Research Issues in Software Fault Categorization

Jan Ploski, Matthias Rohr, Peter Schwenkenberg and Wilhelm Hasselbring
Software Engineering Group, TrustSoft
University of Oldenburg, Germany
jan.ploski@offis.de, {rohr|peter.schwenkenberg|hasselbring}@informatik.uni-oldenburg.de

Abstract

Software faults are a major threat for the dependability of software systems. When we intend to study the impact of software faults on software behavior, examine the quality of fault tolerance mechanisms, or evaluate diagnostic techniques, the issue of distinguishing fault categories and their frequency distribution arises immediately. This article surveys the literature that provides quantitative data on categories of software faults and discusses the applicability of these software fault category distributions to fault injection case studies.

Keywords: Software Faults, Bugs, Software Fault Categorization, Software Reliability, Injection of Software Faults

1 Introduction

Many failures of computing systems are caused by faults in software. The term *software fault* usually refers to design faults introduced into software during any phase of software development, such as specification, programming, or installation [LABK95, p. 48], [LS00]. As the amount of software in critical systems increases, so should the attention given to software faults in dependability engineering and evaluation.

Most software dependability approaches, such as software reliability estimation, or software fault injection, do not distinguish categories of software faults. Arguably, a simple fault density metric is appropriate for software cost estimation and as a decision support tool for releasing non-critical software. However, when we intend to study the impact of software faults on software behavior, examine the quality of fault tolerance mechanisms, or evaluate diagnostic techniques, the issue of distinguishing fault categories and their frequency distributions arises immediately. Intuitively, the conclusiveness of software dependability analysis suffers both from including non-realistic faults and excluding realistic faults. The problem is similar in nature to test case selection in software testing.

The same problem can arise in software fault injection. Fault injection emulates faults in a system to evaluate fault tolerance properties and to estimate dependability. In the past, the focus of fault injection lied on emulating component failures or physical faults in hardware or process failures in distributed systems. Less common is the injection of software faults, that is, design faults, into the application layer of a computing system. However, software fault injection provides a valuable tool to evaluate fault tolerance or failure diagnosis capabilities of research approaches that address software faults or to evaluate real systems. A challenge in the design of such experiments is to decide on the software fault load. Software reliability prediction approaches allow to justify assumptions about fault density. However, one also has to decide on the frequencies of software fault categories in order to conduct a fault injection experiment.

These problems drive our research questions: Which similarities between software faults in different software development projects exist and what categorizations have been applied? What are the factors that determine software fault category distributions? What are the challenges in applying the result of empirical software fault studies in other software development projects? In this paper, we seek answers to the above questions by surveying the literature that provides quantitative data on categories of software faults. After presenting the survey's results, we turn back to discussing their applicability to software fault injection experiments.

2 Past studies of software fault categories and distributions

In this section we summarize and compare previous work on software fault categorization. Other terms used in this context include bug taxonomies, software defect classification and root causal defect analysis. Our goal is to track down past efforts of categorizing software faults and, as far as such a meta-analysis permits, to address their respective strengths and weaknesses.

Rather than just refer readers to numerous literature positions, we contribute comprehensive paper overviews. Our selection of papers was driven by a desire to illustrate the breadth of prior works concerned with software fault categorization. They are characterized by a wide range of underlying motivations: from understanding root causes of programming mistakes, through developing fault tolerance measures, facilitating testing, monitoring the software development process, and improving it through automation.

To organize presentation, our analysis focused on the following aspects of each examined fault categorization schema:

1. Purpose: are the author(s) explicit about the purpose of their categorization schema? If so, what benefits do they expect from applying their categorization?
2. Type of system from which fault data was collected:
   (a) number of projects from which data was collected to derive the schema
   (b) project size (e.g. number of developers, duration, size of examined code)

   (c) number of faults examined to develop the schema
   (d) application domain
3. Information sources used for building fault categories
4. Attributes of faults which influence categorization
5. Intended exclusiveness of the proposed fault categories
6. Possible ambiguities, arbitrariness of fault categories
7. Public availability of data used to derive the schema, in particular
   (a) Is full data from which the schema was derived available?
   (b) Is frequency data available for individual fault categories?
8. Is the categorization schema supplemented by recommended categorization procedures?
9. Number of fault categories proposed
10. Flat or hierarchical categorization

Table 1 at the end of this section provides a corresponding summary of the examined categorization schemas.

2.1 Knuth's errors of TEX

Knuth's paper about the evolution of the typesetting system TEX [Knu89] provides one of the early and often referenced fault categorization schemas, drawing from experiences of the lead programmer on a medium-sized software project collected during a period of 10 years. The categorization schema covers 867 modifications made to TEX in programming languages SAIL and Pascal. Knuth was himself the primary implementor of changes and, as far as we know, the sole user of the fault categorization schema during its inception.

The main stated purpose of the categorization schema is to aid the author's own understanding of the encountered faults and to provide a convenient framework for a discussion of lessons learned from the project. In particular, Knuth wishes to "shed light on the problem of writing large programs". He hopes that the readers learn from his mistakes, and that the information will be "useful somehow to people who study the debugging process". Knuth does not express an ambition to impose his fault categorization schema on other software projects, but applies it consistently throughout his own programming. An important, yet discouraging, conclusion of the paper is that keeping logs did not allow the author to reduce the number of software faults over time. This calls for caution when setting out to categorize software faults for educational purposes. The main benefits mentioned by Knuth are personal: an improvement of his own attitude to programming and reflections on the evolution of medium-sized software systems.

Knuth's categorization schema is used directly by a programmer and applied immediately upon encountering (and logging) a software fault. Knuth defines 9 non-nested faults classes, labelled with capital letters:

A algorithm awry

B blunder or botch

D data structure debacle

F forgotten function

L language liability

M mismatch between modules

S surprising scenario

T trivial typo

Six other classes are provided to cover enhancements. Knuth admits that the distinction between enhancements and bugs is more or less arbitrary. For example, performance optimizations are never considered faults in his schema. Furthermore, the fault categories are proposed "ad hoc", which visibly manifests itself in their non-exclusiveness and the different applied dimensions of classification. Implicitly considered fault attributes include, among others:

- time of fault introduction (e.g., A - during specification of an algorithm, B - during coding),
- fault location (e.g., D - data structure; faults are also assigned to application modules),
- manifestation in source code or lack thereof (errors of commission and omission, e.g. F - forgotten function),
- sort of information misinterpreted by the programmer (e.g., M - module interfaces, L - language rules),
- programmer's ability to avoid the fault at the time of its introduction (e.g., B - easily avoidable blunder, S - difficult-to-predict future scenario).

Although the author names exactly one category for each of the logged faults, it is easily conceivable that a fault belongs to multiple categories: a surprising scenario which required a change in algorithms could be described by both A and S, a forgotten statement which caused an invariant to be violated by F and D, and so on.

Knuth's categorization schema does not explicitly include fault frequencies, although they could be easily derived by counting. Unfortunately, given the overall arbitrariness of the schema, we do not see much value in such an exercise.

The full source code which served to develop the schema is available.

2.2 Beizer's bug taxonomy

Beizer and Vinter [BV01] provide a comprehensive categorization schema for software faults, which originally appeared as a book appendix in [Bei90]. The document does not explicitly state the presented fault categorization schema's purpose. However, given the primary author's background and original publication, it is reasonable to assume that the taxonomy is intended to aid the process of software testing.

According to Beizer, many different sources influenced the taxonomy. The data, collected up to 1989, originates from commercial projects developing systems software in defense, aerospace, and communication domains, written in several programming languages (Assembler, Fortran, Ada, COBOL, and C). In total, 982 bugs were classified to develop the schema.

Like Knuth, Beizer depends on programmers (or testers) to apply the proposed categorization. Another similarity is that the dimensions used to classify faults vary and are not stated explicitly. Some examples include:

- time of fault introduction (e.g., specification, implementation, test)
- effects of fault activation (e.g., undesired control flow, data corruption)
- location (e.g., the entity which is deemed incorrect and requires fixing)
- type of required corrective action (e.g., requirement wrong/undesirable/not needed/ ambiguous, data scope: "local should be global")

Unlike Knuth's, Beizer's schema is hierarchical, with 8 top-level categories and over 100 leaf categories. The numbering convention foresees up to 10 categories at each level, the last category often being the "Other" category. The idea is to organize fault categories according to their semantic distance. Unfortunately, the semantic relationships are not explained in sufficient detail to avoid confusion. Consider, for example, the top-level fault categories:

- Functional bugs: requirements and features (errors during specification of requirements)
- Functionality as implemented (errors in the interpretation of requirements)
- Structural bugs (mistakes during implementation concerning control flow and expression manipulation)
- Data bugs (bugs in definition, structure or use of data)
- Implementation (typographical bugs, violations of standards and conventions, errors in documentation)
- Integration (bugs having to do with interfaces between components)
- System and software architecture
- Test definition and execution

It is not obvious why the "Implementation" category is defined as above, and why bugs related to misinterpretation of requirements are separated at this level in hierarchy from structural or data bugs. Other examples of ambiguity concern the position of leaf nodes in the hierarchy: faults related to string manipulation are in a different major category (Structural bugs) than faults related to manipulation of other data (Data bugs). Even the distinction between control flow and data-related bugs is not trivial. Should we categorize protocol violations as "data-flow anomalies", "control state bugs", or maybe "component invocation" bugs (all leaf categories with different ancestors)?

Beizer provides frequency data for each category. Like in Knuth's case, the usefulness and interpretation possibilities of the quantitative data are not clear due to the inherent ambiguities in the fault categorization schema. Apart from the descriptions of fault categories, no procedure is provided to guide a person in choosing the appropriate category for a particular observed software fault.

2.3 Gray's classification of software faults in production software

Gray [Gra86] provides some insight into software fault classification. The paper's focus is not to develop a software fault categorization schema, but to report on reasons of failures and possible means of improving systems' availability and reliability. As a consequence, software faults are classified according to the ability of considered fault tolerance mechanisms to tolerate them at run time:

- transient faults (called Heisenbugs in the original paper) whose activations can be masked by restoring a consistent initial state and retry; in other words they cannot be simply reproduced by repeated execution
- non-transient faults (called Bohrbugs) whose activations cannot be masked by retry; these are the easily reproducible faults

Gray does not consider any other fault categories. He conjectures that "most production software faults are soft" (transient). His emphasis lies on the words "production software", that is, software which has been extensively tested and used for long periods of time. Gray's hypothesis has some important implications on developing fault categorization schemas:

- software fault categorization based on data from production systems might be extremely difficult, due to non-reproducible activations: the causes of most glitches cannot be determined
- the relative frequencies of software faults are likely to vary with software's maturity; therefore, out-of-context frequency data on fault occurrences is difficult to interpret

Gray's hypothesis is certainly supported by the data collected in his study. Only one of the 132 observed software faults was non-transient. The study covered a remarkable number of installed systems (2000) and operational time (10 million hours). The targets were fault-tolerant transaction processing applications executing on mainframes. Gray notes that while the absolute numbers of failures per time unit vary between conventional systems and fault-tolerant systems, the relative frequencies of root causes are similar. By focusing on "production software", the analysis excluded faults observed during systems' introduction.

In summary, Gray's study differs from the other ones by categorizing software faults according to the character of their run time activations rather than by examining their manifestation in the source code or software development process. The author's observations are useful in the area of refining fault tolerance mechanisms, but fairly pessimistic if our ambition is, like Knuth's, to learn from past mistakes or, like Beizer's, to guide software testing.

2.4 Orthogonal Defect Classification

Chillarege et al. [CBC+92] introduce the Orthogonal Defect Classification (ODC), a concept intended to enable feedback to developers during different phases of software development with the objective to reduce the number of software defects. The concept was developed and consistently applied to at least 50 software projects at IBM. ODC serves as a process measurement tool, which is based upon a growing set of field data, with intent to provide feedback for the development and verification process. ODC can also be used as a framework for the evolution of fault and error models [CC96].

The ODC schema, like others, relies on developers to gather data, but reduces the influence of personal opinion by concentrating on the required fix rather than trying to directly ascribe the fault to a subjectively perceived deficiency in the development process. The information about the fix is encoded in an attribute called *defect type* (also referred to as *fault type*). Additionally, an attribute called *defect trigger* is used to describe the circumstances in which the software fault's activation is observable.

The following set of (exclusive) defect types is provided:

- Function (missing or wrong functionality, may require a formal design change)
- Interface (addresses errors in communication between users, modules or device drivers)
- Checking (faulty or missing validation of data and values in the source code)
- Assignment (addresses faults in the source code such as faulty initialization)
- Timing/Serialization (errors that are corrected by improved management of shared and real-time resources)
- Build/Package/Merge (addresses problems due to mistakes in library systems, management of changes, and version control)
- Documentation (addresses publications and maintenance notes)
- Algorithm (addresses efficiency or correctness problems which can be fixed by re-implementation without the need for requesting a design change)

This categorization schema is supposed to be simple and obvious to programmers. A relatively small set of eight defect types aims at reducing the room for confusion and making the process of classifying defects less error-prone. The proposed fault categorization schema is predominantly intended to be applicable in practice rather than trying to be a universal "phenomenology" of the nature of software faults. Note, however, that ODC is not limited to software faults in the sense of implementation mistakes, as it also offers a category for functional faults whose fixes require changes of the formal design. The categories "Build/Package/Merge" and "Documentation" might be considered as marginal cases, as they do not include software faults according to our terminology. These categories exist because ODC is useful for characterizing and improving the software development *process*. As such, it naturally extends the definition of software faults from faults in the product to faults in the process.

The defect type can be additionally associated with the stage of development during which a bug has been fixed, so that the overall project progress can be measured by comparing distributions of defects in different stages of the process. In this respect, ODC is similar to, but more precise than traditional software reliability growth models.

The defect trigger attribute specifies a condition that allows a defect to surface. Its intention is to provide insight into the verification process. Exemplarily, discovering a significant discrepancy between defect trigger distributions during system test and field test hints at problems with the system test environment.

When a fault has been identified and classified with a defect type, the developer will also specify the circumstance, namely the defect trigger, under which the defect surfaced. Possible trigger types are workload, if the system encountered critical load, startup/restart, if e.g. the system was restarted due to a failure, or timing, if timing problems were observed. These kinds of triggers apply to system tests. Other verification activities, such as function tests, have different kinds of fault triggers.

The discussed classification method identifies different aspects of a fault. The fault type attributes are inherently simple, but fault data is supplemented by a defect trigger attribute, which describes the observed fault activation.

2.5 Eisenstadt's bug war stories

Eisenstadt [Eis97] explores issues related to the elimination of programming errors by examining bug reports from dif-

ferent developers. By taking the programmer's perspective, there are similarities with the approach of Knuth (see Section 2.1). The difference is that Knuth tries to categorize every bug of TEX, whereas Eisenstadt examines 59 self-reports provided by programmers. In the latter case each "bug war anecdote" concerns a single or at most a few related programming errors, resulting in an implicit preselection of data. The work aims at investigating the phenomenology of debugging across a large population of developers and also offers some suggestions on how debugging tools might be improved. Eisenstadt examines responses to his postings on the Usenet and bulletin board systems in which he asked participants for their most remarkable bug hunting stories. His responders are professional developers involved in various types of software projects.

The bug reports were solicited by asking three questions in order to elaborate different aspects of programming errors: "why difficult", "how found", and "root cause". Answers to the first question reveal a large discrepancy between a bug's root cause and observable symptoms as one of the main challenges. Furthermore, responders point out that some bugs "render debugging tools inapplicable". Answers to the second question suggest that data-gathering activities, such as placing print statements into the code and simulating execution "on paper", are the most important bug finding techniques.

The third question, targeting the root cause of a bug, is of more interest here, as it resulted in another categorization schema for software faults (occurrence counts are indicated in braces):

- mem – Memory clobbered or used up (13)
- vendor – Vendor's problem (hardware or software) (9)
- des.logic – Unanticipated case (faulty design logic) (7)
- init – Wrong initialization; wrong type; definition clash (6)
- lex – Lexical problem, bad parse, or ambiguous syntax (4)
- var – Wrong variable or operator (3)
- unsolved – Unknown and still unsolved to this day (3)
- lang – Semantics ambiguous or misunderstood (2)
- behav – End user's (or programmer's) subtle behavior (2)
- *??? (No information)* (2)

Eisenstadt relies on two different approaches to develop the above schema: a "deep plan analysis" approach by [Joh83] and [SSP85], and the phenomenological approach of Knuth. Both approaches maintain that only the programmer can resolve the true cause of a bug. Due to limited information about a bug and its context obtainable from the bug reports, Eisenstadt compiles an own schema in a bottom-up fashion, after inspection of the data and studying Knuth's schema. Memory problems, comparable with Knuth's "data structure debacle", which can result from overwriting reserved memory, turn out to be the most common type of software fault. The proposed schema partially simplifies Knuth's schema, as the category *des.logic*, which addresses faulty design logic (i.e. the programmer did not consider every case), encompasses both "algorithm awry" and "surprising scenario" from the former schema. Also, the category *lex*, which mainly addresses trivial, lexical problems, includes Knuth's *blunder* and *typo*, as the reports did not allow differentiation. The category *var*, for wrong variables or operators, catches errors that lack information whether the error is due to faulty design logic or a simple typographic mistake. Note that not only programming errors are included in Eisenstadt's survey. Wrong behaviour of end users and problems with third-party hardware or software are also considered.

Thus, the main objective of this schema is to allow grouping of software faults discovered and reported by other people. While not being the same as working with source code written by oneself, it focuses on the "lessons learned from the stories" as well. Furthermore, the paper tries to give advice on how future debugging tools should be designed.

Eisenstadt provides the full set of reports and his analysis upon request.

2.6 A grammar based fault classification schema by DeMillo and Mathur

DeMillo and Mathur [DM95] discuss classification schemas of software faults, present an own schema, and quantify faults in TEX. According to the authors, the purpose of fault classification schemas is to support debugging. In particular, they claim that the knowledge about frequently occurring types of faults can be used to choose an appropriate testing technique. The proposed schema supports automated classification based on syntactic comparison of program versions before and after fault removal.

The main criticism of other schemas is that they often suffer from ambiguity. In general, for any given error a chain of causes of an arbitrary length can be identified with some effort. In the presented approach, a fault is defined as a "manifestation of a programming mistake". More precisely, a fault can be identified by a critical string that is inserted or removed from source code. This purely syntactic approach is refined by considering sets of syntactic transformers that map an incorrect program's parse tree to a correct one. As different transformers may be applied to correct a fault, transformers are prioritized to avoid ambiguity. The resulting, formally developed grammar-based categorization schema leads to a hierarchy with four major categories:

- Spurious entity – requires the removal of its characteristic substring (2/0.5%)
- Missing entity – requires the insertion of a syntactic entity (155/53%)

	Primary author	Knuth	Beizer	Gray	Chillarege	Eisenstadt	DeMillo
1	**Purpose**	education	testing	fault tolerance	process improvement	debugging	testing
2a	**No. of projects**	1	multiple	2000	>50	multiple	1
2b	**Project size**	1-8 devs/10 yrs	large	large	large	?	1-8 devs/10 yrs
2c	**No. of faults**	867	982	132	?	51	415
2d	**Application domain**	typesetting	defense aerospace comm.	transaction processing	systems software	unspecified	typesetting
3	**Information source**	developers	developers testers	customers	developers testers	developers	syntactic analysis
4	**Sample fault attributes**	introduction time location in source manifestation type of mistake avoidability	required fix activation effects location introduction time	transiency	required fix activation trigger detection time	location in source type of mistake activation trigger	location in source required fix
5	**Categories exclusive?**	yes	yes	yes	no	yes	yes
6	**Ambiguity risk**	high	high	low	medium	high	low
7a	**Data available?**	Knuth	Beizer	Gray	Chillarege	Eisenstadt	DeMillo
7b	**Freq. data available?**	yes	yes	yes	no	yes	yes
8	**Recommended process?**	no	no	yes	yes	no	yes
9	**No. of leaf categories**	9	>100	2	8+18	10	>20
10	**Layout of categories**	flat	hierarchical	flat	flat	flat	hierarchical

Table 1: Overview of the examined software fault categorization schemas

- Misplaced entity – requires a change in its position (16/5.5%)
- Incorrect entity – for all other faults (118/41%)

The values in braces represent the count of occurrence in the analyzed TEX sources provided by Knuth (Section 2.1) and the corresponding relative fault frequency. The main categories are further subdivided according to the language grammar. Exemplarily, the incorrect entity class is composed of the subclasses *type* for incorrect type declarations or incorrect array sizes, whereas other faults are assigned to the *algorithm* category. The latter again is subcategorized in the form of a tree structure, which contains deeper classes like *statement* (*call* and *loop*) and *expression* (*precedence*, *compound*, *pointer use*, *identifier*, *constant*, and *operator*).

The proposed grammar-based schema differs heavily from others in that the authors consider syntax to be the "carrier of semantics". The purpose of this schema is to allow software fault categorization to become automated using an algorithm presented in the paper. The automation is intended to save time, as manual fault categorization is considered time-intensive, and to reduce ambiguities in selecting appropriate categories.

The application of the proposed method on the TEX code revealed that incorrect identifiers account for 33 out of 291 (11%) considered faults and that 29 of incorrect identifier faults were detected within the first four months of testing, with none of them being detected during the last five years of testing. From this and similar observations, the authors conclude that besides frequency the persistence of a fault type is an important fault characteristic.

Even though the objective of this schema is to avoid ambiguity, problems can arise when for instance an incorrect statement, like $a := a - b$, with '-' intended to be '+', will be "corrected" with $a := a - b; a := a + 2 * b$. This transformation will be interpreted as a missing entity fault instead of an incorrect entity fault. The authors presume that "corrections that mask the true nature of the fault" would be rare in practice and thus the proposed schema will operate fairly accurately. They claim that the schema will be applicable to procedural programming languages, but may be inappropriate for functional or logic programming languages, as particular subcategories of faults may be different.

2.7 Other approaches

We could only cover a small fraction of empirical studies of software faults in this paper.

Empirical studies on software fault densities in large software systems without the categorization of software faults are given by [OW02, FO00, Ada84, NBZ06]. In these papers, empirical support is presented for hypotheses on the relation between fault densities and software metrics or development process metrics.

Lee and Iyer [LI95] study and categorize software faults in the operating system of the Tandem system. The categorization distinguishes nine major classes of software faults: Incorrect computation, data fault, data definition fault, missing operation, side effect of code update, unexpected situation, microcode defect, others, and unclear. Some of the classes have additional subclasses. This categorization schema seems to be developed by the authors. The most common software faults were missing operations and incorrect handling of unexpected situations. The authors expect that software profiles vary based on the operation environment and the software maturity.

Marick [Mar90] summarizes results of several older studies of faults found in software systems. The author notices that detailed comparisons are not possible due to the fact that there are no standard ways of categorizing fault data. The paper mainly focuses on the quantity of software faults related to different classification schemas but does not compare different classification schemas as such in detail. Some

of his conclusions are, for example, that faults in programming logic are common or that faults of omission play an important role.

Du and Mathur [DM98] analyze a set of errors that led to security breaches, and develop a new schema to categorize these errors. The overall target is to evaluate the effectiveness of different measures of code coverage in revealing critical software issues. The categorization consists of three viewpoints: The first viewpoint specifies the cause of a security error (e.g. validation error, authentication error), the second viewpoint indicates the impact of an error, e.g. unauthorized execution of code, or denial of service. The third viewpoint specifies the fix of an error and consists of the same four attributes as the schema of DeMillo and Mathur [DM95] (see Section 2.6).

3 Conclusions

To conclude our survey, we return to the problem of software fault categorization in software fault injection experiments, which provided our original motivation for writing this paper. Traditionally, such experiments focused on discovering unsafe software behavior by perturbing source code or program states at execution time. To conduct a software fault injection experiment, the experimenter must first define the categories of faults to be simulated, decide upon injection points (i.e. which parts of source code or state are perturbed) and the number of faults to be injected. Software fault injection researchers have long acknowledged that the choice of representative software faults is very difficult:

> The point here is that code mutation that tries to simulate programmer faults appears in the extreme to be a futile, intractable process. [VM98, p. 81]

In addition, they claim that the injecting representative faults might be not as necessary as it seems [DTF96] because artificial faults still lead to realistic errors and failures (based on results of a single study).

We remain unconvinced by the existing evidence in support of the claim that software fault categories are irrelevant to software fault injection. For example, if we wanted to use fault injection techniques to generate synthetic data for evaluating performance of an automated fault diagnosis method, it would be difficult to defend the position that *any* fault load would be just as good. Extrapolating from such synthetic data to particular real projects would not be possible. Similarly, if fault injection is to be used to spot weaknesses in error handling mechanisms of an application during testing and quality assurance, then injecting faults that directly perturb these mechanisms would be counterproductive – as any conclusions from such a study would essentially pertain to *different* software than the deployed one.

Notwithstanding the above critique, our survey of existing fault categorization approaches clearly demonstrates why attempts to reproduce "typical" programmer faults in "typical" proportions might not succeed. Different authors propose different fault categories, and relatively few arguments exist that would clearly support their particular choices. The issue of generalization and transfer of quantitative observations concerning software faults among projects remains open and is exacerbated by the ambiguity of some of the discussed fault categorization schemas. Furthermore, we conclude that the distribution of software faults depends heavily on project-specific factors, such as the maturity of the software, operating environment [LI95] or programming language. Unfortunately, researchers have not established the set of such factors yet, and it is not obvious how one would systematically discover them. The already long history of software fault categorization efforts suggests that this task is not likely to be completed in the near future.

Acknowledgments Part of this work is supported by the German Research Foundation (DFG), grant GRK 1076/1

References

[Ada84] E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, January 1984.

[Bei90] Boris Beizer. *Software testing techniques.* Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition, 1990.

[BV01] Boris Beizer and Otto Vinter. Bug taxonomy and statistics. Technical report, Software Engineering Mentor, 2630 Taastrup, 2001.

[CBC+92] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

[CC96] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *FTCS '96: Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)*, page 304. IEEE Computer Society, 1996.

[DM95] Richard A. DeMillo and Aditya P. Mathur. A grammar based fault classification scheme and its application to the classification of the errors of TEX. Technical report, Software Engineering Research Center and Department of Computer Sciences, Purdue University, 1995.

[DM98] W. Du and A. P. Mathur. Categorization of software errors that led to security breaches. In *Proc. 21st NIST-NCSC National Information Systems Security Conference*, pages 392–407, 1998.

[DTF96] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: a real case study involving

real faults and mutations. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 158–171, New York, NY, USA, 1996. ACM Press.

[Eis97] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, 1997.

[FO00] 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, 2000.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of Symposium on Reliability in Distributed Software and Database Systems (SRDS-5)*, pages 3–12. IEEE CS Press, 1986.

[Joh83] W.L. Johnson. An Effective Bug Classification Scheme Must Take the Programmer into Account. *Proceedings of the Workshop on High-Level Debugging, Palo Alto, CA*, 1983.

[Knu89] Donald E. Knuth. The errors of TEX. *Software–Practice and Experience*, 19(7):607–685, July 1989.

[LABK95] Jean-Claude Laprie, Jean Arlat, Christian Beounes, and Karama Kanoun. Architectural issues in software fault tolerance. In Michael R. Lyu, editor, *Software Fault Tolerance*, chapter 3, pages 47–80. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[LI95] Inhwan Lee and Ravishankar K. Iyer. Software dependability in the tandem guardian system. *IEEE Transactions on Software Engineering*, 21(5):455–467, 1995.

[LS00] Bev Littlewood and Lorenzo Strigini. Software reliability and dependability: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, pages 175–188. ACM Press, 2000.

[Mar90] B Marick. A survey of software fault surveys. Technical report, Dept. of Computer Science. U. Illinois at Urbana-Champaign, December 1990.

[NBZ06] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 452–461, New York, NY, USA, 2006. ACM Press.

[OW02] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64. ACM Press, 2002.

[SSP85] J.C. Spohrer, E. Soloway, and E. Pope. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction*, 1(2):163–207, 1985.

[VM98] Jeffrey M. Voas and Gary McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, Inc., 1998.