Why don't we practice what we teach?

Engineering Software for Computer Science Research in Academia.

Andre Oboler, David McG. Squire and Kevin B. Korb

School of Computer Science and Software Engineering, Monash University, Australia

{andre, davids, korb}@csse.monash.edu.au

Abstract

The development process used by academic researchers often seems unsystematic. A Software Development Life Cycle (SDLC) is seldom considered, commenting is scarce, and external documentation consists of erasure marks left on whiteboards. Configuration management is paid lip-service, but is not standard practice. This paper examines reasons behind the apparent large-scale non-adoption of software engineering in academic research. The effects where it was adopted are examined. Finally, we present an SDLC designed for the academic research environment.

1. Introduction

Computer science research usually relies on custom software. This software may implement new algorithms or methods, or facilitate other research. The development process of academic software is often extremely informal. The inherently high-risk and evolving nature of research renders the risk mitigation approaches of most SDLCs, such as the Spiral, inappropriate. The resultant code is often unusable by anyone but its authors, and, even then, only while fresh in their minds. It is all too often throw-away code, yet research is a continuum. There is a dichotomy between the long-term aims of research and the short-term aims of most research programmers.

Watts Humphrey [1] asked "Why don't they [students] practice what we [academic staff] preach?" In this paper, we ask "why don't *we* practice what we teach?" We examine the aims of computer science research and its practitioners. The potential benefits of software engineering in the university setting are introduced, as well as the prohibitive cost of applying it as in industry. We look at current practice, its rationale and problems. Although a perpetual issue in computer science departments, many academics and research students ignore software engineering and feel their work is too small to need it. Past research has suggested improvement in practices across the board, from industry to education [1][2][3][4].

The academic research environment has its own goals, needs, and problems. The current approach, and the problems and impact of software engineering practices, are examined using case studies, surveys and interviews. Although others have recognised the problems of applying industry-style software engineering to academic research, we suggest that writing off software engineering as "industry-only" is wrong, as is using it regardless. We show that there are aspects that work for academic researchers, and that academic research needs a minimum overhead approach that addresses the needs of researchers. We introduce RAISER/RESET, a SDLC that addresses the deficiencies of current approaches when applied in academic settings.

2. Software Engineering and Academia

In 1950 Turing [5] noted that for artificial intelligence research to succeed, improvements in both programming and engineering are needed. As the IT industry developed the "software crisis" emerged. The first conference on software engineering, prompted by this crisis, took place in 1968 [6]. A follow-on conference focused on making software development more "engineering-like" [7].

In 1970 Royce [8] introduced the first SDLC model, the Waterfall. Royce tried to show the steps necessary to bring large-scale software development to an operational state. He first presented a two-step approach: "analysis" leading into "coding". Royce explained that for small projects, in which the software will only be operated by the developers, this is sufficient.

In 1988 Boehm [9] introduced the Spiral SDLC, pointing out that existing models discouraged reuse. The spiral model focuses on the evolving nature of software through prototyping and repeated risk assessment phases.

In 1989 the ACM education board endorsed a report outlining a computer science curriculum including "Software Methodology and Engineering". This featured modular design, abstraction and lifecycles. It aimed to teach how software can be designed for understandability and modifiability [10]. Reuse was not explicitly included.

In 1992 Krueger [11] noted that reuse had failed to become standard practice in industry. In 1996 Devos and Tilman [12] noted that straightforward OOA/OOD focuses on reuse and evolutionary needs too late.

In 1998 Robillard and Robillard [3] compared student development work with industry. They showed that university work was dominated by the programming phase. Humphrey [1] (1998) arrived at the same conclusion, adding that undergraduate students failed to use software engineering practices they had been taught unless directed to do so. Students claimed that class projects were too small. Humphrey described the common student ethic as "ignoring planning, design and quality in a mad rush to start coding". Whether the research student ethic is similar was not discussed.

In 2000 Cook, Ji and Harrison [4] suggested that repeatability might be less relevant for longer term process improvement in software engineering than in other engineering fields. They suggested focusing on design activities and the adaptability of the software process. The evolvability of software is based on the code quality, the evolution process, and the organisational environment in which it takes place. Cook et al. [4] recommended four steps to software evolvability: analyse which parts of the system might need to change, implement the change, restabilise the product (as changes may causes other errors), and test the changed product.

3. A complete approach

We focus on the development of software for research purposes as part of the overall computer science research process in academia.

Our investigation began with two questions:

- Is there room for systematic improvement in the approach computer science researchers take to developing software as part of their research?
- Is there a way to draw on the body of knowledge developed in software engineering and use this as the basis for systematic improvement?

We also consider questions relating to current practice by researchers, their level of knowledge and experiences.

4. Research Methods

We employed both quantitative and qualitative methods, including case studies, surveys, interviews and correspondence with researchers and software engineers.

The survey findings were compared to the experiences of researchers. The case studies were compared to survey trends, and to other projects' experiences as discussed in the interviews. The US survey led to alterations and improvements before the Australian survey was released. Full descriptions of these study components and detailed results can be found in [13].

4.1 Survey

They survey was conducted online, and advertised via e-mail to the heads of the computer science (or similar) departments. From the 255 United States universities e-mailed, 29 survey responses were collected from 17 universities. In Australia, from the 34 universities e-mailed, 35 responses were collected from 13 universities. 15 of these were from Monash University.

Statistical analysis was carried out on the survey results. The Spearman rank order correlation (for ranked responses) and the Pearson product moment correlation (for real values) were used. A t-test was used to calculate the probability that there was no correlation.

4.2 Interviews

All interviews were tape-recorded. They were conducted privately and took place over a one-week period. There were 12 interviews in all, over 13 hours.

4.3 Case Studies

Each case study involved: interviews with developers and users, observations from meetings and presentations, reviews of internal documentation, published literature and meeting minutes and field testing products. Brief descriptions of the three projects studied are given below.

4.3.1 CaMML (Causal Discovery via MML)

CaMML was developed by Wallace and Korb [14]. There have been four versions. Three were based on the original code and developed without any form of software engineering. Korb [15] felt it had suffered for this. The latest implementation effort plans to address issues including maintainability, readability and extensibility.

4.3.2 CDMS (Core Data Mining Software)

CDMS was inspired as "there was no common platform for people to carry-out data mining …different programs spat out different forms of data and no one knew how to use any of the programs apart from the author" [16]. There is a plan to publicly release software and a manual.

4.3.3 The GIFT (GNU Image Finding Tool)

The GIFT [17] is a framework for content-based image retrieval (CBIR) systems. A major goal was to produce a modular, extensible framework of pluggable components,

so that research students could focus on the specific aspect they were researching. For example, in one semester a research student at Monash University extended GIFT's MRML (Multimedia Retrieval Markup Language) to support query-by-region, a task not possible had it been necessary to build an entire CBIR system.

5. Results

5.1 The nature of research and its needs

A clear and constant aim seldom exists in academic software development [15][19]. Development is an opportunistic process, not systematically planned [20], and evolves as the researcher gains knowledge [15][19]. Most research ideas are discarded; it is the exception that something works [15].

Pressman [19] and Brooks [21] observed that the goals and requirements of research software differ from industry. Software engineering research has focused mostly on industry. Wallace [13] observed that, in his 40 years experience, research practices had changed little. Pressman [19] suggested that an approach for the research community should be primarily "agile". The aim of research is to advance the state of knowledge. The aim of researchers is often to publish: academics need to publish articles to build their reputations; research students need to complete their theses. Both groups aim to publish in minimum time [15]. There is usually no incentive to develop robust, extensible and flexible software [15]. Despite this, most would like to start from a well-engineered foundation. We found that much time was lost due to prior obscure coding and a lack of documentation. High quality research work is often shelved once the programmer leaves. The cost for a new person to take over is often prohibitive, or the task nearly impossible.

5.3 Current practice, benefits and costs

Due to a skewed sample population in the US survey, we will concentrate here on the Australian results. These show that an unplanned and non-systematic approach to development dominates (see Figure 1).

Survey participants were asked how regularly they used a variety of software engineering methods, and why. The results indicated that many techniques are considered inappropriate or too costly for research work.

In Australia, both funding and promotion are related to publication rates. Interviews indicated that "the primary aim [in research coding] is to get a flaky prototype working sufficiently to get a few statistics out" [22], in order to publish. The incentive is to "leave it in a half-finished, barely usable, state and go and do something else" [18]. While often beneficial to current researchers, this approach is clearly detrimental to those of the future.

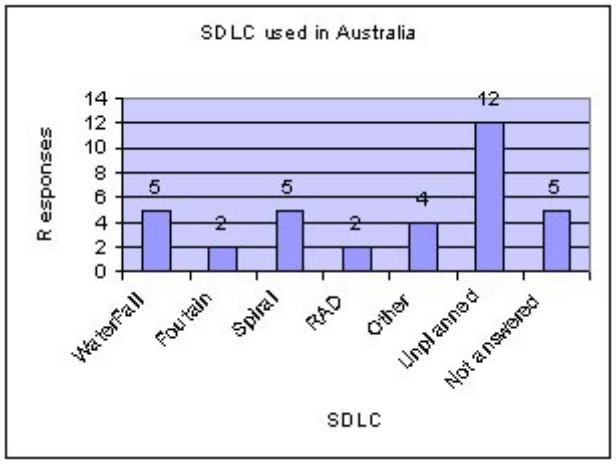

Figure 1. Software Development Life Cycles used in development of academic research software

CDMS revealed the difficulty of funding development activity in Australia. When 90% complete and being used as a platform by other projects, it was almost shelved.

Statistical analysis also highlighted a significant correlation between the willingness to document a project after the research was completed and a higher level of use of certain software engineering practices. The desire to use more software engineering earlier and the use of code and technical reviews are also of interest.

The average number of papers produced by research students was correlated with the number of spin-off projects as well as a history of increased software engineering, a desire to use more software engineering earlier and greater use of code reviews.

Two dominant views appear regarding code reviews: those who know about, but do not use them (11 people), and those who sometimes use them (again 11 people). One interviewee suggested that to review a postgraduate student's code closely might imply mistrust [15].

5.4 Documentation and Communication

Interviews indicated design documents were useful early, but became a burden later [15]. In CDMS, class diagrams were initially useful, but became less so as code familiarity increased and change became more rapid [15]. They were eventually abandoned.

CDMS was developed using a paired-programming approach. "We very rarely sit down and start coding without discussing the issues first… it's interesting and scary the impact that small little issues can have on the system." [16]. This sort of communication was found to be much more effective than documentation [15].

Regarding the CDMS user manual, "We're not sure how this will happen. We were sort of hoping it would happen by magic or be delivered by a stork" [22]. CDMS has high publication potential, but is not yet stable enough

to write about. There are other more pressing things (like PhD theses) requiring attention [15].

In the case of CaMML, completed design documents were ignored. Research students, wanting to code immediately, simplified the design, making it less general and hence less extensible. A gap appeared between documents and code: “We were being typical computer scientists and coding without any specification” [23].

A GIFT user found that the MRML manual was helpful. It was sometimes necessary, however, to consult the authors (via e-mail), or their theses. The decision to use GIFT was based on an expected time saving and its platform-independence.

6. Discussion

An unplanned approach to developing research software is widespread. There is little incentive to continue development once a prototype exists and results published. Many software engineering tools are rejected as inappropriate, or of greater cost than benefit. It appears that Royce’s [8] two-step approach (§2) is still prevalent.

Difficulties of distribution, hardware dependence and cost once limited the reuse of research code. This is no longer the case. Research coding for a single user and purpose is now outdated.

While the Spiral model [9] works well for industry, research often stops after prototyping. An evolutionary approach is needed, but it must take a longer-term view.

Krueger’s observation [11] that reuse seldom occurs is particularly true of academic development. Slight adjustments to past work often require new researchers to start from scratch. The desire for reuse was the inspiration behind both the GIFT and CDMS. Devos and Tilman’s observation [12], that reuse should be factored in earlier, is supported by the success of the GIFT.

We found the development ethic of research students to be much as Humphrey [1] found for undergraduates. For an individual researcher, the view that the cost of software engineering is greater than the benefit, may be accurate. For future researchers, however, well-engineered code and decent documentation can provide a better introduction to the research and reduce the recoding of existing work.

This research supports the suggestion that the evolvability of software is based on code quality, the evolution process and the organisational environment in which it takes place [4]. Both technical and code reviews were associated with larger numbers of spin-off projects, i.e. more evolution. A willingness to document after the project, i.e. to follow through with the third step towards evolvability [4], “stabilising” the development, was associated with a higher level of software engineering and again, with the use of reviews.

7. Preliminary conclusion

The key conflict in computer science research is that between the desire to complete research quickly and that to extend and mature the field. One encourages a "quick and dirty" approach, the other requires a significant amount of planning, effort and engineering. In order to assist, rather than hinder, the research effort, the bulk of software engineering should take place after the research is over. A department's interests are served by developing and supporting high-quality, long-term projects. They attract students, improve the department’s reputation and speed future research by providing a stable framework.

8 A two part approach – Research and Development

The 1993 EDRC workshop concluded that it was time to “adopt a more comprehensive approach to software development—even within a research setting—and for establishing a better infrastructure for software design, maintenance and reuse" [2]. This is still this case.

Research must contain an element of `discovery’. The ideas in the researcher’s mind are subject to constant review and change. A development process for software to aid research must likewise be able to change rapidly [19]. The development process should aid the researcher. The real value is the idea. The burden of implementation must be minimized during the research phase.

Development is the maintenance phase of a research project. New algorithms and functionality should not be added during this time. The development cycle is a restructuring and documenting phase. It aims to provide strong cohesion, so few modules will need to be changed later, and loose coupling allowing a higher degree of reuse and evolvability. Development cleans up the code and leaves it stable for the next team of researchers. Development may also recreate the interface, add user documentation and make the product more useable.

The RAISER/RESET approach (see Figure 3) divides work into research and development cycles. The aim is to ensure that the both tasks occur and support, rather than hinder, each other. The SDLC requires change to the research process, but has a low overhead for research staff. A high overhead for development is required, but such an overhead is currently being paid by research students. The RAISER/RESET SDLC shifts the burden to a more proficient, specialised unit.

8.1 The RAISER/RESET SDLC

In the RAISER/RESET SDLC, research takes place in the top “RAISER” half, while development takes place in the bottom “RESET” half. A distinction is made between initial research and follow-on research. More than one

"initial research project" can go into a single development phase. Likewise, a development phase that produces stable software may spawn many new "follow-on research" projects.

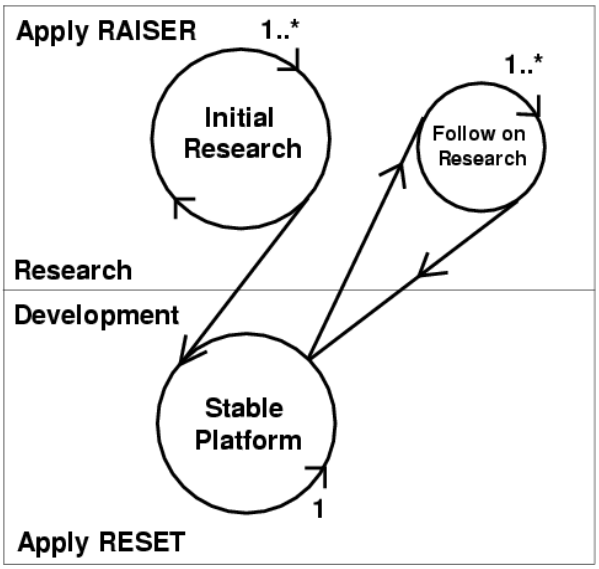

Figure 3. The RAISER/RESET SDLC.

8.1.1 Initial research

Research software begins with an idea. The idea is coded, considered, tested, changed, recoded, etc. and eventually publication occurs. At this point most projects are only half-done. After the initial exploratory research, development is needed to clean code and produce design and user documentation. This "stabilisation" [4] not only makes the work more usable and "evolvable", but also lowers the barrier for future researchers.

8.1.2 Stable Platform

Once the research has come to an end, work may continue in the development phase. Researchers may advise during development, ensuring the scientific integrity of the altered product, and clarifying their code. The "resetting" of the code should involve formal technical reviews. It aims to produce readable, documented, modular and reusable code. Development should take place between research projects and not add an additional burden to the researchers.

8.1.3 Follow-on Research

Follow-on research extends or improves the existing research, through the development of new algorithms or the addition of complementary new functionality. The researcher should work with a stable platform rather than the initial research. Much time is wasted trying to understand other researchers' code. Well-structured, readable code and documentation can greatly reduce this.

8.1.5 RAISER

RAISER (Reactive Assisted Information Science Enabled Research), aims to reduce the burden of software engineering so that it does not interfere with research, yet still raises the evolvability and readability of research code. The four features of RAISER are described below.

Reactive: RAISER is reactive rather than proactive. Changes to ideas will cause changes to code and/or approach. As the project changes, so to may the software engineering tools being used.

Assisted: The code and methodology are there to assist the researcher, not burden them.

Information Science Enabled: This stresses the theoretical research that is behind the software development.

Research: The RAISER phase should only occur while new research is being done. Once all or a significant (publishable) part of the research is completed the project should migrate to the RESET phase.

The software engineering tools used in RAISER should include internal commenting, high-level design documents, algorithms, configuration management, and occasional code reviews at the developers' instigation. Other methods should be added as appropriate. Anything not of benefit to the researchers should be avoided. Other recommendations for RAISER development:

- Code should be written in a modular way
- Code should use header blocks
- Header blocks should additionally contain notes relating to possible future work in that module
- Configuration Management should be used.
- High-level design such as a class diagram or higher-level DFD should be used. While it is useful the diagrams should be updated.
- At least two people should work on a project checking each other's code for readability and requesting clarifying documentation as needed.
- A work schedule taking account of papers to be written should be created.

8.1.6 RESET

As software developed during the RESET phase is based on research software, RESET will differ somewhat from the usual development cycle. The key difference is the existence of a "research prototype" and access to the one or more of its developers. The features of RESET are:

Research Enabled: Software based on a research might not fit any common design patterns. Opportunity may exist for software engineering research to abstract new design patterns.

Software Engineering: The development task is largely one of software engineering. Existing functionality should not change. The prototype must be cleaned up and remoulded, restructuring code for interface and robustness improvements. The software may also be more thoroughly tested and debugged. Finished

design documents may be produced for future developers, both software engineers who will integrate new components and computer scientists who will create them. A user manual may also be produced.

Techniques: The RESET process must vary depending on the level of software engineering employed in the RAISER phase. Suggestions for RESET development include:

- Design and code reviews should be conducted (initially on the researcher's work and later with him/her as a reviewer)
- Existing code should be checked for modularity and restructured as needed
- An interface should be reviewed or created. It will then be documented.
- Design documents should be produced explaining the module structure and responsibilities
- User documents will be produced
- A functional specification detailing current functionality, as well as possible improvements, should be produced.

9. Conclusion

In this paper we have investigated the nature of software development for research in computer science. We have examined the way software engineering is currently employed (or often *not* employed). We suggest that isolated development of research software useable only by the programmer is no longer the best approach, yet is often used. Finally we have presented a new SDLC and approach to software development aimed specifically at the research environment. To recap: research and development should be distinguished. Research requires development. Development requires research. The two must take place in turn, in a regularly repeating cycle and the successful university of the future will require both.

References

[1] Humphrey, W. S. (1998). "Why don't they practice what we preach?", *Annals of Software Engineering*, 1(4): 201-222.

[2] Steier, D., Coyne, R. and Subrahmanian, E. (1993). "Software doesn't transfer, people do—(and other observations from an EDRC workshop on the role of software in disseminating new engineering methods)", Technical report, Carnegie Mellon University.

[3] Robillard, P. N. and Robillard, M. P. (1998) "Improving Academic Software Engineering Projects: A Comparative Study of academic and Industry Projects", *Annals of Software Engineering*, 6:343-363.

[4] Cook, S., Ji, H., and Harrison R. (2000), "Software evolution and software evolvability", Working paper, University of Reading, UK.
http://www.personal.rdg.ac.uk/~sis99scc/desel/desel_result.html

[5] Turing, A. (1950). "Computing machinery and intelligence", *Mind* LIX(236): 433-60.

[6] Naur, P. and Randell, B. (eds) (1969). "Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE", Garmisch, Germany, 7th to 11th October 1968, Scientific Affairs Division, NATO.

[7] Randell, B. and Buxton, J. (eds) (1970). "Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee", Rome, Italy, 27-31 Oct. 1969, scientific Affairs Division, NATO.

[8] Royce, W. W. (1970). « Managing the development of large software systems: Concepts and techniques", 1970 WESCON Technical Papers, Vol. 14, Western electronic Show and Convention, Los Angeles, pp. 1-9. Reprinted in *Proceedings of the Ninth International Conference on Software Engineering*, Pittsburgh, PA, USA, ACM Press, 1989, pp.328-338.

[9] Boehm, B. W. (1988). "A spiral model of software development and enhancement", *IEEE Computer* 21(5): 61-72.

[10]Denning, P., Comer, D., Gries, D., Mulder, M., Tucker, A., Turner A., and Young, P. (1989), "Computing as a Discipline", *Communications of the ACM*, 32(1)

[11] Krueger, C. W. (1992). "Software reuse", *ACM Computing Surveys* (CSUR) 24(2): 131-183.

[12] Devos, M. and Tilman, M. (1996), "Object Oriented and evolutionary Software Engineering", *WS18 workshop, OOPSLA Conference*, San Jose 1996

[13] Oboler, A (2002), "Investigating the use of Software Engineering in Computer Science Research", Honours Thesis, School of Computer Science and Software Engineering, Monash University, Australia. http://www.csse.monash.edu.au/hons/projects/2002/Andre.Oboler/

[14] Wallace, C. S. and Korb, K. B. (1999). "Learning linear causal models by MML sampling", in A. Gammerman (ed.), *Causal Models and Intelligent Data Management*, Springer-Verlag.

[15] Oboler, A., Squire, D. and Korb, K. "Why don't we practice what we teach? Engineering Software for Computer Science Research in Academia", Tech. Report 2003/139, School of Computer Science and Software Engineering, Monash University, Australia, 2003.

[16] Comley, Josh (2002). Interview with Josh Comley, conducted for the USE CSR project.

[17] Squire, David McG., Müller, Henning, Müller, Wolfgang, Marchand-Maillet, Stéphane, and Pun, Thierry, "Design and Evaluation of a Content-based Image Retrieval System", In *Design and Management of Multimedia Information Systems: Opportunities and Challenges*, 7, pp. 125-151, Idea Group Publishing, 2001. http://www.gnu.org/software/gift/gift.html

[18] Wallace, Chris. (2002). Interview with Prof Chris Wallace, conducted for the USE CSR project.

[19] Pressman, R. (2002). Re: SE practise in Comp. Sci. Research, personal communication.

[20] Waite, W. (2002). Re: Use of software engineering in computer science research, personal communication.

[21] Brooks, Jr, F. P. (2002). Re: Use of software engineering in computer science research, personal communication.

[22] Allison, Lloyd (2002). Interview with Dr Lloyd Allison, conducted for the USE CSR project.

[23] Hope, Luke. (2002). Interview with Lucas Hope, conducted for the USE CSR project.