The Role of Inspection in Software Quality Assurance

David L. Parnas, *Senior Member*, *IEEE*, and Mark Lawford, *Member*, *IEEE*

1 Introduction

Despite more than 30 years of effort to improve its quality, software is still released with many errors. Many major products are known to have thousands of bugs. It is not for lack of trying; all major software developers have a software quality assurance effort and attempt to remove bugs before release. The problem is the complexity of the code. It is very easy to review code but fail to notice significant errors.

Researchers have responded to these problems by studying methods of formal correctness verification for programs. In theory, we now know how to prove programs correct with the same degree of rigor that we apply to mathematical theorems. In fact, this is rarely practical and even more rarely done. Most research papers on verification make simplifying assumptions (e.g., a 1:1 correspondence between variables and variable names) that are not valid for real programs. Proofs of realistic programs involve long complex expressions and require patience, time, and diligence that developers do not think that they have. (Interestingly enough, they never have time to verify the program before release, but they must take time to respond to complaints after release.) Inspection methods can be more effective than informal reviews and require less effort than formal proof, but success depends on having a sound and systematic procedure for conducting the inspection. Tools that support this procedure are also important.

The Workshop on Inspection in Software Engineering (WISE),[1] a satellite event of the 2001 Computer Aided Verification (CAV '01) Conference, brought together researchers, practitioners, and regulators in the hope of finding new, more effective approaches to software inspection. The workshop included invited lectures and paper presentations in the form of panel discussions on all aspects of software inspection. Submissions explained how practitioners and researchers were performing inspections, discussed the relevance of inspections, provided evidence of how inspections could be improved though refinement of the inspection process and computer aided tool support and explained how careful design of software could make inspections easier or more effective.

The best ideas from the workshop have been distilled into pairs of papers appearing in linked special issues of *IEEE Software Magazine* (*Software*) and *IEEE Transactions on Software Engineering* (*TSE*).

2 Why Two Linked Special Issues?

As guest editors, we had a very specific goal when we proposed the joint special issues to the publications' editorial boards. We had observed that the practitioners, who read *Software*, tend to neglect the kind of research found in *TSE* on the (sometimes correct) assumption that it is irrelevant to them. On the other hand, researchers tend to write for each other and to lose contact with the realities that practitioners have to face. The linked issues try to narrow this gap. Some of the contributors to WISE were practitioners whose contribution was to explain what they are doing and what problems they encounter. Others were researchers who were trying to discover and verify (either mathematically or experimentally) some fundamental facts. We thought that the researcher authors should write papers that explained to practitioners why the problems that they were studying were relevant to practice. We also thought that the practitioners could communicate to researchers what it is like to try to inspect a program in practice.

To summarize:

1. The purpose of the papers in *Software* is to make practitioners aware of research ideas that they might be able to apply. They do not have to communicate the research results as completely as a normal research paper would.
2. The *TSE* papers do communicate the research results. We considered a paper to make a valid contribution if it shows how known results can be applied to the problem of inspecting software for suitability (fitness for use); the papers are intended to be read by people who are willing to read detailed and careful research papers.

The papers of immediate interest to practitioners (e.g., experience reports) are being published in *Software*, but *TSE* will inform researchers about interesting research problems that are raised by that paper. Similarly, research papers are being published in *TSE*, but practitioners can learn what is relevant to them about the research by reading the *Software* article. The *Software* articles stress what can be done now and the *TSE* articles stress longer term issues. It is intended that there be very little overlap between the two articles.

[1] Not to be confused with the series of Workshops on Intelligent Software Engineering.

- *D.L. Parnas is with the Software Quality Research Laboratory at the University of Limerick, Limerick, Republic of Ireland.*
  E-mail: david.parnas@ul.ie.
- *M. Lawford is with the Software Quality Research Laboratory, Department of Computing and Software, McMaster University, 1280 Main Street West, Hamilton, Ontario, Canada L8S 4K1.*
  E-mail: lawford@mcmaster.ca.

For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 118657.

0098-5589/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society

Authorized licensed use limited to: McMaster University. Downloaded on June 2, 2009 at 10:21 from IEEE Xplore. Restrictions apply.

One reports results and one explains how to use those results or what research is still needed.

We hope that the linked special issues approach can be emulated and improved upon in the future by other guest editors with other topics of importance to both practitioners and researchers. After all, connecting theory with practice is the essence of any type of engineering.

3 What We Mean by Inspection

By "inspection" we mean a systematic approach to examining a program in detail. The goal of such an examination is to assess the quality of of the software product in question, *not* the quality of the process used to develop the product.

In general, "inspection" refers to examining a product by following a prescribed, systematic process that is intended to determine whether or not the product is fit for its intended use. For example, many jurisdictions require a safety inspection for vehicles.[2] They legislate a list of parts that must be examined, measurements that must be made, etc., and criteria for passing the inspection. The word "inspection" usually implies that the process is described in documents, (e.g., checklists, printed forms) that describe exactly what the inspectors have to do during the inspection. The goal of these documents is to make sure that each inspection is so careful and so complete that the failure of an inspection to reveal any defects justifies having great confidence that the product will perform as required.

An inspection process, while systematic and tightly prescribed, is not mechanical; the process description guides the inspectors but is not so prescriptive that the inspections could be done by a machine without human involvement. Success depends on the inspectors understanding the product and the underlying technologies, knowing how to use the appropriate tools, and having considerable experience doing related work.

Because the inspectors, like all of us, have limits on their ability to handle details, the key to inspection of any complex product is a policy of *divide and conquer*, i.e., having the inspector examine small parts of the product in isolation, while making sure that 1) nothing is overlooked and 2) that the correctness of all inspected components implies the correctness of the whole product. The decomposition of the inspection into discrete steps must assure that each step is simple enough that it can be carried out reliably and that one inspection step can be carried out without detailed knowledge of the others.

Inspection can be a very time consuming process. Moreover, no inspection process is perfect. Inspectors may take short-cuts, may not have an adequate understanding of what they are doing, and may identify a product as acceptable when it is not. Nonetheless, a well-designed inspection process can find errors that would be missed by other methods and can engender great trust.

4 The Benefits of Inspection

Testing is widely employed by industry and formal verification widely advocated by the research community as methods of improving software quality. Inspection falls somewhere in between testing and formal verification. Formal verification has yet to catch on with software practitioners while inspection in one form or another has been widely adopted by industry and advocated by leading software practitioners. The main reasons for this difference in acceptance are that inspection can be used directly on the code itself, not abstract models of it, and because it does not require as substantial an investment in training as verification would require. It also does not require the time and formula manipulation ability that verification of typical programs would require.

Inspection seeks to compliment testing. Testing and formal verification help to detect errors and determine mathematical correctness, respectively, but it is possible to have error free (mathematically correct) code that is hard to understand and difficult to maintain. In addition to finding errors in code and related software documents, inspection can also help to determine if coding style guidelines are followed, comments in the code are relevant and of appropriate length, naming conventions are clear and consistent, the code can be easily maintained, etc. When it comes to the cost of building and maintaining large software products, these issues are crucial. To the theorem provers, model-checkers, and automated testing tools, these issues are irrelevant.

There is no need to wait for the code to be complete to reap the benefits of inspection. Early inspection of a document that states system requirements, can help insure that the correct system is built. In our experience, even when mathematical requirements are being used in the formal verification of a product, they may not accurately capture the designer's or customer's intent. Inspection of a requirements document helps to assure that the requirements are capturing the right thing.

5 The Future of Inspection

While many companies are now doing inspection, the effectiveness of inspections can be improved. The *Software* articles provide insights into how software practitioners can improve the effectiveness and applicability of their inspections today. The research articles in the *TSE* provide the theoretical underpinnings of these suggested improvements and offer insights into how inspections might be further improved in the future. The *TSE* articles are evidence that inspection continues to be an active area of academic research.

5.1 Refining the Software Inspection Process

One way that researchers and practitioners are working to address the limitations of current inspection techniques is by refining inspection methods to make them more appropriate for a particular setting and helping inspectors to stay focused on finding the most import problems in the sea of details.

Dunsmore et al.'s "The Development and Evaluation of Three Diverse Techniques for Object-Oriented Code Inspection" proposes three new techniques for the inspection of OO systems and provides some preliminary data on their relative effectiveness.

[2] Some advocates of specific approaches to software inspection assume that their method defines "inspection." In fact, the word was well-defined much earlier.

Authorized licensed use limited to: McMaster University. Downloaded on June 2, 2009 at 10:21 from IEEE Xplore. Restrictions apply.

Reading can be a rudimentary form of inspection. Thelin et al.'s "An Experimental Comparison of Usage-Based and Checklist-Based Reading" describes two different reading techniques and provides a detailed study on their effectiveness.

Both papers illustrate how customizing the inspection process to the task at hand can provide benefits.

5.2 Systems with Real-Time Requirements and Concurrent Activities

Software systems that must deal with a variety of ongoing activities (e.g., device management, user interactions, external event monitoring) have been observed to be less trustworthy than purely sequential programs. Concurrency introduces a form of nondeterminism[3] into the system—external events, which happen at unpredictable times, determine the order of internal events. When nondeterminism is present, an assessor's inability to remain aware of all possible sequences makes inspection more difficult. The nondeterminism makes testing more difficult because a test sequence may cause an error in one case and not in another.

One approach to software quality assessment of systems with real-time requirements in the presence of concurrency that seems worth exploring is restricting the design to place it in a class that is easier to analyze. In what is likely to be the most controversial article in the special issues, Xu's "On Inspection and Verification with Timing Requirements" advocates handling concurrent real-time systems through a preruntime scheduling approach. Xu is asking the designers to accept strong restrictions on their work to make the inspector's job easier.

5.3 Tool Supported Software Inspection

Part of the motivation for organizing the Workshop on Inspection in Software Engineering as a satellite event of CAV '01 stems from the guest editors' belief that computer aided inspection and formal verification techniques represents the area of greatest potential for the future of inspection. From tools to support the work-flow and book keeping of the inspection process through to integrated computer aided verification techniques to allow inspectors to ask the important questions and delegate some of the mechanical details to model-checkers, theorem provers, and other tools, there are many ways opportunities for tools to improve the efficiency and accuracy of inspections. In "Design and Implementation of a Fine-Grained Software Inspection Tool," Anderson et al. describe the theoretical and practical issues underlying a tool that tool can be used to make the inspection of complex software systems more manageable.

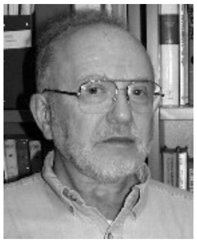
6 Conclusion

In May of this year, a Soyuz TMA-1 spaceship carrying a Russian Cosmonaut and two American Astronauts landed nearly 500km off course after the craft unexpectedly switched to a ballistic re-entry trajectory. Preliminary indications are that the problem was caused by software in the guidance computer in the new, modified version of the spaceship. In the same week, Microsoft's Passport online information repository system was found to have a major security flaw that enabled an attacker to gain access to a user's personal information simply by knowing their e-mail address and constructing an appropriate URL.

These are just the latest examples of lapses in software quality that are shaking the public's confidence that software can be used to build safe, secure systems. In response, both software practitioners and software researchers need to improve the reputation of software and the only way to do that is to improve the quality of software. Inspection is one way to improve software quality. Still, further research is needed to find more practical, effective ways of doing inspections and to measure the effectiveness of those approaches. We hope that these special issues of *Software* and *TSE* on Software Inspection motivate others to take up the increasingly important challenge of improving the effectiveness and practicality of software inspections.

Acknowledgments

The authors would like to thank all of the 2001 WISE participants and, in particular, those that submitted articles for consideration in the special issues. The special issues would not have been possible without the understanding of the reviewers and their invaluable feedback to the editors and authors. Finally, we would like to thank John Knight, Steve McConnell, and Warren Harrison for their encouragement of the linked special issues idea, and the editorial staff of both *Software* and *TSE* for their support.



David L. Parnas received the PhD degree in electrical engineering from Carnegie Mellon University and honorary doctorates from the ETH in Zurich, Switzerland, and the Catholic University of Louvain, Belgium. Dr. Parnas is the director of the Software Quality Research Laboratory, an SFI fellow, and a professor of software engineering at the University of Limerick, Limerick, Republic of Ireland. He is currently on leave from McMaster University. The author of more than 230 papers and reports, he is interested in most aspects of computer system design and has won an ACM Best Paper Award in 1979 and two Most Influential Paper awards from the International Conference on Software Engineering. He is the 1998 winner of ACM SIGSOFT's Outstanding Research award. He is licensed as a professional engineer in the province of Ontario, Canada. Dr. Parnas is a fellow of the Royal Society of Canada, a fellow of the ACM, a senior member of the IEEE, and a member of the IEEE Computer Society.

Mark Lawford received the BSc degree in engineering mathematics from Queen's University, Kingston, Ontario, Canada, in 1989, receiving the University Medal in Engineering Mathematics. His MASc and PhD degrees were obtained from the Systems Control Group, Department of Electrical and Computer Engineering at the University of Toronto, Ontario, Canada, in 1992 and 1997, respectively. Upon completing his studies, he worked at Ontario Hydro as a real-time software verification consultant on the Darlington Nuclear Generating Station Shutdown Systems Redesign project. Since August 1998, he has been an assistant professor in the Department of Computing and Software at McMaster University where he is helping to develop and teach the software engineering programs. His research interests include discrete event systems and application of formal methods to real-time systems. He is a member of both the IEEE Control Systems Society and the IEEE Computer Society.

[3] We will consider a system to be nondeterministic whenever the information available to the observer/assessor is not sufficient to determine the system behavior. In such cases, a system should be designed to handle all possible behaviors.

Authorized licensed use limited to: McMaster University. Downloaded on June 2, 2009 at 10:21 from IEEE Xplore. Restrictions apply.