding manually-seeded faults, in the absence of manual training, although it is impossible to know which oracle combinations yielded the best results without evaluating all of them, manually examining the number of false positives returned by each one [60]. `diff`-based approaches are wrong 70–90% of the time in my experiments. The next section describes how I apply data from unrelated web applications to train a reasonably precise oracle-comparator for a separate target application. Section 4.2.2 describes my experimental setup and results.

4.2.1 Training a Reasonably Precise Oracle-comparator Without the Need for Manual Annotation

In this step I use the same feature-based, linear regression model from Step 1 as my comparator. Instead of training the comparator with manually-annotated data from the application-at-test, I use previously annotated data from unrelated applications. This approach is straightforward and Section 4.2.2 will demonstrate that it is feasible to use training data from unrelated web-based applications to test one of interest.

One complication may arise, however, with this approach: when the application-at-test does not exhibit errors in the same way as the benchmarks used to generate the training data. In such a situation I propose to use defect seeding to supplement the corpus of training data with application-at-test-tailored output. Note that this does not change the automatic nature of the approach significantly, as the process of fault seeding can be automated. To do so I implemented defect seeding through a subset of mutation operators described by Ellims *et al.* [12]. For example, 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`.

Each mutant version of the source code contains only one seeded fault, and is compiled and re-run through the regression test suite. The process of mutation is quite rapid; I am able to obtain 11,000 usable faulty outputs within 90 minutes on a 3 GHz Intel Xeon computer. Section 4.2.2 will show I only need a very small subset of mutants to improve my comparator's performance.

4.2.2 Experimental Setup and Results

I used the same benchmarks from Step 1 as my corpus of training data. My *testing* benchmarks are summarized in Figure 6.7. Although I used ten benchmarks as my training corpus, only two of them (HTMLTIDY and GCC-XML) had a statistically significant number of output pairs that were labeled as errors (given by the "Test Cases to Inspect" column) to qualify as testing subjects. I supplemented these two benchmarks with two open source web applications (CLICK and VQWIKI) as a "worst-case scenario": none of the training benchmarks are web applications, so successful performance on them further supports my claim about wide-reaching application similarities.

My experiment results are summarized in Figure 4.3. My tool is anywhere from over 2.5 to almost 50 times as good as `diff`, and for the web applications I achieve perfect results. While my F_1-score for GCC-XML was three times better than that of `diff`, its recall score of 0.84 implies that I may be missing a significant number of actual errors. For this benchmark I applied the mutation procedure described in the previous section. Figure 4.4 shows my F_1-scores when adding between 0 and 5 defect-seeded output pairs to the set of training data (0 is provided as a baseline). The

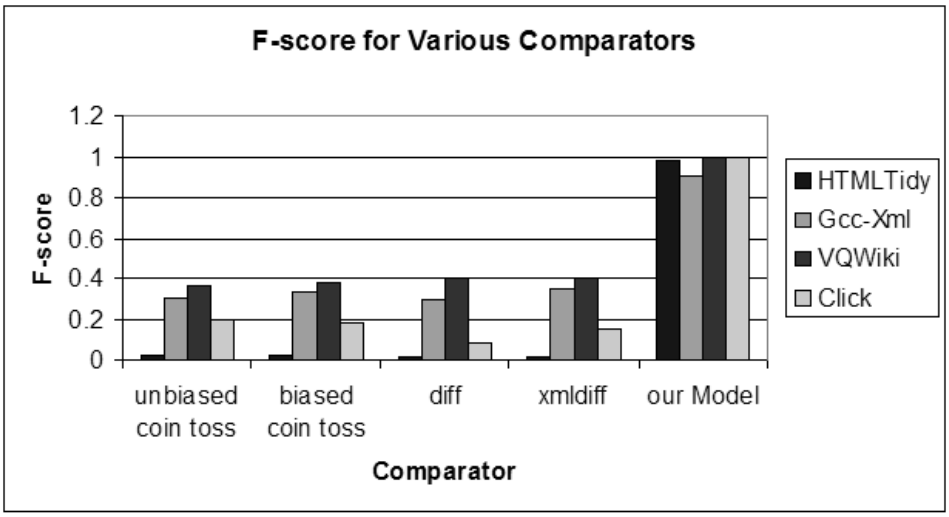

Figure 4.3: F_1-score on each test benchmark (HTMLTIDY, GCC-XML, VQWIKI, CLICK using my Model, and other baseline comparators. 1.0 is a perfect score: no false positives or false negatives.

large margin of error when adding only one mutant output pair implies that performance relies on selecting the most useful mutant outputs to include as a part of the training data set, but selecting any output is always advantageous. Additionally, no performance gains were witnessed after adding 5 mutants, with a near-perfect F_1-score at that point.

4.3 Step 3: Model real-world fault severity based on a human study.

Step 2 suggests that web-based applications have underlying similarities in the way failures manifest. The goal of this step is to build a model of web fault severity through a human study, expanding this concept that errors in web applications have predictable properties. At the time this document was written, this human study was currently under way. Four hundred real-world faults were collected from over 17 open-source PHP, Java, and ASP.NET web applications from different domains, summarized in Figure 6.10. Faults were obtained by systematically browsing the technical forums for each benchmark to include both faults from the beginning of the development of the project, as well as the most recent faults, in equal distribution. In selecting faults, I iterated through the forum entries in order, using each fault where either a screenshot was provided, or the post described the fault in enough detail in order for me to re-create it in a screenshot.

Section 4.4 explains the setup of the human study, as real-world and seeded faults were anonymously combined and presented concurrently to test subjects. Although I have yet to analyze the results of this human study, in manually collecting the faults I have noticed very similar types of faults occurring across my benchmarks and suggest that having 100 faults would have been representative enough to derive a model.

The appendix contains a copy of the survey targeted at developers for estimating the distribution of fault severities in the real world. It uses the same severity rating as the human subject study.

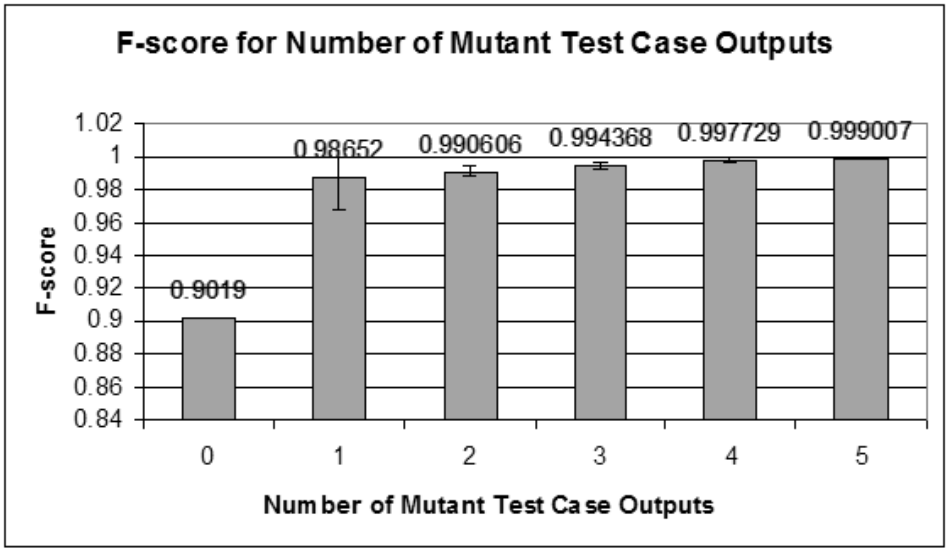

Figure 4.4: F_1-score for GCC-XML using my model with different numbers of test case output pairs from original-mutant versions of the source code. The "0" column indicates no mutant test outputs were used as part of the training data. Each bar represents the average of 1000 random trails; error bars indicate the standard deviation.

4.4 Step 4: Compare the severities of real-world faults to seeded faults using human data.

The goal of this step is to validate or refute the underlying assumption that fault seeding is an accurate way to measure test suite efficacy. In addition to the 400 real-world faults collected, 200 automatically-generated faults, equally distributed in six of the benchmarks in Figure 6.10 (denoted with an asterisk), were introduced through the same source code mutation described in Section 4.2.1. Two hundred manually-seeded faults were similarly obtained for those six benchmarks by instructing three graduate students with programming experience to insert one fault per mutant version of source code according to the fault seeding methodology in [32, 57]. Manually-generated test suites were then replayed for these 6 applications to collect the manually-seeded faults.

These 400 real-world and 400 seeded faults were then combined with 100 correct outputs randomly chosen from the 17 benchmarks, and then divided into eighteen groups of 50 pairs of screenshots a piece. Once the test items were randomized, human subjects were asked to rate the perceived severity of faults they noticed, if any, according to the Likert scale in Figure 6.13. Participants were instructed to use their judgment and past experiences to rate faults; the appendix contains a copy of the instructions provided to them.

Chapter 5 Expected Contributions and Conclusion

This research will explore and analyze errors in web-based applications in the context of fault detection. Although I currently focus on testing web-based applications, the work can be extended to other areas, such as usability and human-computer interaction, as well as other sub-fields, such

as graphical user interfaces. The main contributions are expected to be:

- **Improve fault detection by constructing a reasonably precise oracle-comparator.** My work focuses on the *semantic*, rather than the syntactic, difference between pairs of test case output. In doing so, I can build a model of errors in a web-based application that can be used by a reasonably precise oracle-comparator to reduce the number of false positives returned by more naive approaches. By exploiting similarities across seemingly unrelated applications, I propose to further automate such regression testing by obviating the need to provide (manually-annotated) training data.
- **Develop a model of customer-perceived severities of web application faults.** Severities in web application errors have not been previously explored, despite the customer-oriented nature of these systems. I expect to produce a model that agrees with an average human annotator at least as well as humans agree with each other.
- **Validate or refute fault injection as a standard for measuring web application test suite quality** by assessing whether or not the assumption that all injected faults have the same non-trivial severity holds. If fault seeding is found to be a non-representative application of severity in web application defects, this contribution implies the need to change the metrics by which competing test cases are evaluated in the web testing field. I expect to discover that naïve fault injection does *not* always produce faults of the same severity, as judged by users. I further expect to propose, based on my formal model of fault severity, ways in which fault injection can be guided to produce higher-severity faults.
- **Propose new software engineering guidelines for web application development**. The first set of guidelines will target high-severity fault avoidance during product design. These guidelines will be designed under the assumption that developers are choosing not to test their system, and are therefore orthogonal to testing-based approaches. The second set of guidelines will target making testing efficient. These guidelines may be incorporated into test case design, selection, and prioritization (test suite reduction). In this instance my fault severity model becomes another metric by which testing techniques can me measured. I expect to produce less than a dozen such guidelines.

While the proposed work focuses on web applications, it may be possible to extend some of the results and contributions to other domains. Web-based applications and graphical user interfaces (GUIs) are both used in a visual, interactive manner. It is likely that visible faults in both systems manifest themselves in similar ways. Previous work has made the assumption that fault severities are equal in the domain of graphical user interfaces [64], but to my knowledge no work to date has explored such severities as as a characteristic of the application under test. Similarly, the models constructed in this research may have a general applicability beyond web testing, in areas such as human computer interaction and usability. For example, faults with a certain severity rating can be analyzed for similarities with common usability issues, such as the inability to locate a link. This methodology will allow web application developers to focus their usability analysis on the most critical components of their human interface.

Chapter 6 Appendix

6.1 Web-based Applications

The terms "web-based application" and "web application" are frequently used interchangeably in the web community. For the purposes of this proposal, 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. Testing of such applications has primarily focused on model-based techniques [62].

6.2 Three-tiered Web Applications

An example three-tiered web application is shown in Figure 6.1. The first row in the diagram represents the client-server model. Text in bold are various types of software vendors, many of which are off-the-shelf, opaque components. Example programming languages are associated with each component in the architecture

6.3 Dynamic Content Generation in Web Applications

Figure 6.2 shows server-side dynamic content generation. Adapted from http://blog.search3w.com/dynamic-to-static/hello-world/

6.4 Oracles

An oracle-comparator is shown in Figure 6.3. A human (or in some cases software) provides test input to the system. If capture-replay is being used, these inputs are recorded and then can be re-run on demand. The application is run on the test inputs and produces output, usually in the form of XML/HTML for web-based applications. 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 is passed or failed, and a human judges the acceptability of the output.

6.5 Fault Taxonomies for the Web

Figure 6.4 is a fragment of the initial taxonomy of Marchetto *et al.* [41]. Only selected (sub)characteristics and classes of faults are shown. This table is reprinted from [41].

6.6 Proposed Research Outline

My proposed research outline is shown in Figure 6.5.

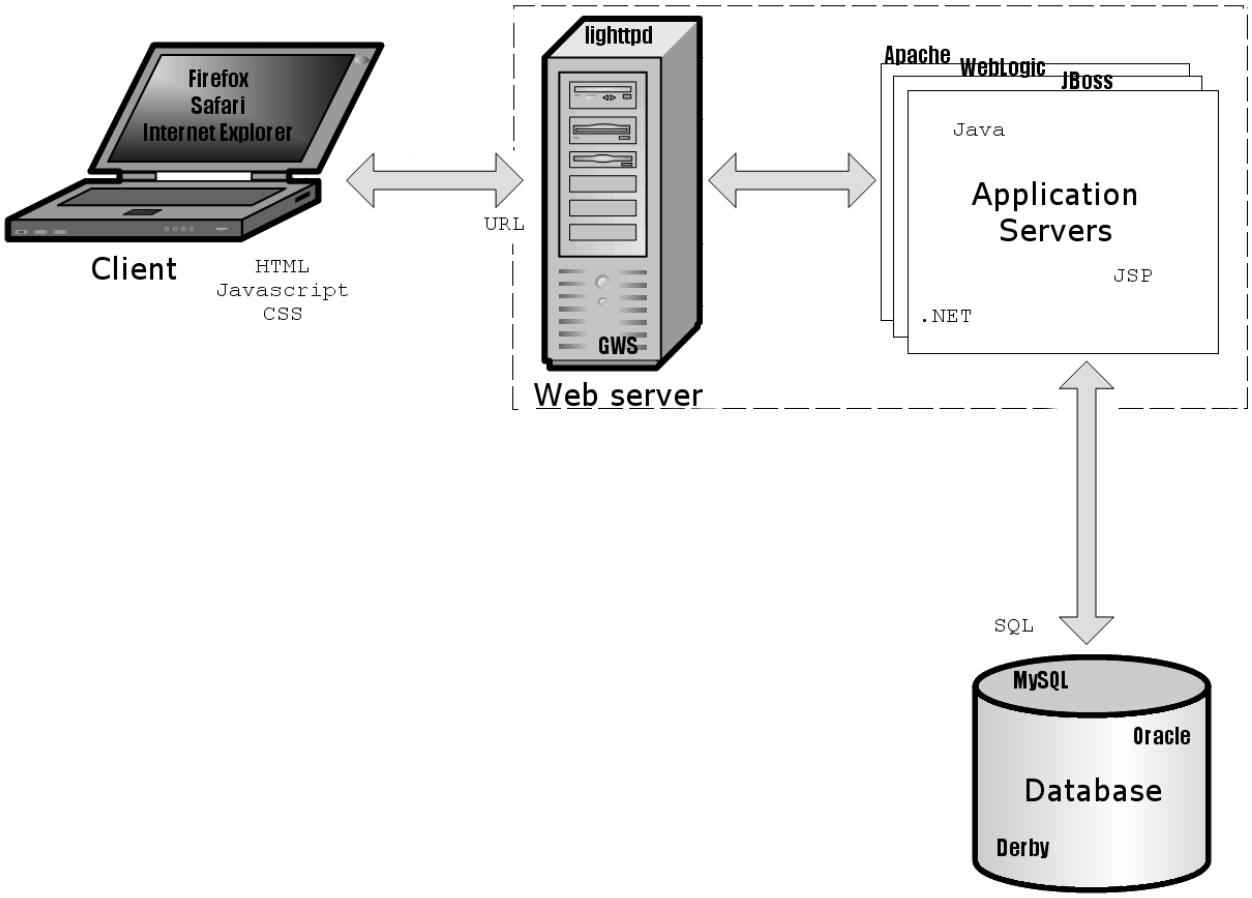

Figure 6.1: Three-tiered web application

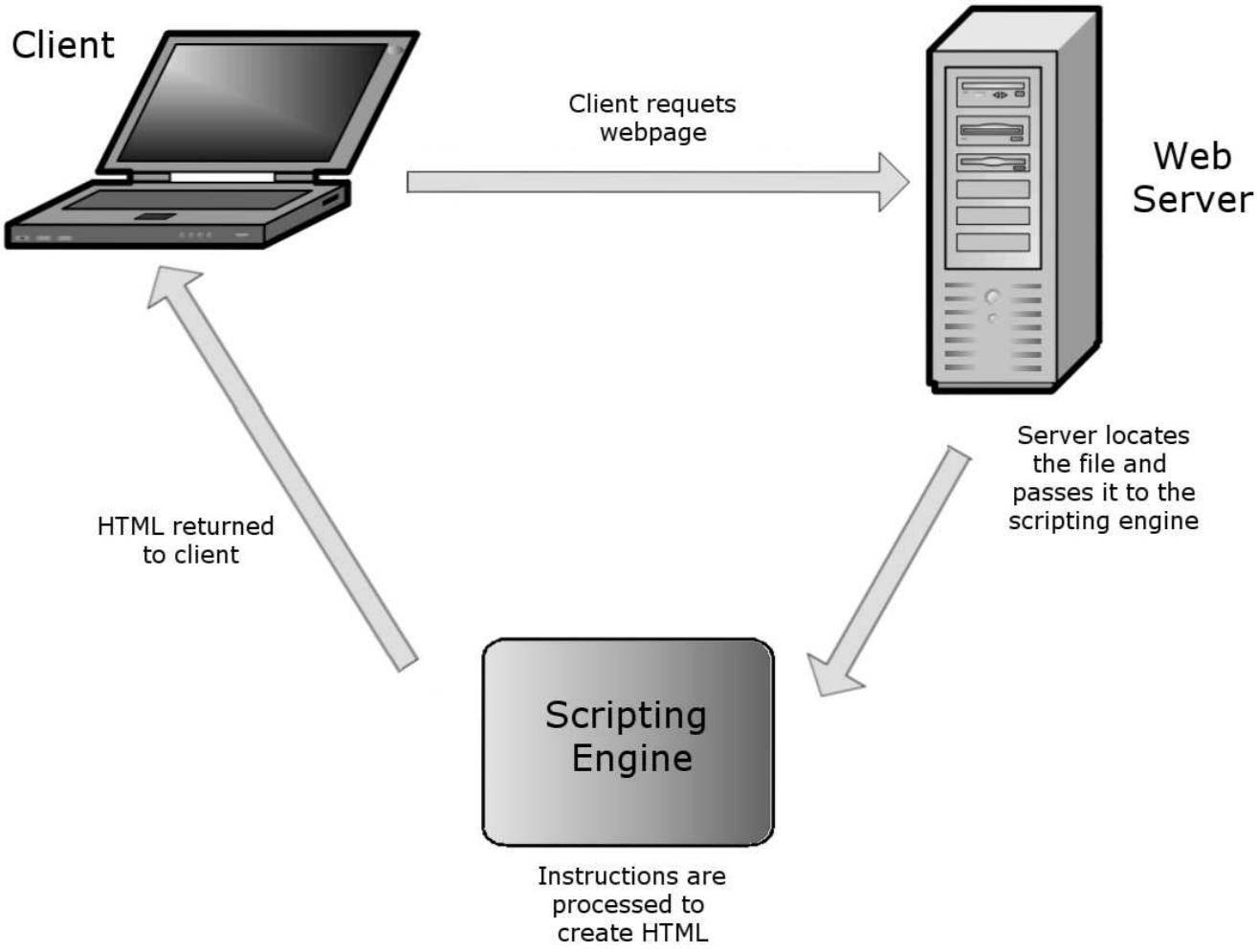

Figure 6.2: Dynamic content generation

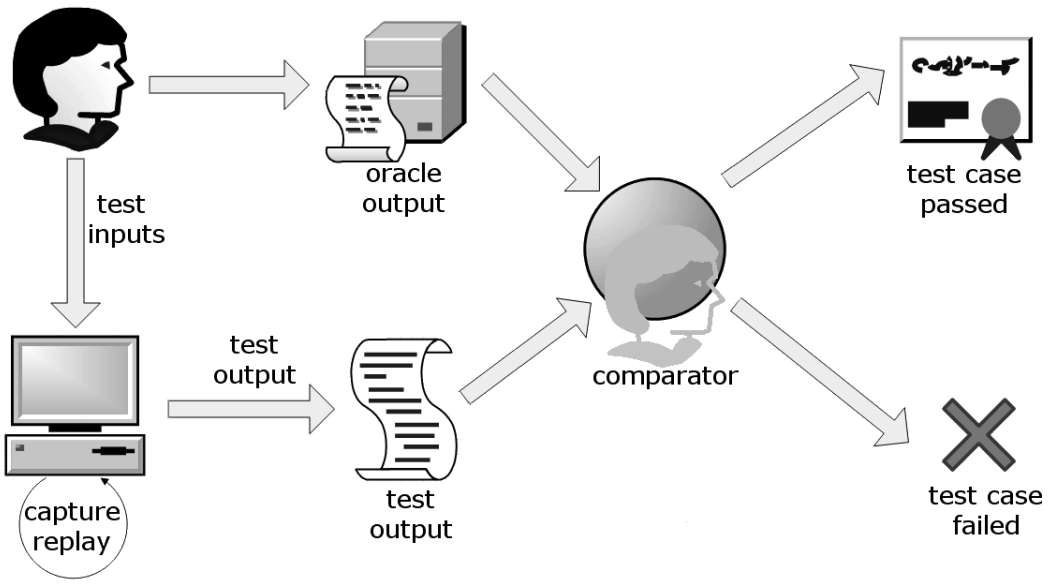

Figure 6.3: The oracle-comparator

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 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 interactions or management 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 be HTML-based	f1.faults related to 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 the 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 of management

Figure 6.4: Fragment of the initial taxonomy of Marchetto *et al.* [41]

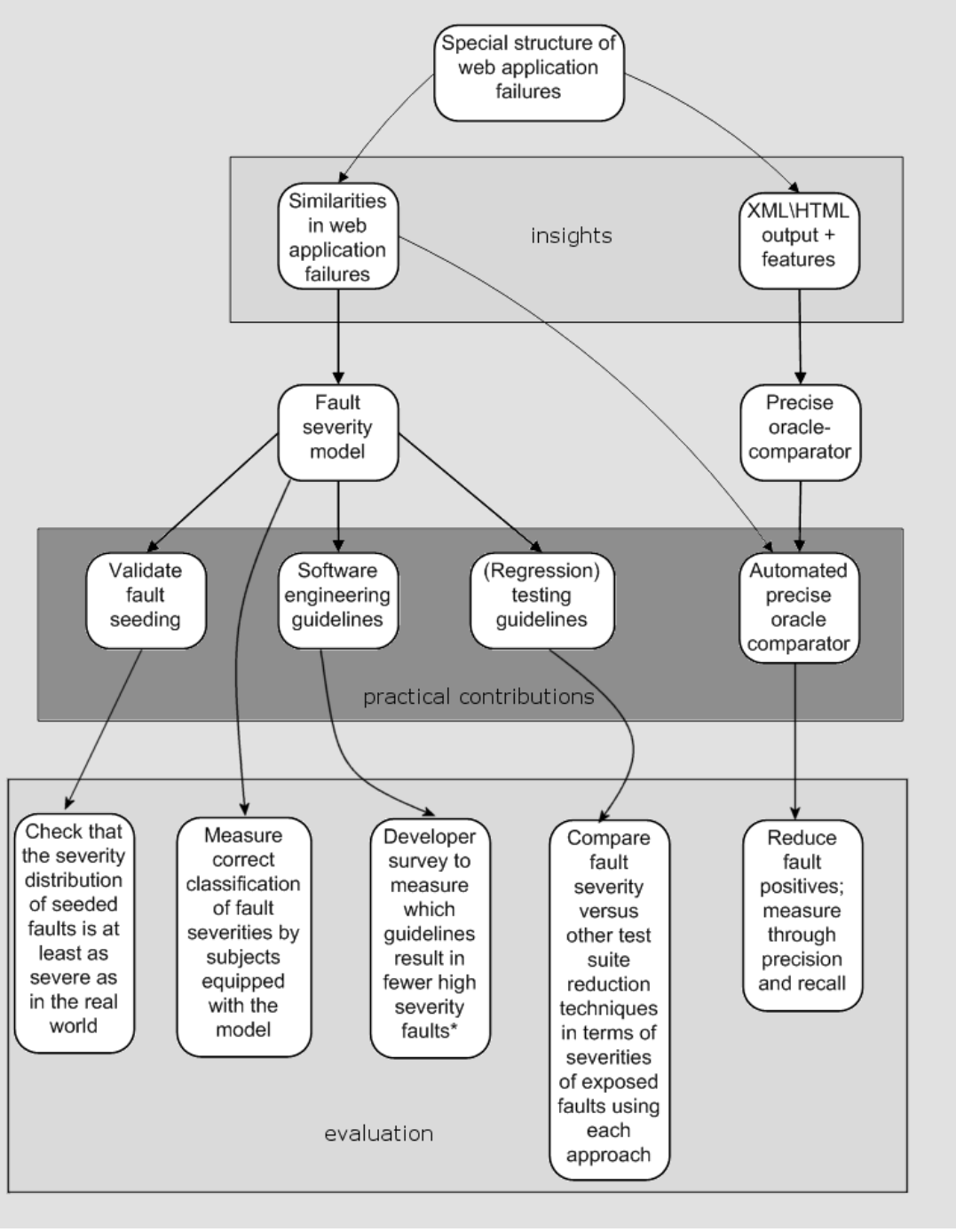

Figure 6.5: Proposed research outline

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 6.6: Benchmarks used in step 1

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 6.7: Benchmarks used in step 2

6.7 Benchmarks in Step 1

Figure 6.6 shows the benchmarks used in our experiments in Step 1. The "Test cases" column gives the number of regression tests we used for that project; the "Test cases to Inspect" column gives the number of those tests for which our manual inspection indicated a possible bug.

6.8 Benchmarks Used in Step 2

The benchmarks used as test data for our experiment are shown in Figure 6.7. The "Test cases" column gives the number of regression tests used; the "Test cases to Inspect" column counts those tests for which our manual inspection indicated a possible bug. When testing on HTMLTIDY or GCC-XML, we remove it from the training set.

6.9 Features used in Steps 1 and 2

Features between pairs of XML/HTML test case outputs used to make comparator judgments are shown in Figure 6.8.

The number of inserts, deletes, and moves required to transform one tree into the other
The number of element inversions of non-text nodes, calculated by removing nodes that are shared in a longest common subsequence in a sorted list of all tree elements
Grouped changes to a set of contiguous elements in the tree
The maximum depth of changes in the tree
Presence of changes to only text nodes
Presence of changes to child ordering
The ratio of displayed text and the ratio of text to multimedia between two versions
The number of programming-language based error keywords (i.e. “exception” or “error”) that occur in the newer version but not the older
Number of changes to functional elements such as buttons
Presence of changed or missing attribute values of an element

Figure 6.8: Features used in steps 1 and 2

6.10 Longitudinal Study Results in Step 1

Figure 6.9 shows the simulated performance of my technique (PC) 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 my manual annotation indicated should be inspected (i.e., might indicate a bug). The ‘Inspected’ column gives the number of tests that my technique 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 my technique becomes profitable (lower values are better).

6.11 Open Source Web Application Benchmarks used in Steps 3 and 4

Figure 6.10 shows open-source applications used to collect real-world faults. Items with an asterisk were benchmarks in which faults were also seeded.

6.12 Web Application Fault Severity Study

6.12.1 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

Benchmark	Release	Test Cases	Should Inspect	True Positive		False Positives		False Negatives		Ratio
				PC	diff	PC	diff	PC	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 6.9: Longitudinal study results in step 1

Name	Language	Description	Real-world Faults
Prestashop*	PHP	shopping cart/e-commerce	30
Dokuwiki*	PHP	wiki	30
Dokeos	PHP	e-learning and course management	22
Click*	Java	JEE web application 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 management framework	30
ASPgallery	ASP.NET	gallery	30
Yet Another Forum	ASP.NET	forum	30
ScrewTurn	ASP.NET	wiki	30
Mojo	ASP.NET	content management system	30
Zen Cart	PHP	shopping cart/e-commerce	30
Vanilla*	PHP	forum	0
other	-	-	9

Figure 6.10: Open source web applications used in steps 3 and 4

University of Virginia. It is possible that our results are biased towards younger people, although these seem 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 6.13. 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. It is also possible that very few or no faults will be rated with the highest severity rating; in this case, levels 3 and 4 can be collapsed into one rating.

This study will attempt to build a predictive model of fault severity by analyzing the human judgments of severities of faults in my dataset. In doing so, I must be confident that the difference in ratings between different faults are due to different true severities of the faults themselves, rather than due to some variation in ratings from the subjects. To reject the null hypothesis that differences in severities across faults are a consequence of random chance related to not having enough human subjects, I propose to conduct a two-way analysis of variance to calculate estimates on the variance due to differences between human ratings on the same fault. I will use these estimates to calculate the intraclass correlation coefficient (ICC) for my dataset: a high ICC score will indicate that raters tend to agree on fault severities. The ANOVA will also provide me with a confidence interval for these values. Should my ICC be low, I will solicit more human subjects until I can be sure that voters agree frequently enough on fault severities with an acceptable level of confidence. Performing the ANOVA analysis will provide me with a value for the variance of my dataset, which will guide towards how many human subjects I need to solicit should my initial results be found to require more.

To compare the distribution of severities across automatically-injected faults, manually-injected faults, and real-world faults, I propose to use a two-sample KolmogorovSmirnov test to reject the null hypothesis that the distribution of severities of any of these three groups of faults in my study is a fixed constant. I will also use this test to test if any of the distributions within these groups are equivalent to any of the others. The KolmogorovSmirnov test will indicate the level of confidence with which each null hypothesis can be rejected.

6.12.2 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

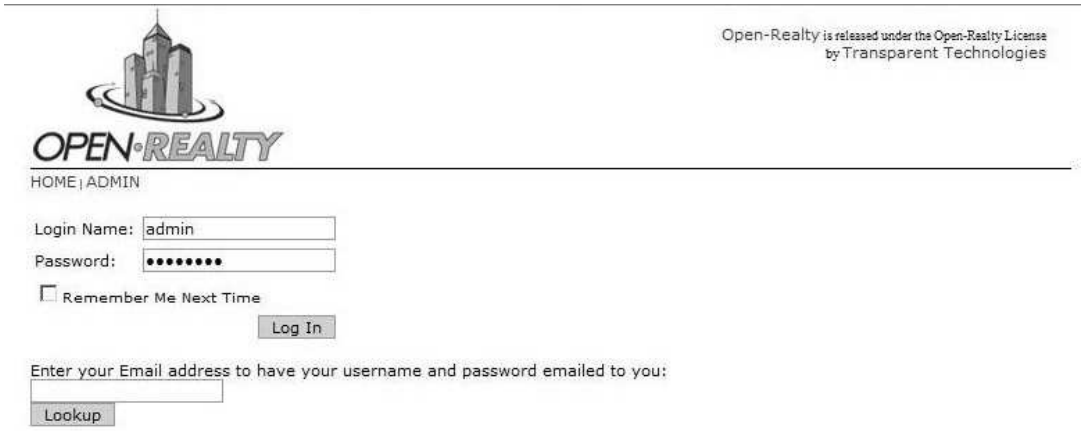

Figure 6.11: The "current" page

6.12.3 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 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 6.11 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

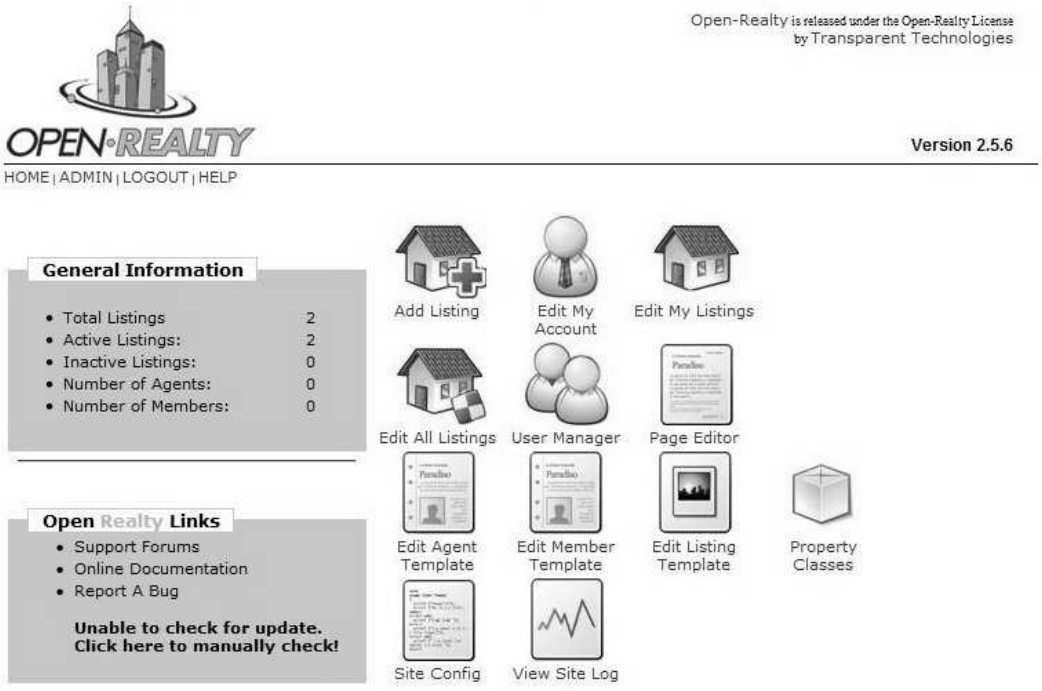

Figure 6.12: The “next” page

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 6.12 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 you believe there is a fault, you will be asked to rate the severity of that fault as we define in Figure 6.13.

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 a 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.

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 6.13: The severity rating used in our human study

6.13 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

6.13.1 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 6.13:

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 begun. 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 6.14 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.

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 6.14: The severity rating used in our human study

Bibliography

[1]

[2] Studying the fault-detection effectiveness of gui test cases for rapidly evolving software. *IEEE Trans. Softw. Eng.*, 31(10):884–896, 2005. Member-Memon,, Atif M. and Student Member-Xie,, Qing.

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

[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] Amazon.com: Help. `http://www.amazon.com/gp/help/customer/display.html`, 2009.

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

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

[8] 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.

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

[10] 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.

[11] 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.

[12] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 402–411, 2005.

[13] 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.

[14] Shay Artzi, Julian Dolby, and Frank Tip. Practical fault localization for dynamic web applications. *IBM Research Report RC24675 (W0810-107)*, October 2008.

[15] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272, 2008.

[16] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 261–272, 2008.

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

[18] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. 1999.

[19] Penelope A. Brooks and Atif M. Memon. Automated gui testing guided by usage profiles. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 333–342, 2007.

[20] 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.

[21] G. A. Di Lucca, A. R. Fasolino, and P. Tramontana. A technique for reducing user session data sets in web application testing. In *WSE '06: Proceedings of the Eighth IEEE International Symposium on Web Site Evolution*, pages 7–13, 2006.

[22] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 411–420, 2005.

[23] 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.

[24] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, 2001.

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

[26] Yuepu Guo and Sreedevi Sampath. Web application fault classification - an exploratory study. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 303–305, 2008.

[27] William G. J. Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In *ESEC-FSE '07: 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.

[28] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

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

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

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

[32] Srikanth Karre. Leveraging user-session data to support web application testing. volume 31, pages 187–202, 2005.

[33] John C. Knight and Paul E. Ammann. An experimental evaluation of simple methods for seeding program errors. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 337–342, 1985.

[34] 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.

[35] David Chenho Kung, Chien-Hung Liu, and Pei Hsia. An object-oriented web test model for testing web applications. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, pages 537–542, 2000.

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

[37] 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.

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

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

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

[41] A. Marchetto, F. Ricca, and P. Tonella. Empirical validation of a web fault taxonomy and its usage for fault seeding. pages 31–38, Oct. 2007.

[42] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

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

[44] J. Offutt, Ye. Wu, X. Du, and H. Huang. Bypass testing of web applications. pages 187–197, Nov. 2004.

[45] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, New York, NY, USA, 2002. ACM.

[46] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, 2004.

[47] S. Pertet and P. Narsimhan. Causes of failures in web applications. Technical Report CMU-PDL-05-109, Carnegie Mellon University, December 2005.

[48] R.S. Pressman. What a tangled web we weave [web engineering]. 17(1):18–21, January/February 2000.

[49] Shruti Raghavan, Rosanne Rohana, David Leon, Andy Podgurski, and Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. pages 188–197, 2004.

[50] Filippo Ricca and Paolo Tonella. Analysis and testing of web applications. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 25–34, 2001.

[51] Filippo Ricca and Paolo Tonella. Testing processes of web applications. *Ann. Softw. Eng.*, 14(1-4):93–114, 2002.

[52] Filippo Ricca and Paolo Tonella. Web testing: a roadmap for the empirical research. In *WSE '05: Proceedings of the Seventh IEEE International Symposium on Web Site Evolution*, pages 63–70, 2005.

[53] Sreedevi Sampath, Sara Sprenkle, Emily Gibson, and Lori Pollock. Integrating customized test requirements with traditional requirements in web application testing. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 23–32, 2006.

[54] Jessica Sant, Amie Souter, and Lloyd Greenwald. An exploration of statistical models for automated test case generation. volume 30, pages 1–7, 2005.

[55] 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.

[56] Luis Moura Silva. Comparing error detection techniques for web applications: An experimental study. In *NCA '08: Proceedings of the 2008 Seventh IEEE International Symposium on Network Computing and Applications*, pages 144–151, 2008.

[57] 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.

[58] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. A case study of automatically creating test suites from web application field data. In *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*, pages 1–9, 2006.

[59] 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.

[60] 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.

[61] 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. volume 0, pages 587–596, 2005.

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

[63] J. Strecker and A.M. Memon. Relationships between test suites, faults, and fault detection in gui testing. pages 12–21, April 2008.

[64] Jaymie Strecker and Atif Memon. Relationships between test suites, faults, and fault detection in gui testing. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 12–21, 2008.

[65] Paolo Tonella and Filippo Ricca. Web application slicing in presence of dynamic code generation. *Automated Software Engg.*, 12(2):259–288, 2005.

[66] Y. Wang, D.J. DeWitt, and J.-Y. Cai. X-diff: an effective change detection algorithm for xml documents. pages 519–530, March 2003.

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

[68] Qing Xie and Atif M. Memon. Model-based testing of community-driven open-source gui applications. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 145–154, 2006.

[69] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for gui-based software applications. *ACM Trans. Softw. Eng. Methodol.*, 16(1):4, 2007.