ADDRESSING HIGH SEVERITY FAULTS IN WEB APPLICATION TESTING

Kinga Dobolyi and Westley Weimer
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
{dobolyi,weimer}@virginia.edu

ABSTRACT

Extreme resource constraints in web application development environments often lead to minimal formal testing. This lack of testing compounds the problem of low consumer retention. In this paper, we analyze an existing study of the consumer-perceived severities of 400 real-world web application faults. We provide concrete guidelines to increase the perceived return-on-investment of testing. We show that even coarse-grained, automated test cases can detect high severity faults. Developers can also mitigate consumer perceptions of fault severity by presenting faults using specific idioms that minimize disruptions to application interaction. Finally, we examine the trade offs between various user-session-based test suite reduction approaches for web applications and the severity of uncovered faults. Overall, we show that even modest testing approaches are able to provide significant predicted gains in consumer satisfaction by focusing on flagging and preventing high severity faults.

KEY WORDS
Web application, design, testing, severity, reduction

1 Introduction

In the United States, over $204 billion were spent on Internet retail sales in 2008 [13], and 73% of the US population used the Internet in that same year [17]. Worldwide, Internet-based transaction orders total several trillions of dollars annually [2, 4]. At the same time, consumers are exposed to several options for online interaction, and customer loyalty to any particular website is notoriously low [22]. Unfortunately, most web applications are developed without a formal process model [26], and consumer satisfaction and retention are rarely formally addressed during the development and testing phases of production. The challenge of maintaining consumer allegiance is complicated by additional factors. For example, web applications have high availability requirements; one hour of downtime at Amazon.com has been estimated to cost the company $1.5 million dollars [25]. About 70% of top-performing web applications are subject to user-visible failures. Web application development is also vulnerable to high developer turnover rates, short delivery times, and rapidly-evolving user needs that translate into an immense pressure to change [27].

Consequently, the testing of web applications often becomes a casualty in the fight to meet deadlines in the dynamic environment such projects are subject to [27]. Although web applications are highly human-centric, consumer-perceived fault severity has not been systematically approached as a metric for selecting development and testing strategies in this domain. In this paper, we study consumer-perceived high-severity faults in web applications to direct development and testing strategies towards their reduction in the context of such demanding development circumstances. The main contributions of this work are as follows:

- We demonstrate that the distribution of consumer-perceived fault severities is *independent* of the type of application. Neither the application type (e.g., shopping cart or online forum), nor, to a lesser degree, the technologies used (e.g., PHP or ASP.net) are predictors of high or low severity faults.

- We show that certain fault *features*, distinct from the context of the fault, are related to the consumer-perceived severity of these faults. Given the same defect in the source code, we demonstrate that the presentation of the fault to the user (e.g., via a stack trace or a human-generated error message) influences the likelihood that the consumer will return to the website in the future.

- We explore the *kinds* of defects that are associated with faults of varying consumer-perceived severity levels. For example, many severe faults are due to configuration errors that are easily detected by running a simple test suite, while less severe faults may require changes to specific lines of application code, and can be more difficult to detect.

- We study the trade offs between various test case *reduction* methodology costs and the severities of the faults they uncover in the context of user-session-based testing.

We supplement our discussion of the above items with concrete suggestions that developers can implement. Our first three contributions provide the foundation for guidelines that assume few resources will be allocated to testing. Our last point provides developer guidance towards choosing an optimal testing approach when performing automated testing. This paper examines the trade offs between various test suite reduction methodologies [31] in terms of the consumer-perceived fault severities revealed by each approach.

The structure of this paper is as follows. Section 2 summarizes a previously-published human study of the consumer-perceived severities of real-world faults. In Section 3 we find no correlation between application type or programming language and the severity of faults. Section 4 demonstrates that approaches such as wrapping errors in popups while maintaining application context can yield lower consumer-perceived fault severity for otherwise-equivalent errors. Section 5 relates fault causes to severities. In Section 6 we analyze various test suite reduction approaches and their ability to produce reduced test suites that locate high-severity faults. Section 7 places our work in context and Section 8 concludes.

2 Consumer-Rated Severities of Real Faults

We hypothesize that web application fault severities have several characteristics that impact consumer retention, many of which can be controlled by developers during product development and testing. In this section, we summarize our previously-published [7] human study of 386 humans rating 400 real-world web application faults. The study itself is not a contribution of this paper. Instead, we analyze the results of the study to demonstrate that fault severities are independent of application type; that different presentations of the same fault yield different consumer-judged severities; that different kinds of faults correlate with different severities; and to evaluate test suite reduction with respect to severity.

Fault severity has been previously explored in the context of industrial systems, but not with the focus on consumers. For example, Ostrand and Weyuker examine faults distributions in large industrial software systems, where fault severities were assigned according to fix priority [23]. In later work [24], they discover that these developer-reported severities were highly subjective, frequently inaccurate, and motivated by political considerations. The human study we summarize uses consumer ratings to avoid such complications. Human subjects were presented with 400 real-world faults and asked to rate the severity of each fault on the 5-point scale in Figure 1. Severity was intentionally left formally undefined.

Each fault was presented as a scenario triple: (1) the *current* webpage (as a screenshot), (2) a *description* of the task the user is trying to accomplish in the scenario, and the action taken, and (3) the *next* webpage (as a screenshot). As a concrete example, a scenario may begin with a screenshot of a login page for a particular website, which includes the description that a valid username and password has been entered, and that the user is going to click the *login* button. The subject is then presented with the *next* page, as if the login button had been clicked. Such a *current-next* paradigm can capture the inherently dynamic context of web applications, and users were freely allowed to toggle between the two views before selecting a severity rating.

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

Figure 1. Severity scale for web application faults.

Four hundred faults were randomly but systematically selected from the bug report databases of 17 open-source benchmarks summarized in Figure 2. The description of the fault was used to obtain or replicate a screenshot and HTML code for the *current* and *next* pages, along with the associated scenario description. Descriptions were supplemented with enough detail so that faults become apparent; for example, if a user had some specific permissions on a website to perform an activity, these were made known to the human subject if they related to the perception of the fault. Users were instructed to rate faults according to their previous experience, and to assume all faults would eventually be fixed upon subsequent returns to the website, although this would occur at an unknown point between the current time and up to one year in the future. Over 12,600 severity scores were recorded from 386 anonymous humans, with at least 12 votes per fault.

Previous work presented this study and constructed a formal model that agrees with average consumer-perceived severities as often as humans agree with each other; a focus was placed on the automatic detection of high-severity faults [7]. In this paper we analyze the results of the study to support claims about the distribution and context of faults, and the efficacy of test case reduction.

3 Fault Severity Distribution by Application

Our study of 400 real-world faults revealed an approximately even distribution of low, medium, medium-high, and severe faults (labeled with average ratings of ≤ 1, ≤ 2, < 2.5, and ≥ 2.5 respectively, see Figure 1) within each of our benchmarks. Although lower-severity faults are likely underrepresented in this population, because they are more frequently not reported or recorded, the relatively high number of severe faults we witnessed provided us with a large data set of severe faults to study.

We first seek to identify correlations between high severity faults and either the type or the web application

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

Figure 2. Real-world web applications mined for faults. All applications were sources for human-reported faults taken from defect report repositories, as well as non-faults taken from indicative usage. An asterisk indicates an application used as a source for manually-injected faults in Section 6. The Vanilla benchmark is used only in Section 6.

(e.g., a shopping cart versus a forum), or the underlying technologies involved (e.g., PHP or ASP.net). Figure 3 presents the distribution of fault severities across our 17 benchmarks, normalized to 100% for each application. Figure 4 shows the Spearman correlation between various features, such as the programming language used or application type, and consumer-perceived fault severity. The application features all hover between no correlation and very weak correlation. While a limited argument could be made that ASP.net and image gallery applications are slightly more likely to have high-severity faults, it is the relative values that are important. Our results generally refute the hypothesis that certain application types have higher user-perceived fault severities. For example, among benchmarks in our dataset with at least 30 faults, both the benchmark with the highest number of severe faults (Zen) and the lowest number of severe faults (Prestashop) were e-commerce, shopping-cart based applications. Similarly, the choice of development language and infrastructure in these benchmarks did not strongly correlate with fault severity.

4 Fault Features Related to Severities

Our primary goal is to provide empirically-backed recommendations to help guide developers to produce reliable web applications under the assumption of limiting testing resources. We begin by demonstrating that faults of varying severities can be classified according to both contextual and context-independent characteristics.

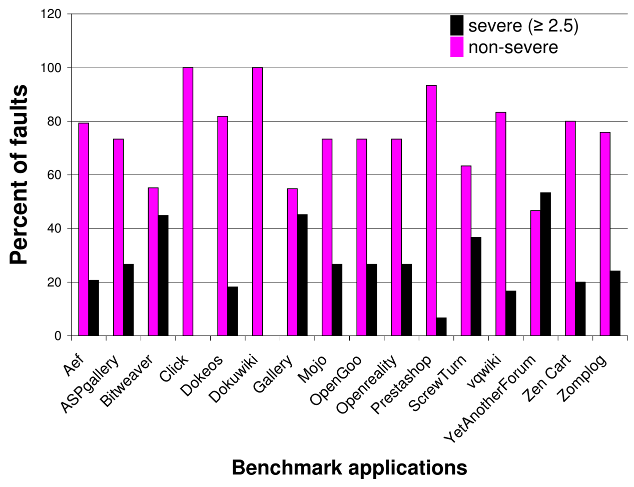

Figure 3. The breakdown of severe faults for each benchmark application. Each application is individually normalized to 100%. There is no apparent pattern between the type of application and its fault severity distribution (see also Figure 4).

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

Figure 4. Spearman's Ranking Correlation Coefficient (SRCC) between an application feature and the severity of its faults. An SRCC of 0.3–0.5 is considered weak, while 0–0.3 indicates little to no correlation [10].

We define contextual features to be visual stimuli or use-case based characteristics exposed to the user as part of the fault manifestation. Contextual features are tied to the underlying origin of the defect in the source code. Examples of such visual stimuli include stack traces, missing images, and small cosmetic errors. Use-case based features associated with faults include authentication, permission, or upload scenarios.

Context-independent features, by contrast, can be viewed independently of the actual context of the fault. Given the same source-level defect, the multiple ways the fault may be presented to users are the various context-independent fault manifestations. Context-independent features, summarized in Figure 5, include displaying the fault on the same page as the current page (in a "frame" style where the header, footer, and side menu bars are preserved), loading a new page that is visually different from the normal theme of the web application, wrapping the fault in a generic or customized human-readable error message, popups, server-generated error messages (such as HTTP

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

Figure 5. Context- and cause-*independent* fault features.

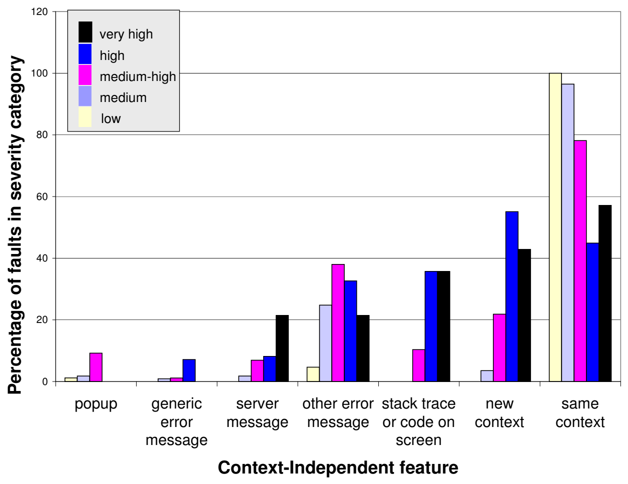

Figure 6. Severity distribution as a function of context- and cause-*independent* fault characteristics.

"404 not found" errors), or displaying a stack trace.

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

A deeper analysis of the human study data [7] reveals that errors presented with a stack trace were viewed as most severe, followed by database errors with visible SQL code, authentication problems, and then error messages in general. Cosmetic errors, such as a typographic mistake, that do not affect the usability of the website were perceived as having the lowest severity, followed by form errors such as extra buttons. Although minor faults, such as typos, formatting issues, and form field problems are often easier to trace in the source code, in that there can be few to no business logic defects involved with these kinds of faults, finding such trivial errors can actually be potentially more challenging when using traditional testing approaches in a web application environment. Consider a situation where a test suite with oracle output exists for an old version of a web application. When the application undergoes development, the HTML output will most likely change, and it becomes difficult to distinguish between faulty output and harmless program evolutions, especially with automated tools [6]. By contrast, severe errors that present with a stack trace or errors messages on the screen are more easily identified, due to the relative larger difference between expected and actual output. Our results highlight the benefits of using even simple, automated testing approaches that search for error keywords [1] in that they are likely to quickly and reliably identify high-severity faults. We return to the issue of fault causes in Section 5.

Modifying the context-independent features of consumer-visible faults is an orthogonal approach to testing strategies in the arena of consumer retention. Although the visual presentation is not the only deciding factor with respect to consumer-perceived fault severity, various fault presentations are associated with different severity levels. When a fault occurs, developers often have the option of managing the way the fault is presented to the user. Consider the case of the inability to upload an image; the set of real-world faults used in the human study includes thirteen such instances. In every case where the fault was judged as severe (5 out of 13), the faulty page was displayed with either a stack trace or a server generated error message, in a new context (i.e., a webpage with a different header, footer, sidebar menus, and layout than the original website template). When the same fault was judged as medium-high severity (4 out of 13 times) a stack trace was present, but in half of those pages the context did not change. When the fault was judged with only medium severity (the remaining 4 out of 13 times), the context remained the same, and a stack trace was usually absent. This example demonstrates how developers can reduce the perceived severity of such faults by preventing the context of the page from changing and wrapping the fault in an error message displayed on the same page.

Figure 6 presents the visual, context-independent characteristics of the real world faults in our study, broken into the four fault severity groups. In general, faults that occur in the *same context*, as opposed to loading a new and visually discordant page, are much more likely to be judged as lower severity. The converse is also true: faults with new contexts are associated with increasing severity ratings. Our analysis of the study revealed that the worst way to present faults to consumers is in the form of a stack trace; wrapping the fault in an error message was associated with a lower severity. Server-generated error messages were also poorly received. Error messages that were displayed as popup windows were regarded as the least upsetting to consumers. Developers should be conscious of the impact on consumer-perceived severity of various fault presentations, and choose options such as maintaining the same context and relying on popups that are associated with

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

Figure 7. Common defect causes from the four hundred real-world faults in the human study. The SRCC column gives the Spearman correlation between faults having that cause and high severity (≥ 2.5). An SRCC above 0.5 is considered moderate to strong correlation [10].

high consumer satisfaction.

5 Fault Causes Related to Severities

In this section, we analyze the causes of defects, independent of their visual or use-case context, to associate faults of varying severities with different components in the code or environment. Our goal is provide developers with ways to target web application design and testing to reduce the frequency of high severity faults by focusing on the potential causes of defects. In our analysis of the human study data we identified nine recurring causes of defects, summarized in Figure 7. Recall that all defects in the study were taken from real-world bug reports filed in bug databases [7].

Figure 8 shows the fault severity distributions associated with the causes from Figure 7. Severe faults are frequently associated with unhandled NULL objects, missing files or incorrect upgrades, database issues, and incorrect configurations. The lower the severity of a fault, the more likely it was due to erroneous logic in the application source code not associated with the database or NULL objects. These results are consistent with those of the previous section, in that exceptions that frequently manifest as stack traces are due to unhandled exceptions, and database access problems are associated with displaying SQL code on the screen. As explained in Section 4, severe faults that are due to exceptions or configuration issues are often easier to detect with even a coarse-grained, automated test suite, because the difference between the expected output and actual application output is more dramatic.

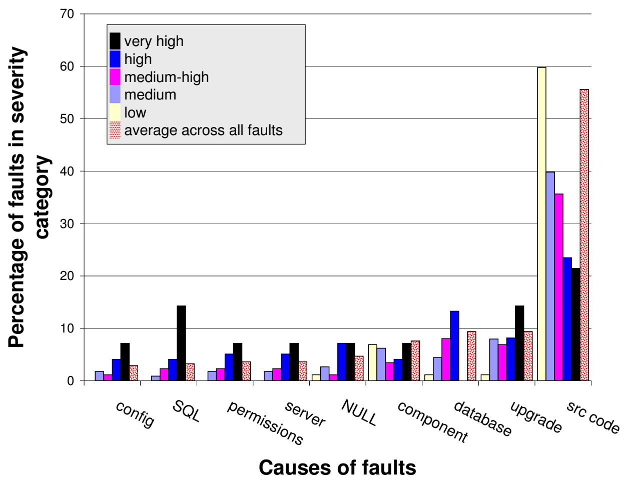

Figure 8. Fault severity as a function of underlying defect causes.

Finding logic errors in program source code poses a greater challenge, as it is less likely that any individual test case will exercise any single line of code, and the difference between the oracle and actual test output may be difficult to recognize as a fault instead of a harmless program evolution [6]. Once again, even a simple, naïve test suite that uses keywords to detect faulty output can be used with a high return on investment when seeking high severity faults. Approaches that orthogonally address the prevention of unexpected NULL objects [5], and database testing rigs [3], and can be used in conjunction with such modest testing techniques to successfully prevent or flag proportionally more severe faults.

6 Test Case Reduction And Severity

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

We consider a scenario in which *user-session-based testing*, a kind of capture-replay technique that is largely automatic, is currently implemented as a company's testing methodology. This approach works by recording user accesses through the server and replaying them during testing [8, 30]. A *user session* is an ordered list of user `URL` requests made to the server. Although a server needs only small modifications to log such sessions, this type of testing has the drawback that large volumes of session data are captured. Replaying all recorded user sessions is often infeasible, and *test suite reduction* with user-session-based testing has been studied extensively [16, 18, 19, 29, 31].

The goal of test suite reduction is to select a subset of all user sessions to replay that will reveal the largest number of faults during testing. While the trade offs between fault detection and test case size have been explored for web applications, the severity of the faults uncovered by various reduction techniques has yet to be addressed.

6.1 Reduction — Experimental Setup

To measure the severities uncovered by various test suite reduction techniques, we manually seeded 90 faults in three PHP applications in Figure 9 denoted by an asterisk (an online store, a forum, and a real estate website). The faults were equally distributed among the applications and were seeded according to the methodology of Sprenkle *et al.* [31]. One hundred and fifty user sessions were then collected from volunteers asked to interact with each application in a typical manner. We here define a test case to be all the URLs in one user session, and each test case was run on a faulty version of a benchmark with one fault injected at a time. All three web applications had database components, the states of which were saved at the beginning of each user session so that when a test case was replayed, it operated on the same relative state.

We chose to implement various user-session test suite reduction strategies explored by Sprenkle *et al.* [31]: *retest-all*, *Harrold-Gupta-Soffa*, and *Concept*. The *retest-all* strategy does not reduce the size of the test suite and serves as a baseline for fault detection. *Harrold-Gupta-Soffa* is a general technique [16] that uses a heuristic which selects a subset of the original test suite by approximating the optimized reduced set (an NP-complete problem). The algorithm chooses test cases from the original test suite one at a time, always choosing the test case that will cover the most untouched URLs next. *Concept* is Sprenkle *et al.*'s orthogonal approach that builds a concept lattice where each node is a test case that inherits the attributes from all previous nodes. The lattice therefore constructs a partial ordering between all test cases, based on the URLs each test case covers. The parent nodes of the bottom of the lattice represent the minimal set of test cases that will cover all URLs using this approach. We adopted two readily-available tools, `concepts` [11] and `RAISE` [12] to implement the *Concept* and *Harrold-Gupta-Soffa* approaches.

Both methodologies need to associate requirements with each test case. We followed the experimental setup in [31], taking a test case to be user session composed of URLs in a specific order, and its requirements to be the respective URLs each user session exercises. Because URLs frequently contain form data as name-value pairs, we follow Sprenkle *et al.* [28, 31] by examining the URLs independent of these values. For example, `http://example.com/order.php?sku=12&id=11` and `http://example.com/order.php?id=15` are considered the same URL and therefore the same requirement, because we ignore all name-value pairs after the `?` in the URL. Conversely, the Openrealty benchmark used the same PHP page for almost all requests, and specified actions via arguments such as `do=Preview`. For this benchmark we also considered the value of this action name-value pair only in mapping the URL to a requirement.

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

Figure 9. Fault severity uncovered via reduced test suites. The "Method" is either *retest-all* (the baseline), *HGS* [16] or *Concept* [31]. The "Test Cases" column counts the number of test cases in the reduced suited produced by that methodology (out of 50). The "Low" through "High" columns count the number of faults exposed in each such severity level. The "Total" column gives the total number of faults across all severities exposed by each technique on each benchmark application.

We tested the fault detection ability of each test suite by cloning each benchmark into 30 versions with one seeded fault each, and running each test suite on each cloned version. We collected the faulty output and expected output for each faulty version of source code by customizing a `diff`-like tool to ignore data such as timestamps and session tokens that would be different between even correct versions of output. Comparing the HTML output of web applications in regression testing is a known problem, due to the inability to distinguish between erroneous output and harmless program evolutions, and has been addressed through partially- and fully-automated tools [6]. We use a customized `diff`-like comparator to eliminate false positives and false negatives.

After collecting the faulty and expected versions of output, we next measure the consumer-perceived fault severity of each defect. We used the formal model derived from the user study [7] to accurately predict the severity between a faulty HTML output and an oracle output by approximating various surface features of the fault and using a manually-constructed decision tree to arrive at a severity.

6.2 Reduction — Experimental Results

Figure 9 shows the number of test cases (user sessions) for each test reduction methodology in our experiment, and the number of uncovered faults of varying severities. Because the severity of a fault depends on the concrete manifestation of the fault in HTML, rather than source code, the same fault may materialize with different severities on different `URL`s. Therefore, Figure 9 shows the number faults in each severity category when considering the average severity rating for any HTML manifestation of a particular fault.

Previous work has shown both the *Concept* [31] and *Harrold-Gupta-Soffa* [16] test suite reduction methodologies to be highly effective at maintaining the fault exposure proprties of the original *retest-all* baseline — when all faults are treated equally. This experiments shows that they are also effective when fault severity is taken into account. Reduced test suites were able to match the same number of exposed faults in cases where the fault were generally of medium-high (Prestahsop) and medium (Vanilla) severity. For our benchmark that happened to be seeded with mostly severe faults (Openrealty), both test suite production approaches had comparable results to the baseline. Although this property of effective fault exposure may not generalize beyond our benchmarks and more experimental work is required, we believe that test suite reduction of user-session based test suites is an effective way to maintain high levels of severe fault exposure while reducing costs.

7 Related Work

The prevalence of failures in web application has been widely studied. A 2005 survey identified 89% of online customers encounter problems when completing online transactions [15, 25]. User-visible failures are common in top-performing web applications: about 70% of top-performing sites are subject to user-visible failures within the first 15 minutes of testing [32].

A number of preliminary web fault taxonomies have been proposed. Guo and Sampath identify seven types of faults as an initial step towards web fault classification [14]. Marchetto *et al.* validate a web fault taxonomy to be used towards fault seeding [21], using fault characteristics such as level in the three-tiered architecture the fault occurred on or some of the underlying, specific web-based technologies (such as sessions). In these fault classifications [14, 21] there is no formal concept or analysis of severity; some categories of faults may produce more errors that would turn customers away, but this consideration is not explored. Our work also divides up faults according to fault localization, but is able to associate consumer-perceived severities with different sources of faults.

Fault severity as a metric is considered by Elbaum *et al.* [9] in test suite construction, but they provide no guidelines for measuring fault severity beyond the time required to locate or correct a fault, damage to persons or property, or lost business, which are difficult to calculate. Ma and Tian present a defect classification framework to analyze web errors with respect to reliability improvement [20]. Unlike our approach, they extract information from web server logs rather than examining browser output. Although they consider defect severity as a classification attribute, like Elbaum *et al.*, no guidelines for how to measure this feature are outlined.

Di Lucca *et al.* have also explored reduction of user-session test suites in [19], where they measure coverage of `URL`s as well as Built Client Pages, which are dynamically built pages generated from user input and application state data. Rather than only considering `URL`s as coverage criteria, they rely on static and dynamic analysis of the web application to generate a reduced test suite smaller than those of *Concept* [31] for their benchmarks. We chose to focus on *Concept*'s and *Harrold-Gupta-Soffa*'s [16] reduction techniques because they rely on simple analysis of `URL`s in collected use cases; in future work we would like to extend our study to reduction techniques such as Di Lucca *et al.* have explored.

8 Conclusion

Formal testing of web applications is frequently a casualty of the extreme resource constraints of their development environments [27]. At the same time, web applications are highly human-centric, and consumer retention is known to be low [22]. In this paper we analyzed the results of a study of the consumer-perceived severity of 400 real-world web application faults with the explicit goal of providing concrete guidelines to increase the perceived return-on-investment of even naive testing approaches. We have shown that the distribution of consumer-perceived fault severities is *independent* of the type or language of the application, and it is therefore possible to deliver high quality software for any use and without constraints on the underlying technologies used. We have also demonstrated that, given the same defect in the source code, different visual presentations of the fault to consumers have different impacts on their perceived severity. Controlling fault presentation by opting for pop-ups and error messages over stack traces and changing page layout is a simple and effective way to reduce the consumer-perceived severity of faults. In studying the causes of various types of faults, we also established the utility of coarse-grained test suites that only detect relatively large differences in HTML output or rely on error keywords as effective ways of capturing many high severity faults, encouraging all developers to invest in at least some minimal testing infrastructure. Finally, we examined the trade offs between the costs of various user-session-based test suites for web applications, and found that reduced test suites were effective at maintaining fault exposure properties across all fault severity levels. Although web application testing has received a relatively lukewarm welcome in industry due to the perceived lack of return on investment within the demanding development environment, we have shown that by focusing on retaining consumers through the prevention of severe faults, even simple testing approaches are able to achieve significant gains in consumer satisfaction.

9 Acknowledgements

This research was supported in part by, but may not reflect the positions of, National Science Foundation Grants CNS 0716478 and CNS 0905373, Air Force Office of Scientific Research grant FA9550-07-1-0532, and NASA grant NAS1-02117, and Microsoft Research gifts.

References

[1] M. Benedikt, J. Freire, and P. Godefroid. Veriweb: Automatically testing dynamic web sites. In *World Wide Web Conference*, May 2002.

[2] D. Boeth. An analysis of the future of B2B e-commerce. In `http://www.ftc.gov/bc/b2b/comments/PPRo%20Statement.pdf`, September 2009.

[3] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[4] ClickZ. B2B e-commerce headed for trillions. In `http://www.clickz.com/986661`, 2002.

[5] K. Dobolyi and W. Weimer. Changing java's semantics for handling null pointer exceptions. In *International Symposium on Software Reliability Engineering*, pages 47–56, Nov. 2008.

[6] K. Dobolyi and W. Weimer. Harnessing web-based application similarities to aid in regression testing. *ISSRE '09: Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE'09)*, November 2009.

[7] K. Dobolyi and W. Weimer. Modeling consumer-perceived web application fault severities for testing. Technical report, University of Virginia, 2009.

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

[9] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *International Conference on Software Engineering*, pages 329–338, 2001.

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

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

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

[13] K. Grannis, E. Davis, and T. Sullivan. Online sales to climb despite struggling economy according to Shop.org/Forrester research study. In `http://www.shop.org/c/journal_articles/view_article_content?groupId=1&articleId=702&version=1.0`, September 2009.

[14] Y. Guo and S. Sampath. Web application fault classification - an exploratory study. In *Intern. Symp. on Empirical Softw. Engin. and Measurement*, pages 303–305, 2008.

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

[16] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[17] Internet World Stats. World internet usage statistics news and world population stats. In `http://www.internetworldstats.com/stats.htm`, Sep 2009.

[18] S. Karre. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, 2005.

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

[20] L. Ma and J. Tian. Web error classification and analysis for reliability improvement. *J. Syst. Softw.*, 80(6):795–804, 2007.

[21] A. Marchetto, F. Ricca, and P. Tonella. Empirical validation of a web fault taxonomy and its usage for fault seeding. In *IEEE International Workshop on Web Site Evolution*, pages 31–38, Oct. 2007.

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

[23] T. J. Ostrand and E. J. Weyuker. The distribution of faults in a large industrial software system. In *International symposium on software testing and analysis*, pages 55–64, 2002.

[24] T. J. Ostrand, E. J. Weyuker, and R. 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.

[25] S. Pertet and P. Narasimhan. Causes of failure in web applications. Technical report, Carnegie Mellon University Parallel Data Lab, December 2005.

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

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

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

[29] S. Sprenkle and E. Gibson. Applying concept analysis to user-session-based testing of web applications. *IEEE Trans. Softw. Eng.*, 33(10):643–658, 2007.

[30] S. Sprenkle, E. Gibson, S. Sampath, and L. Pollock. Automated replay and failure detection for web applications. In *Automated Software Engineering*, pages 253–262, 2005.

[31] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter. An empirical comparison of test suite reduction techniques for user-session-based testing of web applications. In *International Conference on Software Maintenance*, pages 587–596, 2005.

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