Systematically Improving Software Reliability: Considering Human Errors of Software Practitioners

Fuqun Huang, Bin Liu
School of Reliability & System Engineering, Beihang University, China
huangfuqun@gmail.com

Keywords: reliability, defects, human error propensity assessment, framework

Abstract

Human error is the main contributor to software defects, but has not been taken into consideration systematically by the mainstream software reliability engineering. Based on the review of present reliability technologies, this paper proposed an idea to systematically improve software reliability based on human error research. A framework was constructed on how to integrate human error research into software reliability technologies.

1. Introduction

It is known that defect is the major threat to the reliability of software systems. As was reported, software engineers spend an average of 70–80% of their time testing and debugging, which costs the US economy over $50 billion annually (Andrew and Myers, 2005). Since the software crisis in the late 1960s, people developed various approaches to fight against software defects, but the effects are not optimistic, software failure is still the main contributor to system failure. In 1970s, software engineering was proposed, in 1980s software reliability was researched widely, and in early 1990s, software reliability engineering was established. Over 200 models have been developed, but how to quantify software reliability still remains largely unsolved (Pan, 1999).

This might be because software defects are different from hardware defects in nature. Hardware defects are mostly physical defects, while software defects are design defects, which are closely related to human errors and are not solidly understood by traditional software engineers. Just as Weinberg(1971) stated, "programming---sophisticated as it may be from an engineering or mathematical point of view---is so crude psychologically that even the tiniest insights should help immeasurably." It is necessary and meaningful to systematically introduce and develop theories in human error to traditional software engineering, helping reducing software defects and improving software reliability. This paper presents these ideas, with a systematic framework constructed.

2. Definitions

Before starting, we clarify some definitions in this paper as follows.

- *Software defect* is an unacceptable deviation existing in the software (documents, data, programs) from expectations, which is static.
- *software fault* is a machine state that may cause a runtime failure, e.g., a wrong value in a CPU register, branching to an invalid memory address, or a hardware interrupt that should not have been activated.
- *Software failure* is an event that a program's behaviour does not comply with the program's specifications, which is program output status.
- *Error* is human behavior that induces software defects, which is equivalent to human error of software practitioners in this paper.

The result of a software practitioner's error is a defect in the written software; upon activation, the defect becomes active and produces a fault; if and when the faulty data affects the delivered service, a failure occurs.

3. The Gap in Present Reliability Technologies

There are four technologies to achieve reliable software system. Namely, defect prevention, fault tolerance, defect prediction, and defect removal. In these technologies, human error is closely related to the first three issues, but it is barely considered clearly.

Defect prevention

Defect prevention (which is called Defect Casual Analysis in CMMI) is a technology identifying causes of defects and preventing from occurring. Despite its benefits and broad industry adoption, little research is being done regarding developing and validating a systematic taxonomy of root causes. The present taxonomies for defects root causes are concluded in table 1.

Taxonomy Name	Categories
Mays, Jones & Holloway (1990)	Communication, oversight, education, and transcription
Leszak, Perry&Stoll (2000)	Change coordination, lack of domain knowledge, lack of system knowledge, lack of tool knowledge, lack of system kn., lack of tools kn., lack of process kn., Individual mistake, introduced with other repair, communication problems, missing awareness for need of communication, and not applicable
Card(2005); Kalinowski1, Travassos & Card(2008)	Using the Ishikawa categories: methods, which may be incomplete, ambiguous, wrong, or unenforced; tools and environment, which may be clumsy, unreliable, or defective; people, who may lack adequate training or understanding; and input and requirements, which may be incomplete, ambiguous, or defective

Table 1 – The present root cause taxonomies in defect prevention.

The main challenge is that the causal analysis attempts to identify the root causes among a broad range of possible causes. Tools such as cause-effect diagrams and control charts are utilized to facilitate the analysis process, but detailed causes are depended on brainstorm (Moll, Jacobs, Freimut, and Trienekens, 2002; Card, 1998). There are several problems in applying such approach. Defect causes may fall into different categories analyzed by different people, and the causes could be overlapped or omitted. It is no accident that DCA is often misunderstood and misapplied in software industry (Card, 2006). A reliable and practical taxonomy is needed to guide defect prevention, especially in which human error could be considered deeply.

Fault tolerance

Fault tolerance is an effective way to prevent software failure and improve reliability. N-version programming is the most widely used method to pursue fault tolerance. It is well known that a high degree of version independence and a low probability of common-mode failures are important in the success of an N-version programming. The most popular way is to develop versions by independent teams, with different programming languages, different programming environments, different microprocessors, and different processes, followed by specifications formulated in different notations (Littlewood, Popov, Strigini , 2001). However, these approaches are from common-sense vision. How to pursue diversity so as to best reduce common mode defects is confusing and usually lacking convincing bases.

As human is the key factor contributing to software defects, how to build teams with good diversity to avoid common mode defects is an important issue. However, there has been no attempt to link these "diversity seeking decisions" directly to the origin of errors in the minds of software developers.

Currently some preliminary decisions are proposed, but without much theoretical justification. Giving a more solid scientific basis to these decisions based on human error may have great utility.

Defect prediction

Software defect prediction addresses forecasting total number of defects and residual defects, and estimating the distribution of software defects, which provide data and information for software reliability estimation and prediction. Defect prediction provides basis for decision-making on software delivery time, test resources allocation, test plan making, and software project management. It has been proven that precise defect prediction is an effective approach to obtain reliable software systems. As the good application value, this topic is always been researched since 1970s.

Despite the heroic contributions made by the authors of previous empirical studies, serious flaws remain and have detrimentally influenced models for defect prediction. There appears to be little consensus on what the constituent elements of the problem really are (Fenton, 1999). Generally, researchers predict defect by three ways.

One class of prediction models is based on program size and complexity metrics, which generally assume that defects are a function of size or program complexity. Despite the reported high correlations between design complexity and defects, it is wrong to mistake correlation for causation. A good example is the phenomenon that larger modules can have lower defect densities was confirmed, which cannot be explained properly until Hatton introduced theory that short-term memory affects the rate of human error (Fenton, 1999).

Another class of prediction models is based on testing data during early inspection and testing phases, using statistical models. Extremely accurate predictions are claimed (usually within 95 percent confidence limits) in IBM and NASA projects. However, it is rarely to replicate such result in different organizations or projects, as different process and humans are involved in different projects. This is due to the inherent problems of statistical approaches, which is fitting models to data rather than predicting data.

The third class of prediction models is based on process quality metric data, which is constructed to capture the root factors affecting defect rates. The simplest process metric is by five-level ordinal scale SEI Capability Maturity Model (CMM) ranking. However, until recently there was no evidence to show that level (n + 1) companies generally deliver products with lower residual defect density than level (n) companies(Fenton, 1999). COQUALMO is a synthesized model integrating 22 defect introduction drivers, which contain platform factors, product factors, personnel factors and project factors. However, all these factors are ranked by expert judgement, which is somehow subjective and hard to apply by organizations without such experts.

From the analysis above, we can see that there appears to be little consensus on what are the root causes introducing software defects, and how to assess them. There is no doubt that human error is the key factor contributing to software defects, but this factor is not considered in many models. Even considered, it is just by expert ranking approach, without objective evaluation. This may be the gap should be filled in future research.

4. A Systematic Model to Improve Software Reliability Based on Human Error Research

From the analysis above we can see that it is reasonable and necessary to improve all these reliability technologies by integrating human error research. The systematic model is showed in Fig 1.

For defect prevention, root cause taxonomy plays a key role in root cause analysis and data collection. But the present taxonomy is lack of consideration in human error. So a reliable and practical human error taxonomy of software practitioners is needed to be developed and validated.

For tolerant design, how to construct teams to avoid common mode errors in implementing N-version programming is a key issue. It is a folk believe that different people may make different types of errors. What need to be addressed is what factors make people different in error committing. For

example, different levels of experience would be a factor. The differences between experts and novices in programming are widely researched. Experts possess hierarchically organized knowledge, which gives them a better processing capacity than novices (Détienne, 2002). From this view, we conjecture that novices are prone to commit cognitive limitation errors, which maybe tend to manifest themselves as interface defects. And also we conjecture novices may have a higher error rate when solving problems with high complexity. On the contrary, expert have 'compiled knowledge' which does not need to be understood in depth to be accessed at the highest level and perform more 'automatically'. We conjecture that experts may suffer from "strong but wrong" errors. Even in the same experience level, there also could be some basic factors making people different, such as knowledge difference, meta-cognition difference, and personality traits. Take domain knowledge for example. Specifications and constraints come from various knowledge domains and have to be translated into a specific knowledge domain, the programming domain (Chatel &Détienne, 1996), different levels of experience with the application domain can profoundly affect software design outcomes (Adelson & Soloway, 1985).It is reasonable to conjecture that domain knowledge insufficient may induce problem representation errors, and then yield to design defects. Of course, it is not enough just giving out such conjectures, experiments and data analyzing are needed meanwhile.

For software defect prediction, as software is human cognitive product, which may not comply with statistical models. In contrast to model fitting methodology, predicting defects by measuring of inducement factors seems more reasonable. How to design prediction model based on human error assessment may be a meaningful research direction in future. Also, experiments would be implemented to validate prediction model and human error assessment methods.

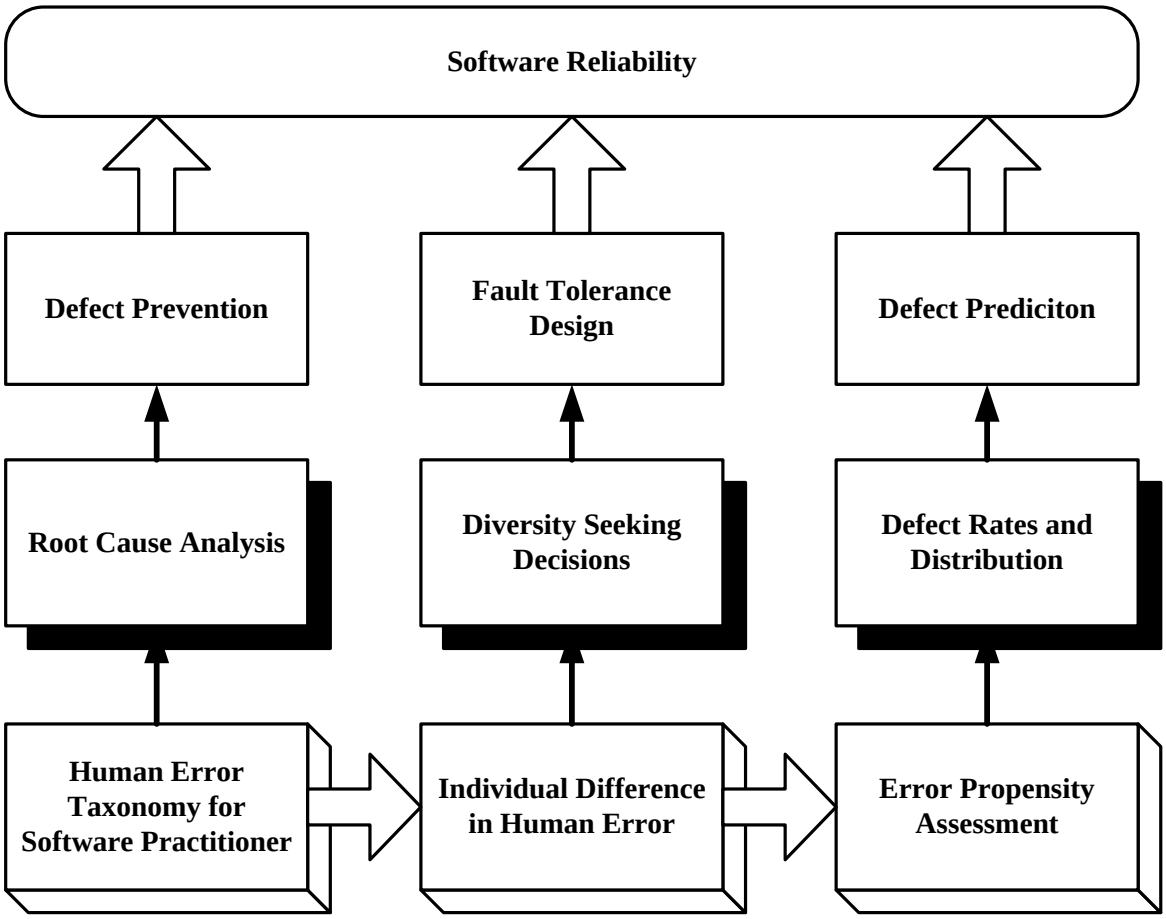

Figure 1-A systematic model to improve software reliability based on human error research

5. Further Research

This paper has just identified the gap in software reliability technologies and proposed an idea to systematically introduce human error research into this field. The research outline was constructed. The human error taxonomy for software practitioners has been developed and validated (Huang, Liu, Strigini). The future work is to implement these research plans, including:

- How to seek diversity decisions based on human error?
- How to construct defects prediction model based on human error?
- How to assess error propensity for software practitioners?

6. Acknowledgements

Thanks Prof. Lorenzo Strigini for enlightening some of the ideas in this paper. Thanks Shreya Keyal for her helpful edit on the previous draft of this paper.

7. References

Andrew J. Ko, Brad A. Myers(2005). A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages & Computing, 16, 41-84.

Jiantao Pan (1999). Software Reliability. http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/.

Gerald M. Weinberg (1971). The Psychology of Computer Programming. New York: VNR Nostrand Reinhold Company, viii.

Mays, R.G., Jones, C.L., Holloway, G.J., Studinski, D.P.(1990). Experiences with Defect Prevention, IBM Systems Journal, 29(1), 4-32

Leszak, M., Perry, D., Stoll, D.(2000). A Case Study in Root Cause Defect Analysis, International Conference on Software Engineering, 428-437.

Card, D.(2005). Defect Analysis: Basic Techniques for Management and Learning, Advances in Computers, 65, 259-295

Marcos Kalinowski1, Guilherme H. Travassos, and David N. Card (2008). Towards a Defect Prevention Based Process Improvement Approach, 34th Euromicro Conference Software Engineering and Advanced Applications. IEEE Computer Society, 199-206

Bev Littlewood, Peter Popov, Lorenzo Strigini (2001). Modelling software design diversity: a review, ACM Computing Surveys (CSUR), 33(2). doi>10.1145/384192.384195

Norman E. Fenton (1999). A Critique of Software Defect Prediction Models. IEEE Transactions on software engineering, 25(5), 675-689.

Francoise Détienne(2002). Software design-cognitive aspects [M] (translated by Frank Bott). UK: Springer-Verlag London Limited, 75-103.

Sophie Chatel , Françoise Détienne(1996). Strategies in object-oriented design. Acta Psychologica, 91(3), 245-269. doi:10.1016/0001-6918(95)00058-5.

Adelson, B., E. Soloway (1985). The role of domain experience in software design. IEEE

Transactions on Software Engineering, 11, 1351-1360.