An Investigation of Java Faults Operators Derived from a Field Data Study on Java Software Faults

Tania Basso[1], Regina L. O. Moraes[2], Bruno P. Sanches[2], Mario Jino[1]

[1]Faculdade de Engenharia Elétrica e de Computação
Universidade Estadual de Campinas (UNICAMP) – Campinas, SP, Brazil

[2]Faculdade de Tecnologia
Universidade Estadual de Campinas (UNICAMP) – Campinas, SP, Brazil

{taniabasso,brunopsanches}@gmail.com, regina@ceset.unicamp.br, jino@dca.fee.unicamp.br

Abstract. *The knowledge of real software faults representativeness is important to allow the emulation of software faults in a more accurate way through software fault injection techniques. This paper presents a field data study to analyze the representativeness of Java software faults, including security faults. The faults are classified according to a previous field study of C faults representativeness and new types of faults are identified due to the specific characteristics of the Java language structure. Results are compared and show that the mistakes most commonly made by programmers follow a pattern, independently of the programming language.*

1. Introduction

Modern society is increasingly dependent on computer services and, consequently, on the software executed to provide these services. According to Avizienis *et al* [2004], dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable. Assuring that software systems are dependable is important especially in critical systems, where faults can cause major damages or loss of human lives [Gage and McCormick 2004],[Pfleeger 2000].

The current scenario of software production requires more complex elements involved in software production, frequent changes and cost constraints within a short limit of development time. These factors can lead to the insertion of faults into the software. Software faults are recognized as the main cause of computational defects [Kalyanakrishnam *et al* 1999], [Lee and Iyer 1995], [Sullivan and Chillarege 1992] and one of the main challenges in this context is to achieve dependability in these systems. Another demand on dependable systems arises from ubiquitous computation. They need to be universally accessed by users around the world (e.g., online trading and home banking). Most of these systems are web applications. In this scenario, the users may find it difficult to report the failures and the unsolved problems can lead them to put in check the organization credibility or allow hackers to attack the systems, which may have a highly negative impact on users. Although other potential causes for vulnerability do exist, the root cause of most security attacks are vulnerabilities created by software faults [Fonseca and Vieira 2008].

One way to ensure software dependability is to apply procedures for fault removal and fault prediction during the software validation process. These procedures enable, respectively, the reduction of the number of faults or the fault severity and the evaluation of the consequences of faults remaining in software products. Software fault injection techniques (i.e., deliberate faults insertion) have been very useful to help the software validation process [Durães and Madeira 2006]. Besides the well known usefulness of the technique for this purpose, the results are even more important if the fault emulation is done in a realistic way, which implies the knowledge of the kind of faults found and their distribution in the operational environment. This means that it is necessary to identify the fault representativeness.

This paper presents a field data study on Java systems in production phase (i.e., systems that have gone through several releases), aiming to identify Java specific fault representativeness. The Java programming language was selected for this field data study because it is widely used in current software applications, including web-based systems. The representativeness of a subset of security faults was also analyzed. 574 faults were analyzed and 67 of them are recognized as security faults. The results are compared with those from a previous field data study [Durães and Madeira 2006].

The proposal of this paper is to evaluate if the fault representativeness of Java applications (based on object-oriented paradigm) follows a pattern similar to the software fault representativeness of C applications (based on structural paradigm); it is an important step for the development of a fault injection tool for Java applications.

Some injection tools have been developed [Barcelos *et al* 1999], [Carreira *et al* 1995], [Martins *et al* 2002], [Neves *et al* 2006], and, however, there is no available injection tool to inject specific java software faults, it means, to change Java code structure. Evaluating fault representativeness of Java applications would enable the emulation of software faults in an accurate and realistic way, leading to useful results in the injection Java faults and, consequently, in the validation process.

In summary, we present a software fault classification based on a set of faults found in Java applications and propose an extension to the classification presented by Durães and Madeira [2006], specific to the object-oriented programming paradigm and the Java language.

The structure of the paper is as follows: Section 2 presents background on software faults; Section 3 describes the approach for analyzing and classifying the software faults; Section 4 contains the comparison with the other field study and presents new types of faults due to the Java language characteristics; Section 5 presents our conclusions and future work.

2. Software faults

Software faults are caused by mistakes made by software product programmers and remain in the program source code. The complete elimination of software faults is a difficult or even impossible goal to be achieved [Lyu 1996], [Musa 1996]. In many cases, waiting for the activation of these faults through normal use is not practicable because it rarely happens (otherwise, the faults would be identified and eliminated by

software tests). Fault injection techniques are one option for activation of these faults in order to evaluate the software behavior during the software validation process.

Knowledge of the software fault representativeness is essential for realistic fault injection and to ensure that results obtained during the experiments are very close to those of an operational environment.

To establish real software faults representativeness it is necessary to classify the most frequent software faults, to understand their origins and the way human mistakes occur in the programming task.

A well known classification of software faults is ODC (Orthogonal Defect Classification) [Chillarege 1995], [Chillarege *et al* 1992]. Durães and Madeira [Durães and Madeira 2006] extended this classification and refined their representation to a point that they can be emulated. The generic technique created by [Durães and Madeira 2006] is the G-SWFIT (Generic Software Fault Injection Technique). This technique makes possible the emulation of the most frequently found software faults. It consists of modifying the compiled binary code of software modules by introducing specific changes which correspond to the code that would be generated by the compiler if the intended software fault were in the high level source code. The emulation is done by means of an emulation operator's library. Each operator in this library consists of two binary code instructions. The first one represents a pattern corresponding to a specific fault and the second one represents the necessary changes for the adequate injection of the fault [Durães and Madeira 2006]. Although this technique is generic, the study was developed using the C language and additional investigation is necessary to extend it to other programming languages.

As software faults are strongly dependent on the paradigm and the programming language structure of the application, not all types of faults found in the present field study are represented in Durães and Madeira study [Durães and Madeira 2006]. Thus, it is necessary to define a new classification and a distribution including the new types of software faults which represent characteristics specific of the Java language.

2.1. Java security software faults

Nowadays, due to the widespread use of web applications, the security vulnerabilities in these systems are being more intensely explored by hackers. The work proposed by Fonseca and Vieira [2008] aims to improve this scenario by investigating security faults in applications developed using the PHP (Hypertext Preprocessor) programming language. They want to achieve an understanding of the relationship between certain software defects and security vulnerabilities.

The vulnerabilities analyzed were Cross Site Scripting (XSS) and SQL injection (see [Fonseca and Vieira 2008] for more details). To understand which code is responsible for the security problems the study was based on vulnerabilities correction patches. Comparison of these patches makes it possible to identify and classify real software faults that lead to security vulnerabilities. The authors also define rules to make the patches analysis coherent and to avoid mistakes during fault classification.

The methodology and patches analysis rules proposed by Fonseca and Vieira [2008] are used in our field data study as a basis for security faults identification.

3. Application analysis and fault identification

The field data study reported in this paper aims to investigate software faults in Java applications in order to define faultloads. To determine these faultloads, Java applications source codes are analyzed; an essential condition for the study to be conducted is the availability of the source code of these applications. We also need previous releases of these applications to compare them and to analyze the corrections made and the programming structures which were used to perform these corrections. The analysis of these structures will indicate the types of faults and, consequently, the types of operators that can be used to emulate these faults making use of the G-SWFIT technique.

To identify faults that lead to security vulnerabilities additional information is necessary to link the vulnerability correction with the specific application release (e.g., information on which files or source code fragments were modified exclusively to correct a particular vulnerability).

In accordance with the necessary conditions above and considering their popularity, six open source applications were selected for the field study: Azureus (Vuze) [Vuze 2009], FreeMind [FreeMind 2009], JEdit [JEdit 2009], Phex [Phex 2009], Struts [Struts 2009] and Tomcat [Tomcat 2009].

Azureus (Vuze) and Phex are client applications for sharing files; they use protocols that, briefly, allow content distribution through the web, optimizing the bandwidth consumption under traffic limitations. FreeMind helps on storing and organizing ideas, contributing, for example, to keep personal knowledge bases, recording information of meetings, brainstorms, presentations, planning, and so on. JEdit is a text editor for software systems programmers. Struts is a framework used to create Java web applications. And, finally, Tomcat is a very popular web application server.

To analyze the differences between the selected applications, a diff tool (i.e., a tool that compares two codes and shows their differences) is applied to the different versions of the same application; the changes introduced in the source code are highlighted. It makes possible to visualize the correction, and, consequently, to identify the software fault. The diff tool used is WinMerge 2.10.0.0 [WinMerge 2009]. A total of 14 versions of applications were analyzed. Table 1 shows the analyzed versions and the faults found through ODC type classification for each selected application.

The applications were selected considering the size (lines of code) and number of downloads. Azureus and Tomcat are in a higher maturity level; they are more popular, have more lines of code and more released versions available. So, they have more number of reported bugs and it implies in more number of faults.

The faults are identified through the comparison between bugs correction code of the current release and the source code of the immediately preceding released version. The bugs' correction codes are identified trough changelog files and it is assumed that different code fragments do correct faults.

Table 1. Applications and analyzed versions

Application	Analyzed versions	ODC type					Total Faults	Size (LOC)
		ALG	ASG	CHK	FUN	INT		
Azureus (Vuze)	3.0.5.2 and 3.1.1.0	46	19	14	36	10	125	576958
FreeMind	0.7.1 and 0.8.0	42	10	2	26	10	90	19397
JEdit	4.2pre15 and 4.3pre16	26	10	8	18	9	71	173798
Phex	3.2.6.106 and 3.4.0.110	12	3	1	2	2	20	159206
Struts	1.2.7, 1.2.8 and 1.2.9	48	18	9	4	20	99	158718
Tomcat	6.0.14, 6.0.16 and 6.0.18	98	21	23	4	23	169	279298

Declarations of new structures (i.e., new classes, new variable/object declaration including the “get” and “set” methods for them) were not identified as faults, because a simple structure declaration does not imply a fault correction. If they are associated with a fault correction, this will be seen in the code, and the fault would be properly classified.

For each fault, we attempt to understand the language construction and the program context around it. Then, we try to correlate the fault with a fault type observed in the previous study [Durães and Madeira 2006]. When it is not possible to classify a fault according to the previous study [Durães and Madeira 2006], a new fault type is created, in accordance with the methodology proposed [Durães and Madeira 2006]. Changes in configuration files (i.e., files with “xml” and “properties” extensions) were not considered.

Under the decisions and assumptions above, we were able to identify and classify all the faults found in this field data study. The final classification should be seen as a complementary extension to the previous classification [Durães and Madeira 2006] and can be used to define specific fault emulation operators to inject faults in Java applications.

3.1. Security vulnerabilities patches analysis

Due to the importance of knowing the faults that lead to security vulnerabilities, a subset of security faults was identified. To identify this fault subset, additional decisions were made as explained in the next paragraph. These decisions are based on the methodology proposed by Fonseca and Vieira [2008].

The Struts and Tomcat applications were selected to identify security faults because they are very popular and have many security vulnerabilities reported. For the other applications there was not security vulnerabilities reported, and, consequently, security faults were not analyzed for them. The vulnerability corrections are available in correction patches. Normally, these patches also present many corrections, including faults not linked to security. Thus, it is necessary to identify files and pieces of source code that were modified exclusively to correct a particular vulnerability. This detailed information was found through an internal control number called “number revision” in

the patches of the applications under analysis. From this number it was possible to identify the files that were modified exclusively to correct the particular vulnerability. It is understood that all the modified source code fragments in these files are connected to the present security faults and it does correct the corresponding security vulnerabilities.

4. Results and discussions

A set of 574 Java faults were analyzed. Table 2 presents the most frequent faults found considering the representative set of Java application shown in Table 1. These most frequent faults represent 78.4% of the total Java faults found.

Table 2. Most frequent Java faults and corresponding ODC fault types

Fault nature	Specific fault types	Faults	ODC type
Missing construct	Missing functionality (MFCT)	87	Function
	Missing If construct plus statements (MIFS)	85	Algorithm
	Missing function call (MFC)	60	Algorithm
	Missing if construct plus else plus statements around statements (MIEA)	20	Algorithm
	Missing if construct around statements (MIA)	17	Checking
	Missing parameter in function call (MPFC)	11	Interface
Wrong construct	Wrong function called with same parameters (WFCS)	30	Algorithm
	Wrong value used in variable initialization (WVIV)	23	Assignment
	Wrong data types or conversion used (WSUT)	19	Assignment
	Wrong value used in parameter of function call (WPFL)	18	Interface
	Wrong variable used in parameter of function call (WPFV)	17	Interface
	Wrong logical expression used as branch condition (WLEC)	15	Checking
	Wrong algorithm - small sparse modifications (WALD)	13	Algorithm
	Wrong return value (WRV)	12	Interface
	Wrong algorithm - code was misplaced(WALR)	11	Algorithm
Extraneous construct	Extraneous function call (EFC)	12	Algorithm
Total faults		450	

The most frequent fault types found in our field study are *Missing Functionality (MFCT)* and *Missing if construct plus statements (MIFS)*. Both types present a percentage much higher than that of other fault types, accounting for 30% of the total of analyzed faults. The third most frequent is the *Missing function call (MFC)*, representing 10.5% of the total faults. This rank is similar to that of a previous field study [Durães and Madeira 2006], where: MIFS is the most frequent type representing 10.7% of the total of faults found in the C language; MFC is the third one, representing 6.7%; and MFCT represents 3.2%. These results indicate that, despite some different frequencies, the mistakes most commonly made by programmers are common to both programming languages (C and Java).

4.1. General Java Faults

From the most frequent fault types in Table 2, seven are also found as the most frequent fault types in Durães and Madeira´s study [Durães and Madeira 2006]. They are MIFS, MFC, MFCT, MIA, WSUT, WPFV and WLEC. Table 3 presents a comparison between the distribution of this field study and the distributions of Durães and Madeira´s field study [Durães and Madeira 2006].

Table 3. General fault distribution across ODC defect-type distribution and comparison with Durães and Madeira´s field study [Durães and Madeira 2006].

ODC defect-type	General Java Faults (GJF)	GJF ODC defect-type distribution (%)	C faults distribution [Durães and Madeira 2006] (%)
Algorithm	272	47.4	40.1
Assignment	81	14.1	21.4
Checking	57	9.9	25
Function	90	15.7	6.1
Interface	74	12.9	7.3
total	574		

Results show that general Java faults distribution is similar from the tendency found in the study by Durães and Madeira [2006]. We observe that Assignment, Interface and Function faults have roughly the same frequency and Checking faults are the least frequent ones. There was a considerable increase of Function faults when compared to the study by Durães and Madeira [2006]. It is our belief that these differences are due to the specific characteristics of the programming language and paradigm; Java programming code tends to be more modularized and encapsulated, and requires more methods constructions to make the corrections. There was also a considerable increase of Interface faults. As Interface faults are faults that produce errors in the interaction among components, modules, device drivers, call statements or parameter lists, this increase may be due to the stronger importance assigned to the interfaces in Java applications and to the security vulnerabilities corrections, since attacks can be done especially through variables and values inputs.

4.2. Security Java Faults

From 574 investigated faults, 67 are found to lead to software vulnerabilities. Table 4 shows the most frequent faults observed in security vulnerability correcting patches. These faults represent 73.2% of the total security faults observed and follow a pattern similar to that of the general Java faults, where the fault types MFC and MIFS are the most frequent ones. Table 5 presents the security Java faults distribution and a comparison with the distribution of Fonseca and Vieira´s field study [Fonseca and Vieira 2008].

Table 4. Most frequent security Java faults and corresponding ODC fault types

Fault Nature	Specific fault types	Faults	ODC Type
Missing construct	Missing function call (MFC)	14	Algorithm
	Missing If construct plus statements (MIFS)	11	Algorithm
Wrong construct	Wrong variable used in parameter of function call (WPFV)	8	Interface
	Wrong value used in variable initialization (WVIV)	7	Assignment
	Wrong value used in parameter of function call (WPFL)	5	Interface
	Wrong algorithm - code was misplaced(WALR)	4	Algorithm
Total faults		49	

Table 5. Security fault distribution across ODC defect-type distribution and comparison with Fonseca and Vieira´s field study [Fonseca and Vieira 2008].

ODC defect-type	Security Java Faults (SJF)	SJF ODC defect-type distribution (%)	PHP faults distribution (%) [Fonseca and Vieira 2008]
Algorithm	36	53.7	86.01
Assignment	11	16.4	6.04
Checking	5	7.5	2.36
Function	1	1.5	0
Interface	14	20.9	5.6
total	67		

The security fault distribution follows a distribution pattern similar to that of general Java faults, except for Function faults. However, significant differences arise when we compare our security fault distribution with that presented by Fonseca and Vieira [2008]. Again, we believe these differences are due to the programming language specific characteristics (PHP faults were investigated in that study).

Table 6 shows, for each fault type, the security fault percentage relative to general Java faults. Interface faults have the highest percentage: 14 of the 74 Interface faults make the software vulnerable, meaning that 18.9% of the Interface faults are security faults.

Table 6. Security fault percentage relative to general Java faults.

ODC defect-type	GJF	SJF	SJF / GJF (%)
Algorithm	272	36	13.2
Assignment	81	11	13.6
Checking	57	5	8.8
Function	90	1	1.1
Interface	74	14	18.9
total	574	67	11.7

4.3. New fault types

In our study new fault types were found due to the Java language specific characteristics and the object-oriented paradigm, representing 7.1% of the total Java faults. As they arise due to the Java language specificity, they must be considered in the creation of new operators and, consequently, in the development of a realistic Java injection fault

tool. Also, some of these new fault types are security faults. Table 7 presents a new Java fault types summary with the corresponding frequencies and the ODC type. A brief description for each new fault type follows.

Table 7. New Java fault types

Fault nature	Fault specific type	Faults	ODC type
Missing contruct	Missing interface implementation (MII)	6	Interface
	Missing throw statement (MTS)	4	Algorithm
	Missing try/catch/finally statement around statements (MTCFAS)	4	Algorithm
	Missing try/catch/finally statement plus statements (MTCFS)	4	Algorithm
	Missing Synchronized statement around statements (MSAS)	2	Algorithm
	Missing throws specification in method (MTSM)	2	Algorithm
	Missing extended class (MEC)	1	Interface
	Missing Synchronized statement (MSS)	1	Algorithm
Wrong construct	Wrong extended class (WEC)	9	Interface
	Wrong parameter passed to an object constructor (WPOC)	5	Assignment
Extraneous construct	Extraneous try/catch/finally statement (ETCFS)	2	Algorithm
	Extraneous Synchronized statement (ESS)	1	Algorithm
Total faults		41	

Interface faults

Missing interface implementation (MII): the omission of the interface specification that will be implemented by the class or even the *implements* clause.

Missing Extended class (MEC): the omission of the super class specification or even the *extends* clause.

Wrong extended class (WEC): the super class specified during the class declaration is wrong.

Assignment faults

Wrong parameter passed to an object constructor (WPOC): a wrong parameter passed to a constructor method when an object is created by the *new* clause.

Algorithm faults

Missing synchronized statement (MSS): the omission of a code fragment with synchronize instruction.

Missing synchronized statement around statements (MSAS): the omission of a synchronize instruction in the existing code fragments.

Missing throws specification in method (MTSM): the omission of the exceptions specification, which will be thrown by particular methods.

Missing throw statement (MTS): the omission of exception handling.

Missing try/catch/finally statement plus statements (MTCFS): the omission of a code fragment with the corresponding exception handling

Missing try/catch/finally statement around statements (MTCFAS): the omission of exception handling in existing code fragments

Extraneous synchronized statement (ESS): a synchronized instruction used mistakenly.

Extraneous try/catch/finally statement (ETCFS): exception handling used in a wrong way.

Fault types relative to the try/catch/finally statements were classified in a generic way and some characteristics of the try/catch/finally block syntax must be explained. It is known that every try block requires at least one catch block or one finally block. For each try block several catch blocks are permitted. The *missing construct* faults relative to the try/catch/finally block can be the omission of one or more catch blocks corresponding to an existent try block; it can also be the omission of one finally block. The extraneous construct faults can be the wrong use of catch and finally blocks even separately.

To classify these new fault types the nomenclature proposed by Durães and Madeira [2006] was followed. Some structures are related to well known Java specific faults presented by Reilly [2009] (Top Ten Errors), but the majority of the new fault types do not represent that kind of mistakes. This can be due to the maturity level of the applications, where more stable versions do not present those common errors.

Most representative faults

The most representative new fault types are WEC, MII and WPOC, accounting for, respectively, 22%, 14.6% and 12.2% of the total new fault types. Their structures are exemplified in Table 8, where the missing or wrong code is marked with gray background.

Table 8. Fault structure of the most representative new fault types

MII	…code before public class Class1 implements Interface1, Interface2 … code after
WEC	…code before public class Class1 extends SuperClass1 … code after
WPOC	…code before Class1 object1 = new Class1 (parameter) …code after

In the object-oriented paradigm an interface establishes a kind of contract which is fulfilled by a class that implements this interface. When a class implements an interface, it is ensured that all the functionalities specified by the interface will be offered by the class. Interfaces define only method and constants variables definitions and represent higher level abstraction than the classes do, i.e., it is possible to design

explicitly all the interfaces of an application before deciding on the more adequate implementation form. A class can implement several interfaces.

The new fault type MII occurs when the programmer misses the *implements* clause or misses the specification of the implementation of some interface. This fact can lead to missing some functionality necessary to the correct behavior of the application. This functionality will be included when the correction of this fault type is done. It also contributes to the increase of the MFCT fault type in this field study, because some interfaces may have several methods definitions that are called in some part of the code.

The WEC fault type occurs due to the concept of inheritance provided by the object-oriented paradigm. This concept permits characteristics common to various classes to be concentrated in only one class (superclass). Extending a wrong class means inheriting wrong characteristics and wrong functionalities to be used by the subclass.

The WPOC fault type is related to the constructor method. This method determines which actions must be executed when an object is created. In the Java language, the constructor is defined as a method whose name is the class name and that does not have a return type, neither *void*. The constructor is invoked only at the moment that the object is created by the *new* clause. The constructor can receive arguments and the same constructor can be defined with different numbers of arguments, using the overloading mechanism.

All the classes have at least one predefined constructor. If no one constructor is defined explicitly by the programmer, a default constructor, which does not receive arguments, is included in the class by the Java compiler. Thus, for the WPOC fault type, if the called constructor has an argument, it means that the programmer wants to write a piece of code to be executed when an object is created (he does not want to use the default constructor). Passing a wrong parameter as an argument means that the code to be executed in this constructor can create an object with wrong initialization values, for example.

Three security faults identified among the new fault types were found. They correspond respectively to the MTSM, MTCFAS and MTCFS fault types. These three fault types are related to the exceptions concept. The Java language offers mechanisms to detect and to treat exceptions.

The existence of an exception means that some exceptional condition has occurred during the code execution. Thus, exceptions are associated with faults that were not identified during the compilation. Catching the exceptional situation is necessary to treat them. For each exception that can occur during the code execution, an exception handler must be specified. The Java compiler verifies and enforces that exceptions have an associated treatment block.

The MSTM, MTCFAS, MTCFS fault types occur due to missing treatment blocks and missing statements related to them. It means that the exceptions can propagate and cause errors in the applications.

5. Conclusions

The field data study reported in this paper analyzed 574 Java software faults. Among them, 67 lead to security vulnerabilities. The results show that the fault types found correspond largely to the fault types found in the field study using the C programming language. The fault representativeness also follows this tendency, meaning that the most frequent Java faults correspond to the most frequent C faults; this shows that mistakes made by computer systems programmers follow a similar pattern independently of these programming languages.

Concerning fault distribution, the Java field study shows a considerable increase of Function and Interface fault types; this increase can be due to Java language characteristics and object-oriented paradigm, security vulnerabilities corrections and a stronger importance assigned to the interfaces in the Java applications analyzed.

New fault types were identified according to Java language specific characteristics and the object-oriented paradigm. These new fault types are important for the validation process of the Java applications, mainly because some of them are security faults that must be considered.

As future work the authors' aim is to develop a fault injection tool to automate the software fault injection process in Java applications. This tool will use the fault representativeness found in the present field study to create faultloads to be used during fault injection experiments, to obtain more realistic and useful results in the software validation process.

References

Avizienis, A., Laprie, J.C., Randell, B. and Landwehr, C. (2004) "Basic concepts and taxonomy of dependable and secure computing". IEEE Transactions on Dependable and Secure Computing, Volume 1, Issue 1, p. 11 – 33.

Azureus (Vuze). (2009). Available in www.azureus.sourceforge.net. Last access on February.

Barcelos, P. P., Leite, F., Silva, T. W. (1999) "Implementação de um Injetor de Falhas de Comunicação". SCTF '99 – VIII Simpósio de Computação Tolerante a Falhas. Campinas, Brazil, p. 225-239.

Carreira, J., Madeira, H. and Silva, J. G. (1995) "Xception: Software Fault Injection and Monitoring in Processor Functional Units". 5º IFIP International Working Conference on Dependable Computing for Critical Applications. Urbana-Champaign, EUA, p. 135-149.

Chillarege, R. (1995) "Orthogonal Defect Classification", Chapter 9 of "Handbook of Software Reliability Engineering", Michael R. Lyu Ed., IEEE Computer Society Press, McGraw-Hill.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D., Ray, B. and Wong, M. (1992) "Orthogonal Defect Classification – A Concept for In-Process Measurement". IEEE Transactions on Software Engineering, vol. 18, n. 11, p. 943-956.

Durães, J. and Madeira, H. (2006) "Emulation of Software Faults: A Field Data Study and Practical Approach". IEEE Trans. on Software Engineering, vol. 32, n. 11, p. 849-867.

Fonseca, J. and Vieira, M. (2008) “Mapping software faults with web security vulnerability”. IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), Anchorage, USA, p. 257-266

FreeMind. (2009). Available in www.freemind.sourceforge.net. Last access on February.

Gage, D. and McCormick, J. (2004) "Why Software Quality Matters", Baseline Magazine, p. 32-59.

JEdit. (2009). Available in www.jedit.org. Last access on February.

Kalyanakrishnam, M., Kalbarczyk, Z. and Iyer, R. (1999) “Failure Data Analysis of a LAN of Windows NT Based Computers”, Symposium on Reliable Distributed Database Systems, SRDS-18, Switzerland, p. 178-187.

Lee, I. and Iyer, R. K. (1995) “Software Dependability in the Tandem GUARDIAN System”, IEEE Trans. on Software Engineering, vol. 21, no. 5, p. 455-467

Lyu, M. (1996) “Handbook of Software Reliability Engineering”, IEEE Computer Society Press, MCGraw-Hill.

Martins, E., Rubira, C. and Leme, N. (2002) “Jaca: A reflective fault injection tool based on patterns”. Proc of the 2002 International Conference on Dependable Systems & Networks, Washington D.C. USA, p. 483-487.

Musa, J. (1996) “Software Reliability Engineering”, McGraw-Hill.

Neves, N., Antunes, J., Correia, M., Veríssimo, P., Neves, R. “Using Attack Injection to Discover New Vulnerabilities”. Proc. of the International Conference on Dependable Systems and Networks (DSN), 2006, p. 457-466.

Pfleeger, S. (2000) "Risky Business: what we have yet to learn about risk management", The Journal of Systems and Software, 53, p. 265-273.

Phex. (2009). Available in www.phex.org. Last access on February.

Reilly, D. (2009). Top Ten Errors Java Programmers Make. Available in www.javacoffeebreak.com/articles/toptenerrors.html. Last access on May.

Struts.(2009). Available in www.struts.apache.org. Last access on February.

Sullivan, M. and Chillarege, R. (1992) “Comparison of Software Defects in DataDatabase Management Systems and Operating Systems”, Proc. of the 22nd IEEE Fault Tolerant Computing Symposium, FTCS-22, p. 475-484.

Tomcat. (2009). Available in www.tomcat.apache.org. Last access on February.

WinMerge. (2009). Avaliable in www.winmerge.org. Last access on February.