Inspection's Role in Software Quality Assurance

David L. Parnas, *University of Limerick*
Mark Lawford, *McMaster University*

Despite more than 30 years' effort to improve software quality, companies still release programs containing numerous errors. Many major products have thousands of bugs. It's not for lack of trying; all major software developers stress software quality assurance and try to remove bugs before release. The problem is the code's complexity. It's 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 reality, this is rarely practical and even more rarely done. Most research papers on verification make simplifying assumptions (for example, a 1:1 correspondence between variables and variable names) that aren't valid for real programs. Proofs of realistic programs involve long, complex expressions and require patience, time, and diligence that developers don't think they have. (Interestingly enough, they never have time to verify a 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

0740-7459/03/$17.00 © 2003 IEEE

proofs, but their success depends on having a sound, systematic procedure. Tools that support this procedure are also important.

The Workshop on Inspection in Software Engineering (WISE), a satellite event of the 2001 Computer Aided Verification Conference (CAV 01), brought together researchers, practitioners, and regulators in the hope of finding new, more effective software inspection approaches. Submissions described how practitioners and researchers were performing inspections, discussed inspections' relevance, provided evidence of how refinement of the inspection process and computer-aided tool support can improve inspections, and explained how careful software design could make inspections more effective. The best ideas from the workshop have been distilled into pairs of articles appearing in linked special issues of *IEEE Software* and *IEEE Transactions on Software Engineering*.

Why two linked special issues?

As guest editors, we had a specific goal when we proposed the joint special issues to the publications' editorial boards. We had observed that the practitioners habitually neglect the kind of research found in *TSE* on the (sometimes correct) assumption that it's irrelevant to them. On the other hand, researchers tend to write for each other and to lose contact with the realities that practitioners must face. The linked issues try to narrow this gap. Some contributors to WISE were practitioners who explained what they're doing and what problems they encounter. Others were researchers trying to discover and verify (either mathematically or experimentally) some fundamental facts. We thought that these researchers should write articles that explained to practitioners why the problems they were studying were relevant to practice. We also thought that the practitioners could communicate to researchers what inspecting a program is actually like. In addition, we asked that the purpose of each paper in one publication be explained to the readers of the other.

To summarize:

- The *Software* articles aim to make practitioners aware of research ideas that they might be able to apply. These articles don't communicate the research results as completely as a normal research paper would.
- The *TSE* papers do communicate the research results. We considered a paper to make a valid contribution if it showed how to apply known results to the problem of inspecting software for suitability (fitness for use). The papers are intended for people who are willing to read detailed, careful research papers.

We intend that each *TSE* paper and its corresponding *Software* article have little overlap. One reports results; the other explains how to use those results and perhaps what research is still needed.

We hope that future guest editors will emulate and improve the linked-special-issues approach for other topics important to both practitioners and researchers. After all, connecting theory with practice is the essence of any type of engineering.

What we mean by inspection

By *inspection* we mean a systematic approach to examining a program in detail. Such an examination's goal is to assess the quality of the software in question, not the quality of the software development process.

In general, inspection means examining a product by following a prescribed, systematic process that aims to determine whether the product is fit for its intended use. For example, many jurisdictions require vehicle safety inspections. (Some advocates of specific approaches to software inspection assume that their method defines "inspection." In fact, the word was well defined much earlier.) They legislate a list of parts that must be examined, measurements that must be made, and so on, and criteria for passing the inspection. The word "inspection" usually implies that the process is described in documents (for example, checklists and printed forms) that explain exactly what the inspectors must do. These documents try to ensure that each inspection is so careful and so complete that an inspection's failure to reveal any defects justifies having great confidence that the product will perform as required.

An inspection process, while systematic and tightly prescribed, isn't mechanical; the process description guides the inspectors but isn't so prescriptive that a machine could perform inspections without human involvement. Success depends on the inspectors understand-

Connecting theory with practice is the essence of any type of engineering.

Early inspection of a document that states system requirements can help insure that the correct system is built.

ing 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—that is, examining small parts of the product in isolation, while ensuring that

- Nothing is overlooked
- The inspected components' correctness implies the whole product's correctness

The inspection's organization as a set of discrete steps must assure that each step is simple enough to carry out reliably and that one step can be carried out without detailed knowledge of the others. Inspection can be time consuming. Moreover, no inspection process is perfect. Inspectors might take shortcuts, might inadequately understand what they are doing, and might identify a product as acceptable when it isn't. Nonetheless, a well-designed inspection process can find errors that other methods would miss and can engender great trust.

Inspection's benefits

Industry widely employs testing and the research community widely advocates formal verification as methods for improving software quality. Inspection falls somewhere between the two. 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 (for example, see Stuart Ball's *Embedded Microprocessor Systems: Real World Design*, Newnes, 2000, pp. 16–24). This difference in acceptance has three main causes:

- You can inspect the code itself, not just abstract models of it.
- Inspection doesn't require as substantial a training investment as verification.
- Inspection doesn't require the time and the formula manipulation ability that verification of typical programs does.

Inspection seeks to complement testing. Testing helps detect errors, and formal verification helps determine mathematical correctness, but you can have error-free (mathematically correct) code that's hard to understand and difficult to maintain. In addition to finding errors in code and related software documents, inspection can also help determine whether coding style guidelines are followed, comments in the code are relevant and of appropriate length, naming conventions are clear and consistent, the code is easy to maintain, and so on. Although these issues are irrelevant to theorem provers, model checkers, and automated testing tools, they are crucial to the cost of building and maintaining large software products.

You don't need to wait until code is complete to reap inspection's benefits. Early inspection of a document that states system requirements can help insure that the correct system is built. In our experience, even when a product's formal verification uses mathematical requirements, they might not accurately capture the designer's or customer's intent. Inspection of a requirements document helps assure that the requirements are capturing the right thing.

Inspection's future

Although many companies now do inspection, they can do better. The *Software* articles in this issue provide insights into how software practitioners can improve their inspections' effectiveness and applicability today. The *TSE* research papers provide the theoretical underpinnings of these suggested improvements and offer insights into how we could further improve inspections in the future. The *TSE* articles are evidence that inspection continues to be an active area of academic research.

Refining software inspection

One way that researchers and practitioners are addressing current inspection techniques' limitations is by refining inspection methods to make them more appropriate for a particular setting. Such refinement will help inspectors focus on the most important problems in the sea of details.

In "Practical Code Inspection Techniques for Object-Oriented Systems: An Experimental Comparison," Alastair Dunsmore, Marc Roper, and Murray Wood propose three techniques for inspecting OO systems and provide preliminary data on their relative effectiveness.

Reading can be a rudimentary form of inspection. In "Prioritized Use Cases as a Vehicle for Software Inspections," Thomas Thelin, Per Runeson, and Claes Wohlin combine use cases and operational-profile testing to assess software from a user's viewpoint. They then use an experimental evaluation to compare their method's effectiveness to that of another well-established reading technique.

Although these papers differ in their conclusions, they illustrate how customizing the inspection process to the task at hand can provide benefits.

Systems with real-time requirements and concurrent activities

Software systems that must deal with a variety of ongoing activities (for example, device management, user interactions, and external-event monitoring) are generally less trustworthy than purely sequential programs. Concurrency introduces a form of nondeterminism into the system—external events, which happen at unpredictable times, determine the internal events' order. (We consider a system to be nondeterministic when the information available to the observer or assessor is insufficient to determine the system behavior. In such cases, a system should be designed to handle all possible behaviors.) When nondeterminism is present, an assessor's inability to remain aware of all possible sequences makes inspection more difficult. Nondeterminism also makes testing more difficult because a test sequence might cause an error in one case but not in another.

One potentially worthwhile approach to quality assessment of systems with real-time requirements in the presence of concurrency is to restrict the design to place it in a class that's easier to analyze. In what's likely the special issues' most controversial article, "Making Software Timing Properties Easier to Inspect and Verify," Jia Xu advocates handling concurrent real-time systems through a prerun-time scheduling approach. He asks designers to accept strong restrictions on their work to make the inspector's job easier.

Tool-supported software inspection

We organized WISE as a satellite event of CAV 01 partly because we believe that computer-aided inspection and formal verification techniques have the greatest potential to improve inspection. Many opportunities exist for tools to improve inspection efficiency and accuracy, ranging from tools that support the inspection process's workflow and bookkeeping, to integrated computer-aided verification techniques that let inspectors ask the important questions and delegate some of the mechanical details to model checkers, theorem provers, and other tools. In "Tool Support for Fine-Grained Software Inspection," Paul Anderson, Thomas Reps, Tim Teitelbaum, and Mark Zarins explain how to use their CodeSurfer tool to make inspection of complex software systems more manageable.

In May of this year, a Soyuz TMA-1 spaceship carrying a Russian cosmonaut and two American astronauts landed nearly 500 km off course after the craft unexpectedly switched to a ballistic reentry trajectory. Preliminary indications are that the problem was caused by software in the guidance computer in the new, modified version of the spaceship. That same week, Microsoft's Pass-

Further Reading

A. Aurum, H. Petersson, and C. Wohlin, "State-of-the-Art: Software Inspections after 25 Years," *Software Testing Verification Reliability*, vol. 12, no. 3, Sept. 2002, pp. 133–154.

M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.*, vol. 15, no. 3, 1976, pp. 182–211.

T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley, 1993.

P. Johnson, *The WWW Formal Technical Review Archive*, 1999, http://www2.ics.hawaii.edu/~johnson/FTR.

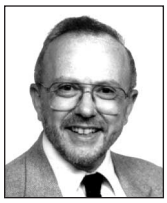
D.L. Parnas and D.M. Weiss, "Active Design Reviews: Principles and Practices," *J. Systems and Software*, vol. 7, 1987, pp. 259–265. Also published in *Software Fundamentals: Collected Papers by David L. Parnas*, D.M. Hoffman and D.M. Weiss, eds., Addison-Wesley, 2001, Chapter 17.

D.L. Parnas, "Inspection of Safety Critical Software Using Function Tables," *Proc. IFIP 13th World Computer Congress*, vol. 3, North-Holland, 1994, pp. 270–277. Also published in *Software Fundamentals: Collected Papers by David L. Parnas*, D.M. Hoffman and D.M. Weiss, eds., Addison-Wesley, 2001, Chapter 19.

port online information repository system was found to have a major security flaw that let an attacker access a user's personal information simply by knowing the user's email address and constructing an appropriate URL.

These are just the latest examples of software quality lapses 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 must improve software's reputation; the only way to do that is to improve software quality. Inspection is one way to do this. Still, we need further research to find more practical, effective inspection approaches and to measure their effectiveness. We hope that these special issues of *Software* and *TSE* motivate others to take up this increasingly important challenge.

About the Authors



David L. Parnas is the director of the Software Quality Research Laboratory, a Science Foundation Ireland Fellow, and a professor of software engineering at the University of Limerick. He is also on leave from McMaster University. He is interested in most aspects of computer system design. He received his PhD in electrical engineering from Carnegie Mellon University and is licensed as a professional engineer in Ontario, Canada. He is a fellow of the Royal Society of Canada and of the ACM, a senior member of the IEEE, and a member of the IEEE Computer Society. Contact him at the Dept. of Computer Science and Information Systems, Univ. of Limerick, Limerick, Ireland; david.parnas@ul.ie.

Mark Lawford is an assistant professor in McMaster University's Department of Computing and Software, where he is helping to develop and teach the software engineering programs. His research interests include discrete-event systems and the practical application of formal methods to real-time systems. He received his PhD in electrical and computer engineering from the University of Toronto. He is a member of the IEEE Control Systems Society and the IEEE Computer Society. Contact him at the Software Quality Research Laboratory, Dept. of Computing and Software, McMaster Univ., 1280 Main St. West, Hamilton, ON L8S 4K1, Canada.

Acknowledgments

We thank all the 2001 Workshop on Inspection in Software Engineering participants, particularly those who submitted articles for consideration in the special issues. The special issues wouldn't have been possible without the reviewers' understanding and invaluable feedback to the editors and authors. Finally, we thank John Knight, Steve McConnell, and Warren Harrison for encouraging the linked-special-issues idea, and the *IEEE Software* and *IEEE Transactions on Software Engineering* editorial staffs for their support and understanding.