Software Engineering for Computer Science Research

An Approach to Facilitate Improved Research Outcomes

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

{andre, davids, korb} @csse.monash.edu.au
School of Computer Science and Software Engineering
Monash University
Clayton Vic 3800
Australia

Abstract. The development process used by researchers often seems to be random and unsystematic. A Software Development Life Cycle (SDLC) is often not considered, internal commenting is scarce, and external documentation takes the form of erasure marks left on whiteboards. Configuration management is paid lip-service, but is not standard practice. This paper examines some reasons behind the apparent large-scale non-adoption of software engineering in academic research. We also look at the effects on some projects where it was adopted. Finally we present a new SDLC designed for the academic research environment, suggest an organisational structure to maximize the benefit to research students, academics, and the institution.

1 Introduction

Computer science research usually relies on custom software, which is developed by researchers or their research students. This software may implement new algorithms or methods, or facilitate other research, through time-savings or improved flexibility. 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 Software Development Life Cycles (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 asked "Why don't they [students] practice what we [academic staff] preach?" [1]. In the this paper, we ask "why don't we practice what we teach?" We examine the aims and goals 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 them as in industry. We look at current practice and the reasoning behind it, as well as current problems. Although software engineering is a perpetual issue in computer

science departments, many academics and research students would rather ignore it. Many feel their work is too small to warrant it. Past research has suggested improvement in practices across the board, from industry to what we teach [2, 1, 3]. We are not aware of prior research on software development in the academic research environment. We show that this environment has its own needs, goals and problems. The current approach, and the problems and impact of various software engineering practices, are examined using three case studies, two surveys (covering both Australian and US institutions) and a number of interviews with academic researchers and research students. Although others have recognised the problems of applying industry-style software engineering to academic research, we suggest that writing off software engineering as an industry-only tool is the wrong approach, as is using it regardless. We show that there are aspects that work for academic researchers and that academic research needs its own approach, that addresses the needs of researchers while adding minimum overhead. We introduce RAISER/RESET, a SDLC that addresses the deficiencies of current approaches when applied in academic settings. We present ideas on implementation and the organisational change needed in order to increase the reuse and software quality.

2 Software Engineering and Academia

Turing [4] noted in 1950 that for artificial intelligence research to be successful, improvements in both programming and engineering would be needed. As the computer industry developed, the "software crisis" slowly emerged. The first conference on software engineering, prompted by this crisis, took place in 1968 [5]. A follow-on conference focused on making the development of applications and programs more "engineering-like" [6].

In 1970 Royce [7] introduced the first SDLC model, the Waterfall. Royce tried to show the steps necessary to bring large-scale software development to an operational state, on-time and on-budget. He first presented a two-step approach: "analysis" leading into "coding". Royce explained that for small projects, where the software will only be operated by the developers, this is sufficient.

Between 1981 and 1985 a group of physicists working in universities created the popular Numerical Recipes books [8]. This was inspired by the gap between best practice "as exemplified by the numerical analysis and mathematical software professional communities" and average practice "as exemplified by most scientists and graduate students" that the authors knew. This example of academics software reuse was an exception.

Boehm [9] introduced the Spiral SDLC in 1988, pointing out that existing SDLC models discouraged reuse and prototyping. The spiral model, that has since become dominant, 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" as one of nine required disciplines. This included modular design, abstraction and life-

cycles. One of the goals was to teach students how software can be designed for understandability and modifiability [10]. Reuse was not explicitly a core part of the curriculum.

Krueger [11] noted in 1992 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 much too late, and as such was less than optimal for evolutionary development.

In 1998 Robillard and Robillard [3]compared student development work with that in industry. They showed that university work was largely dominated by the programming phase.

Humphrey [1] arrived at the same conclusion, but also noted that students (and practising engineers) failed to use software engineering practices they had been taught unless explicitly directed to do so. They did not follow the advice given by their professors and managers because: they already had their own practices that worked (and were reinforced through repeated use); more often than not impressive-sounding new methods and tools disappointed; no one observed how the work was done. Students claimed that class projects were too small. As Humphrey noted, large programming projects are simply a collection of smaller projects, making this style of thinking (viewed as acceptable by Royce) something of a concern. Humphrey described the common student ethic as "ignoring planning, design and quality in a mad rush to start coding". Humphrey's Personal and Team Software Processes (PSP and TSP) are not tailored to academic research, but are aimed rather at industrial-style development and student projects aiming to mirror this. While recommending that staff should adopt the PSP, Humphrey's reason is to enforce the importance of the method to students and not to seem hypocritical. Whether the research student ethic is similar to the undergraduate student ethic is not discussed.

In 2000 Cook, Ji and Harrison [13] suggested that repeatability might be less relevant for longer term process improvement, and the creation of more evolvable software, in software engineering than in other engineering fields. Instead they suggest focusing on design activities and the adaptability of the software process. The evolvability of software is based on the quality of the code, the evolution process, and the organisational environment in which it takes place

Cook et al. [13] recommended four steps to software evolvability: analyse which parts of the system might need to change to meet a new request, implement the change, restabilise the product (as changes in one part may causes errors in another), 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, by both academic researchers and research students.

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 a number of sub-questions relating to current practice by researchers, their level of knowledge and experiences.

4 Research Methods

The research was carried out using quantitative and qualitative methods. This can give a more complete understanding of the phenomenon under investigation. When different collection methods are used, triangulation increases the confidence in the results and reduces the amount of noise due to measurement error [14]. The methods used included case studies, surveys, interviews with researchers, and correspondence with leading researchers and software engineers.

The findings from the surveys were compared to the experiences of researchers, as discussed in interviews and e-mails. The case studies were compared to trends found in the surveys, and also to experiences from other projects discovered in the interviews. The first survey (in the US) led to alterations and improvements before the second (the Australian survey) was released.

4.1 Survey

The survey was conducted online, advertised via e-mail to the heads of the computer science (or similar) departments. The e-mail requested assistance and requested that it be forwarded to departmental staff. 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. 15 of the Australian responses were from Monash University, the remainder were spread between 12 other universities.

The US survey highlighted an important problem: some heads of department, seeing the survey title "Investigating the Use of Software Engineering in Computer Science Research", responded that they would forward the survey to their software engineering staff. Those that responded in this way received further explanation on the research and its desired target (computer science researchers). In the US survey 72% of respondents had taught software engineering. Before releasing the Australian survey the announcement was altered to remove almost all references to "Software Engineering". The Australian survey was released via e-mail to departmental heads as well as via IT-announce[1]. In the Australian responses only 43% had taught software engineering.

Statistical analysis was carried out on the survey results. The Spearman rank order correlation (for data where one or more values were ranked) and Pearson

[1] Australian, largely academic, mailing list: http://www.cs.usyd.edu.au/it-announce/about.htm

product moment correlation (where all data values were real values) were used to measure correlations between responses. A t-test was used in all cases to calculate the probability of the null hypothesis (that there was no correlation).

4.2 Interviews

All interviews were tape-recorded. In order to lower the risk of one interview influencing another, they were conducted privately and scheduled to occur in a one-week period. Interviews typically lasted for one to two hours. The interviews were semi-structured. Questions were provided to the interviewees prior to (or at the start of) the interview. Interviewees were informed that the questions were a rough guide only and the interview could diverge from them at either party's instigation. For some this was divided between a case study interview and more general discussion. There were 12 interviews in all, over 13 hours.

4.3 Case Studies

Three projects were used as case studies. Short descriptions are given below.

CaMML (Causal Discovery via MML) CaMML was developed by Wallace and Korb [15]. There have been four versions of CaMML. Three were based on the original code by Wallace and developed without any form of software engineering. Korb felt it had suffered for this [16]. The latest implementation effort plans to address issues including maintainability, readability and extensibility. The case study included:

- Observations from a 12 week summer vacation scholarship working on the project
- Observations from weekly CaMML meetings and the CaMML mailing list
- The minutes of CaMML meetings
- A field test of an early version of CaMML
- Reading of literature on CaMML
- Discussion with other first-time users of various versions of CaMML
- Interviews with developers of various versions

CDMS (Core Data Mining Software) CDMS was inspired because "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"[17]. There is a plan to publicly release CDMS along with a manual or technical report. The case study included:

- Observation from CaMML meetings that focused on CDMS
- Observations from informal discussion on CDMS
- CDMS presentations
- Draft Papers and documentation on CDMS
- Interviews with users
- Interviews with developers

The GIFT (GNU Image Finding Tool) The GIFT is a framework for content-based image retrieval (CBIR) systems that developed from the Viper project at the University of Geneva and Circus at EPF Lausanne. One of the main aims of the project was to produce a modular, extensible framework of pluggable components for CBIR, so that research students need only code the component for the aspect they were researching, rather than having to build an entire system each time. As an example of this in practice, a Masters minor thesis student at Monash University recently extended MRML (Multimedia Retrieval Markup Language), that underpins the GIFT, to support query-by-region, a task not possible had it been necessary to build an entire CBIR system.

Although system design documents were not initially produced, a high level abstraction of MRML was provided in the manual "configuring and hacking the GIFT"[18]. This was better explained in diagram form six months later in a thesis[19]. The case study included:

- Interview with one developer
- Two theses by members of the GIFT team
- Papers on GIFT
- GIFT web site
- Interview with a GIFT user, not involved with its development

5 Results

5.1 The nature of research and its needs

A clear and constant aim very seldom exists in academic software development [20–24]. Development is an opportunistic process, not a systematically planned one [25], and evolves over time as the researcher gains knowledge [24, 21]. We start with these assumptions, that from others' experience seem true of most research. We also observe that most research ideas are discarded; it is the exception that something works [22].

When asked about applying software engineering in the research environment, Pressman [24] and Brooks [26], both observed that research software has different goals to industry-developed software and as such would have different requirements. Software engineering research has so far focused mostly on industry. Wallace [20] observed that, in his experience (of over 40 years in the field), little had changed in the way in which people go about research. Pressman [24] suggested any new approach for the research community would need to primarily be "agile".

5.2 The aim of research and researcher

While the aim of research is to advance the state of knowledge, the aim of researchers is often to publish. This takes two forms: academics need to publish journal articles and conference papers to build or maintain their reputation; research students need to complete their theses. Both groups aim to produce

something publishable as quickly as possible [27, 20]. There is usually no incentive to develop robust, extensible and flexible software [27]. Despite this, most would like to start working on a project with a solid foundation and well engineered previous work to support it. Interviews showed that students lost large amounts of time due to obscure coding and lack of documentation by their predecessors. In one case two months were lost due to undocumented preconditions [28]. Perhaps more seriously, high quality research work is very often shelved once the programmers leaves the project. 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 the skewed population of the US survey, we will concentrate on Australian results. These show that an unplanned and non-systematic approach to development dominates (see Figure 1).

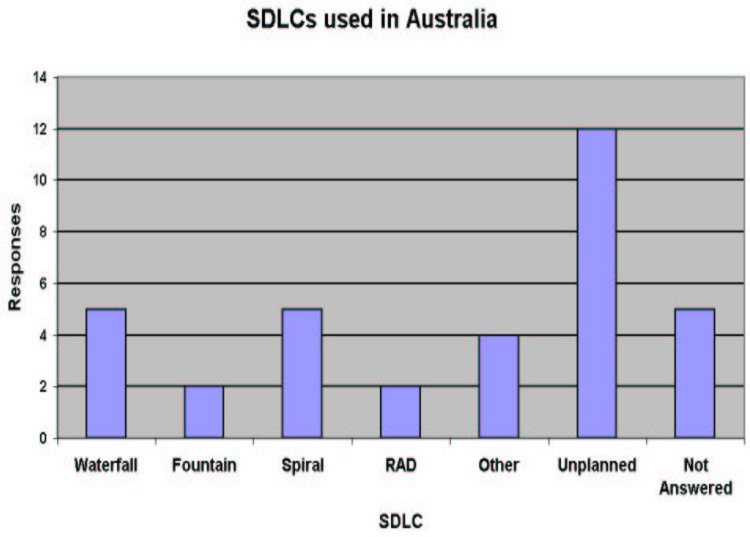

Fig. 1. Software Development Life Cycles used in development of academic research software

Survey participants were asked how regularly they used a variety of software engineering methods. In almost all cases the majority either didn't know the method or knew, but didn't use, it. A noticeable exception was internal commenting, where 57% indicated use. In another question participants were asked why they didn't use the certain methods. The question allows multiple responses from each participant. The results are shown in Table 1, and indicate that many

software engineering techniques are considered inappropriate or too costly for research work.

The number of publications arising from a project was most highly correlated with the its funding and the time spent in planning and development. Other correlations with a high number of publications included the existence of spin-off projects and a large number of software re-writes. This is shown in Table 2. The direction of causation is uncertain. The Australian funding model is significantly based on publication rates, as are some promotional requirements. Publications may attract funding for spin-offs or rewrites.

Interviews indicated that "the primary aim [in development for research] is to get a flaky prototype working sufficiently to get a few statistics out" [27], in order to get publications. After this, the incentive is to "leave it in a half-finished, barely usable, state and go and do something else" [20]. While often to the benefit of the researchers, this approach is clearly not in the best interests of the research or the future researchers who may wish to build on it.

Table 1. Why researchers choose not to use a selection of software engineering methods

Reason	Percent that agree
Never thought about it	14%
Don't know about them	11%
Cost of learning them is too high	17%
They are not appropriate for my work	83%
Cost of using them is higher than the pay off	46%
Organisational Policy against spending time on them	3%

Case studies revealed the difficulty of funding development activity in Australia. When 90% complete and in use as a platform for other projects, CDMS was almost shelved when the students building it completed their PhDs, due to a lack of funding to continue with the development and documentation needed to turn CDMS into a platform usable by the general research community.

Statistical analysis of the Australian results highlighted a correlation between the willingness to document a project after the research was completed and a higher level of use of certain software engineering practices. Of particular interest is the desire to use more software engineering earlier and the use of code and technical reviews (see Figure 2)

The average number of papers produced by research students was correlated to the number of spin-off projects as well as a history of increased software engineering, a desire to use more software engineering earlier and grater use of code reviews (see Figure 3).

Table 2. Number of publications and correlated factors from the Australian survey

Correlated with	Correlation	Degrees of freedom	p value
GrantsValue	0.856	21	0.01
Time in Planning	0.718	29	0.01
Compiler	-0.604	24	0.01
TimeDevelopment	0.541	29	0.01
StartYear	-0.482	29	0.01
Booch Notation	0.658	14	0.05
FountainSDLC	0.547	17	0.05
Technical Reviews	0.547	18	0.05
Rewrites	0.476	28	0.05
Meeting Frequency	-0.424	29	0.05
ProtoTyping	0.416	24	0.05
Spin-off Projects	0.400	29	0.05
UseCases	0.422	19	0.10

Two dominant views appear regarding code reviews: those who know about them, but do not use them (eleven people), and those who sometimes use them (again eleven people). No one indicated they always use them and only one person indicated they use them often. In an interview with Prof. Zukerman the reluctance to use code reviews was explored. It was explained that to review a postgraduate student's work closely might imply mistrust. By the time they become postgraduate students we must expect and trust them to do a good job with the coding [21].

5.4 Documentation and Communication

Interviews indicated that design documents were of use earlier on, but became a burden later projects [29, 17, 27]. In the case of CDMS, diagrams were eventually abandoned: "We were so involved in it that everything was in our heads" [17]. At one point changes were occurring once or twice a week, at this point updating the documents "just got ridiculous" [17]. When development started, class diagrams were found useful for learning about a small amounts of hard-to-follow prototype code. They later became a burden, as code familiarity increased and the information in them was rapidly changing [29, 17, 27]. Eventually they were abandoned.

CDMS was developed using a paired-programming approach. "We very rarely sat down and start coding without discussing the issues first and putting our

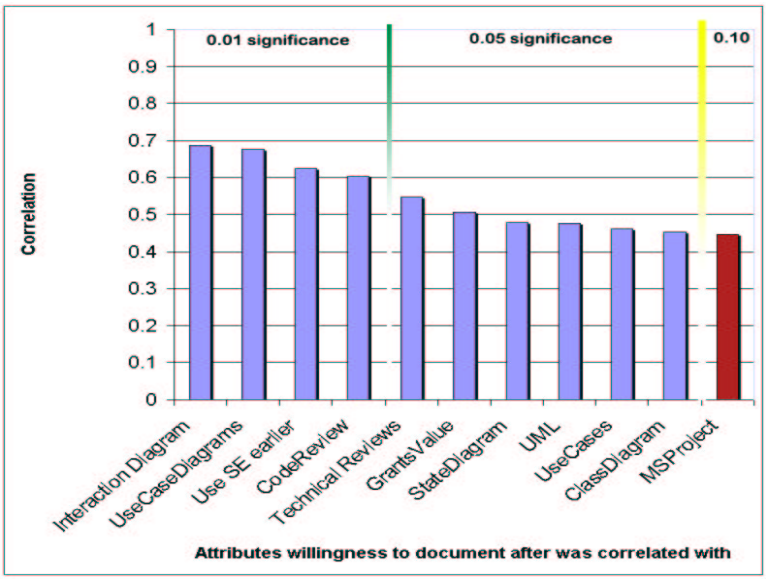

Fig. 2. Correlations between the willingness to document after the research is completed and other factors

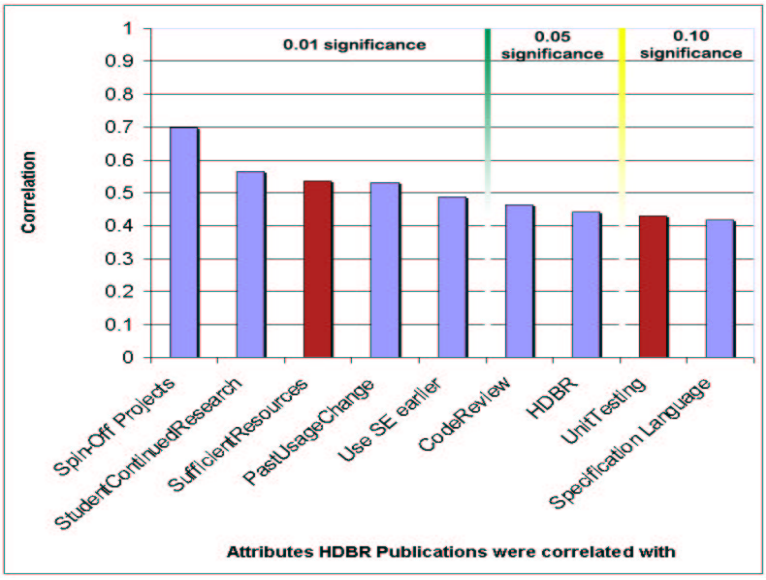

Fig. 3. Correlations between the number of publications by research students and other factors from the Australian Survey

ideas up on the whiteboard ... This goes for big and small things. Its interesting and scary the impact that small little issues can have on the system as a whole. Even the smaller items we bounce off each other" [17]. This sort of communication and teamwork was found to be much more effective than written documentation [29]. "I think we get heaps of benefit working side by side" [17].

Regarding the CDMS User Manual, "We're not sure how this will happen. We were sort of hoping it would happen by magic or delivered by a stork" [27]. CDMS has a high publication potential, but so far it isn't stable enough to write about and there are other more pressing things (like PhD theses) requiring attention [29].

In the case of CaMML, completed design documents were ignored by research students. In an effort to start coding immediately, they opted to simplify the design. The resulting design was less general and hence less extensible. A gap appeared between design documents and code: "We were being typical computer scientists and coding without any specification, ah, with a bit of specification" one researcher remarked referring to design documents that were casually glanced over [30].

In discussion with a user of GIFT not involved with its development, it was found that the MRML manual was helpful and provided the first point of call. It was sometimes necessary, however, to consult the authors (via e-mail), or their theses. The user's decision to use GIFT was based on an expected time saving and its platform-independence [31].

6 Discussion

An unplanned approach to developing research software is widespread. There is an incentive, at least in Australia, for researchers to stop working on projects once a prototype has been developed and results are available for publication. Many software engineering tools are rejected as inappropriate, or of greater cost than benefit. In academia it appears that Royce's [7] two-step approach, "Analysis" leading into "Coding" is still prevalent.

In the past, difficulties of distribution, hardware dependence and computing cost limited the likelihood of another researcher building on existing code. This is no longer true. Distribution via the internet is painless, computing power is cheap and source code is often hardware independent. Research coding for one user with one purpose is now outdated.

The Spiral model is still a dominant approach among users of formal SDLCs. While this works well for industry, research often stops at the creation of the first prototype. As research often has no clear goal, risk assessment makes little sense. An evolutionary approach is needed, but it must take a longer-term view than that used by industry.

Krueger's observation that reuse seldom occurs is particularly true in academic development. Slight adjustments to past work often require new researchers to start from scratch. The need for reuse was the inspiration behind both the GIFT and CDMS. Devos and Tilman's 1996 observation that reuse should be

factored in earlier is supported by the success of the GIFT. The GIFT's original aims, to enable greater reuse and sharing of resources, have lead to development of an easily adaptable platform, that has in turn made more research goals achievable.

Humphrey's 1998 paper did not address the development ethic of research students. We have shown it to be much the same as what Humphry found for undergraduate students. From the perspective of an individual researcher, the view that software engineering has greater cost than benefit, may well be accurate. The benefit is received by future researchers. Well-engineered code and decent documentation can provide a better introduction to the research and reduce the wasted effort spent recoding existing work before building upon it. This benefit to future researchers and the research community was demonstrated in the GIFT case study.

The suggestion that the evolvability of software is based on code quality, the evolution process and the organisational environment in which it takes place (Cook, Ji and Harrison, 2000) is supported by this research. Both technical reviews and code reviews were associated with projects that had larger numbers of spin-off projects, that is, more evolution. A willingness to document after the project, that is to follow through with Cook, Ji and Harrison's third step towards evolvability, "stabilising" the development, was associated with a higher level of software engineering and again, with the presence of technical and code reviews.

7 Preliminary conclusion

Computer Science research in universities has conflicting aims. The key conflict is between the desire to complete research fast and to extend and further develop the field. One encourages a "quick and dirty" approach, the other require a significant amount of planning and engineering. An effort to create more mature research, can increase the number of resultant publications. However there is an initial cost that most researchers cannot afford. On the other extreme, the occasional over-use of industrial-style software engineering adds an additional (large) burden to development, that may have far greater cost than benefit.

In order to assist, rather than hinder, the research effort, the bulk of software engineering should take place after the research is over. Once a project has come to completion, be that because a student's PhD is complete, or because funding has for the moment run out, then the project should be re-engineered and stabilised to enable future researchers to build upon it.

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. The RAISER/RESET approach, a solution to this dilemma, is now presented.

8 A new SDLC

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. Academia has its own requirements and requires a custom approach.

9 A two part approach - Research and Development

Research must contain an element of 'discovery'. The concepts and ideas in the researcher's mind are subject to constant review and change. A development process for software to aid research must likewise be flexible and able to change rapidly [24]. Ideally the development process itself should assist the researcher and provide feedback leading to new research directions. The real value is the idea, and the burden of implementation must be minimized as much as possible during the research phase.

Development is the ongoing maintenance phase of a research project. New algorithms and functionality (i.e. research ideas) should not be added during this time. The development cycle is a restructuring and documenting phase. It aims to provide the research product with 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 generally make the product useable as a tool by other researchers or the public.

The Raiser/Reset approach (see Figure 4) divides work into research and development cycles. The aim is to ensure that the two tasks, with very different aims, both 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 and as a result frees up research students to do research. With stronger "encouragement" from both universities and government for research students to complete in minimum time, such a transfer may in time become a necessity rather than merely a benefit.

9.1 The RAISER / RESET SDLC

The SDLC is divided into two halves and three processes. The top "RAISER" half is where research takes place, while the bottom "RESET" half is where development takes place. A distinction is made between initial research and follow-on research. As indicated by the multiplicity, there can be more than one "initial research project" that goes into a single development phase. Likewise, a development phase that produces a stable piece of software may spawn many new "follow-on research" projects. The different processes will now be described.

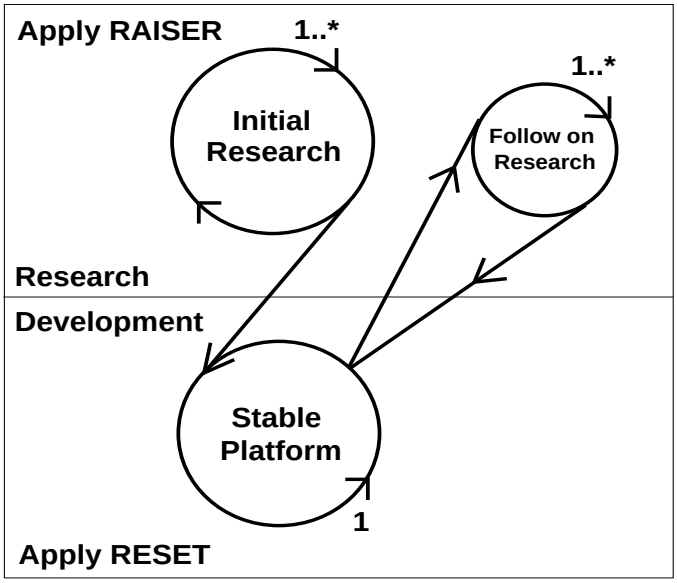

Fig. 4. The RAISER / RESET SDLC

The initial research Research software typically begins with an idea. For the computer scientist it is natural to convert that idea into an algorithm. The algorithm is coded, considered, tested, changed, recoded, etc. and eventually the research comes to a close and papers or theses are written. This approach has a number of flaws, but the most important is that at this point most projects are only half-done. Completing the project will usually not produce more papers, but it may have an impact on the life of the software. In many cases without further work in the short term, the software will cease to be evolvable. Our research suggests that long-term projects produce more papers and more valuable software. After the initial research, project development is needed to clean up the code produced by exploratory programming and produce design and user documentation. This is the "stabilisation" step referred to by Cook, Ji and Harrison [13]. Stabilisation not only makes the work more usable and "evolvable", but also lowers the barrier facing new researchers wanting to extend the work.

A Stable Platform Once the research has come to an end, work may continue in the development phase. One or more researchers may become advisors during development. The researchers ensure the scientific integrity of the altered product and can clarify parts of their code. This is similar to the role of a developer in a code review session. The "resetting" of the code should involve a number of formal technical reviews. As a result of the research process much of the code may initially be difficult to follow. The critical factor when developing from this sort of code (as the EDRC conference pointed out) is access to the

researchers. The development phase should therefore start as soon as research is completed. It aims to remove issues resulting from exploratory coding and leave readable, documented, modular and reusable code. Development should take place between research projects and should not add any significant additional burden to the researchers.

Follow-on Research Follow-on research extends or improves the existing research. It may involve the development of a new algorithm to replace part of the existing code, extend the code to handle new data types (e.g. CaMML), or add new functionality to complement existing research (e.g. the GIFT). Irrespective of the type of follow-on research, the researcher should work with a stable platform rather than the initial research. Large amounts of time are wasted trying to understand other researchers' code, but well-structured, readable code and a decent level of documentation can greatly reduce this.

Further development New research may be incorporated into existing projects during the development phase. Follow-on research may be based on past follow-on research, but this should not be done directly: a restabilised version should be used.

RAISER The aim of RAISER (Reactive Assisted Information Science Enabled Research) is to reduce the burden of software engineering, particularly design work, to a level where it does not interfere with research, yet still raise the evolvability and readability of research code. The four features of RAISER are described below.

1. Reactive
   RAISER is a reactive process rather than a proactive one. Changes to the ideas or algorithm will cause change to code and/ or approach. As the project changes, so too may the software engineering tools being used. For example, class diagrams may initially be of use, but may be dropped once they are no longer of benefit to the researchers.
2. Assisted
   The code and methodology are there to assist the researcher not burden them.
3. Information Science Enabled
   This stresses the theoretical research that is behind the software development. It is thought by some experts to be a more meaningful name than Computer Science [20]. Research not enabled by experimental and theoretical information science may benefit from a heavier approach to software engineering than RAISER.
4. 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 the RAISER phase should meet these requirements. Such tools 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, while 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 or suggestions for replacing the module where appropriate
- Configuration Management, for example CVS, should be used
- High-level design such as a class diagram or higher-level DFD should be used (rather than detailed design). While it is useful the diagrams should be updated.
- At least two people should work on a project. They will check each others' code for readability and request clarifying documentation when anything is unclear to them.
- A work schedule taking account of papers to be written should be created.

RESET As software developed during the RESET phase is based on research software, it will differ somewhat from the usual development cycle. The key difference is the existence of a "research prototype". There should also be access to the one or more of the developers of the prototype. The software itself is likely to have certain characteristics, as a result of exploratory research and experimentation. The RAISER approach will also have an impact. The features of RESET are:

1. Research Enabled
   As the software is based on a research it might not fit any common design patterns. There may be opportunity for software engineering research to abstract new design patterns. The work is also likely to be challenging and vary greatly between projects.
2. Software Engineering
   The development task is largely one of software engineering. The prototype must be cleaned up and remoulded. Existing functionality should not need to change. Rather, code will need restructuring for interface and robustness improvements. The software may also be more thoroughly tested and debugged. Regression testing may be configured for future use. 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.
3. Techniques
   The RESET process must vary depending on the level of software engineering employed in the RAISER phase. Research code can often be the most difficult code to work with.

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 need
- 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 (taken from the researchers' comments), should be produced.

9.2 Implementation

The approach would be best supported by an in-house academic software development laboratory. This laboratory need only have a few permanent staff. Research assistants currently doing stabilising/development work on isolated cases could be seconded to the laboratory where proper training and emphasis on the valuable (to researchers) aspects of Software Engineering can be provided. In addition, interested staff should be encouraged to take a six month sabbatical in the laboratory where they could not only improve and update their programming skills, but would gain detailed knowledge of the department's other research. This would allow new syntheses. The researchers could add their expert knowledge not only of their field, but also of their colleagues needs. This would greatly increase the effectiveness of the laboratory.

Although these recommendations will cost money in the short term, over the longer term time savings may recoup this. In addition, we hypothesise that they would increase the job satisfaction of researchers, decrease the time required to obtain publishable results, allow for an increase in total departmental publication rates, provide organised skill- updating for staff, produce useful products others may adopt and finally, they would allow more research students to complete significant research in a timely fashion.

10 Conclusion

In this paper we have investigated the nature of research in computer science. We have looked at the way software engineering is currently employed (or often not employed). We have shown that isolated development of research software useable only by the programmer is no longer the best approach for research, yet is often used. Finally we have presented a new SDLC and approach to software development aimed specifically at the research development environment, and suggested how this approach might be adopted by an institution. To recap our findings: 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.: Why don't they practice what we preach? Annals of Software Engineering **1** (1998) 201–222
2. Steier, D., Coyne, R., Subrahmanian, E.: 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 EDRC 05-69-93, Carnegie Mellon University (1993)
3. Robillard, P.N., Robillard, M.P.: Improving academic software engineering projects: A comparative study of academic and industry projects. Annals of Software Engineering **6** (1998) 343–363
4. Turing, A.: On computable numbers, with an application to the entscheidungsproblem. Proceedings of the London Mathematical Society **42** (1936) 230–265
5. Naur, P., Randell, B., eds.: Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968, Scientific Affairs Division, NATO (1969)
6. Randell, B., Buxton, J., eds.: Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 Oct. 1969, Scientific Affairs Division, NATO (1970)
7. Royce, W.W.: Managing the development of large software systems: Concepts and techniques. In: 1970 WESCON Technical Papers. Volume 14., Los Angeles, Western Electronic Show and Convention (1970) 1–9 (Reprinted in Proceedings of the Ninth International Conference on Software Engineering, Pittsburgh, PA, USA, ACM Press, 1989, pp.328–338.).
8. Press, W.H., Teukolsky, S.A.: Numerical recipes: does this paradigm have a future? Computers In Physics **11** (1997) 416–425
9. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Computer **21** (1988) 61–72
10. Denning, P.J., Comer, D.E., Gries, D., Mulder, M.C., Tucker, A., Turner, A.J., Young, P.R.: Computing as a discipline. Communications of the ACM **32** (1989) 9–23
11. Krueger, C.W.: Software reuse. ACM Computing Surveys **24** (1992) 131–183
12. Devos, M., Tilman, M.: Object oriented and evolutionary software engineering. WS18 workshop, OOPSLA conference, San Jose (1996)
13. Cook, S., Ji, H., Harrison, R.: Software evolution and software evolvability. Working paper, University of Reading, UK (2000)
14. Patton, M.: Qualitative Evaluation and Research Methods, 2nd edn. Newbury Park, California: SAGE (1990)
15. Wallace, C.S., Korb, K.B.: Learning linear causal models by MML sampling. In Gammerman, A., ed.: Causal Models and Intelligent Data Management. Springer-Verlag (1999)
16. Korb, K.: Interview with Dr Kevin Korb. Interview conducted for the USE CSR project (2002)
17. Comley, J.: Interview with Josh Comley. Interview conducted for the USE CSR project (2002)
18. Müller, W.: Configuring and hacking the GIFT. GNU (2000) Download from the GIFT CVS archive at http://savannah.gnu.org/cgi-bin/viewcvs/gift/gift/Doc/.
19. Müller, W.: Design and implementation of a flexible Content Based Image Retrival framework. Phd, Université de Genève (2001)

20. Wallace, C.: Interview with Prof Chris Wallace. Interview conducted for the USE CSR project (2002)
21. Zukerman, I.: Interview with Prof Ingrid Zukerman. Interview conducted for the USE CSR project (2002)
22. Meyer, B.: Interview with Prof Bertrand Meyer. Interview conducted for the USE CSR project (2002)
23. Farr, G.: Interview with Dr Graham Farr. Interview conducted for the USE CSR project (2002)
24. Pressman, R.: Re: SE practise in Comp. Sci. Research. personal communication, Email to Andre Oboler (2002)
25. Waite, W.: Re: Use of software engineering in computer science research. personal communication, Email to Andre Oboler (2002)
26. Brooks, Jr, F.P.: Re: Use of software engineering in computer science research. personal communication, Email to Andre Oboler (2002)
27. Alison, L.: Interview with Dr Lloyd Alison. Interview conducted for the USE CSR project (2002)
28. Wilkin, T.: Interview with Tim Wilkin. Interview conducted for the USE CSR project (2002)
29. Fitzgibbon, L.: Interview with Leigh Fitzgibbon. Interview conducted for the USE CSR project (2002)
30. Hope, L.: Interview with Lucas Hope. Interview conducted for the USE CSR project (2002)
31. Lim, S.: Re: [use csr] gift software. personal communication, Email to Andre Oboler (2002)