Journal of Web Engineering, Vol. 8, No. 4 (2009) 316–345
© Rinton Press

AN EMPIRICAL VALIDATION OF A WEB FAULT TAXONOMY AND ITS USAGE FOR WEB TESTING

ALESSANDRO MARCHETTO
Fondazione Bruno Kessler—IRST, Trento, Italy
marchetto@fbk.eu

FILIPPO RICCA
Unità CINI at DISI
Laboratorio Iniziativa Software FINMECCANICA/ELSAG spa - CINI, Università di Genova, Italy
filippo.ricca@disi.unige.it

PAOLO TONELLA
Fondazione Bruno Kessler—IRST, Trento, Italy
tonella@fbk.eu

Received February 15, 2008
Revised September 2, 2009

Web testing is assuming an increasingly important role in Web engineering, as a result of the quality demands put onto modern Web-based systems and of the complexity of the involved technologies. Most of the existing works in Web testing are focused on the definition of novel testing techniques, while only limited effort was devoted to understanding the specific nature of Web faults. However, the performance of a new Web testing technique is strictly dependent on the classes of Web faults it addresses.
In this paper, we describe the process followed in the construction of a Web fault taxonomy. We used an iterative, mixed top-down and bottom-up approach. An initial taxonomy was defined by analyzing the high level characteristics of Web applications. Then the taxonomy was subjected to several iterations of empirical validation. During each iteration the taxonomy was refined by analyzing real faults and mapping them onto the appropriate categories. Metrics collected during this process were used to ensure that in the final taxonomy bugs distribute quite evenly among fault categories; fault categories are not-too-big, not-too-small and not ambiguous.
Testers can use our taxonomy to define test cases that target specific classes of Web faults, while researchers can use it to build fault seeding tools, to inject artificial Web faults into benchmark applications. The final taxonomy is publicly available for consultation: since it is organized as a Wiki page, it is also open to external contributions.
We conducted a case study in which test cases have been derived from the taxonomy for a sample Web application. The case study indicates that the proposed taxonomy is very effective in directing the testing effort toward those test scenarios that have higher chances of revealing Web specific faults.

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

1 Introduction

The pervasive nature of Web applications, which have become an integral part of our daily activities in a wide range of domains, spanning from banking to health care, public administration, commerce and amusement, stimulated the investigation of techniques for Web testing [2, 9, 11, 21, 22, 31]. In fact, reliability and more generally quality of such applications is a key issue. While several approaches and techniques have been proposed and studied for Web testing, knowledge about the specific nature of Web application faults, compared to the faults occurring in more traditional software systems, remains quite limited. Without such a knowledge, it is hard to assess the real value of novel testing techniques, proposed to address Web-specific problems.

In this work, we investigate the implications that Web-specific technologies have on testing. Specific faults, requiring dedicated testing techniques, arise as a result of the involved platforms, architectures, frameworks and paradigms. We have analyzed and classified the faults occurring in Web applications, and we have compared them with generic software faults. 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. The specific nature of Web faults demand for ad-hoc testing methods.

Based upon our analysis of Web-specific faults, we defined a fault taxonomy. Existing works on fault taxonomies are focused on generic software faults [14] or present quite preliminary and limited, high-level classifications of Web faults [32]. Other related work deals with the injection of artificial faults into Web applications [21, 34]. However, these works do not resort to a complete, validated Web fault taxonomy for faults generation. In order to support fair and objective comparison among alternative Web testing techniques, an independently constructed Web fault taxonomy is demanded by the Web testing community. The present work aims at initiating the way toward an agreed and commonly accepted Web fault taxonomy.

We built the proposed Web fault taxonomy by:

- **Top down construction**: we organized the higher-level elements in the taxonomy based on the existing literature on Web engineering, looking for Web-specific concerns.

- **Bottom up construction**: we mapped a large set of real Web faults, obtained from bug reporting systems, onto the taxonomy and refined it correspondingly, following an iterative procedure.

- **Analysis**: collection of metrics related to the taxonomy and the process used to build it, so as to gather feedback useful for the next taxonomy construction iteration.

Our empirical validation is based on a set of real bug descriptions retrieved from (on-line) bug tracking systems. After attempting to classify them according to the taxonomy, we determined bugs that remain unclassified and fault classes that are used too often to classify diverse bugs. Refinement actions descend from such an analysis, and include: (1) class addition; (2) class removal; (3) class split. For the final evaluation of the taxonomy we used metrics that quantify important taxonomic properties including completeness and exhaustiveness, mutual exclusiveness, adequate class size (not too-big, not too-small), and non-ambiguity.

The proposed Web fault taxonomy is open for external contributions. Registered users are authorized to enlarge and refine the existing Web fault classes. Being publicly available, the taxonomy can be used by Web testers and by researchers in Web testing. Based on the taxonomy, testers can construct test suites that address Web specific faults. We give an example of this usage of the taxonomy at the end of this paper (Section 5). Researchers may use the taxonomy as the basis for the construction of a Web testing benchmark. This would support quantitative evaluation and comparison of alternative Web testing techniques. The taxonomy is also useful to researchers for a more qualitative assessment of novel Web testing methods. In fact, analysis of the Web faults specifically targeted by a new technique becomes possible thanks to the taxonomy. Finally, researchers can use the taxonomy for the construction of fault seeders [14] able to inject artificial faults that mimic the typical cases of Web specific problems. Fault seeding tools play a central role in the evaluation of the effectiveness of testing techniques [4, 12].

The paper is organized as follows: after a brief summary of notions and concepts used in this work (Section 2), we describe the initial taxonomy and how it was produced (Section 3). In the next section (Section 4) we describe how the built taxonomy was refined and assessed empirically. Then (Section 5) we investigate how to construct a test suite for testing a Web application by using our taxonomy. Related works (Section 6) and conclusions (Section 7) are at the end of the paper.

2 Concepts and notions

Although widely used in testing and software engineering, we prefer to explicitly give the definitions of the following terms, so that whenever they are used in the paper, their meaning is clear and can be referred back to a unique definition.

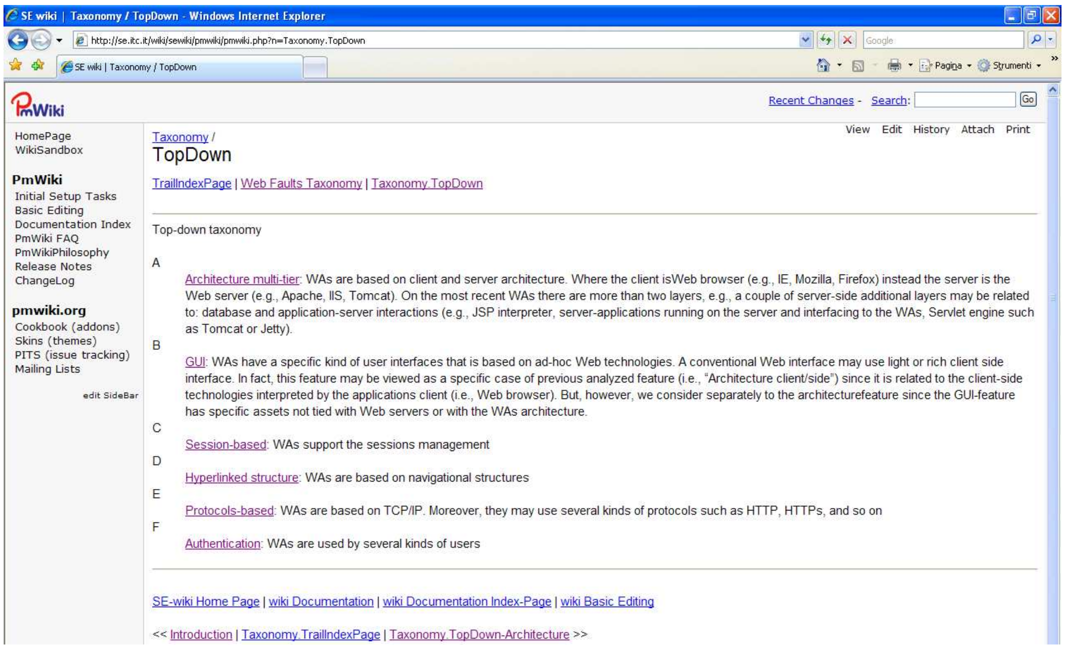
- *Failure*: an observable incorrect behavior or state of a given system, i.e., a behavior or state that violate the specifications/requirements for the system. Thus, occurrence of a failure requires system execution and manifests itself only at runtime (e.g., system crashes when run with certain input data).
- *Fault (also known as: Defect, Bug)*: incorrect implementation (e.g., incorrect instruction in the code) that, under certain execution scenarios, may give rise to a failure.
- *Error*: the developer's mistake (e.g., misunderstanding of an instruction's semantics) that caused the wrong implementation (i.e., the fault).
- *Fault class*: a class of similar faults (e.g., incorrect or missing variable initialization), corresponding to a node in the fault taxonomy.
- *Bug description*: textual description of the manifestation of a fault during execution, usually from the point of view of the user. It can be also regarded as the description of a failure.

Hence, a fault is a particular instance of a fault class and represents a specific way in which the fault class may occur in practice in the code. According to the literature and also in this paper, the terms *fault*, *bug* and *defect* are used as synonyms. Our taxonomy describes fault classes. At the leaf level, it also contains sample faults, which instantiate the fault classes and provide examples of actually occurring defects. The taxonomy was validated using sets of real bug descriptions obtained from the Internet (i.e., from bug tracking systems). The input was a set of bug descriptions, from which faults have been inferred manually, whenever possible.

3 Initial taxonomy

We constructed our 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, as collected from the following sources:

- existing literature about Web fault model such as Sprenkle et al. [34], Karre et al. [21], Ricca et al. [32] and, more generally, also existing literature regarding Web testing such as [2, 9, 11, 21, 22, 31];


SE wiki | Taxonomy / TopDown - Windows Internet Explorer

http://se.itc.it/wiki/sewiki/pmwiki/pmwiki.php?n=Taxonomy.TopDown

Recent Changes - Search:

HomePage
WikiSandbox

PmWiki
Initial Setup Tasks
Basic Editing
Documentation Index
PmWiki FAQ
PmWikiPhilosophy
Release Notes
ChangeLog

pmwiki.org
Cookbook (addons)
Skins (themes)
PITS (issue tracking)
Mailing Lists

edit SideBar

Taxonomy /
TopDown

TrailIndexPage | Web Faults Taxonomy | Taxonomy.TopDown

Top-down taxonomy

A
Architecture multi-tier: WAs are based on client and server architecture. Where the client isWeb browser (e.g., IE, Mozilla, Firefox) instead the server is the Web server (e.g., Apache, IIS, Tomcat). On the most recent WAs there are more than two layers, e.g., a couple of server-side additional layers may be related to: database and application-server interactions (e.g., JSP interpreter, server-applications running on the server and interfacing to the WAs, Servlet engine such as Tomcat or Jetty).

B
GUI: WAs have a specific kind of user interfaces that is based on ad-hoc Web technologies. A conventional Web interface may use light or rich client side interface. In fact, this feature may be viewed as a specific case of previous analyzed feature (i.e., "Architecture client/side") since it is related to the client-side technologies interpreted by the applications client (i.e., Web browser). But, however, we consider separately to the architecturefeature since the GUI-feature has specific assets not tied with Web servers or with the WAs architecture.

C
Session-based: WAs support the sessions management

D
Hyperlinked structure: WAs are based on navigational structures

E
Protocols-based: WAs are based on TCP/IP. Moreover, they may use several kinds of protocols such as HTTP, HTTPs, and so on

F
Authentication: WAs are used by several kinds of users

SE-wiki Home Page | wiki Documentation | wiki Documentation Index-Page | wiki Basic Editing

<< Introduction | Taxonomy.TrailIndexPage | Taxonomy.TopDown-Architecture >>

Fig. 1. A screenshot of the Taxonomy Wiki

- existing literature about fault models, taxonomy, defect types for faults injection in non-Web systems, such as: object-oriented software systems (e.g., [5, 6, 16, 36, 37]), component-oriented systems (e.g., Mariani et al. [29]), service-oriented systems (e.g., Looker et al. [23]);

- existing literature about domain specific fault models and taxonomies proposed in various areas, such as computer security (e.g., Lough [24], Hayes et. al [15]);

- literature and documentation about the most commonly used Web architectures, technologies and languages, such as HTML, JSP, PHP, Javascript, Servlet, EJB, AJAX, XSLT and also communication protocols (HTTP, HTTPS, etc.);

- interviews (on-line and off-line) with Web developers and testers;

- on-line forums dedicated to Web developers;

- our experience/knowledge about Web testing.

Then, the initial taxonomy was refined by analyzing real faults, taken from on-line software repositories, such as Sourceforge[a], that provide the bug reports associated with each supported project. We mapped the reported bugs onto the categories in the taxonomy and refined the taxonomy accordingly.

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 some refinement steps is publicly available as a Wiki[b] page. Figure 1 shows the

[a]http://sourceforge.net
[b]http://selab.fbk.eu/marchetto/tools/pmwiki/pmwiki.php?n=Taxonomy

Characteristics	**Sub-Characteristics**
A.Multi-tier architecture	1.client pages interpreted by browsers
	2.server pages can dynamically generate client pages
	3.server-side components (e.g., JavaBeans) can be used
	4.forms and links are used to exchange data between components
	6.database can be used
	7.client-side pages can be stored in proxies or browser cache
B.GUI	1.interfaces can be HTML-based
	2.client pages can use executable components (e.g., Applets, scripting code)
	3.client pages can be organized on frames and framesets
	4.GUI are events-based
	5.GUI need to be internationalized and multi-languages
	6.GUI elements can send asynchronous requests to Web server
C.Session-based	1.server components can use session objects
D.Hyperlinked structure	1.hypertext and hyperlink are supported
	2.resources are accessed by URL
	3.URL can be static or dynamic
E.Protocols-based	1.secure communications can be used
	2.proxies can be used in the net
	3.different communication protocols (IMAP, POP3, etc.) can be used
F.Authentication	1.authorizations can be used to allow the users to use/access resources

Table 1. The initial taxonomy. High-level characteristics of Web applications.

starting page of the wiki. 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. Table 1 shows both characteristics and sub-characteristics in the initial taxonomy. The main high-level characteristics of Web applications identified in the top-down analysis and composing the taxonomy are:

- (A) Web applications are based on a multi-tier architecture (e.g., client, server, database);
- (B) Web applications use a specific kind of user interface, based on Web languages and technologies (e.g., HTML, Flash, Javascript);
- (C) Web applications support session management;
- (D) Web applications support hyperlink based-navigation and resources are accessed through URLs;
- (E) Web applications use Web protocols (e.g., HTTP, HTTPS);
- (F) Web applications 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.

For each sub-characteristic, we identified a set of related faults that may potentially lead to failures. Table 2 shows a fragment of the taxonomy focused on the classes of faults defined for each sub-characteristic. Some faults (e.g., f3 in A6) are not represented due to space limitation. For example, (f1) browser incompatibility; and, (f2) back button problems, are known as typical faults related to the HTML pages interpreted by Web browsers (sub-characteristic A.1).

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 *generic* faults (e.g., “index out of array bounds”) may occur also in Web applications, they are not represented by a specific category in the taxonomy since they are not associated with any Web-specific features.

4 Validation and refinement of the taxonomy

This section describes the bottom-up validation and refinement of the initial taxonomy, based on real bug descriptions that have been classified according to the taxonomy categories. This classification-based process has been repeated two times with two different sets of real bugs (376 bugs the first time, 300 the second).

A “good” taxonomy [37] 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 faults which 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 empirical study 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 assignment of faults to taxonomy classes, revealing, for instance, if the taxonomy contains overlapping classes or unclear bug descriptions.

RQ4 : *How Web-specific are the 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”.

RQ5 : *How stable is the taxonomy?* This requires to study the refinement steps of the taxonomy, measuring the number of class modifications (e.g., add or remove classes of faults) during the empirical validation. It gives us a preliminary indication of the maturity degree of the taxonomy we built.

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.

Ref. to Sub-Characteristics	**Classes of Faults**
A.1.	f1.faults related to browser incompatibility f2.faults related to back button f3.faults related to the needed plugins
A.2.	f1.faults during client-page construction
A.3.	f4.faults during file-system access f5.faults related to the use of component inputs f7.faults related to the framework configuration f8.faults related to server environment (e.g., Web server) f9.faults in data exchanged among components (e.g., character encoding)
A.4.	f1.faults during form construction
A.6.	f1.faults during database interactions or management f2.faults in loading information in database f4.faults in extracting information from database f7.incorrect database updating
A.7.	f1.faults related to cache managment f4.wrong storage of information in cache
B.1.	f1.faults related to HTML interpretation by the browser f3.faults while manipulating DOM objects f4.faults during CSS managment and use f11.faults during XSLT transformation and use
B.2.	f1.faults during components execution
B.3.	f1.faults on frame synchronization f2.faults on frame loading
B.4.	f1.faults in application presentation or GUI-events managment f2.faults on frame loading
B.5.	f1.faults related to characters encoding f3.unintended jump among languages
B.6.	f1.faults related to the asynchronous requests and response
C.1.	f2.faults in session synchronization f4.faults in persistence of session objects f5.faults while manipulating cookies f6.faults while manipulating session variables f7.faults related to URL rewriting
D.1.	f1.faults related to hyperlinks
D.2.	f1.faults due to the unreached resources f2.faults due to the not available resources
D.3.	f1.faults in URL definition/construction f4.wrong URL parameters settings
E.1.	f1.faults related to the use of encrypted communication f2.faults related to the use of the SSL protocol f3.faults related to the use of the HTTPS protocol
E.2.	f1.proxies do not support a given used protocols
E.3.	f1.faults related to not-supportted communication protocol
F.1.	f2.fault during user authentication f6.faults in accessing/using resources without permission f8.faults in role of management

Table 2. Initial taxonomy: sub-characteristics and fragments of faults classes.

4.2 Metrics

The following metrics are considered 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*). The confidence of classification is coded as a value between 0 and 1, with 1 meaning that the classifier is absolutely sure of the classification. Specifically, the tester tags each classified bug with *not sure*, *partially sure* or *absolutely sure*, coded as *not sure*=0, *partially sure*=0.5 and *absolutely sure*=1, depending on her/his level of confidence. The final metric is obtained as the average over all the classifications.
- *Agreement between classifiers*: percentage of bugs classified in the same fault class by different classifiers (used to answer *RQ3*). This metric should be taken with care because the process of fault classification is, by its own nature, subjective and highly dependent on the people performing the task [17]. A bug classified in the same class by two classifiers indicates that the bug and the taxonomy class are both accurately described and they are not ambiguous. Instead, a bug classified differently by two classifiers could indicate a "poor" bug description or an ambiguous/overlapping fault class. Clearly, there could be other causes, for example a different interpretation by the classifiers. It is usually very difficult to understand the actual cause for a given mismatch.
- *Rate of generic faults*: number of bugs associated with generic (i.e., not Web-specific) faults (used to answer *RQ4*).
- *Maturity degree*: number of refinement operations (adding or removing classes, splitting or merging existing classes, detailing the description of a class) done during different steps of the validation process (used to answer *RQ5*).

These metrics have been computed on the entire set of real bugs considered in all the seven iterations of the validation process. Moreover, *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 676 real bug descriptions from 68 software projects available under three of the most well known on-line software repositories used by the open source community: Sourceforge, Tigris.org[c] and PHP-PEAR[d].

Table 3 details information about a subset of the selected Web applications. In detail, in the table are shown: in which iteration (1-7) the Web application has been considered; the technology used; the application size, measured in number of files and LOC (Line Of Code); the number of

[c] http://www.tigris.org/
[d] http://pear.php.net/

Web application	**Iteration**	**Technology**	**Size**		**Libraries/Frameworks**	**Bugs**
			(# files)	(LOCs)	(number of)	(selected)
PHPAdmin	2	PHP	732	345298	3	15
AjaxAnywhere	3	Java	36	3152	0	5
ZK - Simply Ajax	4	Java	1618	120790	3	10
Sarissa	4	Java	170	19640	1	2
WebCalendar	2	PHP	498	105820	2	15
Websvn	5	PHP	126	17404	0	15
ServicesAmazon	5	PHP	14	2382	1	8
Alfresco	6	Java	1998	307400	6	10
OmniPortal	7	.NET	250	26368	1	14
SharpWebMail	7	.NET	320	64312	6	12
Workflow	6	Java	205	7500	0	7

Table 3. Some of the selected Web applications (11/68)

libraries/frameworks used; and the number of bugs selected for that Web application. For instance, in iteration 5, we have used 15 bugs of Websvn[e], a Web interface for SVN repositories, written in PHP.

Usually, bug descriptions for these software systems can be accessed in the software repository (e.g., Sourceforge) thanks to the bug tracking systems in use. For each Web application, a bug tracking system is typically managed by the development team, so that the other users and developers can raise bugs using informal descriptions via natural language, possibly adding some source code fragments or suggested code changes and patches.

The whole set of 676 collected bugs are distributed among three main Web technologies: (1) .NET; (2) PHP; and, (3) JSP and Java-like. Javascript, HTML and other common technologies are shared among most selected applications. The following rules have been applied to select the set of bugs to be used:

- the person that makes the selection of the set of bugs is different from the person that makes the classification and he/she has no knowledge about our Web fault taxonomy;

- the "selector" needs to have knowledge and experience about Web engineering and testing;

- the number of extracted bugs needs to be balanced across the considered Web applications. For instance, we never took a large number of bugs associated with one particular system, since the resulting taxonomy would be focused on the related domain and technologies. On the contrary, we need to get few bug descriptions, but from as many software systems as possible;

- a bug description needs to be discarded (e.g., not considered) if the description of the bug is not clear. "Clear" means that the description: (i) documents a bug (thus, requests for new features and similar requests are not considered); (ii) contains a detailed description of the documented bug; (iii) contains a limited number of nonsensical words or sentences (e.g., poor English, typos, or incorrect sentences); (iv) contains a limited number of words or sentences that are not consistent or pertinent with the rest of the bug description.

[e]http://websvn.tigris.org

4.4 Experimental Procedure

The entire process involved two (human) classifiers. It is divided into two phases. In the former phase (*Phase-1*), the initial taxonomy and a subset of 376 real bugs (randomly selected from the whole set) was used. In the latter phase (*Phase-2*) the other subset of 300 bugs and the taxonomy refined during *Phase-1* were considered. Between the two phases some time has been intentionally let pass in order to consolidate the knowledge about the taxonomy and the already executed classifications. The experiment has been performed in seven iterations, involving 100 bugs each (except for the fourth iteration, that was based on 76 bugs).

The procedure consists of the following steps (all of them were executed by the first classifier, while the second classifier accomplished only steps 1 and 2, with the purpose of measuring only the agreement between classifiers, to answer *RQ3*):

1. Each bug description is analyzed by the classifier to understand the fault that leads to the described failure or deviation from the expected behavior. In case the bug description is difficult to understand, the code of the Web application is also considered. In case the fault remains unclear, the bug description is discarded.
2. The fault is classified as generic or Web-specific. If the fault is Web-specific, it is assigned to a fault class in the taxonomy or it is marked as *Unclassified*.
3. The classifier declares the *Confidence of classification* for the given bug, by tagging the classification with: *not sure*, *partially sure* or *absolutely sure*.
4. If necessary, the classifier improves the fault class description to make it more precise.
5. If appropriate, according to the average bugs per class, the classifier marks the fault class as a candidate for splitting.
6. If the bug is unclassified, the classifier may record it and define a new fault class as a candidate for addition to the taxonomy.

After each iteration, the following operations are executed:

1. The 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* of the bugs distribution and produces more meaningful groups of bugs into fault classes.
3. Based on *Unclassified bugs* and on the candidates for addition identified by the classifier, new classes are added to the taxonomy. When possible, unclassified bugs are assigned to newly created classes.

At the end of Phase 1 and 2, a final *removal step* is conducted. Based on the final bugs distribution in the taxonomy (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 the removal step following Phase 2. All metrics are then recomputed, while the agreement between classifiers can be computed only at this point. 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. Finally, according to the results of the two-phase refinement, the *Maturity degree* is computed.

	Classes candidate to be				**Classes actually**			
Steps	**Split**	**Removed**	**Added**	**Refined**	**Split**	**Removed**	**Added**	**Refined**
	Phase 1							
Iteration 1	3	43	6	22	0	0	4	23
Iteration 2	4	31	3	17	0	0	0	17
Iteration 3	6	23	1	9	0	0	0	9
Iteration 4	0	23	0	2	0	9	0	2
All	13	23	10	46	0	9	4	51
	Phase 2							
Iteration 5	0	36	3	5	0	0	2	5
Iteration 6	0	20	0	2	0	0	0	2
Iteration 7	0	15	0	1	0	9	0	1
All	0	15	3	8	0	9	2	8
	Phase 1 ∪ Phase 2							
Final	13	26	13	47	0	18	6	47

Table 4. Data about the taxonomy refinement process

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 during *Phase-1* and 2 more classes during *Phase-2*. Moreover, 23 and 15 classes have been candidates for elimination given that none bug have been mapped to these classes during *Phase-1* and *Phase-2*. Following the opinion of the expert involved in the classification, only 9 of these classes have been actually removed from the taxonomy during *Phase-1* and other 9 have been removed after *Phase-2*. Hence, the taxonomy size was reduced to 64 after the *Phase-1* and to 57 at the end of *Phase-2*.

Table 4 shows the data collected after each iteration. The last row of the Table 4 summarizes the executed operations. In detail:

- 6 new classes have been added to the initial taxonomy, out of the 13 proposed;
- 18 classes have been removed, out of the 26 candidates;
- none of the 13 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.
- several classes of the taxonomy (51 in *Phase-1* and 8 in *Phase-2*) have been refined by the classifiers to improve the description of the captured type of faults (during *Phase-1* some extra classes in addition to the candidates have been refined).

Furthermore, in *Phase-1*, 79% of the classes of faults contain at least one bug, while 70% of bugs have been mapped onto 16% of fault classes of the taxonomy. Instead, in *Phase-2*, 61.5% of the classes of faults contain at least one bug, while 70% of bugs have been mapped onto 50,8% of fault classes of the taxonomy.

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 particular, 54 hours were spent to select

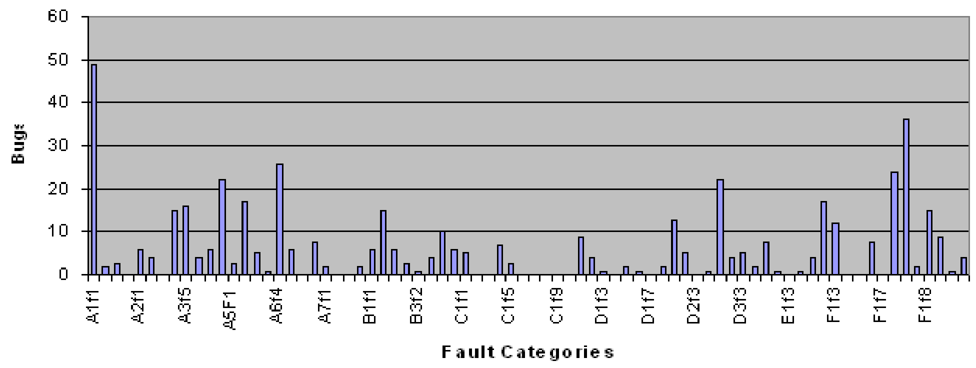


Fig. 2. The bugs distribution for the whole set of bugs considered

376 bugs of *Phase-1* (on average, 0.14 hours for bug) and then classified in 15.5 hours (on average 0.041 hours for bug). While during *Phase-2*, 38 hours were spent to select 300 bugs (on average 0.12 hours for bug) and then classified in 11.2 hours (on average 0.037 hours for bug). 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 deduce the fault behind the signaled misbehavior. To complicate the task, some descriptions are not really bug descriptions but just comments, questions and suggestions for improvement, although they were inserted into the bug tracking systems. Recognizing these kinds of descriptions and excluding them involved substantial effort.

4.6 Experimental results

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

Research Question RQ1 (Bugs distribution)

The bugs distribution was inspected after each iteration, to identify opportunities for class split or removal. The final taxonomy does not need to be fully balanced (e.g., rare defects may occupy small size classes), but we would like to avoid over-generic classes, attracting a huge number of defects, or ad-hoc classes, defined for specific instances. Figure 2 shows the overall bug distribution, according to the classes in the taxonomy, after all the 7 iterations (both *Phase-1* and *Phase-2*). It still contains peaks (e.g., A1f1) and flat zones (e.g., E1f3), but the classifier considered the related fault classes sufficiently accurate and meaningful. Most of the classes contain between 0 and 15 bugs, with a few exceptions. On average there are 6.26 bugs per class. Overall, the taxonomy looks well-balanced and classes were judged accurate enough.

Table 5 summarizes the most frequent classes of bugs, considering the whole set of bugs and both phases of the validation process. A1f1 (browser incompatibility) is the largest class, with 49 bugs mapped to it. Next comes a relatively small set of classes: A3f9 (faults in data exchanged among Web components), A6f4 (faulty extraction of information from database), A3f8 (faults related to server environment configuration) and A4f1 (faults during form construction), containing between 36 and 22 bugs.

However, the set of the most frequent bugs changes considering the distributions in the two phases.

Most Large Classes of Faults	
Class	**Faults**
A1f1 (faults related to browser incompatibility)	49
A3f9 (faults in data exchanged among Web components)	36
A6f4 (faults in extracting information from database)	26
A3f8 (faults related to server environment)	24
A4f1 (faults during form construction)	22

Table 5. Most frequent faults for the whole set of bugs

Largest Classes of Faults	
.NET	**Faults**
A3f5 (faults related to components inputs)	10
E1f2 (faults related to the use of the SSL protocol)	8
A4f1 (faults during form construction)	8
PHP	**Faults**
A1f1 (faults related to browser incompatibility)	30
A3f9 (faults in data exchanged among Web components)	13
D3f1 (faults in URL definition/construction)	10
JSP	**Faults**
A3f9 (faults in data exchanged among Web components)	16
A6f4 (faults in extracting information from database)	15
A1f1 (faults related to browser incompatibility)	14

Table 6. Most frequent faults per technology according to the whole set of bugs

For example, the largest class in *Phase-1* is A1f1 (browser incompatibility) with 30 bugs, while in *Phase-2* the largest class is A3f9 (faults in data exchanged among Web components), with 24 bugs. If we compute Pearson correlation and covariance on the five most frequent classes of bugs in the two phases, we obtain: correlation equal to -0.1 and covariance -6.75. Hence, the two sub-distributions are not correlated. Probably, this depends on the different types of technologies used in the Web applications chosen in the two phases. For instance, a lot of PHP applications were considered in *Phase-2* (e.g., *ServicesAmazon* in Table 3), while in *Phase-1* such kind of applications were less represented.

The most frequent classes of faults in the taxonomy have been analyzed also considering the technologies (i.e., .NET, PHP, JSP). Table 6 summarizes the most frequent bugs per technology considering both phases of the validation experiment. Again, we computed correlation and covariance for the bugs distributions associated to the three technologies. Since both correlation and covariance are substantially greater than zero, we conclude that the three bug distributions are all positively correlated. In detail, we measured the following *(correlation, covariance)* pairs: (0.61, 4.4) for .NET and PHP, (0.67, 5.8) for .NET and JSP and (0.76, 7.5) for PHP and JSP. This indicates that the considered bug distribution is quite representative of typical Web applications written in a given language.

Table 6 details the bug distribution of the whole set of bugs according to the technologies considered in the validation process. Analyzing the most frequent fault classes, we can observe that they are not the same across languages. For instance, A1f1 (browser incompatibility) is one of the most common bugs for PHP and JSP, but is not so much of a problem when .NET is used. Probably, this datum may indicate that bug A1f1 refers mainly to portability to the IE browser, which is ensured

Technology	**Bugs**								
	Phase-1			***Phase-2***			***Phase-1∪2***		
	Total	**Generic**	**Web**	**Total**	**Generic**	**Web**	**Total**	**Generic**	**Web**
.NET	100	22	77	100	19	75	200	41	152
PHP	166	63	100	100	17	75	266	80	175
JSP	110	36	73	100	16	75	210	52	148
all	376	121	250	300	52	255	676	173	505

Table 7. Generic vs Web-specific faults

to hold for .NET, since this technology is by construction tied to the Windows platform and to the IE browser. This datum could also explain why the correlation is stronger between PHP – JSP than between .NET – PHP and .NET – JSP.

After the execution of both phases of the process, 45 classes (out of 57) contain at least one bug. This means that approximately 79% of the taxonomy was validated empirically by at least one real bug mapped to one of its fault classes. The remaining 21% is still to be validated, notwithstanding the large number of bugs analyzed in the seven iterations conducted in this study. This datum, together with the fact that on average there are only 6.26 bugs per class, indicates 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. For this reason, the built taxonomy is publicly accessible and editable through a Wiki page.

Research Question RQ2 (Unclassified bugs)

The number of *Unclassified* bugs was 13, 3, 1, 0 respectively in the 4 iterations of *Phase-1*, and 8, 9, and 6 in the 3 iterations of *Phase-2*. Some of them, about 10 bugs, caused the addition of new classes to the initial fault taxonomy (e.g., fault in interface construction by means of XSLT; fault in management of different user roles and privileges; faults related to asynchronous communications between browser and Web server), while 5 (2%) and 20 (8.8%) remained unclassified after *Phase-1* and *Phase-2* respectively. These last bugs 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.

The classifier decided that the classification performance of the taxonomy was already "good", with only 5.2% of overall unclassified bugs, and that the addition of new classes for such unclassified bugs would be detrimental for it.

Research Question RQ3 (Ambiguity)

The average *Confidence of classification* for the first classifier is 91.5% (the confidence of classification was not collected for the second classifier). This value indicates high confidence in the classification performed.

Considering only Web specific faults, the classification *Agreement* between the two classifiers is 43%. After a round of discussion, the two classifiers agreed on a set of changes to their original classifications of bugs. Such changes produced an increase of the *Agreement* up to 77%. Differences in the classifications were mainly due to: (i) difficulties in interpreting the bug descriptions; (ii) overlaps among classes in the Web fault taxonomy; and, (iii) errors made by classifiers.

For example, one bug description of the first iteration was: "*editing a query with $>$ in the sql, write incorrectly to db when you have a query with sql that contains a $>$ (greater than) (possibly a less than*

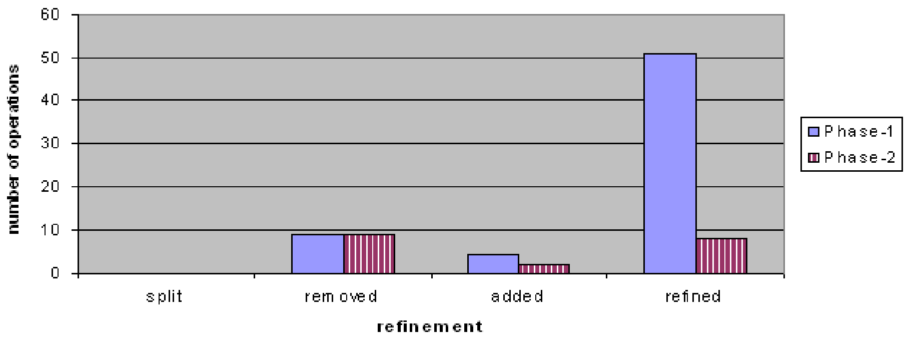


Fig. 3. Refinement operations by phase

will do this too). It writes it to the data base as a &gt; which then crashes the bug view". The first classifier classifies it with A6f7 (incorrect updating of databases in response to service requests) while the second with A6f1 (faults during database interactions or management). After a brief discussion, the classifiers agreed that the most accurate classification was A6f7, being A6f1 quite general.

Practical experiences with taxonomy construction and validation [37] indicate 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 and consistent with previous empirical studies [17]. More experimental work is of course necessary to confirm our findings and further validation conducted in other research labs would be more than welcome.

Research Question RQ4 (Generic vs Web-specific faults)

Table 7 details the number of generic and Web-specific faults for each considered technology. The overall *Rate of generic faults* (considering both phases) is 25.6% or 173 bugs out of 676. Inspecting these 173 generic faults, we found that a lot of them 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.

The interesting datum is that a large proportion of the bugs reported for Web applications, precisely 74.4%, is Web-specific. This indicates that Web-specific testing approaches are highly needed and extremely important to ensure the quality and reliability of Web applications.

Research Question RQ5 (Maturity degree)

Overall, during the classification process the classifier accomplished 71 operations of refinement of the taxonomy. Precisely 18 classes have been removed, 6 added and 47 refined (none has been split). In the first phase, 64 refinement operations were carried out, while in the second they were only 19, hence their number decreased between the two phases (see Figure 3). This indicates that the classification is approaching stability with the taxonomy's maturity. Moreover, the number of classes added in *Phase-2* was only 2, indicating that the taxonomy already captured most of the bugs considered in the second phase. The number of classes removed from the taxonomy was 9 in *Phase-1* and 9 in *Phase-2*. This may indicate that further iterations may be useful. In fact, the final taxonomy contains about 21% of classes (i.e., 12 classes) not associated with any real bug, i.e., classes not yet validated.

4.7 Threats to validity

In this section, we discuss the main threats to validity [39] that can affect our results: internal, construct, conclusion, and external validity threats.

- *Internal validity* concerns the existence of factors which may have affected the results and have not been properly taken into account. The obtained results might have been influenced by several factors. An internal validity threat is in the subjective steps conducted during the experiment, i.e., how the Web applications and the bugs have been chosen, how a bug is classified (Web-specific or generic), the assignment to the fault classes, the declaration of the confidence of classification and the operations of refinement of the taxonomy (add, remove, split and refine a class). To limit this threat: (i) a big number of Web applications (68) and related bugs (676) were analyzed; (ii) the person who made the selection of the set of bugs was different from the person who made the classification; (iii) a classifier expert in Web application testing was employed to execute the entire process; (iv) a second classifier was employed for augmenting the confidence of classification and measuring the agreement between classifiers; (v) a rigorous procedure was followed. Another factor that may have affected the results is the poor quality of the *Sourceforge* bug descriptions. In many cases they were not quite understandable, containing a big number of nonsensical words/sentences (e.g., poor English, typos, or incorrect sentences) or sentences non consistent with the rest of the bug description. To limit this threat, unclear bug descriptions were discarded. Another source of internal validity threat is the use of only two classifiers. We plan to replicate the study with more classifiers.

- *Construct validity* concerns the metrics or observations used to address the research questions. We selected a set of metrics (*Bugs per class*, *Unclassified bugs*, *Confidence of classification*, *Agreement between classifiers*, *Rate of generic faults* and *Maturity degree*) with a straightforward, clear meaning. As already said, some degree of subjectivity affects the computation of *Confidence of classification* and *Agreement between classifiers*. It is possible that these metrics do not provides the best and unique means to evaluate the fault taxonomy and its process.

- *Conclusion validity* concerns the possibility to derive legitimate conclusions from the observations. Since we could obtain no statistical measure of confidence, but only subjective confidence estimates, our conclusions rely mainly on the interpretation of the metrics and on the *Confidence of classification* declared by the classifiers. We complement this metric with the measurement of *Agreement*, which gives us another way to estimate the effects of the subjectivity.

- *External validity* concerns the generalization of the findings, i.e., to the possibility of applying our results to other Web applications. The selected applications are real Web applications belonging to different domains and technologies (e.g, .NET, PHP, JSP). Moreover, the chosen Web applications use a big number of software frameworks and libraries, in line with current trends. This makes the context quite realistic, despite further studies with other applications and also Web sites related to different domains are necessary to confirm or confute the obtained results. The representativeness of the considered bug dataset can be limited by: (i) number of bugs considered in the experiment (676 in total); and, (ii) the type of the bug descriptions, coming from the user community of open source software. We plan to consider also commercial Web applications and the related bug tracking systems in our future work.

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

5 Testing based on fault taxonomy

In this section, we show an example of how our taxonomy can be used for testing; specifically, how it can be used to derive test cases. We apply functional testing first and then taxonomy-based testing. The former is based on the analysis of system requirements and builds a test suite exercising the system functionalities. The latter uses both taxonomy and system requirements to build a test suite that exercises the application characteristics which can be mapped onto the taxonomy according to the system functionalities. Finally, a comparison between the two approaches is drawn, based on a set of documented faults.

The TAB 3.0 Web Application

TAB (**The Address Book reloaded**) is a Web system to manage contact lists. TAB can be downloaded from Sourceforge[f]. The version we considered (3.0) is based on PHP, Javascript, and HTML. Users can manage (e.g., add, remove, update) lists of contacts (e.g., friends, customers, employees) in which they store details (e.g., email, website, phone number) about them. Figure 4 shows the main panel with the commands to manage the lists. Users can visualize these lists and perform different queries to extract/search information or to select groups of contacts according to specific criteria. The basic version of *TAB* can be enriched with a set of utilities by means of existing plug-ins, such as the utility to generate PDF files that contain whole or part of the lists. Two roles are supported for the user: *administrator*, who manages plug-in activation and other users (Figure 5 shows the panel with administrator commands) and *registered user*, the owner of the contact lists. User roles, registration and authentication are managed by *TAB* in order to let the user access her/his resources and services.

TAB consists of 266 files for a total of about 49450 lines of codes. Overall, the application uses a set of five frameworks: Prototype, Scriptaculous, SortTable, Tabber, and PHPMailer (see http://tab-2.sourceforge.net for further information about how these frameworks are used in TAB).

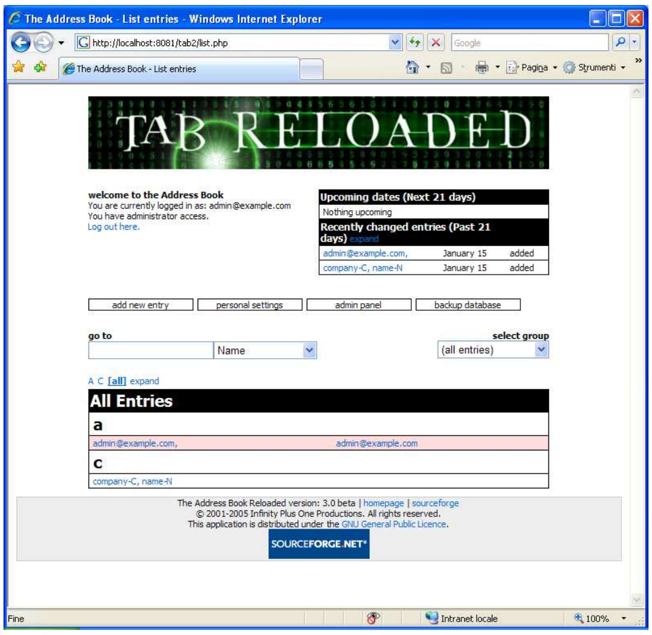
In the Sourceforge Website for *TAB* it is possible to access the application's documentation, consisting of:

- an informal use-case description that details the application requirements;
- an on-line demo that shows how the application works;
- a bug-tracker that describes 66 bugs of *TAB* 3.0.

Figure 6 shows a snapshot of the bug-tracker. Each bug is described, through an identifier code (e.g., "*1588078*"), a bug summary (e.g., "*adminPanel crashes, if plug-in not configured*"), the date in which the bug was raised (e.g., "*2006-10-31*"), the priority (an index in the range of *1–10*), the state of the bug ("*open or closed*"), and information about who submitted and resolved (if closed) the bug.

Among the 66 available bugs in the bug tracker, 27 have not been considered, because they are difficult to interpret (i.e., the description of the bugs is unclear) or reproduce in the considered version of *TAB*. The remaining 39 bugs are used to compare functional and taxonomy-based testing approaches, by counting the number of related bugs revealed by each technique.

[f] http://tab-2.sourceforge.net


The Address Book - List entries - Windows Internet Explorer

http://localhost:8081/tab2/list.php

welcome to the Address Book
You are currently logged in as: admin@example.com
You have administrator access.
Log out here.

Upcoming dates (Next 21 days)		
Nothing upcoming		
Recently changed entries (Past 21 days) expand		
admin@example.com,	January 15	added
company-C, name-N	January 15	added

add new entry | personal settings | admin panel | backup database

go to: Name — select group: (all entries)

A C [all] expand

All Entries

a

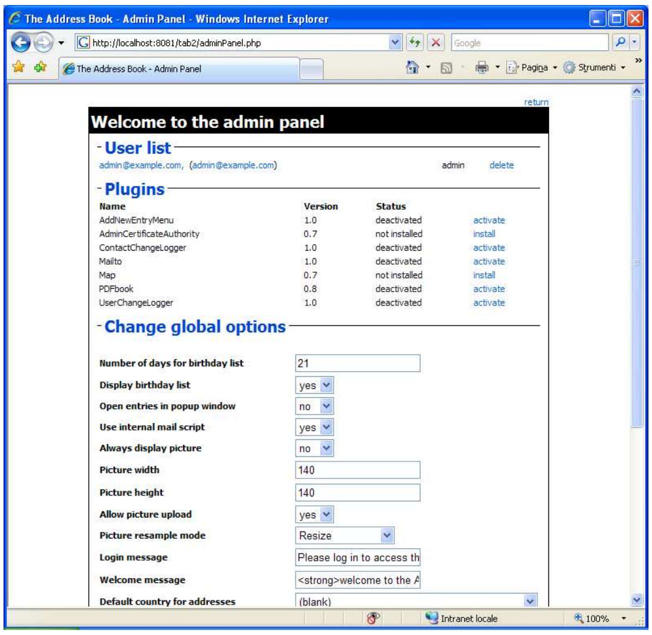
admin@example.com, admin@example.com

c

company-C, name-N

The Address Book Reloaded version: 3.0 beta | homepage | sourceforge
© 2001-2005 Infinity Plus One Productions. All rights reserved.
This application is distributed under the GNU General Public Licence.

Fig. 4. Snapshot of *TAB*, user panel


The Address Book - Admin Panel - Windows Internet Explorer

http://localhost:8081/tab2/adminPanel.php

Welcome to the admin panel

User list

admin@example.com, (admin@example.com) admin delete

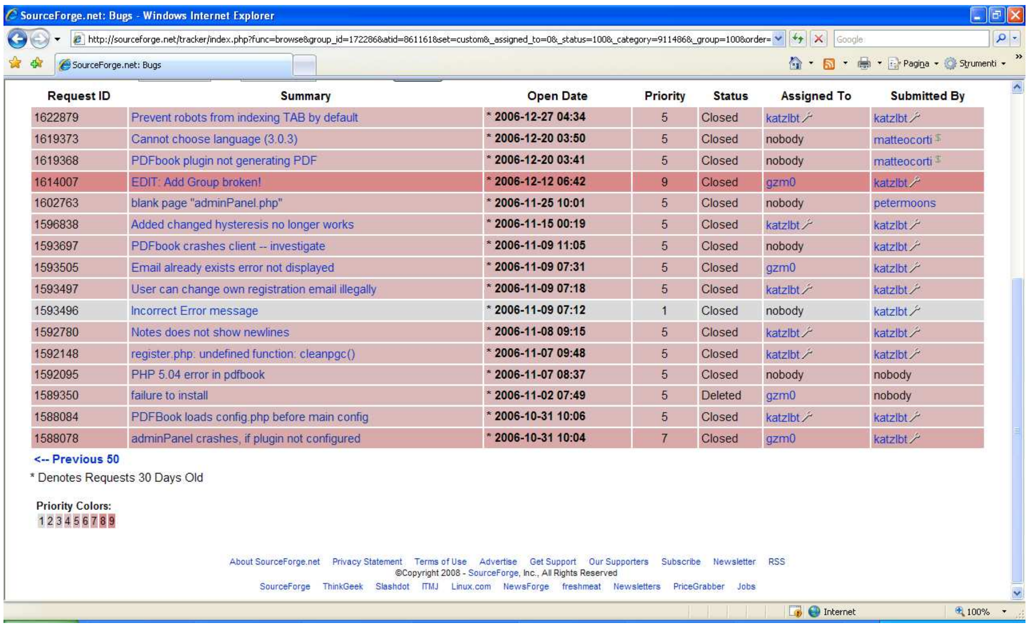
Plugins

Name	Version	Status	
AddNewEntryMenu	1.0	deactivated	activate
AdminCertificateAuthority	0.7	not installed	install
ContactChangeLogger	1.0	deactivated	activate
Mailto	1.0	deactivated	activate
Map	0.7	not installed	install
PDFbook	0.8	deactivated	activate
UserChangeLogger	1.0	deactivated	activate

Change global options

Number of days for birthday list	21
Display birthday list	yes
Open entries in popup window	no
Use internal mail script	yes
Always display picture	no
Picture width	140
Picture height	140
Allow picture upload	yes
Picture resample mode	Resize
Login message	Please log in to access th
Welcome message	<strong>welcome to the A
Default country for addresses	(blank)

Fig. 5. Snapshot of *TAB*, administrator panel


SourceForge.net: Bugs - Windows Internet Explorer

http://sourceforge.net/tracker/index.php?func=browse&group_id=172286&atid=861161&set=custom&_assigned_to=0&_status=100&_category=911486&_group=100&order=

Request ID	Summary	Open Date	Priority	Status	Assigned To	Submitted By
1622879	Prevent robots from indexing TAB by default	* 2006-12-27 04:34	5	Closed	katzlbt	katzlbt
1619373	Cannot choose language (3.0.3)	* 2006-12-20 03:50	5	Closed	nobody	matteocorti
1619368	PDFbook plugin not generating PDF	* 2006-12-20 03:41	5	Closed	nobody	matteocorti
1614007	EDIT: Add Group broken!	* 2006-12-12 06:42	9	Closed	gzm0	katzlbt
1602763	blank page "adminPanel.php"	* 2006-11-25 10:01	5	Closed	nobody	petermoons
1596838	Added changed hysteresis no longer works	* 2006-11-15 00:19	5	Closed	katzlbt	katzlbt
1593697	PDFbook crashes client -- investigate	* 2006-11-09 11:05	5	Closed	nobody	katzlbt
1593505	Email already exists error not displayed	* 2006-11-09 07:31	5	Closed	gzm0	katzlbt
1593497	User can change own registration email illegally	* 2006-11-09 07:18	5	Closed	katzlbt	katzlbt
1593496	Incorrect Error message	* 2006-11-09 07:12	1	Closed	nobody	katzlbt
1592780	Notes does not show newlines	* 2006-11-08 09:15	5	Closed	katzlbt	katzlbt
1592148	register.php: undefined function: cleanpgc()	* 2006-11-07 09:48	5	Closed	katzlbt	katzlbt
1592095	PHP 5.04 error in pdfbook	* 2006-11-07 08:37	5	Closed	nobody	nobody
1589350	failure to install	* 2006-11-02 07:49	5	Deleted	gzm0	nobody
1588084	PDFBook loads config.php before main config	* 2006-10-31 10:06	5	Closed	katzlbt	katzlbt
1588078	adminPanel crashes, if plugin not configured	* 2006-10-31 10:04	7	Closed	gzm0	katzlbt

<-- Previous 50

* Denotes Requests 30 Days Old

Priority Colors:
1 2 3 4 5 6 7 8 9

About SourceForge.net Privacy Statement Terms of Use Advertise Get Support Our Supporters Subscribe Newsletter RSS
©Copyright 2008 - SourceForge, Inc., All Rights Reserved
SourceForge ThinkGeek Slashdot ITMJ Linux.com NewsForge freshmeat Newsletters PriceGrabber Jobs

Fig. 6. Snapshot of the *TAB*, bugs list

Functional Testing

Functional testing (or black-box testing) is based on execution scenarios and use cases [18, 20, 33, 40] derived from the system's functionalities. In the literature, some empirical investigations (e.g., Basili [4]) show that the functional testing is one of the most effective testing approaches.

Use cases can be defined during the analysis phase of the system's development. They are used to fully describe a sequence of actions performed by the system, to provide an observable result of value to a person or another system. In other words, each use case represents one functionality of the system in execution. When not available, use cases can be extracted manually from other sources of documentation or using reverse-engineering (see Bojic et al. [7] and Di Lucca et al. [25]). Given a use case for a system, a scenario is an instance of it, or a complete and executable path through the use case. A scenario is an execution of a functionality of the system described in the use case, e.g., the execution of a specific event flow (the basic or a combination of basic and alternate flows), among those described in the use case.

Use cases are expressed by means of a structured, textual description, following a specific format and using natural language. They can be visually represented in use case diagrams [30], where ovals represent use cases, stick figures represent "actors", which can be either humans or other systems, and lines represent relationships between these elements.

We organized the use cases for TAB according to the following guidelines:

- The textual description of a use case has the following format: (i) name of the use case; (ii) brief description; (iii) flow of events (describing what the system does regarding that use case); (iv) special requirements (describing all requirements, such as non-functional ones, that are not modeled in the use case); (v) preconditions; and (vi) postconditions.
- The flow of events is divided into: (a) basic flow, that describes what "normally" happens when

the use case is performed; and, (b) the alternate flows of events, that describe the behavior of the system in case of optional or exceptional variations of the "normal" behavior.

To extract suites of test cases from the use cases, we applied the *three-steps approach* proposed by Heumann [18]:

- (step F1) for each use case, a representative set of scenarios is generated;
- (step F2) for each scenario, a test case and the conditions that will make it "executable" are identified;
- (step F3) for each test case, data values able to exercise the corresponding execution of the system are selected.

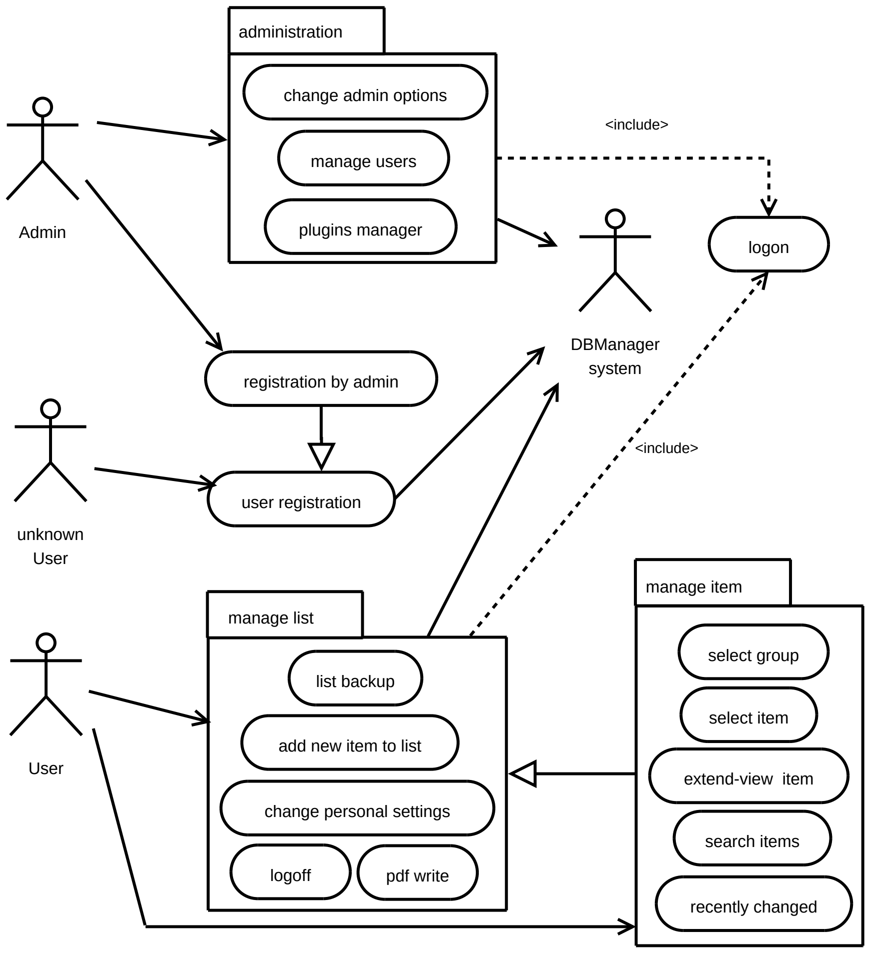
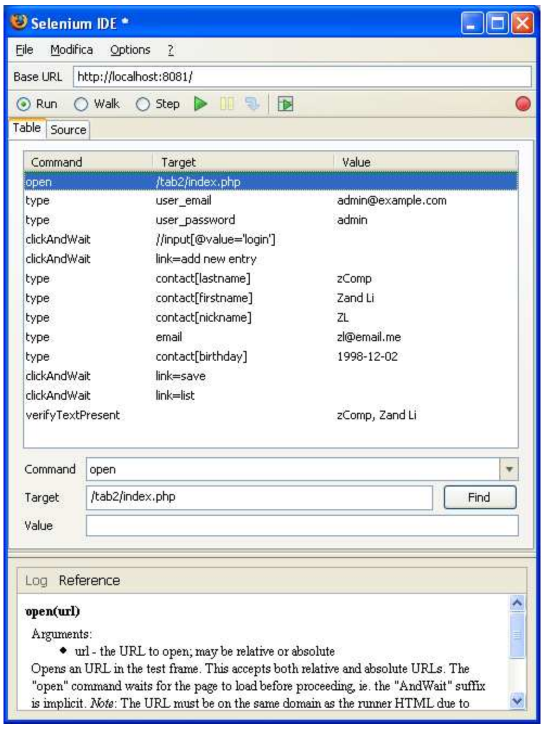


Fig. 7. Use Case Diagram for *TAB*

Using the available documentation, we recovered and refined — by executing the Web application — the use cases for TAB and we built the use case diagram. The result is shown in Figure 7, which shows the main functionalities implemented by the *TAB* application.



Base URL: http://localhost:8081/

Command	Target	Value
open	/tab2/index.php	
type	user_email	admin@example.com
type	user_password	admin
clickAndWait	//input[@value='login']	
clickAndWait	link=add new entry	
type	contact[lastname]	zComp
type	contact[firstname]	Zand Li
type	contact[nickname]	ZL
type	email	zl@email.me
type	contact[birthday]	1998-12-02
clickAndWait	link=save	
clickAndWait	link=list	
verifyTextPresent		zComp, Zand Li

Command: open
Target: /tab2/index.php
Value:

open(url)

Arguments:

- url - the URL to open; may be relative or absolute

Opens an URL in the test frame. This accepts both relative and absolute URLs. The "open" command waits for the page to load before proceeding, ie. the "AndWait" suffix is implicit. *Note*: The URL must be on the same domain as the runner HTML due to

Fig. 8. A Selenium test case for *TAB*

As an example, let us consider the use case "add new item to list" and let us apply to it the tree-step approach. The basic flow (*Bf*) of events for it is:

- Bf.1: the user clicks the *add new entry* tab;
- Bf.2: a form-based page for the new contact is loaded by *TAB* replacing the starting page of the application;
- Bf.3: the user fills the form with information regarding the new contact that he/she wants to add to the list (e.g., first name, last name, nickname, address, email, website link, phone number, photo, birth date); in particular, the following information: name, email, and birth date are mandatory to create a new entry, while the others are optional;
- Bf.4: the user clicks the *save* link in order to persistently store the contact information;
- Bf.5: the system persistently stores the inserted data about the new contact into the database;
- Bf.6: the system loads the starting page, listing also the new contact.

The unique pre-condition to exercise this use case is that a user can *log on* to the system, by providing login and password. The post-condition is that a new contact (e.g., friend) is added to the user list of contacts (with detailed information about it).

An alternate flows of events for the use case "add new item to list" can be the following:

- Alt.1: (alternate to Bf.3) the user fills the form with contact information but he/she does not fill, at least, one of the mandatory data for a new contact and so the system cannot add the new contact to the current list;
- Alt.2: (alternate to Bf.4) the user clicks the *cancel* link and leaves the new contact page.

Let us consider both the basic (*Bf*) and alternate (*Alt*) flows of events. The following set of scenarios can be defined (step F1):

- Scen.1: add one new contact (Bf.1, ..., Bf.6);
- Scen.2: partially fill contact information so no new contact is created (Bf.1, Bf.2, Alt.1, Bf.4);
- Scen.3: leave the new contact page after filling contact information (Bf.1, Bf.2, Bf.3, Alt.2);
- Scen.4: partially fill contact information and leave the new contact page (Bf.1, Bf.2, Alt.1, Alt.2);

Based on these scenarios (such as Scen.1, ..., Scen.4) a set of test cases — at least one for scenario — have been defined to exercise the *TAB* application. Following the *three-steps approach* [18], we identified the conditions that make each scenario executable (step F2) and the related input data (step F3). For instance, Figure 8 shows a test case that realizes Scen.1 (*add one new contact*) using the Selenium[g] tool. This test case aims at testing the TAB functionality "add new item to list". It is composed of a set of user "actions" (user interactions with a Web page) described as commands provided by Selenium ("Command" column). These commands are applied to specific elements of the current Web page ("Target" column) with the specified inputs ("Value" column). The last row contains an assertion (the test oracle) to verify the content of the resulting Web page.

In detail, Scen.1 is realized — in the test case shown in Figure 8 — by means of the following actions: log-on as a registered user, click the "add new entry" link, fill the mandatory fields of the new contact page, save the information regarding the new contact and verify whether the added contact is in the new displayed list.

By applying the same procedure to the other use cases, we built a functional test suite for TAB. It consists of 64 test cases expressed as Selenium test scripts and it exercises 15 out of 16 use cases. We decided not to test the "registration" use case because this functionality requires an external mail server. The resulting test suite revealed 26 out of the 39 known bugs (i.e., 66.7%).

Taxonomy-based testing

We derive a test suite from our Web fault taxonomy as follows:

- (step T1) Select the fault classes that apply to the system under test. For instance, if an application does not use Javascript or SSL for secure communications, the classes of faults related to these technologies will be discarded.
- (step T2) Define at least one usage scenario of the application under test for each selected class of faults of the taxonomy.

[g]http://www.openqa.org/selenium/

- (step T3) Build, for each scenario, a test case that exercises the application, by specifying the input values that make the scenario executable.

Some classes of faults have been excluded, since the related technologies are not used in *TAB* (*step T1*). For instance, the entire characteristic *E* (Protocol-Based) is excluded, since no specific protocols (e.g., secure connections) are used. The fault classes *A1f3* (browser plug-ins) and *B1f11* (related to GUI built using XSLT transformations) are also not considered, since the application does not require any ad-hoc browser plug-in (e.g., Flash) and it does not apply any XSLT transformation to build the GUI.

Some of the characteristics, sub-characteristics and fault classes considered are the following:

- The sub-characteristic *A6* (Database interactions) is relevant, since *TAB* uses a database to permanently store information about users and contact lists. Thus, the interactions with the database are tested.
- *B1* (browser interpretation) and A1 (browser portability) are taken into account.
- *C* (Session-based) needs to be tested since *TAB* uses session objects (e.g., the contact list). Since the application stores objects during the navigation session of each authorized user, the fault class *C1f4* (faults in persistence of session objects) is used to derive some test cases.
- *F* (Authentication) is relevant, since the application implements an authentication mechanism to give users proper privileges (e.g., *TAB* plug-ins are managed only by the administrator).

Then, (*step T2*) for each selected fault class we defined at least one usage scenario. Finally, (*step T3*) for each scenario, at least one test case has been produced.

The resulting test suite consists of 45 test cases, which correspond to exercising 14 out of 16 overall use cases (defined for functional testing). Only the use cases “registration” and “recently changed” are not exercised by the taxonomy-based test suite. The first is also not exercised by the functional test suite, for similar reasons. The taxonomy-based test suite has been executed by means of Selenium. Running the test suite, a set of 35 bugs, out of 39 known bugs (89.7%) has been revealed.

Comparison and lessons learned

We investigate the following two research questions:

- *RQ6*: Does the fault taxonomy increase the ability of finding faults?
- *RQ7*: Is there any additional effort involved in using the fault taxonomy during testing?

Table 8 summarizes the results obtained from our testing activity. The functional test suite is composed of 64 test cases and it revealed 26 out of the 39 known bugs (i.e., 66.7%) (see Table 9 for details about the revealed bugs). Taxonomy-based testing produced a suite of 45 test cases that revealed 35 out of 39 known bugs (i.e., 89.7%) (see Table 9 for details). Looking at these data we can conclude that in this example the use of the taxonomy was quite effective. The test suite based on the taxonomy contains less test cases than the functional test suite (45 vs 64) and it was able to reveal more faults (35 vs. 26). Another interesting result is that 24 faults have been revealed by both techniques while 2 faults have been revealed only by functional testing — 1643709 (unprivileged

Testing	**Test Cases**	**Use-Cases Exercised**	**Non-Considered Fault Classes**	**Found Bugs**	**Bugs Found only by one technique**
Functional	64	15 (out of 16)	25 (out of 57)	26 (out of 39)	2
Taxonomy-Based	45	14 (out of 16)	13 (out of 57)	35 (out of 39)	11

Table 8. Report on *TAB* test suites

view of contacts) and 1832517 (admin hidden value) — and 11 faults only by taxonomy-based testing. Therefore, our taxonomy was able to help the tester to find 11 more faults otherwise not revealed.

Moreover, Table 8 shows that when using functional testing 25 classes of faults in the taxonomy (out of 57) are not considered, instead of the 13 that are not considered with taxonomy-based testing. This means that several classes of faults (such as browser portability, back button, Web server configuration, data encoding, session synchronization, etc.) have not been considered at all in the test suite based on the systems functionalities. This could explain the different results obtained, in terms of: (i) bugs found by the two testing techniques; and (ii) bugs found by only one of the two techniques. For instance, (see Table 9) bugs such as 1619673 (GUI languages), 1674651 (browser GUI), 1671569 (CSS) and 1743848 (UTF-8 characters) haven't been found by the functional testing technique since they do not regard directly any functional aspect of the application. The same faults have been revealed by the taxonomy-based approach because this technique considers also non-functional and technological aspects of the Web application under test.

Regarding the effort, we have measured the hours employed in the preparation of the two test suites (functional and taxonomy-based). Building the use case diagram (Figure 7) and defining the use cases (textual descriptions) starting from the TAB Web application and related documentation, took approximately 6 hours. In functional testing, we employed 25 hours for analyzing the 15 use cases and producing the 64 Selenium test cases. Instead, for taxonomy-based testing we employed 6 hours to analyze the code of the application and 7 hours to analyze the taxonomy and discard the fault classes that cannot be applied to TAB (step T1 of taxonomy-based testing). In the other two steps of our procedure, to define the 45 Selenium test cases, we employed about 27 hours (approximatively 35 minutes for each test case). Overall, the construction of the functional test suite took 31 hours while the construction of the taxonomy-based test suite took 40 hours. Looking at these data we can conclude that, in our example, the use of the taxonomy involved additional effort. Precisely, the effort added by the use of the taxonomy was 9 hours more (i.e., 22.5%)

With reference to the research questions *RQ6* (effectiveness of taxonomy in finding faults) and *RQ7* (additional effort), according to this experience, we can conclude that:

- *RQ6*: The use of the fault taxonomy increases the ability of finding faults. In fact, a tester, guided by the fault classes included in the taxonomy, can verify not only the functional requirements but also the technological and non-functional aspects of the system.

- *RQ7*: Additional effort is required in taxonomy-based testing, since a tester needs to take into account new elements with respect to functional testing: the fault classes and the implementation of the Web application. The implementation needs to be considered because fault classes describe technological aspects of a Web system, such as the use of specific protocols or session mechanisms. Such issues are not considered in use cases and functional requirements.

Since we conducted a case study, not an experiment with replications, it is not possible to assess the external validity of our findings through statistical analysis (i.e., the obtained results might not generalize to other projects). However, we expect that similar results are obtained on other case studies having similar features as the presented one. The considered technology (PHP, Javascript and HTML) is widely used. The application domain is quite representative (personal data management through the Internet), although different domains may involve different functionalities. The application size (around 50,000 lines of code) is pretty typical of existing Web applications. The main threat to the generalization of our results is the level of knowledge of the fault taxonomy possessed by the test engineer involved in the case study. In fact, the study was carried out by one of the two classifiers who contributed to the taxonomy refinement described in Section 4. Hence, we can expect a comparably effective usage of the taxonomy in test case definition only under the assumption that the test engineer has a deep knowledge of the taxonomy classes and how they can be instantiated into actual faults occurring in the code. In other words, we expect that some training and learning on the taxonomy classes is required to achieve the performance reported in this paper.

According to the documented experience, the main lessons learned are the following: (i) the fault taxonomy can be effectively used for Web testing; (ii) taxonomy-based testing can reveal faults regarding non-functional and technological aspects otherwise unnoticed or very hard to detect using the functional testing technique; (iii) functional testing is still useful, in that it can reveal complementary faults; and, (iv) the effort involved in taxonomy-based testing — measured in working hours — is increased, compared to functional testing.

6 Related works

In software testing and verification, some effort has been spent by the scientific community to build fault taxonomies, categories and models or to apply them in different domains (e.g., object-oriented and service-oriented software systems) and contexts (e.g., software quality or security). The main reason is that knowing the types of faults that can occur for a specific kind of software system is useful to understand: (i) what type of testing or verification approach can be applied; (ii) whether a new one is needed; (iii) how adequacy and effectiveness of testing and verification techniques can be evaluated and compared; (iv) how effective is a given suite of test cases; (v) how a test suite can be prioritized or reduced; (vi) how the ability of a test suite in finding faults can be increased; and (vii) how to evaluate the relationships among: faults, test cases and software technologies.

Several fault taxonomies and models have been presented to classify programming errors for specific kinds of software systems. For instance, Chillarege et al. [8] and the IEEE 1044-1993 Standard Classification for Software Anomalies [19] define two of the most well-known and high-level software faults classification systems tied with the software development process and implementation. Basili et al. [4] define a fault classification for structured (e.g., Fortran) programs. Basili defines a fault model that groups faults according to two schemes: (i) faults commission (due to the presence of wrong code) and omission (due to the absence of pieces of code); and (ii) faults divided into six classes: initialization, computation, control, interface, data, and cosmetic (according to the target component and/or structure associated with the fault). Mahaweerawat et al. [27] use an ad-hoc fault model to define a metric-based prediction system for identifying faults in object-oriented systems. Mariani [29] defines a fault taxonomy for component-based applications. It is composed of six high-level groups of faults (syntactic, semantic, non-functional, connectors, infrastructure, topology, other) divided according to the architecture of a typical component-based application. For each identified group, Mariani

Sourceforge Bug		**Testing based on**	
Id	**Summary**	**Functionalities**	**Taxonomy**
1588078	adminPanel crashes	Yes	Yes
1588084	PDFbook loads wrong file	Yes	Yes
1592148	undefined function in registering	Yes	Yes
1592780	Notes does not show newline	Yes	Yes
1593496	Incorrect error message	Yes	Yes
1593505	*email not displayed*	*No*	*Yes*
1593697	PDFbooks crashes the client	Yes	Yes
1614007	add group in edit breaks	Yes	Yes
1619368	PDF file not generated	Yes	Yes
1619373	*cannot choose language*	*No*	*Yes*
1622879	prevents robots from indexing TAB	No	No
1626976	*style option ignored*	*No*	*Yes*
1632092	sorting of property	Yes	Yes
1632366	Map plug-in breaks	No	No
1635749	duplicate error message	Yes	Yes
1639671	country sort is wrong	Yes	Yes
1641671	*PHP strict mode error*	*No*	*Yes*
1641674	registration takes long time	Yes	Yes
1642015	*database structure error*	*No*	*Yes*
1643709	*unprivileged view of contacts*	*Yes*	*No*
1644572	plug-in manager loads wrong file	Yes	Yes
1652523	problems in uploading images	Yes	Yes
1654299	*name and group can overlap*	*No*	*Yes*
1659576	*performance of database query*	*No*	*Yes*
1663334	login failure	Yes	Yes
1663451	*URI missing in vCard export*	*No*	*Yes*
1664418	HTML code in info fields	Yes	Yes
1666760	single quotes in notes	Yes	Yes
1671569	*invisible links in CSS style*	*No*	*Yes*
1674651	*IE7: text/image overlap*	*No*	*Yes*
1678821	Resend confirmation email fails	No	No
1684678	Error message in edit options	Yes	Yes
1684707	save data fails	Yes	Yes
1688158	ZIP code forced to zero	Yes	Yes
1743848	*UTF-8 characters break*	*No*	Yes
1745972	user account creation fails	Yes	Yes
1760221	Hide entry button	Yes	Yes
1830345	*login failure*	*No*	*Yes*
1832517	*admin hidden value*	*Yes*	*No*

Table 9. Report about the revealed bugs for *TAB* 3.0

details several low-level categories of faults. Bruning et al. [35] and Looker et al. [23] built two fault taxonomies for service-oriented applications and Web services respectively. In both cases, the main architectural details of typical SOA-applications are identified (e.g., Bruning identifies such features: publishing, discovery, composition, binding, execution) and then detailed using a multi-layer system which helps in identifying typical faults.

Other taxonomies have been developed with different purposes. For example, Utting et al. [36] built a taxonomy to classify model-based testing methods, while, Lough [24] built another taxonomy to classify computer vulnerabilities and security issues.

Several works, such as Harrold et al. [14], Artho et. al [3], Voas et. al [38], and Aidemark et al. [1], study fault categorization approaches and use them in fault injection systems. They present sets of defect types in the form of changes to be made into the code (e.g., object oriented code), with the purpose of simulating the occurrence of *real* software faults. To this aim, often, two approaches are applied, fault seeding and code mutation. Code mutation uses a set of mutant operators to make small syntactical changes in the application code. Fault seeding tries to inject more realistic and semantics-oriented faults. Thus, the main difference is that mutation works at the syntactical level, instead of the semantical level of fault seeding. Ma et al. [26] present a mutation system working at class level for Java system, while Harrold et al. [14] present a fault model for fault seeding in Java software.

Fault injection is recognized as an important approach to experimentally validate the dependability of software systems and evaluate the effectiveness in finding faults of testing techniques. Eldh et al. [12] propose an experimental framework for comparison of test techniques with respect to efficiency, effectiveness and applicability. The first step of that framework is fault injection. Basili et al. [4] uses their fault model to do fault injection and compare different testing techniques (i.e., code reading, functional and structural testing) according to their ability in finding faults. Heimdahl et al. [16] use fault injection to evaluate a test suite reduction technique applied to test cases generated using a model-based approach.

Another aim of fault models and taxonomies is to increase the effectiveness of a given test suite, by evaluating it on a set of faulty applications. In other terms, fault models are used to artificially inject faults into applications with the aim of generating a set of applications similar to the original one but with one fault injected in each. These faulty-applications are tested to evaluate the effectiveness in finding faults of the given test suite and to improve it according to its capability of revealing faults — in fact, the number of faults revealed is used as a coverage criterion. Vijayaraghavan et al. [37] present a survey of bug taxonomies used to improve software testing. Bieman et al. [6] use a fault injection mechanism, based on assertion violation (pre- and post-conditions of functions), to increase the coverage of test suites.

Other works investigate the relationship between faults and software components, by taking measures such as the fault density. The aim is to study the distribution of faults in system structures and components in order to, e.g., identify the most fault-prone pieces of code and build fault-prediction systems. Hayes et al. [15] study the relationships that exist between faults and system modules. They build both fault and module taxonomy and then study the match-relationship between them. The idea is that the types of faults introduced by programmers depend on the types of modules that are being developed. The result is that this relationship can be used to guide code reviews, walkthroughs, and verification. Another work, by Fenton et al. [13], documents an experiment in which the relationships among faults, failures and software components is evaluated in different versions of software systems. The results show that modules that are more fault-prone in a pre-release often become the least fault-prone after release, and viceversa.

Only a few very preliminary works considered fault taxonomies for Web applications. Ricca et al. [32] present a very preliminary Web fault taxonomy, consisting of a single level of quite general fault categories. Other works, such as Sprenkle et al. [34] and Karre et al. [21], use ad-hoc and limited defect injection systems, used to evaluate the effectiveness of testing approaches, which are not based on any explicit and exhaustive Web fault taxonomy. Ecott et al. [10] present a case study that compares fault injections done by manual fault seeding and code mutation for a given Web application. The results show that manual seeding creates more realistic faults than mutants, and that traditional mutation operators (e.g., like those available in MuJava [26]) are not sufficient, a Web system needs specific operators (e.g., for forms, links and Web page display faults).

The present work aims at providing a detailed, complete Web fault taxonomy, to be used by practitioners defining test cases and researchers comparing alternative Web testing approaches. Our taxonomy is the first Web fault taxonomy constructed according to all main structural characteristics of Web systems and validated empirically using a large set of real bugs, taken from online bug tracking systems. This work extends our previous one, Marchetto et al. [28], in several respects: taxonomy validation was extended, with 300 more bugs and one additional phase (Phase 2); we investigated one more research question (RQ5); we used the taxonomy for test suite construction and we assess its effectiveness on a case study.

7 Conclusions and future work

After defining an initial Web fault taxonomy top-down, from high-level to specific features of Web applications, we validated and tuned it empirically by asking two classifiers (expert testers) to map a set of 676 real bugs onto fault classes. Refinement of the taxonomy was conducted in seven iterations by one classifier, based on its balance and on the existence of unclassified bugs.

In the final taxonomy, (i) bugs are quite evenly distributed among 57 fault classes; (ii) according to the average of bugs per class, fault classes are not-too-big and not-too-small; (iii) only few bugs (25 out of 475 Web faults) cannot be mapped onto any fault class; (iv) fault classes are not ambiguous since the two classifiers agreed on most classifications; (v) around 2/3 of the bugs were classified as Web-specific, thus indicating the relevance of this kind of faults and the importance of testing techniques specific for Web applications; (vi) after seven iterations, the maturity degree of the taxonomy is reasonably acceptable.

Constructing a comprehensive and agreed Web fault taxonomy is a huge task that cannot be afforded by only one research group. We intend to involve as many researchers as possible, in order to continue validating and enhancing the taxonomy produced in this work. For this reason, we make the taxonomy available as a Wiki page.

Testers can use our taxonomy to define test cases that target specific classes of Web faults, while researchers can use it to build fault seeding tools, to inject artificial Web faults similar to real one into benchmark applications. In this paper, we describe a case study in which test cases have been derived for a real Web application from the fault classes in our taxonomy. The case study indicates that: (i) the proposed taxonomy is a useful complement for testers to devise test scenarios that have higher chances of revealing Web specific faults; (ii) the effort involved in taxonomy-based testing is increased if compared to functional testing.

Future works will be devoted to continue the empirical validation and enhancement of the taxonomy produced in this work. We plan to conduct a comparative experiment, to estimate the effectiveness of taxonomy-based testing with respect to other testing techniques specifically proposed for Web applications. We are also working on the construction of 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.

Acknowledgements

Filippo Ricca is partially supported by the project "Iniziativa Software" Finmeccanica (Research unity of Software Engineering coordinated by Prof. Egidio Astesiano, DISI, Genova, Italy) funded by Elsag-Datamat.

References

1. Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Goofi: Generic object-oriented fault injection tool. International Conference on Dependable Systems and Networks (DSN), July 2001.
2. A. Andrews, J. Offutt, and R. Alexander. Testing Web Applications by Modeling with FSMs. Software and System Modeling, Vol 4, n. 3, July 2005.
3. Cyrille Artho, A. Biere, and S. Honiden. Enforcer-efficient failure injection. Formal Methods (FM), 2006.
4. Victor R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. IEEE Transactions on Software Engineering, 13(12):1278–1296, December 1987.
5. Boris Beizer. Software Testing Techniques. Van Nostrand Reinhold, New York, USA, 1990.
6. James M. Bieman, D. Dreilinger, and Lijun Lin. Using fault injection to increase software test coverage. International Symposium on Software Reliability Engineering (ISSRE), October 1996.
7. Dragan Bojic and Dusan Velasevic. A method for reverse engineering of use case realisations in uml. Australasian Journal of Information Systems, 8(2), 2001.
8. R. Chillarege, S. I. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, K. Ray Bonnie, and MY. Wong. Orthogonal defect classification - a concept for in-process measurement. Transaction of Software Engineering, 18(11):943–957, 1992.
9. Giuseppe A. Di Lucca, Anna Rita Fasolino, Francesco Faralli, and Ugo De Carlini. Testing Web applications. International Conference on Software Maintenance (ICSM), October 2002.
10. Stacey Ecott, Sara Sprenkle, and Lori Pollock. Fault seeding vs. mutation operators: An empirical comparison of testing techniques for web applications. In Grace Hopper Celebration of Women in Computing 2006, October 2006.
11. S. Elbaum, S. Karre, and G. Rothermel. Improving Web application testing with user session data. International Conference on Software Engineering (ICSE), pages 49–59, May 2003.
12. Sigrid Eldh, Hans Hansson, Sasikumar Punnekka, Anders Pettersson, and Daniel Sundmark. A framework for comparing efficiency, effectiveness and applicability of software testing techniques. Testing: Academic & Industrial Conference - Practice And Research Techniques (TAIC PART), 2006.
13. Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering, 26(8):797–814, December 2000.
14. Mary Jean Harrold, A. Jefferson Offutt, and Kanupriya Tewary. An approach to fault modeling and fault seeding using the program dependence graph. Journal of Systems and Software, March 1997.
15. Jane Huffman Hayes, Inies C. M. Raphael, Vinod Kumar Surisetty, and Anneliese Amschler Andrews. Fault links: Exploring the relationship between module and fault types. European Dependable Computing Conference (EDCC), April 2005.
16. Mats P. E. Heimdahl and Devaraj George. Test-suite reduction for model based testing: Effects on test quality and implications for testing. IEEE international conference on Automated software engineering (ASE), 2004.
17. Kennet Henningsson and Claes Wohlin. Assuring fault classification agreement - an empirical evaluation. In IEEE International Symposium on Empirical Software Engineering (ISESE), pages 95–104, USA, August 2004. IEEE Computer Society.

18. Jim Heumann. Generating test cases from use cases. Technical report, Rational Software, June 2001.
19. IEEE. 1044-1993. standard classification for software anomalies. Transaction of Software Engineering, 1993.
20. Cem Kaner. An introduction to scenario testing. Technical report, Florida Tech, University, June 2003.
21. Srikanth Karre, Sebastian Elbaum, Gregg Rothermel, and Marc Fisher II. Leveraging user session data to support web application testing. IEEE Transactions on Software Engineering, March 2005.
22. C-H Liu, D. C. Kung, P. Hsia, and C-T Hsu. An object-based data flow testing approach for web applications. International Journal of Software Engineering and Knowledge Engineering, 11(2):157–179, April 2001.
23. Nik Looker, Malcolm Munro, and Jie Xu. Simulating errors in web services. Journal of Simulation: Systems, Science & Technologny, 2005.
24. Daniel L. Lough. A Taxonomy of Computer Attacks with Applications to Wireless Network. PhD Thesis, Virginia, USA, 2001.
25. Giuseppe Antonio Di Lucca, Massimiliano Di Penta, Giulio Antoniol, and G. Casazza. An approach for reverse engineering of web-based applications. In 8th Working Conference on Reverse Engineering (WCRE), USA, October 2001.
26. Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava : An automated class mutation system. Journal of Software Testing, Verification and Reliability, 15(2):97–133, June 2005.
27. Atchara Mahaweerawat, Peraphon Sophatsathit, Chidchanok Lursinsap, and Petr Musilek. Masp - an enhanced model of fault type identification in object-oriented software engineering. Journal of Advanced Computational Intelligence and Intelligent Informatics, 10(3), December 2006.
28. Alessandro Marchetto, Paolo Tonella, and Filippo Ricca. Empirical validation of a web fault taxonomy and its usage for fault seeding. In IEEE International Symposium on Web Site Evolution (WSE), Paris, France, October 2007. IEEE Computer Society.
29. Leonardo Mariani. A fault taxonomy for component-based software. International Workshop on Test and Analysis of Component Based Systems (TACOS), April 2003.
30. Rational Software Corporation. Unified Modeling Language, Version 1.0. 1997.
31. F. Ricca and P. Tonella. Analysis and testing of Web applications. In Proc. of ICSE 2001, International Conference on Software Engineering, Toronto, Ontario, Canada, May 12-19, pages 25–34, 2001.
32. Filippo Ricca and Paolo Tonella. Web testing: a roadmap for the empirical research. IEEE International Symposium on Web Site Evolution (WSE), 2005.
33. Johannes Ryser and Martin Glinz. A scenario-based approach to validating and testing software systems using statecharts. In International Conference on Software and Systems Engineering and their Applications (ICSSEA), Paris, France., April 1999.
34. Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for web applications. IEEE international conference on Automated software engineering (ASE), 2005.
35. Stephan Weißleder Stefan Bruning and Miroslaw Malek. A fault taxonomy for service-oriented architecture. In IEEE High Assurance Systems Engineering Symposium (HASE). IEEE Computer Society, November 2007.
36. Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing. Working paper series. University of Waikato, April 2006.
37. Giri Vijayaraghavan and Cen Kramer. Bug taxonomies: Use them to generate better test. Software Testing Analysis and Review (STAR EAST), May 2003.
38. Jeffrey Voas, Gary McGraw, Lora Kassab, and Larry Voas. A crystal ball for software liability. IEEE Computer, 1997.
39. C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers, 2000.
40. Peter Zielczynski. Traceability from use cases to test cases. Technical report, Rational Software, Febraury 2006.