Improving Software Quality - The SQRL Approach

Prof. David Lorge Parnas, P.Eng.
Department of Computer Science and Information Systems
Director of the Software Quality Research Laboratory
College of Electronics and Informatics
University of Limerick
Limerick, Ireland

1 Introduction

Many decades of research and experience have made it clear that there is neither a magic tool nor any other easy path to real improvements in software quality. Good software can be achieved only by disciplind hard work. That work includes careful attention to:

- precise specification of the requirements that the software must satisfy
- decomposition of the software into components (modules)
- design of module interfaces
- writing hierarchically structured programs
- precise documentation of design decisions

followed by:

- systematic inspection of all code and documents, and
- disciplined testing

The **S**oftware **Q**uality **R**esearch **L**ab at the University of Limerick (**SQRL**), established with the generous support of the **S**cience **F**oundation of **I**reland (**SFI**), conducts both fundamental and applied research in support of professional software developers. SQRL's goal is to develop methods and tools that can be used to assess and improve the quality of industrially developed software. The methods must be sound, i.e., based on mathematics and computer science, yet practical, i.e., compatible with the realities of software development. The tools must support developers when they apply SQRL methods, must be trustworthy, and must have well-designed user interfaces,

SQRL can achieve its goals only if those who work at SQRL share a fundamental philosophy about software development. This document is intended to be a living guideline for all who work at or with SQRL. Suggestions for clarifications, revisions, or additions are always welcome.

2 SQRL Software Design Principles

This section discusses each of the activities listed above. It provides a brief description of the SQRL approach to each activity and references to more detailed descriptions.

2.1 Specification of System Requirements

High quality software performs exactly the functions required and provides appropriate interfaces to users and external systems. Design decisions about the external effects of the software should not be taken by software developers while programming; they should be fully

specified before the programming begins. A full requirements specification must be available for every high quality software product. This allows decisions about user-visible behaviour to be reviewed and revised by people who are not programmers.

Trustworthy descriptions of externally visible behaviour are useful long after the development is complete. They provide input to those who write user documentation, are helpful in trouble shooting, and surprisingly useful when revising existing system behaviour. The SQRL approach to requirements documentation was first described in [2] and illustrated by a complete example in [3]. Documents such as [20] and [22]provide a more formal explanation of the approach.

2.2 Decomposition into components

Since successful software is almost always changed, software should be organized for ease of change. Each component must be easy to understand; the structure must allow system changes to be carried out component-by-component with changes to each component made without knowledge of internal aspects of other components. This type of "agility" cannot be achieved without explicit analysis of the set of likely changes and reflecting the results of that analysis in the "architecture" of the system.

Consequently, SQRL expects that a design decision[1] that might have to be reconsidered should be the "secret" of a single module, so that only that one module would be affected by a revision of that decision. Moreover, each module should have only one such secret[2]. This principle, known as "information hiding", is also known as "separation of concerns", "data hiding", and reduction of "coupling" [4]. Consistent and systematic application of this principle leads to very unusual structures. An extensive example is discussed in [5].

Adherence to this principle results in simpler internal interfaces and makes the software easier to understand and inspect. Consequently, programming errors are less likely and the expected cost of revision is lower.

However, there is more to decomposition than simply modularisation. There are many other structures (divisions of a product into parts with specified relations between them) that are relevant to software quality. Some of these are described in [6] and the most important is discussed in [7]. A well-thought-out (and well documented) decomposition assists us in developing "program families" or "product lines" as discussed in [8].

2.3 Interface Design

Software interfaces should be precisely specified abstractions that hide the secret of the module while allowing all essential operations to be performed. Interface specification is the "flip-side" of information hiding, Without it, information hiding can never succeed. Programmers developing other modules must be able to use the interface specification in lieu of the design details that are hidden. To achieve this, interface specifications must be precise and complete.

---

[1] Examples of design decisions would be algorithms, data structures, or external interfaces.

[2] Two decisions would be considered part of a single secret if it is unlikely that one would be changed without a corresponding change in the other.

The interface must be designed so that it need not be changed if the design decisions that constitute the secret of the module change. SQRL advocates a systematic procedure for designing, specifying and reviewing such interfaces. The basic concepts behind this approach are discussed in [9] with detailed examples provided in [10]. A method that works in the vast majority of practical cases was introduced and illustrated in [11]. More general, but less intuitive approaches, such as that discussed in [12] and [13], have been proposed. There has been a vast amount of research on algebraic approaches but none of those approaches is ripe for regular use in industrial development; more work is clearly needed.

2.4 Program Structure

Individual sequential programs (the constituents of the modules discussed above) should be "well -structured". Well-structured programs can easily be decomposed into sub-programs whose effects can be summarized by a "program function", allowing their usage to be understood without looking at internal details or other sub-programs. Well structured programs are more easily documented and inspected. This topic is discussed and illustrated in [14] and [15].

2.5 Detailed and Systematic Inspection

Inspecting software to determine whether or not it will function as required is not simply a matter of reading the code hoping to notice errors. Inspection must be performed in a systematic way so that no cases are overlooked and each case can get careful examination. SQRL's document driven inspection procedures are described in [15] and [16].

2.6 Disciplined Testing

In the pursuit of higher software quality, testing and inspection are complementary. Testing can never be complete; inspection can never reveal the fact that an externally provided component does not satisfy its specification. SQRL develops methods and tools for documentation-driven software testing. The precise documentation used for inspection can be also be used to derive test cases, evaluate test coverage, and to generate test "oracles" (programs that evaluate test results [17]), and systems that monitor real-time systems [18]. If operational profiles are available, test results obtained in this way can be used to estimate the in-service reliability of a product.

2.7 Documentation of design decisions

A feature that distinguishes SQRL's approach to improving software quality from others, is the central role that it assigns to precise, highly structured, internal documentation.

Even well-structured software will deteriorate if it is not well documented. Without good documentation, its behaviour will not be understood, changes will not be made properly and keeping the software useful (often called maintenance) will require a steadily increasing amount of effort. [19]. To reduce this "aging" effect, documentation must be complete, precise, and organized in a way that makes information retrieval easy and trustworthy. The contents of the set of documents recommended by SQRL are described in [20].

Tabular expressions make it possible to produce documents that have both mathematical precision and reasonable readability [21], [22]. Tabular expressions can be tested for completeness and lack of ambiguity. Tabular expressions can also be used in pre-implementation simulation. Precise documents that contain tabular expressions are used in SQRL's testing and inspection processes.

3 Research approach

SQRL regards the ideas described above as a starting point; its goal is to improve the methods and build a new generation of developer support tools. Researchers will be expected to start from the principles discussed above, and improve them.

SQRL researchers will be:

- developing methods, notations, and tools to support software developers.
- evaluating the above by working with industry on specific products.
- publishing and presenting scientific papers and reports that describe SQRL's progress.

Each SQRL researcher will have specific tasks and responsibilities. However, those tasks will be part of a single overall project; researchers will review and discuss each other's work regularly.

All SQRL employees are expected to publish their results in reputable scientific journals but they will also be expected to prepare more detailed reports suitable for use by developers. On occasion SQRL will also publish designs that are considered to be of high quality and illustrate the application of good design methods.

3.1 Tool Development

The fact that the SQRL methods and notations are based on sound mathematics and computer science will allow researchers to build powerful tools that can assist developers when they are using the methods.

- Some tools will assist developers in producing better documents. For example, tools can check documents for completeness and consistency.
- Other tools can make these documents more useful. For example, tools can generate test oracles, generate test cases, and evaluate test coverage based on these documents.
- The documents that are produced using SQRL methods can also form the basis of a systematic inspection procedure that can be supported by inspection management tools.

The Software Engineering Research Group at McMaster University in Canada produced a variety of prototype tools which demonstrated the feasibility of such tools [23] and taught us a great deal. SQRL will be developing a new generation of tools building on this experience using a new platform and kernel.

3.2 Improved method descriptions

The papers referenced in this document were written over three decades. During that time, the field has changed a great deal, our understanding has improved, and the notation has been greatly improved. For example, the example in [4] contains a subtle, insight-provoking, design error that

has never been discussed in print. The notation used in [11] has been repeatedly revised and has evolved to the point where many do not see that the original notation was essentially a special case of the more recent approaches. Moreover, the terminology used in the field has changed, as has the type of software being written so that the examples used in such papers as [4], [6] and [7] may no longer seem relevant to software designers. SQRL will publish a series of papers and reports that present these methods as part of a coherent whole using modern terminology and notation.

3.3 Case studies

SQRL conducts pragmatic research. Since our goal is to develop methods and tools that support software developers, we will evaluate our work both by the reception of our publications and by how well our results actually do support industrial developers in practice. While some SQRL researchers are developing new methods and tools, other SQRL researchers will be working with developers to evaluate the usefulness of those tools and methods. Information gained by using the methods and tools will be "fed back" to the rest of the group and will lead to improvements in both tools and method descriptions.

Our work with industry will provide our industrial partners with useful information about the products that we evaluate as well as showing them how they can improve their methods. In return, the developers will provide SQRL with essential feedback about the weak points of our methods. Two-way exchange with industry is essential to SQRL's success.

4 SQRL: Where software researchers can "kick start" research careers.

People interested in research careers are often torn between working in industry or working in academia. Those who choose academia find their ability to do research limited by the fact that they must devote a great deal of time to offering high quality education. Those who choose industry find that project and market pressures interfere with their ability to perform and publish truly sound research. SQRL provides an opportunity to perform research that is expected to meet the highest academic standards without teaching responsibilities. SQRL researchers will have ample opportunity to publish and will not be forced to compromise with quality because of market pressures.

5 References

[1] Hoffman, D.M., Weiss, D.M. (eds.), "*Software Fundamentals: Collected Papers by David L. Parnas*", Addison-Wesley, 2001, 664 pgs., ISBN 0-201-70369-6,.

[2] Heninger, K.L., Specifying Software Requirements for Complex Systems: New Techniques and their Application, IEEE Transactions Software Engineering, Vol. SE-6, January 1980, pp. 2-13.

- Reprinted as chapter 6 in item [1]

[3] Heninger, K.L., Kallander, J., Parnas, D.L., Shore, J.E., Software Requirements for the A-7E Aircraft, NRL Memorandum Report 3876, United States Naval Research Laboratory, Washington D.C., November 1978, 523 pp. and subsequent versions published by the U.S. Naval Research Laboratory.

[4] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15, 12, December 1972, pp. 1053-1058.

- Republished in *Classics in Software Engineering,* edited by Edward Nash Yourdon, Yourdon Press, 1979, pp. 141-150.
- Republished in *Great Papers in Computer Science,* edited by Phillip Laplante, West Publishing Co, Minneapolis/St. Paul 1996, pp. 433-441.
- Reprinted as chapter 7 in item [1]
- Reprinted in *Software Pioneers: Contributions to Software Engineering*, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, Berlin - Heidelberg, 2002, pp. 481 - 498, ISBN 3-540-43081-4.

[5] Parnas, D.L., Clements, P.C., Weiss, D.M., "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering*, March 1985, Vol. SE-11 No. 3, pp. 259-266 (special issue on the 7th International Conference on Software Engineering).

- Also published in *Proceedings of 7th International Conference on Software Engineering*, March 1984, pp. 408-417.
- Reprinted in IEEE Tutorial: "Object-Oriented Computing", Vol. 2: *Implementations* edited by Gerald E. Peterson, IEEE Computer Society Press, IEEE Catalog Number EH0264-2, ISBN 0-8186-4822-8, 1987, pp. 162-169.
- Reprinted as chapter 16 in item [1].

[6] Parnas, D.L., "On a 'Buzzword': Hierarchical Structure", *IFIP Congress '74*, North Holland Publishing Company, 1974, pp. 336-339.

- Reprinted as chapter 8 in item [1]
- Reprinted in *Software Pioneers: Contributions to Software Engineering*, Manfred Broy and Ernst Denert (Eds.), Springer Verlag, Berlin - Heidelberg, 2002, pp. 501 - 513, ISBN 3-540-43081-4.

[7] Parnas D.L., "Designing Software for Ease of Extension and Contraction", *IEEE Transactions on Software Engineering*, March 1979, pp. 128-138.

- Also in *Proceedings of the Third International Conference on Software Engineering*, May 1978, pp. 264-277.
- Reprinted as chapter 14 in item [1]

[8] Parnas, D.L., "On the Design and Development of Program Families", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976, pp. 1-9.

- Reprinted as chapter 10 in item [1]

[9] Britton, K.H., Parker, R.A., Parnas, D.L., "A Procedure for Designing Abstract Interfaces for Device Interface Modules", *Proceedings of the 5th International Conference on Software Engineering*, March 1981, pp. 195-204.

- Reprinted as chapter 15 in item [1]

[10] Parker, R.A., Heninger, K.L., Parnas, D.L., Shore, J.E., "Abstract Interface Specifications for the A-7E Device Interface Modules", NRL Report 4385, November 1980, 176 pgs.

[11] Parnas, D.L., "A Technique for Software Module Specification with Examples", *Communications of the ACM*, 15, 5, May 1972, pp. 330-336.

- Republished in *Writings of the Revolution,* edited by Edward Nash Yourdon, Yourdon Press, 1982, pp. 5-18.
- Also in *Software Specification Techniques* edited by N. Gehani & A.D. McGettrick, AT&T Bell Telephone Laboratories, 1985, pp. 75-88 (QA 76.7 S6437).
- Translated into Russian - book "*Danniye v yazikach programmirovania*" Moscow, Mir (Publishing House), 1984, pp. 9-24.

[12] Bartussek, W., Parnas, D.L., "Using Assertions About Traces to Write Abstract Specifications for Software Modules", UNC Report No. TR77-012, December 1977, 26 pgs.

- Also in Lecture Notes in Computer Science (65), *Information Systems Methodology, Proceedings ICS*, Venice, 1978, Springer Verlag, pp. 211-236.
- Also in *Software Specification Techniques* edited by N. Gehani & A.D. McGettrick, AT&T Bell Telephone Laboratories, 1985, pp. 111-130 (QA 76.6 S6437).
- Reprinted as Chapter 1 in item [1].

[13] Parnas D.L., Wang, Y., "Simulating the Behaviour of Software Modules by Trace Rewriting Systems", *IEEE Transactions of Software Engineering,* Vol. 19, No. 10, October 1994, pp. 750 - 759.

[14] Parnas, D.L., Madey, J., Iglewski, M., "Precise Documentation of Well-Structured Programs", *IEEE Transactions on Software Engineering*, Vol. 20, No.12, December 1994, pp. 948 - 976

[15] Parnas, D.L. "Inspection of Safety Critical Software using Function Tables", *Proceedings of IFIP World Congress 1994, Volume III"* August 1994, pp. 270 - 277.

- Reprinted as chapter 19 in item [1]

[16] Parnas, D.L., Asmis, G.J.K., Madey, J., "Assessment of Safety-Critical Software in Nuclear Power Plants", *Nuclear Safety*, vol. 32, no. 2, April-June 1991, pp. 189-198.

[17] Peters, D., Parnas, D.L., "Using Test Oracles Generated from Program Documentation", *IEEE Transactions on Software Engineering*, Vol. 24, No.3, March 1998, pp. 161 - 173

[18] Peters, D.K., Parnas, D.L., "Requirements-Based Monitors for Real-Time Systems", *IEEE Transactions on Software Engineering*, Vol. 28, No.2, February 2002 pp. 146 - 158.

[19] Parnas, D.L., "Software Aging", in *Proceedings of the 16th International Conference on Software Engineering*, Sorento Italy, May 16 - 21 1994, IEEE Press, pp. 279 - 287

• Reprinted as chapter 29 in item [1].

[20] Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering" published in *Science of Computer Programming* (Elsevier) vol. 25, number 1, October 1995, pp 41-61.

[21] Parnas, D.L., "Tabular Representation of Relations", CRL Report 260, McMaster University, Communications Research Laboratory, TRIO (Telecommunications Research Institute of Ontario), October 1992, 17 pgs.

[22] Janicki, R., Parnas, D.L., Zucker, J., "Tabular Representations in Relational Documents", in "*Relational Methods in Computer Science*", Chapter 12, Ed. C. Brink and G. Schmidt. Springer Verlag, pp. 184 - 196, 1997, ISBN 3-211-82971-7.

• Reprinted as chapter 4 in item [1].

[23] Parnas, D. L., Peters, D. K., "An Easily Extensible Toolset for Tabular Mathematical Expressions", in *Proceedings of the Fifth International Conference on Tools And Algorithms For The Construction and Analysis Of Systems (TACAS '99)*, volume 1579 of Lecture Notes in Computer Science (LNCS) pages 345 - 359. Springer Verlag, 1999.