2011 35th IEEE Annual Computer Software and Applications Conference Workshops

Debugging Debugging

Kinsun Tam
Department of Accounting and Law, School of Business
University at Albany, State University of New York
1400 Washington Ave., Albany, NY 12222, U.S.A.
e-mail: tam@albany.edu

Abstract— **When a program fails to accomplish its intended task, debugging is conducted to identify and remove any bugs. The debugging operation itself is not immune to flaws. Empirical evidence suggests many bugs are found after shipping of software, which calls into question the effectiveness of the present debugging operation. When failing to accomplish its mission, the debugging operation itself needs to be debugged. Challenging the traditional view on debugging, this paper identifies misdirection and inadequacies of the present program debugging operation. To improve debugging, it critically reviews selected aspects of the debugging operation and the system development life cycle, and explores linkages connecting debugging to other environments (e.g. auditing, business, and education). Suggestions for improvement of debugging are made.**

Keywords- bug prevention, debugging, internal control, COSO

I. INTRODUCTION

When a programmer is puzzled by a mysterious bug, a usual approach is to bring in a colleague for new insights on bug hunting. In a similar spirit, this study brings in new perspectives on the control of errors and irregularities from the auditing discipline to help control defects (or bugs) in software development. In particular, this paper applies COSO's (The Committee of Sponsoring Organizations of the Treadway Commission) internal control integrated framework to software quality control. An internal control system to guard against a firm's errors and irregularities (such as production defects and miscounting of inventories) resembles a development team's software quality control to guard against program bugs. COSO was formed in 1985 to study fraudulent financial reporting and to provide recommendations for the U.S. Securities Exchange Commission, public companies, independent auditors, etc. In September 1992, COSO released the "Internal Control -- Integrated Framework" to help define, assess and improve internal control. Internal control, as defined by COSO, is a process to provide reasonable assurance that firms achieve their intended objectives (e.g. effectiveness and efficiency of operations). If an internal control framework helps firms achieve their intended objectives, a similar framework applied to software can arguably promote development of mission-accomplishing software. COSO's integrated framework encompasses five components of internal control, namely (1) control environment, (2) risk assessment, (3) control activities, (4) information and communication, and (5) monitoring. These components are summarized below:

Control Environment. The control environment sets the tone of an organization, influencing the control consciousness of its people.
Risk Assessment. Risk assessment is the identification and analysis of relevant risks to achievement of the organization's objectives, forming a basis for determining how the risks should be controlled.
Control Activities. Control activities are the policies and procedures taken to address risks, which could be preventive, detective, or corrective in nature.
Information and Communication. Information about pertinent events, activities, and conditions must be effectively identified, captured and communicated in a form and timeframe that enable all people to carry out their control responsibilities. All people must understand their own roles in the internal control system, as well as how individual activities relate to the work of others.
Monitoring. Internal control systems need to be monitored to assess their performance over time. Internal control deficiencies should be reported upstream to the proper level of management.

II. SOFTWARE QUALITY CONTROL INTEGRATED FRAMEWORK

Applying COSO's integrated framework to software development produces a framework of software quality control. The following subsections will first normatively describe each component of this software quality control framework, and then positively review corresponding current literature including scholastic publications, professional books and articles, industrial research and white papers, developer forums, and other online documents.

Control Environment. The control environment relates to the ethical values, attitude, and consciousness of the software company's people on software quality. For instance, a stronger control environment should be expected for development of medical device software and nuclear power plant monitoring software than for computer game software. Examples of relevant questions to be considered include: Does management take software quality seriously? Is top management more eager to rush out software to generate sales than to fix known bugs?

Current Literature. In system development, the term "environment" is taken overwhelmingly to mean a software application providing tools and facilities for program development, design, and debugging (i.e. IDE). The impact of people's attitude on bugs is seldom discussed.

Risk Assessment. Risk in software development refers to the risk of producing buggy software (i.e. quality risk), which in turn diminishes the likelihood that software accomplishes its intended mission. Examples of relevant questions are: How do inexperienced development team members affect quality risk? Have new system

978-0-7695-4459-5/11 $26.00 © 2011 IEEE
DOI 10.1109/COMPSACW.2011.93

vulnerabilities been discovered that subject software products to more risks? Are there new developments in programming languages to avoid a certain class of bugs?

Current Literature. There are more studies and discussions on risk assessment than on the control environment. Fault proneness prediction is an actively researched topic [1, 15]. Other discussions of risk analysis, however, tend to focus on the risk of computer attacks [e.g. 13, 17].

Control Activities. Control activities are policies and procedures that help to reduce software defects. Preventive control activities prevent bugs from entering into the source codes. Detective control activities discover bugs after they have been introduced. Corrective control activities remove bugs and correct source codes after bug discoveries. Examples of relevant questions are: What quality control activities do developers adopt to enhance software quality? How do developers prevent, detect, and correct software errors?

Current Literature: Preventive control. The lack of discipline and structure in professional programming style is believed to be the major reason for the poor reliability of larger software products [21]. Sensible programming style and practice are programmers' first line of defense against bugs. Bug prevention is accomplished by following structured analysis, design, and implementation techniques [20]. Widely accepted coding standards include Atlas, Verilog and VHDL, SUN, Doug Lea, Ellemtel etc. Making source codes clear, unambiguous, and logical helps the programmer avoid creating bugs. Source codes should have self-documenting variable names, avoid negative boolean variable names (e.g. isValid instead of isNotValid), have clear comments, and adhere to logical indenting structure [14]. Developer training, clear system specification, and defensive coding in anticipation of all execution conditions are sound preventive controls [19]. Programming languages, if designed to be immune to some classes of bugs (e.g. Java's immunity to pointer arithmetic errors), can contribute to bug prevention.

Current Literature: Detective Control. Peer code reviews, source code scanning and analysis technology, and unit testing are widely regarded as effective means to detect program defects [18, 19]. Some critics contend that software testing comes into a project too late, contributes too little, and costs too much [16]. However, testing is being redefined to emphasize its defect prevention role. Shifting testing to earlier stages of the software development life cycle brings about early testing and early error detection. According to Lewis' [12] "greater testing" approach, it is important to test not just the program, but also the requirements, design, code, and the tests themselves. Moreover, modular coding breaks up a program into smaller but simpler and more manageable pieces, thus reducing the chance of obscure bugs hiding in thick code sections. Likewise, piecewise coding makes testing easier.

Current Literature: Corrective Control. The process of correcting program defects is performed manually (as in walkthroughs or code inspections) or with the help of debugging tools [20]. There has been active research on developing more capable debugger programs to help identify program defects. Divergent views on the usefulness and benefits of debuggers exist. In addition, some professionals express concerns over the quality of the correction, which if not done right, may create new risks.

Information and Communication. Communication of information (e.g. testers' bug reports and user's satisfaction surveys) pertinent to software quality can help develop better software. Examples of relevant questions are: Do individuals within a software firm communicate with each other efficiently and effectively? How do they communicate with external parties such as customers, subcontractors, computer security organizations (US CERT, SANS Institute, etc.), and programming language developers?

Current Literature. As testing moves upstream, testers work more closely with designers and programmers. Information sharing and communication between these professionals improve, resulting in clearer system specification, richer tests, and higher quality codes [16]. Recent efforts to integrate debugging with other software development activities, a topic of IWPD 2011, will benefit from better information sharing and communication.

Monitoring. Monitoring assures that necessary control policies and procedures are in place to address software quality risk. Examples of relevant questions are: Is software quality control monitored? Are monitoring procedures effective? How frequent are the evaluations and what do they cover?

Current Literature. Software development monitoring is often interpreted as keeping the development team and management up to date with development progress and any schedule deviations. There is little discussion on the supervision over software quality control. Evaluations are performed to detect and correct errors, rather than to monitor software quality control.

III. INSIGHTS FROM INTEGRATED FRAMEWORK

The above review of current literature on software quality control reveals uneven emphases on different components. In particular, "control activities" is the most discussed component of the integrated framework. There is contention over the benefits of debuggers. Literature on "risk assessment" and "monitoring" covers narrow topics. Discussion on "control environment" and "information and communication" is light and sparse. This state of software quality control is biased and unhealthy. To improve software quality, emphases on the control environment, risk assessment, information and communication, and monitoring components need to be strengthened.

In addition, the massive research efforts on developing more capable debugger software are arguably misdirected. Debugging is costly. Considerable efforts and time are needed to (1) recognize that a bug exists, (2) identify cause of bug, (3) isolate source of bug, (4) determine bug fixes, and (5) fix and test. In contrast, bug prevention avoids all these costs by dealing with bugs at the earliest and least expensive stage. Design of programming languages that are immune to a certain class of bug, for instance, is a promising bug-prevention research direction.

IV. LESSON FROM THE AUDITING DISCIPLINE

Debugging makes up most of software development efforts. To show the disparity of emphasis on bug prevention versus debugging, this study conducts multiple google searches on April 30, 2011 to compare the counts of matching articles. Google scholar searches made at `http://scholar.google.com/` search Google's database of scholarly literature, and reflect the emphases of scholarly researchers. Google Web searches performed at `http://www.google.com/` search all Web pages indexed by Google, and reflects the emphases of all Websites. The `intitle:` operator requires key words to appear in the title of matching documents. When the `intitle:` operator is not used, key words may appear anywhere in matching documents (hence ANYWHERE as column label in the two tables below). The `site:edu` operator limits searches to only the .edu domain, and includes both scholarly research and teaching documents. When the `site:edu` operator is not used, matching results may come from any domain (hence ANYDOMAIN as column label). Since "bug prevention" and "prevent bug" are synonymous, their counts are summed together. Similarly, the counts of "debugging" and "debug" are totaled. Additional search operators `(~program -"bed bug" -insect)` assure that matching documents contain the word "program" or its synonym, and do not contain "bed bug" or "insect". The bottom row of the following table presents the ratios of prevent-related matches over debug-related matches. These ratios show appalling bias in the emphases of research scholars, educators, and others in favor of the issue of debugging.

search type	**google scholar**	**google scholar**	**google Web**	**google Web**
search operator	**Intitle:**	**ANY-WHERE**	**site:edu**	**ANY-DOMAIN**
bug prevention	1[a]	88[c]	138[e]	28400[c]
prevent bug	0[b]	32[d]	2270[f]	43200[d]
total	1	120	2408[g]	71600
debugging	6650[g]	195000[i]	385000[k]	14100000[i]
debug	1630[h]	96600[j]	294000[l]	24200000[j]
total	8280	291600	679000	38300000
prevent /debug ratio	**0.000121**	**0.000412**	**0.003546**	**0.001869**

Corresponding google searches:
[a] `intitle:"bug prevention" ~program -"bed bug" -insect`
[b] `intitle:"prevent bug" ~program -"bed bug" -insect`
[c] `"bug prevention" ~program -"bed bug" -insect`
[d] `"prevent bug" ~program -"bed bug" -insect`
[e] `site:edu "bug prevention" ~program -"bed bug" -insect`
[f] `site:edu "prevent bug" ~program -"bed bug" -insect`
[g] `intitle:debugging ~program -"bed bug" -insect`
[h] `intitle:debug ~program -"bed bug" -insect`
[i] `debugging ~program -"bed bug" -insect`
[j] `debug ~program -"bed bug" -insect`
[k] `site:edu debugging ~program -"bed bug" -insect`
[l] `site:edu debug ~program -"bed bug" -insect`

As a comparison, a similar table is compiled for preventive control, detective control, and corrective control in the context of auditing (hence `~audit` as search operator). The ratios of preventive control over the total of detective and corrective control suggest preventive control is emphasized far more in the context of auditing than is bug prevention in the context of programming.

search type	**google scholar**	**google scholar**	**google Web**	**google Web**
search operator	**Intitle:**	**ANY-WHERE**	**site:edu**	**ANY-DOMAIN**
preventive control	2[a]	455[b]	1740[c]	55800[b]
detective control	3[d]	200[f]	407[h]	14800[f]
corrective control	1[e]	152[g]	700[i]	14700[g]
total	4	352	1107	29500
prevent/(detect +correct) ratio	**0.5**	**1.292614**	**1.571816**	**1.891525**

Corresponding google searches:
[a] `intitle:"preventive control" ~audit`
[b] `"preventive control" ~audit`
[c] `site:edu "preventive control" ~audit`
[d] `intitle:"detective control" ~audit`
[e] `intitle:"corrective control" ~audit`
[f] `"detective control" ~audit`
[g] `"corrective control" ~audit`
[h] `site:edu "detective control" ~audit`
[i] `site:edu "corrective control" ~audit`

The value of both preventive and detective controls in auditing is time-proven. For instance, the Sarbanes-Oxley Act of 2002 emphasizes both preventive and detective controls, and requires firms to identify whether any control is designed to prevent or detect errors. The above google search comparisons suggest that (1) bug prevention is under-utilized and (2) there is disproportionately strong emphasis on detecting and correcting software bugs. Therefore, more emphasis needs to be shifted from debugging to bug prevention for a more balanced software quality control integrated framework.

V. PROMINENT COMPUTER SCIENTISTS' VIEWS

Computer scientists have been warning against a debugging-based model (i.e. testing for bugs and removing them) of software quality since some forty years ago. Debugging has its limits. In particular, Dijkstra [5] notably comments on debugging as an inadequate means for achieving software quality goal: "Program testing can be used to show the presence of bugs, but never their absence".

Dijkstra, Kernighan [10], and Linus Torvalds argue against debugging software. Dijkstra [7] puts the blame of buggy programs on the programmer: "If debugging is the process of removing bugs, then programming must be the process of putting them in", and "The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it disguises that the error is the programmer's own creation." Dijkstra [5] calls for identifying not the best debugging aid, but identifying and removing careless programmers referred to as "the more productive bug-generators!" According to Dijkstra [5], the best way to remove bugs is to avoid creating them in the first place: "If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with." Kernighan supports the idea of using careful thinking, not a debugger program, as defense against bugs: "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements." Kernighan's view is echoed by Ganssle [9], "The most important debugging tool is a healthy dose of common sense, a sometimes rare commodity in the frenzy of debugging."

There is a common concern that debuggers may encourage developer to apply brute force quick-fixes without sufficiently understanding the true logic of the codes, thus jeopardizing the reliability of the software. Torvalds criticizes debuggers as discouraging programmers from thinking carefully and gaining full understanding what the program does. Using careful thinking as defense against mistakes is echoed in many famous quotes of Dijkstra. Concerned that debugger users could be inclined to regard program logic as a black box, Dijkstra [5] asks developers to understand the program, "… as long as we regard the mechanism [i.e. program logic] as a black box, testing is the only thing we can do. The conclusion is that we cannot afford to regard the mechanism as a black box, i.e. we have to take its internal structure into account." For this reason, Dijkstra [3, 6] emphasizes many times that simplicity is a prerequisite for reliability, that source codes should be more readable and understandable, and that programmers need a clear and systematic mind, to prevent introducing bugs.

VI. CONCLUDING THOUGHTS

Computer professionals have expressed grave concerns over the quality of software products. This study debugs the debugging operation. It suggests adopting a software quality control integrated framework, putting more emphasis on other components of software quality control besides control activities, and shifting research focus from debuggers to bug prevention.

Solving problems is a high-visibility process; preventing problems is of low-visibility [9]. Just as doctors should get credit for illness prevention, bug prevention efforts should be encouraged and rewarded. Future research is needed on metrics to fairly measure the efforts and success in bug prevention.

Schools have not emphasized bug prevention enough, and are too eager to consign to the debugger. Does the debugger instill helpful experience for new programmers? Dijkstra [4, 8] cautions that the wrong experience may easily corrupt the soundness of our judgment, and as long as programming is done by people that don't master techniques of effective reasoning, the software crisis will remain incurable. On the other hand, with a mathematical inclination, a logical mind, and effective reasoning techniques, a student can become a competent programmer [6].

If one thinks carefully before coding, many errors can be avoided. Therefore, perhaps the most important lesson from debugging the debugging operation is to extend the system development life cycle backwards to include academic training. SDLC should start with proper programming education, which should include logical (effective) thinking, manual debugging, and introduction to the fun of programming and debugging. The ability to think carefully and logically is a great asset in manual debugging. A few authors [16] have even alluded to the fun in programming and manually tracking down bugs. Knuth [11] refers to the beautiful, artistic, and enjoyable side of programming. If students with the right reasoning techniques are introduced to the fun side of debugging, they may develop a passion in debugging, which will be another formidable defense against bugs.

Is manual debugging quickly losing its viability as a practical option? This is true as long as developers decide to let the market dictate its wishes on them. Does spoiling-the-market work? The current software development model hands over the software release schedule to the economics of the market, but the market still sadly ends up with inferior software products. Dijkstra [7] says, "It is not the task of the University to offer what society asks for, but to give what society needs," and warns, "… the academic world … has to refine and to teach to the best of its abilities how computing should be done; would it ever yield to the pressure [of computer companies] to propagate the malpractice of today, it had better fold up." Should developers offer what the market asks for, or should they give what the market needs?

REFERENCES

[1] C. Catal and B. Diri, "A systematic review of software fault prediction studies", Expert Systems with Applications, Volume 36, Issue 4, May 2009, pp. 7346-7354.

[2] COSO, Internal Control - Integrated Framework, 1992.

[3] E. Dijkstra, "Notes On Structured Programming. On The Reliability of Mechanisms", 1970, EWD249.

[4] E. Dijkstra, "On the reliability of programs", 1971-1973, EWD303.

[5] E. Dijkstra, "Turing Award Lecture: The Humble Programmer", Communications of the ACM 15 (10), October 1972: pp, 859-866.

[6] E. Dijkstra, "How do we tell truths that might hurt?", 1975, EWD498.

[7] E. Dijkstra, "On the cruelty of really teaching computing science", 1988, EWD1036.

[8] E. Dijkstra, "Answers to questions from students of Software Engineering", 2000, EWD 1305.

[9] J. Ganssle, The Art of Designing Embedded Systems, Second Edition, Elsevier/Newnes, UK:Oxford, 2008.

[10] B. Kernighan, Unix for Beginners. Second Edition, Bell Laboratories, Murray Hill, New Jersey 07974, 1979.

[11] D. Knuth, "Computer Programming as an Art", CACM, vol. 17, p.667 , 1974.

[12] W. Lewis, Software Testing and Continuous Quality Improvement, Auerbach Publications, 3 edition,December 22, 2008.

[13] G. McGraw, "IEEE Security & Privacy", in G. McGraw (Ed.), Building Security In: Software Security vol. 2, no. 2, pp. 80-83, 2004.

[14] R. Stephens, Bug Proofing Visual Basic: A Guide to Error Handling and Prevention, John Wiley & Sons, 1998.

[15] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, "A General Software Defect-Proneness Prediction Framework," IEEE Transactions on Software Engineering, 12 Oct. 2010.

[16] H. Robinson, "Predicting the Future of Testing", StickyMinds.com Weekly Column,12/8/03.

[17] D. Verdon and G. McGraw, "IEEE Security & Privacy", in G. McGraw (Ed.), Building Security In: Risk analysis in software design. pp. 32-37, 2004.

[18] J. Viega, G. McGraw, T. Mutdoseh, and E. W. Felten, "Statically scanning Java code: finding security vulnerabilities", Software IEEE, vol. 17 issue 5, pp.68 - 77, Sep/Oct 2000.

[19] J. Whittaker, Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design. Addison-Wesley Professional, Aug. 2009.

[20] K. Wiegers, Software Engineering: The Software Metric System. ST-Log Issue 35b, September 1989, pp. 50.

[21] N. Wirth, The programming language PASCAL, 1970.