Software Fault Characteristics: A Synthesis of the Literature

Tracy Hall[a] David H. Bowes[b] Steve Counsell[a] Leon Moonen[c] Aiko Yamashita[c]

[a]Department of Computer Science
Brunel University London

[b]STRI
University of Hertfordshire

[c]Simula Labs
University of Oslo

Abstract—Faults continue to be a significant problem in software. Understanding the nature of these faults is important for practitioners and researchers. There are many published fault characteristics schemes but no one scheme dominates. Consequently it is difficult for practitioners to effectively evaluate the nature of faults in their software systems, and it is difficult for researchers to compare the types of faults found by different fault detection techniques. In this paper we synthesise previous fault characteristics schemes into one comprehensive scheme. Our scheme provides a richer view of faults than the previous schemes published and presents a comprehensive, unified approach which accommodates the many previous schemes. A characteristics-based view of faults should be considered by future researchers in the analysis of software faults and in the design and evaluation of new fault detection tools. We recommend that our fault characteristics scheme be used as a benchmark scheme.

1 Introduction

Software faults continue to have a significant impact on people and on the success of businesses. The serious consequences of software faults are regularly reported in press headlines, a prime example of which includes the recent failure of NATS (the UK's National Air Traffic Services)[1]. The cost of finding and fixing faults is high. Operational faults in commercial products have been reported to cost $7k per fault to fix (Ohlsson and Alberg 1996) with software developers spending nearly half their time fixing faults (LaToza et al 2006).

To address this situation it is essential to have a detailed understanding of the nature of software faults and the characteristics of faults that occur in particular development contexts. Currently this understanding does not exist. This is partly because no widely accepted, comprehensive and consistently used fault characteristics scheme exists with which to analyse faults. Our aim is to present a benchmark fault characteristics scheme. This scheme is based on a synthesis of the diversity of previous schemes. We demonstrate our scheme as both comprehensive and usable.

Applying our fault characteristics scheme could have important implications for future research in fault detection. One of the biggest challenges is that current techniques are reported to have reached a ceiling in their fault detection capabilities. For example, in defect prediction the best models recall a maximum 80% of faults (Menzies et al 2008, Hall et al 2012). A stubborn 20% of faults remain difficult to predict. Understanding the characteristics of those 80% faults detected and those 20% not detected may provide future researchers with vital information with which to improve the performance of their detection techniques.

Another major problem with current fault detection techniques is that they do not maintain their performance when transferred from one system to another. This lack of transferability is because each software system tends to have specific features. These features stem from context factors including application domain, development environment, language, development team etc. Context variation is also likely to result in different systems having populations of defects with different characteristics. Variations in the characteristics of defect populations for different systems are thought to explain why detection techniques do not transfer well; a technique may work well on a specific set of faults but not so well on another set of faults. For example, our own previous prediction models based on program slicing metrics found only a subset of faults; not surprisingly, this subset tended to be faults with dependence characteristics.

[1] http://www.caa.co.uk/docs/2942/v3%200%20Interim%20Report%20-%20NATS%20System%20Failure%2012%20December%202014.pdf

Currently we do not understand how populations of faults vary from one system to the next. Applying our scheme could not only provide researchers and practitioners with important information on the populations of faults in different systems, but could provide vital insight into how detection techniques need to adapt to improve transferability.

Current studies generally consider faults as homogenous. The performance of fault detection techniques is usually based on counts of defects detected and not detected. Sometimes, especially in defect prediction, performance evaluation does not even include a defect count and is merely based on a categorical assessment of faults present or absent. This current approach is flawed. Faults are far from homogenous. They are not all the same and vary greatly. Some faults are structural problems in the code, some are because data is being handled incorrectly, some are logic problems, some are just minor typographical problems, but many are related to requirements issues or issues in how the system interacts with 3rd party systems. The characteristics of faults are likely to influence the ease with which individual faults can be detected and the most suitable detection techniques to apply.

The characteristics of faults have been identified as important for many years. In 1975 Albert Endres (1975) published a paper analyzing the types of faults found in systems developed by IBM. He concluded that such an analysis, though difficult, was an effective way in which to prevent future faults. Forty years later the analysis of types of faults remains under-exploited by practitioners and researchers alike.

Our current knowledge on the characteristics of faults is largely based on work conducted in the 70's, 80's and early 90's. Chillarege et al.'s (1992) Orthogonal Defect Classification (ODC) could be described as the culmination of this early work. Software and the development context have changed significantly since this early work and the characteristics of faults are also likely to have changed. More recently several fault classification schemes have been devised for particular domains (e.g. (Yin et al 2010) with many additional schemes developed over the last 10 years.

This proliferation of fault classification schemes means that it is difficult for researchers and practitioners to know which scheme to select. The relatively few studies analyzing the characteristics of faults (e.g. (Mantayla and Lassenius 2009), (Ray et al. 2014)) usually design their own schemes. Consequently we have no up-to-date comprehensive scheme that is consistently used to analyse faults. This lack of consistency makes comparing and benchmarking the outcomes of fault studies unsatisfactory. It also makes it difficult to develop detection techniques which explicitly target defects with specific characteristics.

In this paper we synthesis previous published defect categorization schemes to produce a benchmark scheme. We have used concept mapping and card sorting to rationalise the hundreds of fault characteristics in the published schemes into our scheme.

The remainder of this paper is organised as follows: Section 2 describes related work. Section 3 describes the design of the study. Section 4 presents our fault characteristics scheme. Section 5 discusses our future work with the scheme and concludes.

2 Background

The characteristics of faults have been studied for over 30 years during which time a large collection of fault characteristics schemes have been published. Ploski et al (2007) provide an overview of these schemes. Some schemes are high level, offering a general view of faults others are detailed and contain hundreds of fault classifications. Some schemes are application area specific (e.g. Palix et al's (2014) Linux scheme), while others are lifecycle phase specific (e.g. Beizer's (1992) testing scheme). ODC is probably the most popular fault categorisation scheme. Despite this popularity ODC is criticised for failing to be orthogonal and for being difficult to customise to specific contexts (e.g. (Seaman et al 2008). Furthermore, previous schemes generally include poorly defined categories. Exceptions to this are Grady (1992), OCD and IEEE Standards which provide definitions of their categories.

Very few studies of faults consider the characteristics of the faults investigated. Exceptions to this include considering fault severity (e.g. (Khoshgoftaar et al 2005). Categorising faults is difficult and usually involves extensive manual effort. Collecting fault data manually does not necessarily ensure that quality data is collected. Strecker and Memon (2012) provide some useful criteria to ensure that fault data can be collected consistently and objectively across multiple systems.

Automatically classifying faults enables the large scale analysis of faults across many different systems. Li et al. (2005) developed a machine learning approach which classified 29,000 Mozilla and Apache faults with 87% precision for semantic faults. The classification was based on analysing the natural language contained in fault logs written by developers. This approach may have limitations as such developer logs have been shown to be unreliable (e.g. (Li et al 2006)). DeMillo and Mathur (1995) developed an automatic analysis of fault fixes in order to categorise system faults. This automatic process is based on establishing the syntactic differences in the tree structure of code before and after a fix. This approach is more promising as it is based on actual changes made to the code, however it is a very complex approach to implement.

3 METHODS

3.1 Producing the Characteristics Scheme

We initially identified 34 papers published before July 2014 that contained fault categorisation schemes, fault types, fault characteristics or fault taxonomies. These papers were identified by searching the titles and abstracts of papers in IEEExplore and the Web of Science databases using key words: [fault OR defect OR bug] AND [categorization OR type OR characteristics OR taxonomy]. References from the papers returned from these searches were then followed up. Four researchers each independently read these 34 papers to decide whether they contained a fault scheme that could be integrated into a benchmark scheme. These four researchers then met to discuss disagreements and finalise a set of papers on which all agreed.

Each of these papers contained schemes with many individual fault characteristics. We extracted each characteristic from each paper and ended up with 220 characteristics. Each characteristic was printed onto paper cards. We used card sorting[2] to rationalize these cards into a preliminary scheme. This card carding process entailed five of the authors independently sorting their cards into groups. These five authors then came together and amalgamated their groupings of characteristics. Disagreements were discussed and resolved and a preliminary scheme identified. This process was done manually because none of the automated tools to support card sorting could handle so many cards.

Many of the characteristics in the preliminary scheme were related, overlapping and at varying levels of granularity. Together the five authors refined this preliminary scheme to introduce hierarchical structure, eliminate replication and produce a rationalized scheme. Further refinement was done when we tested our scheme out by categorizing the faults in four systems. This refinement included an additional fault characteristic not identified from our literature synthesis. During implementation we also introduced additional dimensions to a fault characteristic. These dimensions involved also collecting the following data for each fault:

- *Basis of the fault*. Whether a fault was as a result of a conceptual design error or concrete implementation error. This information identifies whether the fault is because the developer has misunderstood what is needed or simply made an error in implementing what is needed. This differentiation follows the mistakes, slips and mishaps identified in error theory (Reason 1990]). All software faults can be classified in this way, for example, an API fault may be because the developer mis-understood how the API works (i.e. a conceptual design error) or may be because the developer made a minor error in their API implementation (a concrete implementation problem).

- *Manifestation of the fault.* Each fault should be classified both in terms of the symptom of the fault and the root cause of the fault. A fault may appear to have particular characteristics on first glance at the changes made, e.g. the interface may be incorrect, however when that fault is investigated further the real cause of the fault is something else, e.g. the system is interacting with a third party system incorrectly.

In all, three pieces of information should be collected about each fault:

1. Basis of fault: conceptual design error or concrete implementation error.
2. Symptom category.
3. Root cause category.

3.2 Applying the Benchmark Scheme

Currently it is not possible to automate the categorisation scheme we have developed. The scheme is manually implemented by inspection of both the change logs and the code changes made for a fault fix. For each fault report listed in the fault repository the corresponding fault fix in the change repository is identified. The comments made in both the fault report and the comments logged by developers in the change repository are first considered. Inspection of these comments is likely to generate the fault's *symptom classification*. An inspection is then made to the code that has been changed to fix the fault. Such an inspection usually provides a more in-depth understanding of the fault and generates the *root cause classification* of the fault.

4 THE CATEGORISATION SCHEME

Appendix 1 shows the fault categorisation scheme that we sythesised from the previous schemes published in the literature. The scheme comprises 26 individual fault categories divided into four high level groupings (Construction, Environment, QA and Other). Lower level groupings of categories exist within these.

5 CONCLUSIONS AND FUTURE WORK

We have presented a synthesis of the published fault characteristics schemes. This forms the first version of our benchmark categorisation scheme. This version of the scheme remains to be fully validated. We will validate the

[2] Card sorting: a definitive guide http://www.steptwo.com.au/papers/ext_cardsorting/

categorisation of data from fault repositories both in terms of third parties (usually researchers) categorizing faults from these repositories; but also in terms of originating developers categorising their own faults using the scheme. We intend to ensure that the scheme is usable by both groups as this will mean that the scheme is useful in research but also in practice.

ACKNOWLEDGMENT

This work was partly funded by a grant from the UK's Engineering and Physical Sciences Research Council under grant number: EP/L011751/1. We would also like to thank Dr Joost de Folter for his help in the early stages of this work.

REFERENCES

B. Beizer, Software Testing Techniques, International Thomson Press, second ed., 1990

R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Wong, "Orthogonal defect classification- a concept for in-process measurements" Software Engineering, IEEE Trans on, vol. 18(11), pp. 943–956, 1992.

R. DeMillo and A. Mathur, "A grammar based fault classification scheme and its application to the classification of the errors of tex" Private communication, Software Engineering Research Center, Purdue University, West Lafayette IN, 1995.

Albert Endres. 1975. An analysis of errors and their causes in system programs. In *Proceedings of the international conference on Reliable software*. ACM, New York, NY, USA, 327-336

Robert B. Grady, Practical software metrics for project management and process improvement, Prentice-Hall, 1992

Hall T, Beecham S,Bowes D, Gray D, and Counsell S, "A systematic literature review on fault prediction performance in software engineering" Software Engineering, IEEE Trans on, vol. 38, no. 6, pp.1276 –1304,2012.

T. Khoshgoftaar, N. Seliya, K. Gao, "Assessment of a new three-group software quality classification technique" Empirical Software Engineering, vol. 10(2), pp. 183–218, 2005.

LaToza T, Venolia G, DeLine R. Maintaining mental models: a study of developer work habits. In Proc. ICSE, 2006.

S. Li, L. Tan, X. Wang, S. Lu, Y. Shou, C. Shai, "Have things changed now?" Architectural and system support for improving software dependability. ACM, 2006, pp.25–33.

P. Li, J. Herbsleb, M. Shaw, "Finding predictors of field defects for open source software systems in commonly available data sources: a case study of openbsd" in Software Metrics, IEEE Intern'l Symposium, sept. 2005, pp. 10 pp. –32.

Mantyla, M.V.; Lassenius, C., "What Types of Defects Are Really Discovered in Code Reviews?," *Software Engineering, IEEE Transactions on* , vol.35, no.3, pp.430,448, May-June 2009

T. Menzies, B. Turhan, A. Bener, G. Gay, B. Cukic, and Y. Jiang, "Implications of ceiling effects in defect predictors" in Procs of the 4th internl workshop on Predictor models in software engineering, ser. PROMISE '08. New York, NY, USA: ACM, 2008, pp. 47–54.

Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Gilles Muller, and Julia Lawall. 2014. Faults in Linux 2.6. *ACM Trans. Comput. Syst.* 32, 2, Article 4 (June 2014),

Ohlsson N and Alberg H (1996). Predicting Fault-Prone Software Modules in Telephone Switches. IEEE Trans on Soft Engineering. 22(12):886-894.

Ploski, Jan, et al. "Research issues in software fault categorization." *ACM SIGSOFT Software Engineering Notes* 32.6 (2007): 6

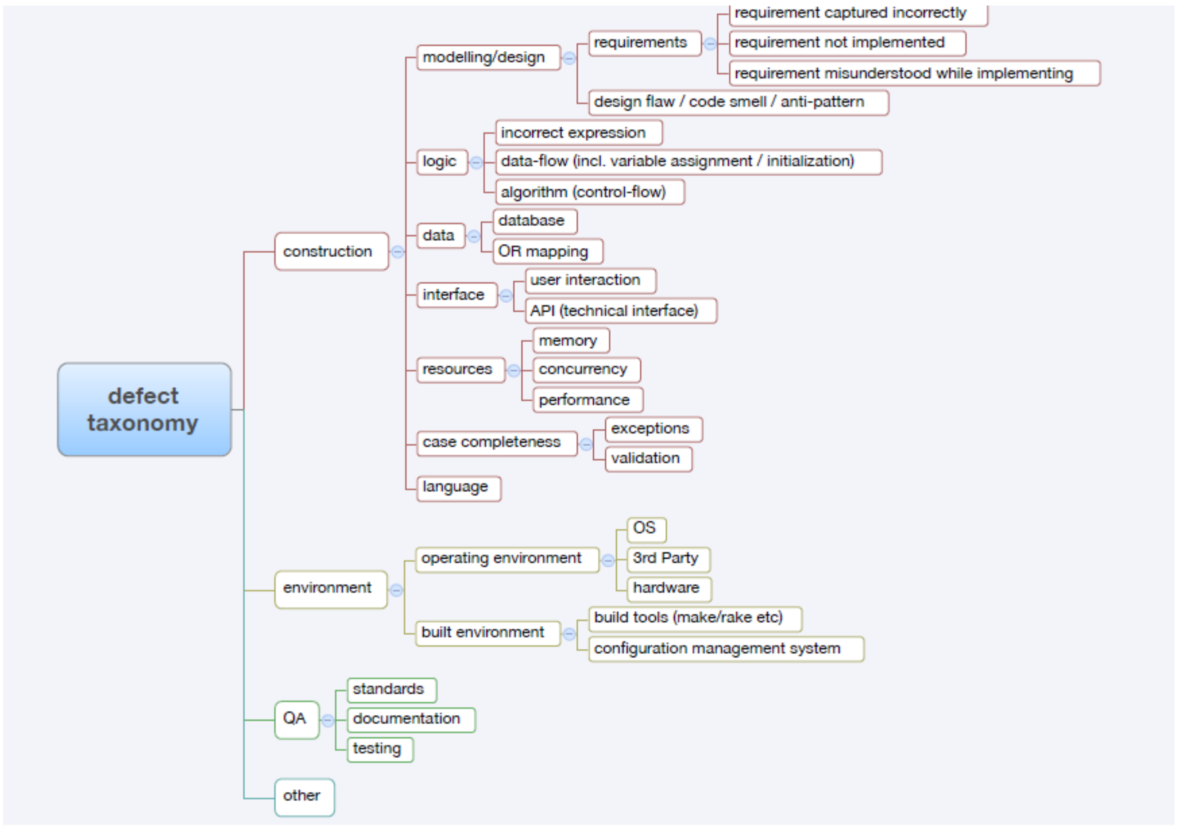
Reason, J. 1990 "Human error. 1990." Cambridge Press

Carolyn B. Seaman, Forrest Shull, Myrna Regardie, Denis Elbert, Raimund L. Feldmann, Yuepu Guo, and Sally Godfrey. 2008. Defect categorization: making use of a decade of widely varying historical data. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement* (ESEM '08). ACM, New York, NY, USA, 149-157.

J. Strecker, A. M. Memon, "Accounting for defect characteristics in evaluations of testing techniques" ACM Trans. Softw. Eng. Methodol., vol. 21(3), pp. 17:1–17:43, 2012.

S. Yin, M. Caesar, and Y. Shou, "Towards understanding bugs in open source router software" ACM SIGCOMM Computer Communication Review, vol. 40(3), pp. 34–40, 2010.

Appendix 1: The fault categorisation scheme


defect taxonomy
construction
modelling/design
requirements
requirement captured incorrectly
requirement not implemented
requirement misunderstood while implementing
design flaw / code smell / anti-pattern
logic
incorrect expression
data-flow (incl. variable assignment / initialization)
algorithm (control-flow)
data
database
OR mapping
interface
user interaction
API (technical interface)
resources
memory
concurrency
performance
case completeness
exceptions
validation
language
environment
operating environment
OS
3rd Party
hardware
built environment
build tools (make/rake etc)
configuration management system
QA
standards
documentation
testing
other