Empirical Validation of a Web Fault Taxonomy and its usage for Fault Seeding

Alessandro Marchetto¹, Filippo Ricca², Paolo Tonella¹
¹Fondazione Bruno Kessler—IRST, Trento, Italy
²Unità CINI at DISI;
marchetto@itc.it, filippo.ricca@disi.unige.it, tonella@itc.it

Abstract

The increasing demand for reliable Web applications gives a central role to Web testing. Most of the existing works are focused on the definition of novel testing techniques, specifically tailored to the Web. However, no attempt was carried out so far to understand the specific nature of Web faults. This is of fundamental importance to assess the effectiveness of the proposed Web testing techniques.

In this paper, we describe the process followed in the construction of a Web fault taxonomy. After the initial, top-down construction, the taxonomy was subjected to four iterations of empirical validation aimed at refining it and at understanding its effectiveness in bug classification. The final taxonomy is publicly available for consultation and editing on a Wiki page. Testers can use it in the definition of test cases that target specific classes of Web faults. Researchers can use it to build fault seeding tools that inject artificial faults which resemble the real ones.

Keywords: Web applications testing, Fault taxonomy, Fault seeding and Empirical Validation.

1 Introduction

Web applications mediate a huge number of operations which are critical for our life and involve virtually every domain, from banking to health care, public administration, commerce and amusement. Such a widespread usage triggered the investigation of methodologies and techniques to improve the quality of Web applications. Notably, several Web testing techniques have been proposed in the literature [2, 9, 10]. On the other hand, the specific nature of Web application faults, compared to the faults occurring in more traditional software systems, has not been studied in detail so far. As a consequence, the fault revealing capabilities of Web-specific testing techniques can be hardly determined and it is difficult to assess the additional contribution they give, compared to standard and consolidated software testing methods.

In this paper, we describe the initial results of a research activity which aims at analyzing the testing issues that affect Web applications in a specific way. Web applications are built on top of a set of specific technologies, which give rise to specific faults, which in turn require ad-hoc testing methods to be revealed. Hence, the first step to understand the testing issues for Web applications consists of the analysis of the Web faults, compared to the generic software faults occurring whenever a program is written. For example, a malformed URL in a dynamically constructed Web page is a Web-specific fault, while an index outside the array bounds is a generic fault, which can occur in any program, regardless of any Web technology involved. Web-specific faults may require ad-hoc testing methods.

We are investigating the nature of Web-specific faults by defining a taxonomy for them. After constructing an initial taxonomy top-down (Section 3), we refined it through four iterations of empirical validation (Section 4). At each iteration, a set of real bugs retrieved from bug tracking systems was manually classified according to the taxonomy. The presence of unclassified bugs or the mapping of a large number of bugs onto the same fault class have been used to identify candidate refinement actions on the taxonomy, such as class addition, removal or split. The final taxonomy was assessed through metrics, showing that it meets properties such as: completeness and exhaustiveness, mutual exclusiveness, adequate class size (not too-big, not too-small), and non-ambiguity.

The Web fault taxonomy that we constructed is publicly available on a Wiki page. It can be used by Web testers to define a set of test cases specifically designed to reveal Web faults (as compared to test suites designed for generic faults). Testers can produce one or more test cases for selected classes in the taxonomy (e.g., the most frequent fault classes). Another usage of the taxonomy, in line with the final goal of our research activity, is fault seeding. Fault
seeding [6] can be used to artificially produce faulty versions of an application that are used to evaluate comparatively alternative testing techniques. Higher confidence in the results of the comparison can be achieved if the faults generated artificially resemble the real faults. By basing the fault seeding on an empirically validated fault taxonomy, we can generate better artificial faults to inject. Then, there are at least two possibilities: the fault seeder can be a human, who is guided by the taxonomy in the selection of the faults to insert, or it can be a tool, which turns each fault category into a set of defects that can be inserted through program transformation. The final part of the paper is devoted to this latter possibility (Section 5).

2 Related works

Several related works, such as Harrold et al. [6], Artho et. al [3] and Voas et. al [20], are focused on fault injection and present defect types in the form of changes to be made into non-Web software code (e.g., object oriented), with the purpose of simulating the occurrence of real software faults. Eventually, fault injection aims at increasing the effectiveness of testing, by evaluating its effects on artificial faults [4, 8, 19]. Bieman et al. [4] use a fault injection mechanism, based on assertion violation, during software testing to increase the testing coverage. Heimdahl et al. [8] apply fault injection to reduce the suite of test cases generated using a model-based approach. Vijayaraghavan et al. [19] proposes a survey of several bug taxonomies used to improve software test. Unfortunately, only a few fault taxonomies and models are available to classify programming errors for specific kinds of software systems, such as Web services [11], component-based applications [13], and object-oriented software systems [1]. Hayes et. al [7] study the relationships that exist between faults and systems modules. They build both fault and module taxonomy and then study the relationships between them. The idea is that the type of fault introduced by programmers depends on the type of module that is being developed. Thus, this “link” can be used to guide code reviews, walkthroughs, and verification.

Fault taxonomies for Web applications are still to be investigated in depth, in that only a few very preliminary works considered them. For instance, Ricca et al. [14] present a very preliminary Web fault taxonomy, consisting of a single level of quite general fault categories. Other works, such as Sprengle et al. [15] and Karre et al. [9], use ad-hoc defect injection systems to evaluate the effectiveness of testing approaches, which are not based on any explicit Web fault taxonomy. With the present work, we intend to fill this gap by providing a preliminary, yet complete, Web fault taxonomy, to be used to help practitioners define the test cases and researchers compare alternative Web testing approaches.

3 Initial taxonomy

We constructed the Web fault taxonomy by following a mixed top-down and bottom-up approach. An initial taxonomy was defined by analyzing the high level characteristics of Web applications. Then, this taxonomy was refined by analyzing real faults, taken by the Sourceforge¹ bug reports, and trying to map them to taxonomy categories. The focus of this section is the top-down definition of the initial taxonomy, while its refinement through bottom-up empirical validation is described in the next section. The whole Web fault taxonomy obtained after refinement is publicly available as a Wiki² page. Being a Wiki, it is open to collaborative enhancements and evolution by the scientific community.

We have defined the taxonomy by identifying the set of main high-level structural characteristics of Web applications. Then, for each one, we have listed the set of sub-characteristics that detail them. For each sub-characteristic, we have identified the set of typical faults that are related to the specific sub-characteristic of a Web application and can potentially lead to its failure. We recall that this taxonomy is intended to be a Web fault taxonomy, i.e., it is specifically tailored to the features that characterize Web applications, as compared to other software applications. Thus, even if general faults (e.g., “index out of array bounds”) may occur also in a Web application, they are not represented by a specific category in the taxonomy since they are not associated with Web-specific features.

Table 1 shows a fragment of the taxonomy. The main high-level characteristics of Web applications identified in the top-down analysis are: (A) Web applications are based on a multi-tier architecture (e.g., client, server, database); (B) they use a specific kind of user interface, based on Web languages and technologies (e.g., HTML, Flash, Javascript); (C) they support session management; (D) they support hyperlink based-navigation and resources are accessed through URL; (E) they use Web protocols (e.g., HTTP, HTTPS); (F) they are executed concurrently by several users and thus user roles and authentication need to be managed. For these characteristics, the taxonomy provides a set of sub-characteristics that better detail them. For instance, a typical Web architecture (A) is composed of: client pages; server side pages; server side components; database; and HTML forms, used to exchange data between client and server. Table 1 gives also some examples of typical types of faults, identified for each sub-characteristic. For example, (f1) browser incompatibility and (f2) the back button problems are known as typical faults related to the HTML pages interpreted by Web browsers (sub-characteristic A.1).

¹http://sourceforge.net
²http://se.itec.it/wiki/sewiki/pmwiki/pmwiki.php?n=Taxonomy
### Characteristics

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

Table 1. Fragment of the initial taxonomy. Only selected (sub-)characteristics and classes of faults are shown.

### 4 Validation and refinement of the taxonomy

This section describes the bottom-up validation and refinement of the initial taxonomy, based on real bugs that have been classified according to the taxonomy categories. A “good” taxonomy [19] should be: (1) general, (2) complete and exhaustive, (3) mutually exclusive (i.e., with no overlap among classes), (4) not-too-large and not-too-small, and (5) not ambiguous. In order to evolve the initial taxonomy so as to meet these properties, we considered the bug reports that cannot be accurately classified in the taxonomy. These trigger refinement actions such as: adding or removing classes, splitting or merging existing classes, detailing the description of a class.

#### 4.1 Research questions

In addition to producing the refined taxonomy, the experiment was also aimed at addressing the following research questions:

**RQ1**: Are bugs evenly distributed among the fault classes of the taxonomy? This requires to study frequency and distribution of the bugs across the classes of the taxonomy.

**RQ2**: How many non-classifiable bugs are there? This requires to analyze the set of bugs that cannot be classified and to understand why this occurs. It pertains to the completeness of the taxonomy.

**RQ3**: How mutually exclusive and ambiguous is the taxonomy? This requires to study ambiguity and exclusiveness of the fault assignment to the taxonomy classes, revealing, for instance, if the taxonomy contains overlapping classes or unclear fault descriptions.

**RQ4**: How Web-specific are the faults associated with bugs reported for real Web applications? This requires to analyze the type of faults to understand if they can be considered “Web-specific” or “generic”.


The answers to research questions RQ1, RQ2 and RQ3 were the key to guide the refinement of the taxonomy, in an iterative procedure aiming at the improvement of the taxonomy accuracy in classifying real bugs.

4.2 Metrics

The following metrics are calculated to answer the research questions:

- **Bugs per class**: number of bugs associated with each fault class of the taxonomy (used to answer RQ1).

- **Unclassified bugs**: number of bugs that cannot be classified using the taxonomy (used to answer RQ2).

- **Confidence of classification**: average confidence declared by the classifier (i.e., expert tester) over all performed classifications (used to answer RQ3). This index is in the range 0-100, with 100 meaning that the classifier is absolutely sure of the classification.

- **Agreement between classifiers**: proportion of bugs classified in the same fault class by different classifiers (used to answer RQ3). This metric should be taken with care. A bug classified in the same class by two classifiers indicates that the bug and the taxonomy class are respectively accurately described and non ambiguous/overlapping. Instead, a bug classified differently by two classifiers indicates a poor bug report or an ambiguous/overlapping fault class.

- **Rate of generic faults**: number of bugs associated with generic faults (used to answer RQ4).

These metrics have been computed on the entire set of real bugs considered in all iterations of the validation process. Bugs per class and Unclassified bugs have been evaluated also after each validation iteration, to assess the effects of taxonomy refinement over the successive iterations.

4.3 Objects

We extracted 376 real bugs from 32 software projects (such as PHPAdmin, AjaxAnywhere, ZK - Simply Ajax, Sarissa, etc.) available under Sourceforge. Bugs can be accessed in Sourceforge thanks to the bug tracking system in use. The collected bugs are distributed among three main Web technologies: (1) ASP/ASP.NET; (2) PHP 4/5; and, (3) JSP and Java. Javascript, HTML and other common technologies are shared among most selected applications.

4.4 Experimental Procedure

The experiment has been performed in four iterations, repeated for each of the two (humans) classifiers participating in the experiment. Every iteration performed by each classifier involves the following steps:

1. Each bug is analyzed by the classifier to understand the fault that leads to the described failure or deviation from the expected behavior. The fault is classified as generic or Web-specific. If it is Web-specific, it is assigned to a fault class in the taxonomy or it is marked as Unclassified.

2. The classifier declares the Confidence of classification for the given bug.

3. If necessary, the classifier improves the fault class description to make it more precise.

4. If appropriate, according to the average of the bugs per class, the classifier marks the fault class as candidate to be split.

5. If the bug is unclassified, the classifier may record it and define a new fault class as a candidate for addition to the taxonomy in the next iteration.

After each iteration, the following operations are executed:

1. Metrics Bugs per class and Unclassified bugs are computed.

2. Based on Bugs per class and on the list of classes marked as candidates for split, the taxonomy is refined by splitting selected classes. This increases the balance and produces more meaningful groups of bugs into fault classes.

3. Based on Unclassified bugs and on the candidates for addition created by the classifier new classes are added to the taxonomy. When possible, unclassified bugs are assigned to newly created classes.

After all iterations, a final removal step is conducted. Based on the final taxonomy balance (i.e., metrics Bugs per class), empty or too small fault classes are taken into account and examined for possible removal from the taxonomy. The final taxonomy is obtained after this removal step.

All metrics are then recomputed while, the agreement between classifiers is computed for the first time. Notice that final Unclassified bugs is not just the sum of the unclassified bugs at every iteration, since the creation of new classes may have reduced their number.
4.5 Taxonomy refinement process

During the refinement process we collected data to quantify the evolution of the taxonomy and the involved effort. The initial taxonomy contained 69 classes of Web-specific faults. During the refinement process, 4 classes have been added to the taxonomy and 23 have not been mapped to any bug, being thus candidates for elimination. However, following the opinion of the experts involved in the classification, only 9 of these classes have been actually removed from the taxonomy. Instead, 50 classes of faults contain, at least, one bug while, the 70% of bugs have been mapped onto 16% of fault classes of the taxonomy. Hence, the taxonomy size has been reduced to 64. Table 2 shows detailed data collected after each iteration. None of the classes marked as candidates for split has been eventually subjected to this operation. The reason is that these classes are associated with typical and frequent faults and a high number of bugs can be reasonably expected for them.

The effort spent in this experiment is mainly due to two tasks: (1) selection of the bug dataset; and, (2) classification of the bugs according to the taxonomy. In detail, 54 hours were spent to select the 376 bugs used in the experiment and 15.5 hours to classify them (average time computed over all classifiers). Surprisingly, a larger proportion of the total effort is devoted to bug selection instead of classification. The reason is twofold: several bug descriptions are unusable, since it is impossible to guess the fault behind the signalled misbehavior. Other descriptions are not really bug reports, although they were inserted into the bug tracking system. For example, they contain just comments, questions and suggestions for improvements. Recognizing these kinds of reports and excluding them a-priori involved substantial effort.

4.6 Experimental results

In this section, we analyze the results and try to answer the research questions described above.

Research Question RQ1 (balance)

The bug distribution was inspected after every iteration, to identify opportunities for class split or removal. Figure 1 shows the final bug distribution after all 4 iterations. It still contains peaks and flat zones, but the classifiers considered the related fault classes already accurate and meaningful. A1f1 (browser incompatibility fault) is the largest class, with 30 bug reports mapped to it. Most of the other classes contain between 0 and 10 bugs, with a few exceptions (above 10). Overall, the taxonomy looks well-balanced and classes were judged accurate enough. The most frequent classes of faults in the taxonomy have been divided by programming language (see Table 3). It is interesting that there is indeed a dependency on the programming language, in that the most frequent fault classes are not the same across languages. For example, the most frequent fault A1f1 (browser incompatibility) is not so much of a problem when ASP is used. Since this language is naturally tied to the Windows platform and the IE browser, this datum may indicate that A1f1 bug reports refer mainly to portability to IE, which is ensured to hold for ASP. A6f4 (database extraction fault) is more or less evenly distributed across languages. A4f1 (form construction fault) is also less frequent in PHP than in ASP and JSP, possibly indicating a more robust and easier form management in PHP.

Table 4 shows that 50 classes (out of 64) contain at least one bug. This means that the 74% of the taxonomy was validated empirically by at least one real bug mapped to
of the major fault classes. The remaining 36% is still to be validated, notwithstanding the large number of bug reports analyzed in the four iterations conducted in this study. This indicated that the validation of a Web fault taxonomy is a major, challenging task, which probably should be carried out by the research community as a whole, not by one team in isolation.

Research Question RQ2 (unclassified)

The number of Unclassified bugs was 13, 3, 1, 0 respectively in the 4 iterations. Some of them have been used to add new classes (e.g., fault in interface construction by means of XSLT; fault in management of different user roles and privileges) to the initial fault taxonomy, while 5 (or 1.3%) remained unclassified. These have been judged too high level (e.g., generically addressing the quality of the application) or too specific to deserve a dedicated class in the taxonomy. Classifiers decided that the classification performance of the taxonomy was already very good, with only 5 unclassified bugs, and that addition of new classes for such unclassified bugs would be detrimental for the taxonomy.

Research Question RQ3 (ambiguity)

The average Confidence of classification declared by the two classifiers involved in the experiment was 96.2%. This value indicates high confidence in the classification performed. The classification Agreement between the two classifiers was also very high: 85%. Practical experiences with taxonomy construction and validation [19] shows that defining a clear, accurate and non-ambiguous taxonomy is a challenging task and several reasons may cause overlapping and multiple possible classifications. Hence, the values that we measured for Confidence of classification and Agreement are encouraging, indicating a sharp and clear categorization of the possible Web faults.

Research Question RQ4 (Web-specific faults)

The Rate of generic faults for this experiment was 32.6% (or 121 bugs). Often, these faults are associated with server-side components, such as JavaBeans. For them, it is reasonable to assume that testers would use traditional, non Web-specific, testing approaches. However, the interesting datum is that a large proportion of the bug reported for Web applications are Web-specific. This indicates that Web-specific testing approaches are highly needed and extremely important to ensure the quality and reliability of Web applications.

4.7 Threats to validity

As common with empirical studies, we analyze the main threats to the validity of this experiment.

- External validity concerns the generalization of the findings. The representativeness of the considered bug dataset can be limited by: (i) number of bugs considered in the experiment (376 in total) and (ii) the bug descriptions in the bug reports, coming from the user community of open source software, not from customers paying for a product.

- Construct validity concerns the metrics or observations used to address the research questions. We selected a set of metrics (Bugs per class, Unclassified bugs, etc.) with a straightforward, clear meaning. Some degree of subjectivity affects the computation of Confidence of classification and, of course, the classification of the bugs.

- Internal validity concerns the existence of factors which may have affected the results and have not been properly taken into account. The main source of internal validity threats is the use of only two classifiers.
Conclusion concerns the possibility to derive legitimate conclusions from the observations. Since we could obtain no statistical measure of confidence (such as a p-value), but only subjective confidence estimates, our conclusions rely on the interpretation of the metrics and the Confidence of classification declared by the classifiers. We complement this metric with the measurement of Agreement, which gives us a more objective way to estimate the effects of the necessarily involved subjectivity.

Clearly, not all threats to validity are properly addressed, but the nature of a Web fault taxonomy is such that it is only thorough replication of the experimental validation of the taxonomy that we can gain more confidence in its capability to classify real bugs.

5 Fault seeder

The described Web fault taxonomy can be used to associate its fault classes to concrete defects that can appear in the source code and implementing a fault seeder which actually injects them into the code. Availability of source code with known faults is fundamental when conducting studies on the effectiveness of alternative testing techniques. However, often real faults are not available or known for a given Web application. Hence, usage of a fault seeder [6, 9] represents a viable option. The problem is that the faults generated artificially by the seeder may be substantially different from the real ones. If the faults injected by the seeder have been produced in accordance with an empirically validated Web fault taxonomy, they are more likely to resemble real faults. Hence, the results obtained in the evaluation of the testing techniques are more meaningful and more likely to generalize to real faults if the involved fault seeder was constructed based in a validated fault taxonomy.

A taxonomy-based fault seeder gets in input the code of an application and a class of fault selected in the taxonomy. The seeder chooses one or more defects that instantiate the selected fault class and injects such defects into the application code, by means of automated program transformation. This means that in order to let the fault seeder use the fault taxonomy, we have to expand it so that every fault class be associated with a set of defects which instantiate it.

In the extended taxonomy, we describe defects in terms of changes which, once applied to the code, are highly likely to introduce the selected fault. In particular, each defect instance is documented as: (1) original code fragment; and, (2) code after fault injection. Let us consider some examples of defects associated with classes of faults described in Table 1.

A Web application can contain fragments of code devoted to recognizing the browser used by the user. To inject a browser incompatibility fault (A1f1 in Table 1), the fault seeder corrupts such code fragment. Figure 2 shows the injection of this fault into a JSP page. In this case, the block of statements used to recognize the “MSIE” browser is deleted, so that this kind of browser cannot be recognized by the faulty code. Often, to extract information from a database the SQL language is used in server-side components. Thus, to inject a fault related to the information extraction from a database (A6f2) we can corrupt the SQL queries performed by server components. Figure 3 shows an instance of the defect used to inject this fault into a Java statement containing an SQL query. In this case, we remove the “WHERE” clause of the query, so that the resulting set of data extracted from the database is wrong. Such a defect is typical of an SQL query which is constructed without all necessary constraints. Web applications use HTML forms...
to exchange data between components. Thus, to inject a fault related to forms (A4f1), we can corrupt the fragment of code devoted to building a form. In Figure 4, the form contains three input fields, identified by their name. The name of one of these inputs is changed with the name of another one, as may actually happen if this HTML line of code is produced by copy-paste of the previous line.

6 Conclusions and future work

After defining an initial Web fault taxonomy top-down, from high-level to specific features of Web applications, we validated it empirically by asking two classifiers (expert testers) to map a set of real bugs onto fault classes. Refinement of the taxonomy was conducted in four iterations, based on its balance and on the existence of unclassified bugs. In the final taxonomy, bugs are quite evenly distributed among fault classes; and, according to the average of bugs per class, fault classes are not-to-big and not-to-small; only few bugs cannot be mapped onto any fault class; fault classes are not ambiguous since the two classifiers agreed on most classifications; around 2/3 of the bugs were classified as Web-specific, thus indicating the relevance of this kind of faults.

Constructing a comprehensive and agreed Web fault taxonomy is a huge task that we cannot afford on our own. We intend to involve as many researchers as possible, in order to continue validating and enhancing the preliminary taxonomy produced in this work. For this reason, we make the taxonomy available as a Wiki page. Currently, we are working on a fault seeder based on the taxonomy. In turn, availability of the fault seeder will enable comparative studies of alternative Web testing techniques, as well as the development of novel techniques which cover fault classes not otherwise addressed by the existing testing methods.

References

