ACSE 2003

3rd International Workshop on Adoption-Centric Software Engineering

ICSE 2003 IEEE/ACM International Conference
on Software Engineering
Portland, Oregon
May 3-11, 2003

editors:
Robert Balzer, Teknowledge Corporation
Jens-Holger Jahnke, University of Victoria
Marin Litoiu, IBM Canada Ltd.
Hausi A. Müller, University of Victoria
Dennis B. Smith, Software Engineering Institute
Margaret-Anne Storey, University of Victoria
Scott R. Tilley, Florida Institute of Technology
Kenny Wong, University of Alberta
Anke Weber, University of Victoria

June 2003

SPECIAL REPORT
CMU/SEI-2003-SR-004

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

ACSE 2003

3rd International Workshop on Adoption-Centric Software Engineering

ICSE 2003
IEEE/ACM International Conference on Software Engineering
Portland, Oregon
May 3-11, 2003

CMU/SEI-2003-SR-004

editors:
Robert Balzer, Teknowledge Corporation
Jens-Holger Jahnke, University of Victoria
Marin Litoiu, IBM Canada Ltd.
Hausi A. Müller, University of Victoria
Dennis Smith, Software Engineering Institute
Margaret-Anne Storey, University of Victoria
Scott R. Tilley, Florida Institute of Technology
Kenny Wong, University of Alberta
Anke Weber, University of Victoria

June 2003

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

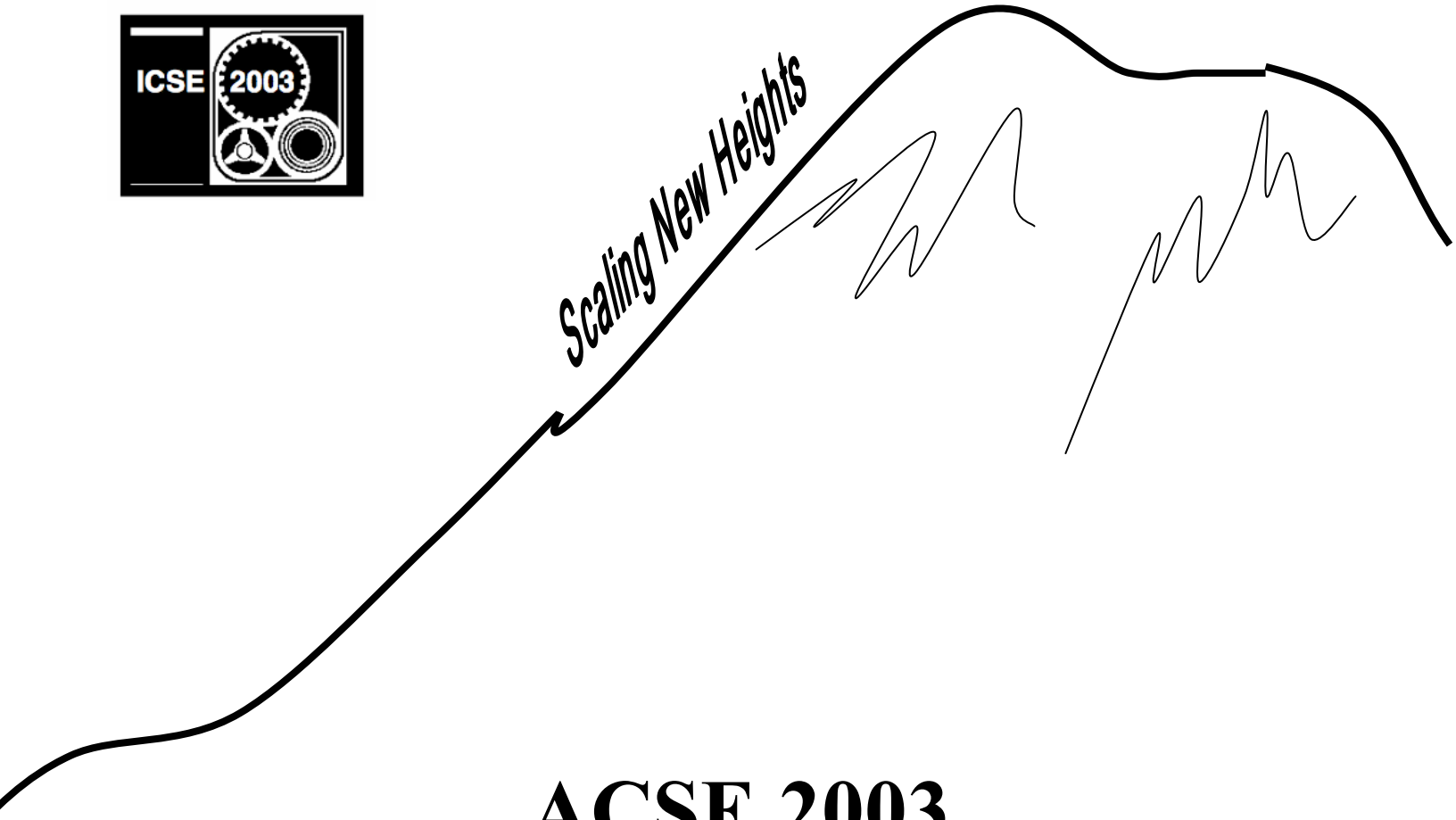
FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).



ACSE 2003
3rd International Workshop on Adoption-Centric Software Engineering

ICSE 2003
IEEE/ACM International Conference
on Software Engineering
Portland, Oregon
May 3-11, 2003

ACSE 2003 PROCEEDINGS

3rd International Workshop on Adoption-Centric Software Engineering

9 May 2003 • Portland, Oregon, USA

Workshop at ICSE 2003
25th IEEE/ACM International
Conference on Software Engineering

Table of Contents

LESSONS LEARNED: Case Studies and Experiences

Abstract

This report contains a set of papers that were presented at the Third International Workshop on Adoption-centric Software Engineering (ACSE). The papers focused on overcoming barriers to adopting research tools. Such barriers include the user's lack of familiarity with the tools, the mismatch between the tools and the users' cognitive models, a lack of interface maturity, limited tool scalability, poor interoperability and limited support for complex software engineering development tasks. The workshop papers explored innovative approaches to the adoption of software engineering tools and practices in particular by embedding them with middleware products and other commonly available commercial products.

Organizing Committee

Dr. Robert Balzer, Teknowledge Corporation, USA
After several years at the Rand Corporation, Dr. Balzer left to help form the University of Southern California's Information Sciences Institute (USC-ISI) where he served as Director of ISI's Software Sciences Division and Professor of Computer Science at USC. In 2000 he joined Teknowledge Corporation as their CTO and Director of their Distributed Systems Unit, which combines AI, DB, and SE techniques to automate the software development process. His current research includes wrapping COTS products to provide safe and secure execution environments, extend their functionality, and integrate them together; instrumenting software architectures; and generating systems from domain specific specifications.

Dr. Jens-Holger Jahnke, University of Victoria, Canada
Dr. Jahnke is an Assistant Professor at the University of Victoria, Canada. He holds a doctoral degree (summa cum laude) from the University of Paderborn, Germany. He received the E. Denert Software Engineering Award in 2000 and has been appointed an Industrial Research Fellow by the Advanced Systems Institute of British Columbia. He is a Principal Investigator of the Consortium for Software Engineering Research (CSER). His current research focuses on network-centric aspects of software engineering, in particular system mediation, system reverse engineering, embedded systems, data reengineering, and connection-based programming.

Dr. Marin Litoiu, IBM Canada Ltd., Canada
Dr. Litoiu is member of the Centre for Advanced Studies at the IBM Toronto Laboratory where he initiates and manages joint research projects between IBM and Universities across the globe in the area of Application Development Tools. Prior to joining IBM (1997), he was a faculty member with the Department of Computers and Control Systems at the University Politechnica of Bucharest and held research visiting positions with Polytechnic of Turin, Italy, (1994 and 1995) and Polytechnic University of Catalunia (Spain), and the European Center for Parallelism (1995). Dr. Litoiu's other research interests include distributed objects; high performance software design; performance modeling, performance evaluation and capacity planning for distributed and real time systems.

Dr. Hausi A. Müller, University of Victoria, Canada
Dr. Müller is a Professor at the University of Victoria, Canada. He is a Visiting Scientist with the Centre for Advanced Studies at the IBM Toronto Laboratory and the Carnegie Mellon Software Engineering Institute. He is a principal investigator of CSER. Together with his research group he investigates technologies to build adoption-centric software engineering tools and to migrate legacy software to object-oriented and network-centric platforms. Dr. Müller's research interests include software engineering, software evolution, reverse engineering, software reengineering, program understanding, software engineering tool evaluation, and software architecture. He is GC for IWPC-2003. He was GC for ICSE-2001.

Dr. Dennis B. Smith, Carnegie Mellon Software Engineering Institute, USA
Dr. Smith is a senior member of the technical staff in the Product Line Systems Program at the Software Engineering Institute. He is the technical lead in the effort for migrating legacy systems to product lines. In this role he has integrated a number of techniques for modernizing legacy systems from both a technical and business perspective. Dr. Smith has been the lead in a variety of engagements with external clients. He led a widely publicized audit of the FAA's troubled ISSS system. This report produced a set of recommendations for change, resulting in major changes to the development process, and the development of an eventual successful follow-on system. Earlier, Dr. Smith was project leader for the CASE environments project. This project examined the underlying issues of CASE integration, process support for environments and the adoption of technology. He is also a co-editor of the IEEE and ISO recommended practice on CASE Adoption. He has been general chair of two international conferences, IWPC'99 and STEP'99.

Dr. Margaret-Anne Storey, University of Victoria, Canada
Dr. Storey is an Assistant Professor at the University of Victoria. Her main research interests involve understanding how people solve complex tasks, and designing technologies to facilitate navigating and understanding large information spaces. With her students and she is working on a variety of projects within the areas of software engineering, human-computer interactison, information visualization, social informatics and knowledge management. Dr. Storey is a fellow of the ASI and as susch collaborates with the IBM PDC on HCI issues for eCommerce and distributed learning applications, and with ACD systems. She is a principal investigator for CSER developing and evaluating software migration technology and a visiting researcher at the IBM Centre for Advanced Studies.

Dr. Scott R. Tilley, Florida Institute of Technology, USA
Scott Tilley is an Associate Professor at the Florida Institute of Technology. He is also Principal of S.R. Tilley & Associates, a Southern California-based information technology consulting boutique. He maintains an appointment as Visiting Scientist with the Software Engineering Institute at Carnegie Mellon University. He was PC Chair for SIGDOC 2001, and is GC of the WSE 2003.

Dr. Kenny Wong, University of Alberta, Canada
Ken Wong is an assistant professor at the University of Alberta. His main areas of research are software architecture, integration, evolution, and visualization. This research includes conducting case studies, building and using integrated environments for reverse engineering, and exploring a framework for continuous, collaborative program understanding. Current industrial collaborations include IBM, KLOCwork Inc., and Intuit Canada. He is a principal investigator of CSER and ASERC. He co-manages a Canadian Foundation for Innovation facility to study distributed software development, with connected, experimental laboratories at the University of Calgary and University of Alberta. Dr. Wong is also PC Chair for IWPC 2003 and WSE 2003.

3rd International Workshop on Adoption-centric Software Engineering ACSE 2003

http://www.acse2003.cs.uvic.ca

Robert Balzer,[1] Jens Jahnke,[2] Marin Litoiu,[3] Hausi A. Müller,[2]
Dennis B. Smith,[4] Margaret-Anne Storey,[2] Scott R. Tilley,[5] Ken Wong[6]
[1]*Teknowledge Corporation, USA;* [2]*University of Victoria, Canada*
[3]*IBM Canada Ltd., Canada;* [4]*Carnegie Mellon Software Engineering Institute, USA*
[5]*Florida Institute of Technology, USA;* [6]*University of Alberta, Canada;*
balzer@teknowledge.com, jens@cs.uvic.ca, marin@ca.ibm.com, hausi@cs.uvic.ca,
dbs@sei.cmu.edu, mstorey@cs.uvic.ca, stilley@cs.fit.edu, kenw@cs.ualberta.ca

Abstract

The key objective of this workshop is to explore innovative approaches to the adoption of software engineering tools and practices—in particular by embedding them in extensions of Commercial Off-The-Shelf (COTS) software products and/or middleware technologies. The workshop aims to advance the understanding and evaluation of adoption of software engineering tools and practices by bringing together researchers and practitioners who investigate novel solutions to software engineering adoption issues.

1. Workshop theme and goals

Understanding adoption of software engineering tools and practices is critical for the software and information technology sectors, which are continually challenged to increase their productivity. Recent advances in effective standards and interfaces for tool extension and customization have opened new research avenues, which allow software engineering tools and technologies to be incorporated into commonly used Commercial Off-The-Shelf (COTS) products and middleware platforms and adopted as extensions of those COTS products.

The key objective of this workshop is to explore approaches where software engineering tools and practices are implemented as extension of COTS software products and middleware technologies that work in conjunction with software engineering tools as well as mined components. The workshop aims to advance the understanding and evaluation of adoption of software engineering tools and practices.

Research tools in software engineering often fail to be adopted and deployed in industry. Important barriers to adopting these tools include the user's lack of familiarity with these tools, their mismatch with the users' cognitive models, their lack of interface maturity, their limited scalability, their limited support for complex work products of software development, their poor interoperability, and their limited support for the realities of system documentation engineering. Developing and deploying innovative research tools and ideas as extensions to modern, commonly used platforms may ease these barriers.

2. How can the workshop advance software engineering research in practice?

One key problem in software engineering research is the integration of research tools into industrial software development processes. Tools developed by the software engineering research community often remain orphans due to adoption problems since research tools are rarely built for an industrial setting. Developing effective techniques and strategies to overcome this problem is timely and will have great value to the software and information technology sectors. Injecting more of the leading-edge software engineering research results into industrial practice has a potentially significant impact on the production of quality software. Thus, this research addresses three diverse markets: the software developers, who need to understand and document existing software systems, the researchers, who want to inject and validate their research tools in industrial development processes, and the tool users, who want to leverage their personal work environment in software engineering tools.

3. Background and related work

The notion of building software systems from existing building blocks, components, or parts has been around since the Sixties. Communities, such as Software Reuse,

Commercial-of-the Shelf Components (COTS), or CBSE (Component-Based Software Engineering) have investigated many approaches and developed effective solutions to this problem. Their approaches differ in many aspects, including the granularity, genericity, or wrapping of the components. The goal of ACSE is to take a significant step back and approach this perennial problem from a radically different perspective. The idea is to select the host components according to a variety of *adoption criteria.*

Shaw observed that systems, such as interactive graphics applications, devote less than 10% of their code to the overt function of the system and more than 90% to the user interface [1]. Reiss leveraged FrameMaker, a COTS editor, for all editing aspects in his Desert software development environment [2]. Sullivan, Knight and Coppit use the term Package-Oriented Programming (POP) to support tasks, such as document embedding and scripting [3, 5]. The Software Bookshelf, built on top of Netscape, exploits the familiar Web interface [4].

In several recent conference keynotes, Balzer has advocated that software engineering researchers should exploit large COTS products for building software engineering tools rather than constructing stand-alone tools [7, 9]. Egyed and Balzer proposed an integration-architecture for COTS products, which provides access and visibility into the document information contained within a COTS tool [6]. The information that is shared with external tools allows users to track user actions and provide analysis and automation services for the user within the COTS tool. For example, they exploited this architecture to add semantics to PowerPoint diagrams and Word documents and to build defenses against malicious e-mail attachments.

The mandate of the Technology Transition Practices (TPP) group at Software Engineering Institute (SEI) is to identify, develop, promote, and apply *practices* that result in more rapid, affordable, and sustained transition of innovative software engineering technologies [10].

The hypothesis of the ACRE project [8] is that developers will more likely adopt tools that leverage the *cognitive support* and *interoperability mechanisms* of tools they use daily and know intimately (e.g., Lotus Notes, Office XP, or StarOffice). To increase their productivity, developers accumulate sripts, macros, and shortcuts in their personal work environment. Reusing such hard-won and treasured cognitive support features is a central idea of this project.

Beyond user-to-tool compatibility concerns of cognitive support, there must also be tool-to-tool compatibility. Towards this end, interoperability mechanisms for data, control, and presentation integration are critical factors to tool adoption. Recently developed middleware standards and technologies can offer unprecedented possibilities to seamlessly integrate new research tools into existing, familiar environments.

Since it is difficult to evaluate tool adoption in the course of a research project, ACRE concentrates on investigating how to leverage cognitive support and interoperability mechanisms from COTS products for software exploration and visualization tools.

4. References

[1] M. Shaw, "Prospects for an Engineering Discipline of Software," *IEEE Software*, 7(6):15-24, Nov. 1990.

[2] S.P. Reiss, "Simplifying Data Integration: The Design of the Desert Software Development Environment," *Proc. 18th IEEE/ACM Int. Conf. on Software Engineering,* Berlin, Germany, pp. 398-407, March 1996.

[3] K.J. Sullivan and J.C. Knight, "Experience Assessing an Architectural Approach to Large-Scale, Systematic Reuse," *Proc. 18th IEEE/ACM Int. Conf. on Software Engineering,* Berlin, Germany, pp. 220-229, March 1996.

[4] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, "The Software Bookshelf," *IBM Systems Journal,* Vol. 36, No. 4, pp. 564-593, Nov. 1997

[5] D. Coppit and K.J. Sullivan, "Multiple Mass-Market Applications as Components. *Proc. 22nd IEEE/ACM Int. Conf. on Software Engineering*, Limerick, Ireland, pp. 273-82, June 2000.

[6] A. Egyed and R. Balzer, "Unfriendly COTS Integration: Instrumentation and Interfaces for Improved Plugability," *Proc. 16th IEEE Int. Conf. on Automated Software Engineering (ASE 2001),* San Diego, USA, pp. 223-231, Nov. 2001.

[7] R. Balzer, "Tolerating Inconsistency Revisited," *Proc. 23rd IEEE/ACM Int. Conf. on Software Engineering (ICSE 2001),* Toronto, Canada, p. 665, May 2001.

[8] H.A. Müller, M.-A. Storey and K. Wong, "Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE)," CSER Research Proposal, Nov. 2001. www.acse.cs.uvic.ca

[9] R. Balzer, "Living with COTS," 24th IEEE/ACM Int. Conf. on Software Engineering (ICSE 2002), Orlando, USA, p. 5, May 2002 and 2nd Int. Conf. on COTS-Based Software Systems, Ottawa, Canada, Feb 2003.

[10] Technology Transition Practices (TPP) Group, Software Engineering Inst., 2002. www.sei.cmu.edu/ttp/

On the Challenges of Adopting ROTS Software

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Shihong Huang
Department of Computer Science
University of California, Riverside
shihong@cs.ucr.edu

Tom Payne
Department of Computer Science
University of California, Riverside
thp@cs.ucr.edu

Abstract

One of the reasons why research tools often remain lab orphans is that it is so difficult for third parties to adopt the solution and make efficient use of it in their own work. This paper outlines some of our experiences in adopting research-off-the-shelf (ROTS) software in the application domain of optimizing compilers. While it is true that there are always difficulties using prescribed solutions in complex applications, there are unique challenges inherent in ROTS software. These include understandability (e.g., a lack of high-quality documentation), robustness, (e.g., an implementation that is not quite ready for prime time) and completeness (e.g., a partial solution due to an implicit focus on getting "just enough" done to illustrate the feasibility of a solution, rather than going the "last mile" to bring the prototype to market). In this context, we offer several recommendations meant to address the challenges of adopting ROTS software.

Keywords: *adoption, research-off-the-shelf (ROTS) software, optimizing compilers, frameworks*

1. Introduction

The problems associated with technology transition and adoption are many and manifold. In our opinion, adoption is one of the most important, yet perhaps least appreciated, areas of interest in academic computer science circles. The importance of managing adoption issues in the context of software engineering is attested to by Carnegie Mellon's Software Engineering Institute: "Technology Adoption" is one of their three main top-level focus areas (the other two being "Management Practices" and "Engineering Practices") [6]. Indeed, it can be argued that "transitionability" as a quality attribute should receive more emphasis in most software projects [8].

In the book *Crossing the Chasm* [5], Geoffrey Moore describes the challenges in bridging the gap between two groups. The first group is the early adopters of new and promising technology. The second group is the vast majority of people who are part of the mainstream market that will wait until the technology is proven, not just promising. In the context of software produced in an academic or research setting, which we call "research-off-the-shelf" (ROTS) software, this problem is very much present, albeit in a slightly different form.

The early adopters of ROTS software are usually other academics. For example, the next generation of graduate students who will continue the work begun by their supervisor or by the previous students who have completed their degrees. The mainstream market for such ROTS software may therefore not necessarily be the general public (as is the case with a commercial application), but rather other groups and labs in related communities.

However, truly widespread dissemination (while rarer) is still possible – and potentially very lucrative when it does occur. Since 1980, American universities have spun off more than 2,200 startups whose sole purpose is to commercialize results that began in research labs, resulting in a contribution of over $40B annually to the U.S. economy [10]. Indeed, for research areas that are more applied, such as software engineering, broad adoption is often an important (long term) goal of the project. For example, it may be a measure of success for the results of an academic project to be adopted by an industrial partner and used on a regular basis. It is therefore critically important for people involved in applied software engineering research to be cognizant of some of the challenges that they will face when it comes to convincing others to adopt their results.

Unfortunately, such awareness is not the norm. In the continuum of technology transition phases, "adoption" is the fourth phase (after "contact", "understanding", and "trial use", and before the last phase of "institutionalization") in a product's acceptance [7]. Examples of transition mechanisms that are applicable to the adoption phase are handbooks, third-party case studies, and quantitative data. For many research projects involved in producing ROTS software, such transition mechanisms are rarely addressed.

The next section discusses some of the challenges that are intrinsic to the problem domain of ROTS software. Section 3 discusses more prominent challenges that directly affect the adoption of ROTS software by

other users. Section 4 provides a brief set of recommendations that might help alleviate these problems. Finally, Section 5 summarizes the paper and outlines possible avenues for further work in the area.

2. Challenges with the Problem

By their very nature, the problems that academics and researchers work on are complex. Otherwise, there would be little interest in working on the problem in the first place. This gives rise to unique challenges with the problem domain – challenges that have direct impact on adoption challenges with the corresponding ROTS solution. For example, the problem being addressed may be so removed from the current needs of potential users that any solution to this problem will have great difficulties in being adopted (at least in the short term). Christensen refers to this phenomenon as "Principle #4: Technology Supply May Not Equal Market Demand" in the book *The Innovator's Dilemma* [3].

By definition, complex subjects require a deep understanding of the key issues pertaining to the specific problem being studied. For most computer topic areas, the amount of secondary and tertiary knowledge required to properly master a modern topic is quite significant. Consider bioinformatics: a researcher working in this area needs to be adept at computer science topics (such as algorithm analysis and design, complexity theory, and search strategies), biology topics (such as understanding the structure of DNA, gene sequencing techniques, and bio-chemical evolution of cell material), and have the ability to relate one area to another with ease.

For those working in software engineering, the scale and scope of today's problems is equally challenging. In fact, it has been said that progress in the area can only be made by those who are multi-specialists [10]. This means someone who is skilled in the underlying areas of computer science, engineering discipline, and information technology, and adept at moving between the three of them.

Consider our own experience performing research and teaching courses in the area of optimizing compilers. At the graduate level, students are expected to have already mastered the basics of compiler technology (which is no small feat in itself). The focus of the second compiler design course is on developing algorithms for various types of program optimization, such as dead code elimination, and then implementing and evaluating the efficacy of these algorithms. Just understanding the theory behind some of these code optimizations is quite challenging; properly implementing them is extremely challenging (especially in a time-constrained ten-week quarter).

Any project to implement various optimizations requires a framework in which to test those routines. This framework must be capable of creating symbol tables and lists of intermediate instructions. It must also be able to display the optimized intermediate code for comparison with the original. As part of a Master's thesis, one of our graduate students developed the RIF (Riverside Intermediate Format), a portable, human-readable, and machine-processable format for medium-level program representation [9]. It is based on C-- (pronounced "see minus minus"), a close subset of the C programming language. Since it is based on C, most programmers can quickly learn the syntax and semantics of the C-- language. Moreover, existing tools such as the Gnu C compiler (gcc) can be used to process the C-- source, thereby leveraging the investment made in existing toolsets. Nevertheless, there are considerable challenges of adopting the RIF as ROTS software by other students for their own use in the class projects. The next section outlines some of these challenges.

3. Challenges with the Solution

Developing software solutions in an academic setting is both qualitatively and quantitatively different than developing similar solution in a commercial setting. These differences lead to numerous adoption challenges with the ROTS solution. This section discusses three of the most common challenges: understandability, robustness, and completeness, using the RIF as a concrete example.

3.1 Understandability

The first challenge of adopting ROTS software is understandability (or the lack thereof). The typical ROTS solution is rarely ready to be used by anyone other than its author (and sometimes not even then). The short-term goal of ROTS software, especially that produced as part of a thesis or dissertation, is usually to produce a "proof of concept" solution – not a shrink-wrap product suitable for others to use "off the shelf". If the intention is not to produce software for others to use, then it is extremely difficult to reengineer ROTS software so that others can adopt it after the fact.

In the case of the RIF, using the scaffolding it provides to carry out code optimization experiments is nontrivial. The RIF is implemented in standard C++ and relies heavily on the Standard Template Library (STL) [1]. The implementation relies on C++ templates extensively, to enrich its intermediate representation and make it more type-safe, to make the library more flexible and convenient to use, and to decrease the amount of library source code that must be maintained. It also utilizes multiple inheritance to maximize reuse and to most effectively represent unique concepts with distinct classes.

While these design goals are laudable and in fact reflect current thinking in modern object-oriented design, they can also make the code very difficult to understand. Unless one is well-versed in the nuances of advanced generic programming using the STL, it is not obvious which classes should be used, how they should be used to make the program efficient, and how to debug the result when something inevitably goes wrong.

A reference manual for the RIF is available. However, it is not complete. Most of the students find themselves poring over the RIF source code, occasionally consulting the RIF author's thesis itself, to obtain guidance on how to use the facilities the RIF provides.

3.2 Robustness

The second challenge of adopting ROTS software is that the people coding the solution are rarely professional software developers. This may not be the case if the person doing the coding is, for example, a research associate whose primary responsibility may be to create a robust ROTS solution that is closer to commercial quality than the norm.

However, the programmer is more often a graduate student, someone who is trying to do his or her best, in a short period of time, and just get "something" running. Since most computer science students do not take software engineering classes (to say nothing of students in other disciplines, such as bioinformatics), they have little guidance or background upon which to draw best practices. The not-very-surprising result is that much ROTS software is of very poor quality.

Interestingly, this was not the case with the RIF. The code is generally quite well written. That may be because a robust implementation was one of the key goals of the thesis, and hence was something upon which the value of the student's work was judged. However, robust code is not necessarily the most usable or complete code.

3.3 Completeness

The third challenge of adopting ROTS software is that there is very little incentive for the researcher to create a complete solution. After all, if the main reason the program was written was to test a hypothesis, or provide enough evidence that a "real" solution to the problem under investigation could be engineered given enough time and effort, then once this more modest goal has been achieved, most researchers will move on to the next problem. For students working on a thesis, once the minimum amount of required coding is done, it's done.

In the case of the RIF, the library was adopted as the development platform for students to use in subsequent offerings of the graduate compiler class. While the student who did the bulk of the implementation was primarily concerned with getting the minimal functionality working "just enough" to complete the degree requirements, there was also an incentive to construct the RIF to be as usable as possible. But that was only because this was one of the criteria for success for the thesis itself. In most other cases, ROTS software is a means to an end, not an end in itself.

4. Recommendations

We believe that to properly address the challenges of adopting ROTS software, the academic community must address several fundamental issues. Obviously, these recommendations only apply to those problem domains and research projects working in more applied areas, or that have as one of their main goals technology transfer. This is not true of all academic efforts, nor should it be. There will always be a place for pure science and investigation for its own benefit. However, for areas like software engineering, adoption is becoming more and more important. In fact, we believe that it should be considered essential to any declaration of success.

4.1 Improve Programming Skills

The first recommendation is to train students to be better software developers. Although the ultimate goal might be to instill the same level of discipline and rigor in students as exists in professional and conscientious software engineers working in mature organizations, this might not be realistic. However, there are concrete steps that can be taken to improve the quality of the code they produce.

The latest draft of the joint ACM/IEEE SEEK (The Software Engineering Education Body of Knowledge) document lists "software construction" as a key knowledge area [4]. Three years ago the University of California, Riverside introduced an elective course called "CS 100: Software Construction" in support of this goal. Of course, taking a course in software engineering would be extremely beneficial as well, but students who already have an interest in the area typically take this. A course on software construction, which focuses more on individual programming skills and acumen, could be more broadly beneficial.

Programming skills should be considered as essential to a software engineer researcher's success as a mastery of technical communication.

4.2 Require Empirical Evidence of Efficacy

The second recommendation is to move towards requiring empirical evidence of the efficacy of the ROTS software solution. There is a growing awareness of the need to employ evidence-based arguments to support the

practices of software engineering, rather than arguments based upon advocacy [2]. An objective measure of the efficacy of a ROTS software solution would facilitate adoption by providing an independent predictor of its likely benefits.

One of the paradoxes of software engineering is that, although it extensively employs widely-accepted concepts and practices that are drawn from experience and observation, we rarely possess any solid audit trail that can provide a validation of these ideas and that could link theory and concepts to observed practices. By requiring evidence of ROTS efficacy, problems related to both technology adoption and the maturity of the software engineering field would be partially addressed.

4.3 Change the Academic Reward Structure

The third recommendation is related to the typical academic reward structure. One of the prime currencies for most academics is publication. Once a paper describing (often preliminary) results from a project has appeared in a public forum, such as a conference proceeding or journal paper, there is often little incentive to continue the work. Quite the contrary in fact; subsequent publications on the same topic are often viewed as derivative work by reviewers and hence may not get into print.

This creates a clear disincentive for the principal investigator and the rest of the team to continue working on the project. Unless the goal was specifically technology transition to an industrial partner, in many cases this part of the project will be declared complete and a new line of investigation will begin. For software engineering research and ROTS software, this phenomenon is very unfortunate, since it perpetuates the adoption challenges outlined in Section 3.

One way to address this problem would be to reward researchers who take ROTS software from the proof-of-concept stage to something that is nearing commercial viability. It is well known that such an endeavor is both academically challenging and (as outlined in Section 1) potentially very rewarding. To see this recommendation come to fruition, the community as a whole would have to recognize the importance of this "last mile" of software engineering research.

5. Summary

ROTS software holds much promise, but only if it is adopted by people other than its original developers. The diffusion of technology remains more of an art than a predictable process, but is it essential to move the field forward. Nowhere is this truer than in applied software engineering research, where the adoption of results by the community-at-large should be regarded as a necessary but not sufficient sign of success.

Our own experience with an object-oriented framework for developing routines in the domain of optimizing compilers suggests that there are several possible areas for improvement. As outlined in Section 3, these were understandability, robustness, and completeness of the ROTS application. There is no easy answer to these challenges, but the recommendations suggested in Section 4 begin to address them.

References

[1] Austern, M. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1998.

[2] Budgen, D.; Hoffnagle, G.; Müller, M.; Robert, F.; Sellami, A.; and Tilley, S. "Empirical Software Engineering: A Roadmap." To appear in *Proceedings of the 10th International Conference on Software Technology and Engineering Practice* (STEP 2002: Oct. 6-8, 2002; Montréal, Canada). IEEE Computer Society Press, 2003.

[3] Christensen, C. *The Innovator's Dilemma*. Harvard Business School Press, 1997.

[4] IEEE Computer Society. "Computing Curricula for Software Engineering". Online at http://sites.computer.org/ccse/.

[5] Moore, G. *Crossing the Chasm*. HarperBusiness, 1991.

[6] SEI Technology Adoption program. Online at http://www.sei.cmu.edu/adopting/adopting.html.

[7] SEI Technology Transition Practices group. "Fundamentals of Transition Mechanisms". October 2002. Online at http://www.sei.cmu.edu/ttp/presentations/fundamentals-transition/.

[8] SEI Technology Transition Practices group. Online at http://www.sei.cmu.edu/ttp/.

[9] Sirko, E. "RIF: A Language and Toolkit Supporting Research and Education in Optimizing Compilers." Master's Thesis, Department of Computer Science, University of California, Riverside. September 1999.

[10] The Economist. "Innovation's Golden Goose." *The Economist Technology Quarterly*, December 14, 2002.

[11] Tilley, S. and Huang, S. "On the Emergence of the Renaissance Software Engineer." *Proceedings of the 1st International Workshop on Web Site Evolution* (WSE'99: Atlanta, GA: October 5, 1999).

Tool Adoption: A Software Developer's Perspective

Johannes Martin
Johannes Gutenberg-Universität Mainz
Psychologisches Institut
Mainz, Germany
jmartin@notamusica.com

Abstract

Much of the work in adoption centric software engineering has focussed on the aspect of presenting and manipulating documentation and information on source code of software systems. These are tasks that are usually done by managers and system designers, and thus an integration into the office tools those people use is very appropriate.

Programmers usually use quite a different set of tools, either integrated development environments or powerful text editors and command line tools. While managers and system designers are satisfied with an infrequently updated high-level view of their software systems, programmers need exact information on the details of a system in its current state.

This position paper surveys a number of current software engineering tools with respect to their support for programmers' requirements to formulate properties that ease the adoption of software engineering tools by programmers.

1. Introduction

The main task of many programmers is to respond to various features requests and problem reports. While high-level design documents may help in narrowing down the sources of a problem or the parts of the code affected by a feature request, that information is usually not sufficient to solve these problems. They need an intimate knowledge of the source code they are working with, down to where a particular function or variable is used. Even though expert programmers can remember a lot of this information, they require ways to lookup information in parts of the software system that they are unfamiliar with or that have recently changed.

Software reengineering tools have addressed the problem of discovering properties of source code for some time already. Recently, development tools have also tried to support programmers in this respect. In the following section, we survey a number of these tools and consider their success in term of their adoption by programmers.

2. Tool survey

2.1. Editor & search tool

Traditionally, programmers have used more or less sophisticated text editors to write and maintain source code. These editors usually offer simple search and replace tools that programmers use to locate definitions and references of source code artifacts. External search tools such as *grep* are used to locate artifacts and their relationships in source code distributed over several files and directories.

Even though these tools are much less sophisticated than integrated development environments and software engineering tools, they have some big advantages of those, for example their level of availability. In the rare case that they are not preinstalled on a system, popular editors such as *vi* and *emacs* and search tools (*grep*) are easy to install and available on virtually all hardware and software platforms. As they have a very limited range of basic functionality, they are easy to master and fast, and therefore work well on typical low-end machines available to programmers. Another big advantage is that they always present the current state of the system, as they work directly on the source code of the system rather than a on a repository of artifacts extracted from a more or less recent version of the source code.

The disadvantage of the search tools is their impreciseness. When searching for strings of matching a variable or function name, they turn up all source code artifacts that match that name. The programmer then has to filter the search results in order to eliminate these false positives. Depending on the size of the system, this might be a difficult and errorneous task.

2.2. Cross reference tools

In the early days of programming when searching in program source code was difficult, programmers often used printed cross references of source code artifacts to help in understanding and debugging their program code. As searching became easier, these cross references became neglected. With the increasing size and complexity of modern software systems and thus the high number of false positives obtained using search tools such as *grep*, programmers start to recognise the need for such cross references again. They no longer appear on paper but in the form of source code annotated with hyperlinks in a web browser. Users of these systems can navigate through the source code and find definitions and uses of source code artifacts. The more precisely a cross referencing tool parses the source code, the less likely it is to turn up false positives in a search for source code artifacts. On the other hand, a precise tools will likely fail to analyse source code that contains syntax errors.

The Rigi reverse engineering environment comes with a set of parsers and tools that can be used to generate a precise hyperlinked version of source code [8, 11]. LXR is another such project, originally intended for cross referencing the Linux kernel sources [4]. It has a less precise parser and can therefore handle a wider variety of source code dialects and even some errors in the code. It is now used for a number of open source projects in addition to the Linux kernel such as Mozilla and KDE.

Some effort and expertise is required to install these tools. Since they work on an intermediate repository of source code artifacts that needs to be recreated after the source code has changed, they do not always reflect the current state of the source code. So, even if they parse the source code precisely, they may report false positives or fail to report some results. Depending on the amount of change a system undergoes and the time it takes to update the intermediate repository, this might pose a problem. An advantage of these systems it they run entirely an a host system, the programmer only needs a standard web browser on his own system. While a programmer can thus read and navigate through the source code in his web browser, he has to switch back to his regular programming environment to continue editing the source code, requiring him to locate the source code already displayed in the web browser in his editor. A tighter integration of browsing and editing would be helpful.

2.3. Integrated development environments

Integrated development environments (IDE) combine editors with other tools programmers frequently use. Traditionally little more than a wrapper for these tools, modern IDEs integrate these tools into the editor and support advanced features such as code browsing with automated and precise cross-referencing using an internal and proprietary repository [2, 9, 10, 5]. These advanced features are easy to use for programmers already familiar with these IDEs, and therefore adoption is almost guaranteed.

A big problem of these IDEs, however, is their lack of scaleability. As they rely on building an internal repository, they require all or a large part of the code to be compiled on the programmer's machine. For industrial size applications, this is often not possible. As these IDEs sometimes try to offer every imaginable gadget, they become bloated, requiring more RAM and CPU time an average programmer can offer on her workstation. Often they are limited to a certain programming language and operating system, making them even more difficult to use in existing software projects. Also, programmers who are used to a different programming environment might be reluctant to learn a new environment.

3. Goals

We have identified a number of properties that should ease the adoption of software engineering tools by programmers. We will discuss these in the following sections of this paper.

3.1. Correctness of the results

A software engineering tool must respond to a query by returning all appropriate responses. Programmers are used to filtering false positives from search result, so a small number of false positives within the search result is acceptable. There must not be any false negatives, since programmers will not accept a tool that will require them to verify results externally.

The tool must reflect the current state of the source code as much as possible. If it is not possible to update the internal repository of the tool with every change of the source code, the programmer should be able to force an update when needed. A small change in the source code should only require a minor update operation in the repository. It is usually acceptable to a programmer if the repository update takes about as long as a recompilation of the changed parts of the program. Ideally, the repository update should be triggered automatically whenever a compilation is performed.

This requirement is met by most modern IDEs, unless a software product exists in different configurations. IDEs usually consider only one particular configuration of a program as they depend on their integrated compiler to populate their repository. The problem of parsing source code independently of its configuration needs to be addressed bet-

ter in IDEs. However, it is a problem that is limited mostly to the C and C++ programming languages.

3.2. Responsiveness

Unless a software engineering tool provides a significant advantage over her current tools, a programmer will not use it. She will evaluate the tool by how it helps her in completing her tasks on time. For example, if a search tool requires considerably longer to present a search result than it would take the programmer to filter the output of a search using *grep*, she will revert to using the faster *grep* over the more exact tool.

3.3. Version control

Programmers frequently have to work on different versions of a program. They might have to correct a problem in the released version of a product one day and implement a new feature in the development version of the product the next day. Most software engineering tools do not support versioning natively. The internal repository of the tool will have to be recreated in order to get an accurate description of any one version of a software system. Research on how to store historical data on source code in repositories needs to be done.

3.4. Flexibility

Most current software engineering tools come with a proprietary user interface and therefore require their users to familiarise themselves with that interface. The developers of these tools often overlook that fact that every programmer has a different style of accomplishing his work. While some prefer spartanic environments and are happy with a simple editor, others prefer a windowed environment and a colourful IDE. Software engineering tools should offer different frontends to suit a variety of usage style. One programmer should be able to query a repository using a command line client, while another programmer should be able to perform the same query from within his IDE. The command line client interface could further be modeled after *grep* to help the programmer in the transition, just as the IDE's query dialog closely resemble the IDE's regular search dialog. Most importantly, the software engineering tools need to be available on the programmer's platform of choice.

4. Conclusion

In this paper, we surveyed and categorised tools used by programmers and formulated some criteria for the adoption of software engineering tools by programmers. While some of these tools, namely IDEs, are already being adopted by programmers quite well, they are often limited to small or medium size projects that can be compiled within the programmers' development environment. Those tools that can handle larger volumes of source code do not yet integrate well with common development environments. Many of the currently available tools do not offer support for version control and limit the programmer in her choice of programming environment.

In the recent past, many technologies and tools have been developed that can help solve some of the problems assessed in this paper. A common exchange format for software artifacts has been defined, and a number of parsers for popular programming languages that use this exchange format are now available on various platforms [1, 6, 3]. In the *Ovid Project*, we collect these tools to integrate them according to the goals we have formulated [7]. The resulting tool sets will help in everyday software maintenance, software reengineering, refactoring, and language migration.

References

[1] GXL Home Page. http://www.gupro.de/GXL, November 2001.

[2] Eclipse.org. Eclipse Web Site. http://www.eclipse.org.

[3] R. Ferenc, F. Magyar, A. Beszédes, A. Kiss, and M. Tarkiainen. Columbus — Tool for Reverse Engineering Large Object Oriented Software Systems. In *Proceedings of SPLST 2001*, pages 16–27, Szeged, Hungary, June 2001. http://ferenc.rgai.hu/research/ferencr_columbus.pdf, November 2001.

[4] A. G. Gleditsch and P. K. Gjermshus. Cross-Referencing Linux. http://lxr.sourceforge.net/.

[5] M. Karasick. The Architecture of Montana: An Open and Extensible Programming Environment with an Incremental C++ Compiler. In *Proceedings of the Conference on Foundations of Software Engineering*, Orlando, FL, Nov. 1998.

[6] A. J. Malton. CPPX Home Page. http://www.swag.uwaterloo.ca/~cppx/, November 2001.

[7] J. Martin. The Ovid Project. http://ovid.tigris.org/.

[8] H. A. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988.

[9] L. R. Nackman. CodeStore and Incremental C++. *Dr. Dobb's Journal*, pages 92–95, Dec. 1997.

[10] D. Soroker, M. Karasick, J. Barton, and D. Streeter. Extension Mechanisms in Montana. In *Proceedings of the 8th IEEE Israeli Conference on Computer Systems and Software Engineering*, Herzliya, Israel, June 1997. IEEE Computer Society Press.

[11] University of Victoria. Rigi Web Server. http://www.rigi.csc.uvic.ca, June 2001.

The Need for Adoption Issues in Enterprise Integration

Dennis Smith and Liam O'Brien
Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213 USA
+1 412 268 7727
{lob, dbs}@sei.cmu.edu

Abstract

The ability to provide integration between business functions that may be supported across multiple applications is a critical need for modern organizations. Although significant technical issues need to be addressed to address the issue, many failures have resulted from not adequately addressing adoption issues. This paper identifies adoption issues that need to be addressed in effectively addressing the enterprise integration problem.

1 Introduction

Enterprise Integration has the goal of providing timely and accurate exchange of consistent information between business functions to support strategic and tactical business goals in a manner that appears to be seamless. The initial automated applications that were developed during the 1950s and 1960s tended to focus at the level of an organizational unit. Over time, the scope of requirements has increased, along with an unfortunate tendency for the information systems to become brittle, difficult to manage, and hard to understand. This in turns led to the inability of users to integrate critical new applications into the existing solution set, or to mix-and-match the capabilities provided by the systems to solve new problems.

As automated systems became more pervasive within organizations, and as organizations reorganized, split or were acquired and reacquired over the years, the need for integration between applications over a broad enterprise has become increasingly important, and in fact is often a critical success factor for the survival of the enterprise.

Rather than acting as independent programs, integrated systems can provide better business value by sharing data, communicating results, and improving overall functionality. Integration of information systems is expensive and time consuming. Between 20% and 40% of labor costs can be traced to the storage and reconciliation of data. In addition, 70% of code in corporate software systems is dedicated to moving data from system to system [1]. The challenge has always been how to realize this goal.

This paper focuses on the adoption issues that need to be addressed in enterprise integration. Section 2 outlines some of the highly publicized failures that have been experienced in enterprise integration projects. Section 3 identifies adoption problems that have been experienced in implementing Enterprise Resource Planning (ERP) solutions. Section 4 summarizes organizational problems. Section 5 identifies migration planning issues that need to be addressed. Section 6 discusses the adoption issues that need to be resolved to address the problems.

2 Adoption Problems in Addressing Enterprise Integration

Although enterprise integration is critical for achieving organizational goals, the track record of implementations has been spotty. A number of publicly documented failures [2] include:

- Hershey Foods Corp., $115 million SAP installation to replace "scores of legacy programs running everything from inventory to order processing to human resources"; during busiest season of the year (Halloween), "Hershey warehouses piled up with undelivered Kisses, Twizzlers and peanut-butter cups. The upshot: third-quarter sales dropped by a staggering 12.4 percent … and earnings were off 18.6 percent."
- Whirlpool, Dow Chemical, Boeing, Dell Computer, Apple Computer and Waste

Management experienced similar disappointment.

- W. L. Gore & Associates sued Deloite Consulting for breach of contract, fraud and negligence to recover $3.5 million in fees; also names PeopleSoft for certifying incompetent party.
- SunLite Casual Furniture sued Deloitte in Arkansas for maliciously "indoctrinating in SunLite a total dependency on D&T that D&T hoped would result in lucrative fees for years to come.
- FoxMeyer Drugs blames "botched implementation of SAP's F/3 software for pushing it into bankruptcy back in 1996." Suing Andersen and SAP for $500 million, also only spending $30 million for the project.

Two analyses [3, 4] attribute the reasons for failures to 7 adoption related reasons:

- Miscommunication
- Hazy goals
- Poor project management
- Scope creep
- Modifying ERP software prior to pilot testing
- inadequate training
- insufficient implementation support

3 Adoption Problems and Enterprise Resource Planning (ERP) Implementations

Enterprise resource planning (ERPs) solutions are essentially COTS products that provide support for standard enterprise needs in such areas as finance, human resources, and logistics. A number of vendors provide ERP solutions, including PeopleSoft, SAP, Baan, and Oracle. ERPs are popular because they offer the promise of enabling an organization to leverage the research and development efforts of the ERP vendors. Functional areas such as taxes, purchasing, and human resource management have a significant amount of commonality between organizations, and there are strong arguments for purchasing a ready solution rather than developing an application from scratch.

ERPs can make good sense for an organization. However, the benefits are far from automatic. In fact, a number of disasters in ERP implementations have occurred. Often these problems occur because of mismatches between the COTS ERP products and the business practices of the target organization. A decision to implement an ERP requires careful analysis of the following factors:

- an understanding of the gap between the underlying object, data, or functional models of the ERP solution and those currently supported by an organization's legacy systems (often substantial effort is required to customize the ERP or change the organizational processes to match those of the ERP)
- an understanding of the role of data, control and presentation integration in making ERP solutions more effective
- an understanding of specific ways in which the ERP will interface with legacy systems, other ERPs, and future development efforts
- an understanding of migration issues, such as user training, data migration, phasing in of the ERPs, and phasing out of the legacy systems
- development of realistic cost and schedule estimates reflecting realistic expectations

A number of open issues concerning the adoption of ERP products need to be addressed. These include:

- To what extent is it possible to share services between different ERPs?
- To what extent is it possible to use a common framework to support different ERPs?
- To what extent can the user interface be separated from the core functionality of ERPs?
- If core functionality can be separated from user interfaces, how do new versions of the ERPs interact with the new user interface?

4 Organizational Issues

Organizational issues in enterprise integration are significant. An enterprise-integration effort affects an entire organization, and it is necessary to have long-term management support and financial commitment, realistic plans, and systematic migration planning.

The outstanding organizational issues and problems include:

- a need for an effective methodology for enterprise integration including a clear distinction between the scoping that an enterprise architecture provides and the detailed blueprint for development that a software

architecture for an application provides. This includes clear guidelines for the deferral of details from the enterprise architecture to the software architecture

- lack of organization-wide solutions to integration problem (integration solutions tend to be local)
- the identification of interfaces from existing systems and identification of side effects
- need for clear guidance for management decision making
- A need to recognize that integration cannot be mandated, legislated or assumed. It needs to be nurtured. There is often a lack of understanding of the cost of integration at upper management levels. Cost, risks, and potential harm need to be fed back to the upper levels.
- A common failure to clearly define the scope of an "enterprise" to integrate. This can result in a shifting definition of the enterprise, overlapping organizations, and turf battles. There is a great deal of pressure to define an enterprise too broadly, and thus to make it difficult to partition the problem into manageable entities.
- A need to recognize that any technology solution should derive from business drivers using technology as an enabler, as opposed to viewing technology as a primary driver. Many failures result from a "technology first" or solution first" approach, and from the failure to adequately address organizational and cultural issues.
- There is often a failure to take a long term view. Some of the factors include the practice of rapid rotation of leadership, budget instabilities, and failure to plan for long-term maintenance and upgrading costs.
- A need to consider the total cost of ownership for an Enterprise Resource Planning (ERP) solution. The total cost includes not only the ERP software cost but the other associated costs including integration, training, system analysis, customization, maintenance, etc. The associated costs may be of the order of magnitude of 5 to 7 that of the software cost.
- There can be a tension between the requirement of developing an enterprise architecture and the pragmatic demands of individual ERP/COTS solutions, leading to the impression that technology is the solution.
- When making a decision on an ERP, low level analysis of the details needs to be done to determine a match in some cases changing existing business processes to match those of an ERP may be the most cost-effective approach. However, this needs to be done with careful analysis which unfortunately does not occur often.
- There is a strong need for a coordinated migration plan for the existing systems to move towards integration

5 Disciplined Migration Planning

Enterprise integration can also be considered to be a complex migration problem. Although the initial step of developing an enterprise integration plan establishes a blueprint for the final goal, in general these efforts have not developed adequate plans. In general there is an assumption, which may be unrealistic, that legacy systems will be replaced over a period of time. As a result, such efforts have sometimes been big bang approaches – without substantial intermediate deliverables. There is a need to have a greater focus on migration plans from current legacy systems and to clearly relate the perceptions and needs of end users to the long-range plans that are developed.

Bergey, O'Brien and Smith [5] have addressed the issue of migration planning, and recommend addressing a set of issues, including:

- Identifying all relevant stakeholders and involving them throughout the project
- Ensuring there is a common understanding of the problem to be solved
- Determining that the initiative is commensurate with the maturity of the organization's software practices
- Define all aspects of the software architecture and its constraints on existing and new systems
- Perform a thorough analysis of legacy systems, their interfaces, and changes required
- Break the problem into bite size chunks that are phased in incrementally
- Do a pilot effort before committing to a large scale plan

6 Conclusion: Bridging the Gap between Need and Reality

A number of recent trends provide a foundation that can lead to future success. Middleware technologies have advanced rapidly over the past 10 years, and these enable more options for integration than had been previously available. The maturing of markup languages such as XML enable more effective integration, particularly between structured data, such as data from databases, and non-structured data, such as email. The Web can serve as a common front end for integrating a variety of applications, and it can enable effective presentation integration. Web services are maturing as an important mechanism for integration of legacy systems, new applications and ERPs. The emergence of enterprise portals over the past several years demonstrates the strong interest and need for effective presentation integration. In addition, ERP applications, while still displaying significant problems, have become more mature, and their interfaces are better able to share data with other applications.

However, despite progress, the overall status of the field is immature. There is a significant gap between the desired state and present reality. In order to bridge this gap the following critical issues need to be addressed:

- Determining an effective scope for an integration effort as well as the development of a proven method for developing an effective scope for integration efforts. Currently, many efforts flounder because they fail to define an effective scope. Often current efforts develop very ambitious scopes that are difficult, or impossible to successfully implement.
- Aligning the integration effort with the mission and high level goals of the enterprise, and developing commitment and sponsorship at the appropriate levels
- Understanding the role of adoption issues in implementing enterprise integration efforts.
- Understanding the appropriate role of an enterprise architecture, and its relationship to a software architecture
- Understanding appropriate role of frameworks and standards
- Addressing the technology issues of data integration, control integration and presentation integration
- Decision rules for making choices on the types of technology that are most appropriate for specific types of efforts. Although many technology solutions are available, there are not easily accessible guidelines for when to use different types of solutions.
- Determining the type of technology that is most appropriate for different types of programs
- Understanding when ERP solutions are appropriate, and when they are not appropriate
- Breaking down an overall project into realistic parts
- Developing realistic sets of plans for the effort
- Addressing issues of contracting, funding and oversight management within government organizations
- Migration and integration of legacy systems

References

1. Zachman, J. “Enterprise Architecture: The Issue of the Century.” *Database Programming and Design*, March, 1997.
2. Osterland, A. *ERP Disasters.* CFO, The Magazine for Senior Financial Executives, Jan. 2000.
3. Nash, K. “Companies Don’t Learn From Previous IT Snafus”, *Computerworld*, October, 2000.
4. McAlary, S. “Three Pitfalls in ERP Implementations”, *The Managers Publication of Data Solutions*, March, 2000.
5. Bergey, J. O’Brien, W. and Smith, D. *DoD Software Migration Planning*. Carnegie Mellon University, Software Engineering Institute, CMU/SEI-2001-TN-012.

Adoption-Centric Knowledge Engineering

Neil A. Ernst
Department of Computer Science, University of Victoria
PO Box 3055, STN CSC, Victoria, BC, Canada V8W 3P6
nernst@cs.uvic.ca

Abstract

Cognitive issues in software engineering are relatively well-documented and well-understood when compared with the domain of knowledge engineering. Current knowledge engineering tools often feature high barriers to the use and re-use of both the tool and its products. An adoption-centric knowledge engineering approach is suggested to deal with these issues. Adoption-centric knowledge engineering is the design of knowledge engineering tools and knowledge engineering processes to ensure the widest possible adoption. In turn, wide adoption will benefit the projects that choose to focus on it by increasing the user-base. One way to make a tool more adoption-centric is to provide increased cognitive support to the potential user.

1. Introduction

The introduction to a well-known project site for adoption-centric software engineering (ACSE) [1] makes the case for ACSE as follows: "Research tools in software engineering often fail to be adopted and deployed in industry." This is equally true of tools in the discipline of knowledge engineering. User-centered software engineering has seen a wealth of research compared to similar projects in knowledge engineering. This research has produced a body of work which describes theories for how software engineering is practiced, although by no means an exhaustive amount. Knowledge engineering, the design of knowledge-based systems, be they theorem provers, expert systems, or intelligent agents, is not as well documented. I am not referring here to the logical foundations of expert systems, as this is a much-studied area; see [2] or [3] for examples of seminal knowledge-engineering design projects. These papers document in detail the mechanics of designing a knowledge-based system. Unfortunately, the knowledge acquisition field has traditionally ignored the user perspective in these areas. Practitioners have been more concerned with designing a system to solve a problem – say, to diagnose a specific medical condition – than with the actual methods used to create the system. This is well illustrated in [3]; namely, that few efforts in the field are focused on developing tools for users, being more concerned with knowledge modelling and knowledge elicitation, often at the expense of end-user usability concerns.

We should be concerned with end-user adoption and usability (where the end-user, in this case, is the system designer) because developing good applications is directly related to how simple the chosen tool is to use: the tool should be unobtrusive, a fact shown in certain software engineering studies [4]. Knowledge engineering needs a similar focus. Adoption-centric knowledge engineering (ACKE) would be focused, like its sibling ACSE, on delivering tools that leverage existing user knowledge rather than requiring learning yet another new product; this can provide a significant advantage to developers. In this position paper I first discuss the background of knowledge engineering, particularly with respect to software engineering; section 3 illustrates a typical tool for adding to cognitive understanding of large knowledge bases; finally, section 4 argues that adoption-centric knowledge engineering is of vital importance to many projects.

2. Background

This section describes the fundamental ideas of knowledge engineering and looks at the intersections of both knowledge engineering and software engineering. I have found that empirical studies of cognitive support for knowledge engineering are lacking, and seek to leverage comparable studies in software engineering.

2.1 Knowledge Engineering

There has been an increased focus in recent years on knowledge engineering, particularly in response to the Semantic Web initiative of the World Wide Web Consortium (W3C) [5], [6]. The Semantic Web initiative is concerned with the "abstract representation of data on

the World Wide Web" [7] such that additional, machine-comprehensible metadata might be created. The formation of global standards such as the Resource Description Framework (RDF) [8], the Web Ontology Language (OWL) [9], and XML, combined with the power of distributed application development via the Internet, has led to renewed interest in knowledge-based systems, to perform any number of tasks, such as making inferences on web site metadata to intelligent e-commerce shopping agents [10].

The term knowledge engineering refers to the design and construction of knowledge-based systems, much like software engineering refers to the design and construction of software systems. Traditional knowledge engineering has followed a number of processes, which typically contain some of the following elements [11] (figure 1):

a) The knowledge base design phase. This is a model of the proposed system (for example, a medical protocol for cancer treatment); this phase is akin to a software modelling phase where requirements are gathered but no actual code is written;
b) the knowledge acquisition and knowledge elicitation phase, in which domain experts are interviewed by knowledge engineers and data instances are created;
c) the knowledge entry phase, in which the knowledge engineer enters the newly acquired data into the knowledge base;
d) the knowledge maintenance phase, where the system is updated to reflect new facts and rules.

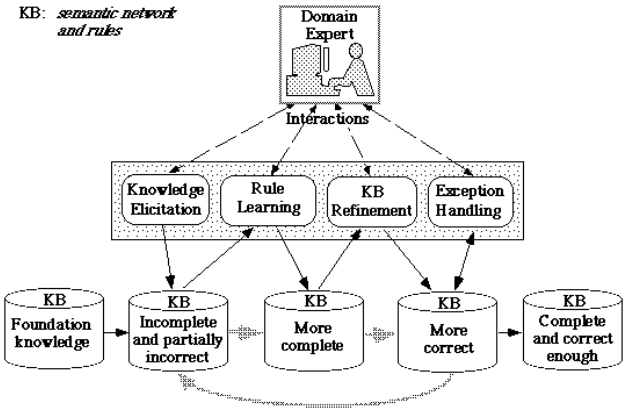


Figure 1 - The knowledge acquisition process ([12])

This process is somewhat non-linear, as indicated by the grey arrows. Knowledge engineering, like software engineering, can be very iterative; as the system is tested and faults are found, the model may be changed or the data instances modified.

In knowledge engineering there is a third party involved in a significant way: the domain experts are a crucial and significant part of the knowledge engineering process. Since the domains are so complicated, and the goals so high-level, a second expert is often needed to explain this, particularly where the knowledge engineer has no domain knowledge. Furthermore, the modelling of knowledge-based systems often takes place at the knowledge level [13], rather than the symbol level; this higher degree of abstraction leads to problems in deciding exactly what the system is capturing and modelling.

Knowledge engineering tools are systems designed to automate some aspect of this complex process. They range from tools which help with the design of knowledge-based systems [14] and the elicitation of knowledge [15], to tools which provide a mechanism to maintain and upgrade the knowledge base ([16], [17]). There are two broad classes of users for these tools. One is an end-user, for instance, the doctor or engineer accessing a knowledge-base for decision support with some task. The second category, with which my research is most concerned, is the system engineer, either the knowledge engineer who created the system, or a maintenance engineer who seeks to ensure accuracy, speed, and other performance measures as defined by the application specification. This class of user has obvious and natural equivalencies with his software engineering counterpart, for example, a maintenance programmer. Both users require cognitive support for understanding the model for which they are responsible. To further narrow the types of use-case we seek to model, I have wilfully ignored the cases where we seek to enter knowledge (knowledge acquisition), focusing instead on maintaining knowledge-bases. This is akin to the software maintenance and program comprehension tasks. One of the big difficulties in this area is the lack of comprehensive research in the field. While there is a fair amount of work in the areas of knowledge engineering methodology, such as ensuring accuracy and performance, comparatively little has been done on knowledge base maintenance, and there are no theories on cognitive support for knowledge base engineering, unlike some work in software engineering [18].

2.2 Knowledge Engineering and Software Engineering: Perspectives

What are the differences and similarities between knowledge engineering and software engineering? I believe there are two perspectives to take on this relationship; one is to examine knowledge-based software engineering, the other to consider software-centred knowledge engineering. Software engineering can often be said to be knowledge-centric, in that it seeks to construct a knowledge-based model of a particular application domain. This would apply to projects which leverage the knowledge of end-users to support decisions

they might make at a later date. Such systems often contain a large degree of knowledge, such as the forest stand lifecycle data of the Sortie project [19], within data structures of the program, whatever the programming language.

Clearly, the greatest overlap occurs in languages such as Prolog, which is typically used by software engineers to construct knowledge-based systems. We describe these systems as software-centered, because their functionality arises from a software-based tool or solution. In this perspective, software tools are used to construct a symbol-level representation of the knowledge-level model.

Other systems exist outside these perspectives. There are software projects, such as an operating system, which make less use of specific knowledge from end-users. Requirements in this type of project are largely functional and quantitative. The Linux project, for example, is not as concerned with capturing user-specific information as it is with providing certain functionality, such as USB support. The inverse of this is the knowledge base which uses no or few software engineering techniques to design the tool. We might characterize knowledge bases which use domain-specific tools in this manner. An example of this is a controlled terminology which is used to standardize the vocabulary of a particular domain. On their own, these terminologies use no specific software engineering approaches; however, they are often combined with other tools, such as Internet application servers, to deliver the product.

3. Our Approach

Our research group is developing a tool called Jambalaya [17]. Jambalaya is the integration of a software understanding tool, SHriMP, with a knowledge engineering tool, Protégé [20], to attempt to provide some cognitive support for developers of software-centric knowledge bases. We have recently completed a user questionnaire on how people might use the visualizations of Jambalaya for enhancing their comprehension of the (often-complex) knowledge structures in Protégé, in order to provide a better understanding of these issues: a paper describing these results is in progress. Amongst the things that emerged were:

- the large number of domains being worked on;
- the different sizes of ontologies which users manage;
- the relative lack of usable visual representations of the knowledge structures.

These points seem to indicate that a major challenge in both designing tools for knowledge engineering and, perhaps more importantly, increasing the adoption of those tools, will be in bridging the number of domains and approaches which exist. The reason better cognitive support is needed remains relatively unclear except on an ad-hoc level, and needs to be addressed through user evaluations and interviews. Some preliminary work, however, seems to illustrate the need fairly clearly. Blythe et al. [21], for example, identifies some typical concerns that users may have when adding new knowledge to an intelligent system:

- Users do not know where to start and where to go next;
- Users do not know if they are adding the right things;
- Users often get lost as it takes several steps to add new knowledge.

Our goal is to attempt to address some of these concerns by providing enhanced cognitive support for developers in understanding the nature of the knowledge-base they are maintaining. For example, Jambalaya provides a series of different graph layout algorithms to allow users to maintain different perspectives on the model. One area of research of great interest is on what techniques knowledge engineers use to understand the model. For example, research in software engineering indicates several strategies for program comprehension, such as top-down, bottom-up, knowledge-based, as-needed, and integrative [22]. We are interested in exploring whether such techniques translate readily to the knowledge engineering domain.

4. The Need for Adoption-Centric Knowledge Engineering

Our tool is currently integrated with a reasonably popular knowledge engineering framework. Protégé has fairly widespread support, but is by no means a universal tool. Much like the popular software development environment Eclipse (eclipse.org), Protégé has a number of adoption-centric advantages, including an extensible plug-in architecture, an open-source licence, and a lengthy history in the community. Nevertheless, Protégé has limitations in specific areas. For example, Protégé uses a frame-based knowledge representation, which is only one of many formalisms, each of which has its own advantages. The frame-based representation is somewhat similar to object-oriented software engineering paradigms, and Protégé uses one version of it, much like Eclipse offers Java integration. Some large applications, such as the U.S. National Cancer Institute's controlled terminology, prefer instead to use different representations, for a variety of reasons, just as other projects may use pure first-order logic or variants thereof. The particular formalism a tool uses may affect the type of cognitive assistance required: for example, in a frame-based system like Protégé, a hierarchical object-centric view may be appropriate; in a first-order logic rule-based

expert system, a visualization of how the rules were fired may be of most interest. ACKE tools should provide support for the engineer, and not the formalism chosen (just as ACSE should not differentiate between programming languages). In other words, ACKE tools should be flexible enough to support different and varied environments and formalisms, as the user may require. This may not be possible in all domains and areas of interest, naturally; rather, what is being proposed is a user-centered approach to the design of these tools, as much as possible. Many current tools focus only on the semantics of the formalism they are attempting to implement – that is, does the tool fulfil the formal model and syntax specification of the formalism – and not on how usable the tool may be for developing different applications.

In particular, the Semantic Web initiative defines only representation mechanisms and not tools to perform knowledge engineering operations. While Protégé seems likely to be a part of Semantic Web application development, it is unreasonable to assume that it would be the only tool used, which illustrates the need for ACKE tools. If a new platform for editing Semantic Web services becomes widespread, we should not force users who have a need for the enhanced cognitive support that Jambalaya may offer to adopt Protégé as well. Instead, we should focus on adapting our tool to the user requirements, rather than vice-versa. The ACSE approach suggests that the best method for providing developers of knowledge-based systems with tool support is to focus on developing either simple add-ins to commonly used platforms, or providing interfaces to those platforms. `One approach might be to embed the tool in another product with a larger base of users, much as we have done with Protégé and SHriMP, but another example might be to offer SHriMP-like functionality in an SVG-based web tool. This would allow users to continue using their web user-agent, such as Internet Explorer, with whose functions they are very familiar, while also accessing more complex functionality to support the creation and browsing of sophisticated knowledge-based applications. An interface approach might suggest developing a standard import/export mechanism, allowing for interoperability between tools. A relatively recent example of this is the ability of Protégé to export XMI serializations of its models, allowing Protégé users to access a wider range of products, such as Rational Rose [23]. The challenge for the developers of cognitive aids, such as the SHriMP team, is to reduce the feature set and user interface challenges of the tool to a point where the essential features are preserved, yet the cognitive overhead of learning the tool is still low enough to encourage adoption. It is not sufficient to merely embed the tool in a knowledge-engineering platform; it must still be compelling and intuitive – in other words, it must rapidly answer the user's question, What does this do for me? I believe previous knowledge engineering tools, while their formal *utility* may have been high, nevertheless required a great deal of learning before they met simple *usability* criteria, forcing users to learn not only new user interfaces (Protégé, for example, defines its own look and feel, and is one of the simpler and better-designed UIs in the area), but also to understand and leverage new syntax and semantics. The hurdles imposed by the RDF syntax, for example, are high enough without forcing users to learn a complex tool as well.

5. Conclusions

A primary goal of adoption-centric software engineering is to increase the number of users of software engineering tools, to increase awareness of the potential benefits these tools offer. My position is that such a focus is equally important for knowledge engineering tools, particularly with the emerging focus on the development of knowledge-aware applications on the Internet. Recent trends suggest that the divisions between software engineering and knowledge engineering may be blurring, and the demand for more powerful application design and construction tools, such as Eclipse and Protégé, is growing, whether the application is knowledge-centric or software-centric.

If the vision of the Semantic Web is to be realized, it will likely arise in distributed fashion, much like its forefather the World Wide Web has done. To leverage the true capabilities of the Semantic Web, we will see increasing returns with more and more providers making information – knowledge – available to other applications. The vision has many other hurdles, among them privacy and provenance, but the lack of easy to use tools is one which should be straightforward to overcome. I propose making ACKE a focus for new knowledge engineering tools to support this vision.

7. References

[1] A. Weber, *Adoption-Centric Software Engineering*, available at: http://www.acse.cs.uvic.ca/, Department of Computer Science, University of Victoria: 2003

[2] B. G. Buchanan and E. H. Shortliffe, *Rule-Based Expert Systems: The MYCIN experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley, 1984.

[3] N. F. Noy, R. W. Fergerson, and M. A. Musen, "The knowledge model of Protege-2000: Combining interoperability and flexibility," in *Proceedings of the 2nd International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000)*, Juan-les-Pins, France, 2000.

[4] M.-A. D. Storey, K. Wong, F. Fracchia, and H. Mueller, "On Integrating Visualization Techniques for Effective Software

Exploration," in *Proceedings of the InfoVis '97*, Phoenix, AZ, 1997.

[5] T. Berners-Lee, M. Fischetti, and M. Dertouzos, *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by its Inventor*. San Francisco: Harper, 1999.

[6] N. F. Noy, M. Sintek, S. Decker, M. Crubézy, R. W. Fergerson, and M. A. Musen, "Creating Semantic Web Contents with Protégé-2000," *IEEE Intelligent Systems*, pp. 60-72, 2001.

[7] E. Miller, R. Swick, D. Brickley, B. McBride, J. Hendler, and G. Schreiber, *W3C Semantic Web*, available at: http://www.w3.org/2001/sw/, World Wide Web Consortium: 2003

[8] F. Manola and E. Miller, *RDF Primer Working Draft*, available at: http://www.w3.org/TR/rdf-primer, World Wide Web Consortium: 2002

[9] M. K. Smith, D. McGuiness, R. Volz, and C. Welty, *Web Ontology Language (OWL) Guide version 1.0*, available at: http://www.w3.org/TR/owl-guide, World Wide Web Consortium: 2002

[10] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," in *Scientific American*, 2001.

[11] S. Russell and P. Norvig, *Artificial Intelligence: A modern approach*. New Jersey: Prentice-Hall, Inc., 1995.

[12] G. Tecuci, *Constructing and Refining Knowledge Bases: A Collaborative Apprenticeship Multistrategy Learning Approach*, available at: http://lalab.gmu.edu/Projects/HPKB/boston-briefing/HPKB-Boston-index.htm#Index, Learning Agents Laboratory, Computer Science Department, George Mason University: 1997

[13] A. Newell, "The Knowledge Level," *Journal of Artificial Intelligence*, vol. 18, 1982.

[14] Protege-2000, *The Protege-2000 website*, available at: http://protege.stanford.edu, Stanford Medical Informatics: 2003

[15] P. Clark, J. Thompson, K. Barker, B. Porter, V. Chaudhri, A. Rodriguez, J. Thomere, S. Mishra, Y. Gil, P. Hayes, and T. Reichherzer, "Knowledge Entry as the Graphical Assembly of Components," in *Proceedings of the 1st International Conference on Knowledge Capture (K-Cap '01)*, 2001.

[16] I. Horrocks, "FaCT and iFaCT," in *Proceedings of the International Workshop on Description Logics (DL'99)*, P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. Patel-Schneider, Eds., 1999, pp. 133-135.

[17] M.-A. D. Storey, M. A. Musen, J. Silva, C. Best, N. Ernst, R. Fergerson, and N. F. Noy, "Jambalaya: Interactive visualization to enhance ontology authoring and knowledge acquisition in Protege," in *Proceedings of the Workshop on Interactive Tools for Knowledge Capture, K-CAP-2001*, Victoria, B.C. Canada, 2001.

[18] A. Walenstein, *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework,* Unpublished Ph.D. thesis, Computer Science, Simon Fraser University

[19] M.-A. D. Storey, S. E. Sim, and K. Wong, "A collaborative demonstration of reverse engineering tools," *ACM SIGAPP Applied Computing Review*, vol. 10, pp. 18 - 25, 2002.

[20] N. F. Noy, R. W. Fergerson, and M. A. Musen, "The knowledge model of Protégé-2000: combining interoperability and flexibility," SMI Technical Paper.

[21] J. Blythe, J. Kim, S. Ramachandran, and Y. Gil, "An integrated environment for knowledge acquisition," in *Proceedings of the Int. Conf. on Intelligent User Interfaces*, 2001.

[22] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive Design Elements to support the Construction of a Mental Model During Software Exploration," *Journal of Software Systems, special issue on Program Comprehension*, vol. 44, pp. 171-185, 1999.

[23] H. Knublauch, *XMI Backend: Storing Protégé ontologies in XMI*, available at: http://protege.stanford.edu/plugins/xmi/, Stanford Medical Informatics: 2003

On the Yin and Yang of Academic Research and Industrial Practice

Shihong Huang
Department of Computer Science
University of California, Riverside
shihong@cs.ucr.edu

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Zhou Zhiying
Department of Computer Science
Tsinghua University
zgzy-dcs@tsinghua.edu.cn

Abstract

Transitioning results from academic research into industrial practice should be a goal of modern software engineering. However, technology adoption is not something for academia alone to worry about; industry also has an important role to play in the relationship. Each supports one another, yet each is often in conflict with the other at the same time. This paper looks at the adoption problem by modeling the situation using the ancient Chinese philosophy of Yin and Yang. Using this model shows how academic research and industrial practice react to one another in a continual and ever-changing relationship that nevertheless exhibits timeless patterns of conflict and cooperation.

Keywords: *adoption, yin yang, academic research, industrial practice*

1. Introduction

The transition of academic research tools and methodologies to industrial practice is necessary to push science and technology forward. The maturation of prototype into product is needed to make the research results have an impact on the real world. For example, effective research results from biology laboratories need to be adopted by clinical practice in order to benefit patients; efficient mechanical engineering designs need to be put into factory production to improve manual labor; and software engineering research tools and techniques need to be adopted by industry as standard practice to make the research more meaningful.

However, realizing this adoption is not easy. From a software engineering point of view, to make research tools easily adoptable by industry has unique challenges. These include understandability of the solution (e.g., complex and poorly-structured source code), robustness (e.g., an implementation that is not quite ready for prime time), and completeness (e.g., a partial solution due to an implicit focus on getting "just enough" done to illustrate the feasibility of a solution) [8].

However, to thoroughly understand technology transition issues and to increase the likelihood of adoption of results from research by industry, a broader perspective may be required. There is still a need to study the problem from the academic perspective, but there is also a need to look at the same problem from the opposite side: the industrial perspective. Only by looking at technology adoption in such a holistic manner will it become more commonplace. Fortunately, there is a well-established model that can be used to reason about modern technology adoption: the ancient Chinese philosophy of Yin and Yang (also known as Tai Ji).

2. The Yin and Yang Philosophy

The ancient Chinese philosophy of Yin and Yan has its origins in modeling the unchanging rules of the changing universe. The picture shown at right of the Yin and Yang represents the cycles of the sun, the changing of the four seasons, and the entire celestial phenomenon. In more modern times, the Yin and Yang symbol has become more generally applicable to represent situations other than the natural cycles.

The essence of the Yin and Yang philosophy is in two opposite principals, Yin and Yang, which simultaneously oppose and rely on one another. In each principal there is a little of the other. In fact, one principal can give rise to the other and yet also cause the destruction of the other.

The Yin and Yang represent all the opposite forces in the universe [2]. Under Yang are the attributes of maleness, the sun, creation, light, etc; under Yin are the attributes of femaleness, the moon, completion, darkness, etc. Each of these opposites aspect produces the other: creation occurs under the principle of Yang, the completion of the created thing occurs under Yin, and

vice versa; the end of light is darkness, when darkness progress produce light.

The changing from Yin to Yang and from Yang to Yin happens constantly and cyclically, like a spiral that keeps spinning but each time reaching a level higher. This ensures that neither Yin nor Yang dominates or decides the other. All the phenomena we experience in our life, such as day and night, success and failure, conquer and defeat can be explained as the temporary dominance of one side over the other. Given the nature of Yin and Yang's dynamic change, each side will eventually change into their opposite in one way or the other.

This cyclical nature of Yin and Yang, the opposing forces of change in the universe, means several things [1]. First, all phenomena change into their opposite in an eternal cycle of reversal. For example, birth is followed eventually by death; economic booms are followed by recession; Spring turns into Summer, Summer into Fall, Fall into Winter, and Winter back into Spring again.

Second, because one side will produce its opposite side, all temporary phenomena have the seeds of their opposite side within them. For example, success contains the seed of failure, failure contains the seed of success; wealth contains the sees of poverty, poverty contains the seeds of wealth. This implies that no principal is "pure"; each contains the promise of the other.

Third, even though an opposite may not be seen to be present, no phenomenon is completely devoid of its opposite state. For example, no season is ever completely fallow, since within it are the seeds of growth for the seasons to follow. One principle produces the other.

3. The Yin and Yang of Research and Practice

When discussions of technology adoption from research into practice occur, there is inevitably an undercurrent of feelings from the academic side of "Why don't they [industry] see how good this is?" At the same time, for the same tool or technique, someone from industry might ask themselves "Why [or how] do they expect me to use this?" Both parties know that they have a symbiotic relationship with one another, yet they seem unable to truly understand what each other needs.

The academic needs an industrial partner to adopt the research results so that the prototype can be validated (or refuted), creating a feedback loop so that the next iteration of the solution is that much better. The industry person knows that academic research is potentially very valuable to them, since the results from the work could be matured and integrated into existing processes to improve important product quality attributes. So why doesn't adoption happen more easily? The reason may lie in the Yin and Yang model of technology adoption.

Academic research and industry practice can be described as two halves of the same whole [9]. They have opposite attributes: academic vs. industry; research vs. practice; theory vs. application. On the academic side, at the beginning is basic research that is conducted in the lab. The ultimate goal of the research may be to solve some real world industry problem (the white dot inside the dark part). When the research produces preliminary results, they may be adopted as industrial practice. At this period of time, the research portion is getting smaller and smaller, and the industrial practice is getter larger and larger. After industry starts to use the results from academic research, new problems will appear, so industry brings the new problems back to academia. During this time, the portion of industry is getting smaller and smaller until it disappears. Then new research starts to solve new problem. This process keeps on going on, reaching ever higher according to the constant and changing Yin and Yang cycle.

In [6] it is argued that syntax and semantics are the Yin and the Yang of the Web, and should be complementary to each other rather than independent – or worse, incompatible – from one another. Similarly, academic research and industrial practice can be viewed as the Yin and Yang of technology adoption: interdependent and complimentary. Harrison makes a similar argument in a recent article [5], in which he advocates the need to discuss technology transition from both the academic and industrial points of view.

3.1 The Yin: Research

To make research results easier to adopt by industry, there needs to be a much clearer communication channel between the academic researchers and industrial practitioners. In many ways, this is standard requirements engineering and as software engineers it should be common practice. Alas, it is more often an example of "Do as I say, not as I do."

It is also important for researchers to have conclusive results that provide unequivocal evidence as to the efficacy of the results. Many research papers attempt to illustrate the potential of their results through needlessly complicated mathematical formulas. Mathematics has an important role to play in explaining theory and verifying characteristics of a system. However, the equations should clarify the results, not obfuscate them. Too often the mathematics in many software engineering research

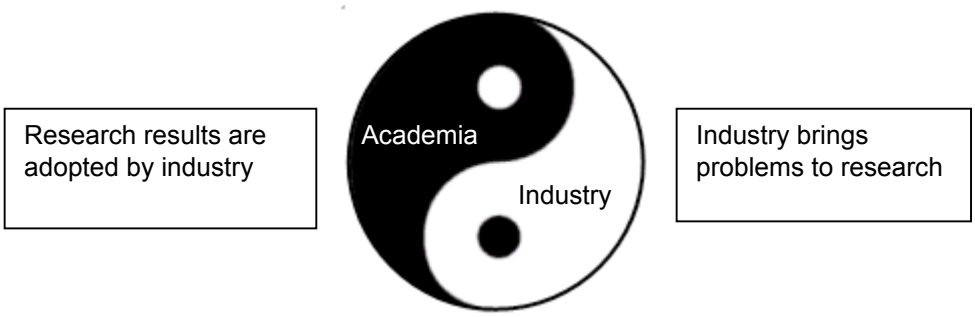


Figure 1: Yin and Yang

papers appear to accompany the prose purely for the sake of providing the illusion of formalism where none was warranted.

Empirical studies can provide objective measures of efficacy that industry can readily understand. If the studies are accompanied by detailed instructions of on how replicate the experimental results, or how to tailor the process for trials at the industrial partner's facility, all the better. Unfortunately, empirical studies remain an under-utilized tool in service of technology adoption [3].

3.2 The Yang: Practice

As the recipient of research results, industry needs to play a more active role in the problem of technology adoption. For example, providing clearer requirements would greatly help the researchers steer their work towards outcomes with a better chance of success than if they are working without any guidance. However, requirements engineering requires two parties, and just as software engineering researchers are often poor masters of requirements elicitation, industrial partners are often not very good and clearly communicating their needs.

Part of the problem may lie in the fundamentally different goals of academic research and industrial practice. Researchers tend to try for a 100% solution; practitioners may often be satisfied with an 80% solution, leaving the remaining 20% for the next iteration of the Yin and Yang cycle. This partial solution is often preferable to a full theoretical solution that is correct in all cases on paper, but impractical when it comes to realization.

Cryptography is an excellent example of such an area. It relies on basic and fundamental theoretical research. Yet for it to be put into practice requires quite a different mindset. In the book *Practical Cryptography* [4], Ferguson and Schneier state:

> "Building real-world cryptographic systems is vastly different from the abstract world of most books on cryptography, which discuss a pure mathematical ideal that magically solves your security problems. Designers and implementers live in a very different world, where nothing is perfect and where experience shows that most cryptographic systems are broken due to problems that have nothing to do with mathematics."

This quote is particularly interesting, since Schneier is the author of one of the best-known textbooks on theoretical cryptographic methods [7]. Software engineering researchers could benefit by learning from his change of philosophy in the last eight years.

3.3 Yin and Yang: Research and Practice

As shown in Figure 1, the Yin and Yang model consists with static model and a dynamic model, both part of the same whole [10]. In the static model, the world as a whole consists with a pair of two connected but opposite halves, each as half of the small opposite spot inside with the opposite outside. This model can be used to explain many everyday situations. For example, modeling the pair of employee and employer describes the relationship of administrator (under control) and administrated (incontrollable).

In the dynamic model, the pair of opposites changes and evolves in continuous manner, transitioning in two directions (forward and backward) simultaneously and recursively. The dynamic view helps to consider how and when enforcing or adopting the opposite for reaching one's own goals is the best choice.

When academic research and industrial practice are viewed using Yin and Yang as the whole world, the relationships between the two become apparent. The static model indicates that "Academic Research" always takes advantage of some industrial practice gains as the

small spot inside, and the positive or negative impact of "Industrial Practice" as the outside opposite partner. The dynamic model indicates that neither "Academic Research" nor "Industrial Practice" should ever overcome one another for too long; if this happens, the cycle of innovation begins again.

The balance between academic research and industrial practice must be maintained. The equilibrium between the two is their natural state; it is perturbed when new requirements arise and new results are made available. Using such a holistic approach that incorporates the Yin views of research and the Yang views of practice can foster technology adoption.

References

[1] "Chinese Philosophy: Yin and Yang" Online at http://www.wsu.edu:8080/~dee/CHPHIL/YINYANG.htm.

[2] "Where does the Yin and Yang Symbol come from?" Online at http://www.chinesefortunecalendar.com/yinyang.htm.

[3] Budgen, D.; Hoffnagle, G.; Müller, M.; Robert, F.; Sellami, A.; and Tilley, S. "Empirical Software Engineering: A Roadmap." To appear in *Proceedings of the 10th International Conference on Software Technology and Engineering Practice* (STEP 2002: Oct. 6-8, 2002; Montréal, Canada). Los Alamitos, CA: IEEE Computer Society Press, 2003.

[4] Ferguson, F.; and Schneier, B. *Practical Cryptography*. John Wiley & Sons, 2003.

[5] Harrison, W. "The Marriage of Research and Practice." *IEEE Software*, pp. 5-7, March/April 2003. IEEE Computer Society Press, 2003.

[6] Patel-Schneider, P; and Siméon, S. "The Yin/Yang Web: XML Syntax and RDF Semantics." *Proceedings of WWW 2002* (May 7-11, 2002; Honolulu, HI). ACM Press, 2002.

[7] Schneier, B. *Applied Cryptography: Protocols, Algorithms, and Source Code in C* (2nd Edition). John Wiley & Sons, 1995.

[8] Tilley, S.; Huang, S.; and Payne, T. "On the Challenges of Adopting ROTS Software." *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering* (ACSE 2003: May 9, 2003; Portland, OR).

[9] Trochim, W. *Research Methods Knowledge Base*. Online at http://trochim.human.cornell.edu/kb/.

[10] Zhou, Z. "CMM in Changing Environment with Uncertainty – Injecting Ancient Chinese Philosophy into Modern Science and Technology." To appear in *Communications of the ACM*, 2003.

Two good reasons why new software processes are not adopted

Stan Rifkin
Master Systems Inc.
2604B El Camino Real
Carlsbad, California 92008 USA
✆+1 760 729 3388 sr@Master-Systems.com

Abstract

There are many reasons we do not adopt software engineering processes, including those associated with tools. This paper presents two of the most persuasive reasons, based on a literature review of 175 references.

The archetypal dimensions of adopting a new things are: attributes of the thing itself (classically relative advantage, compatibility, complexity, trialability, and observability), qualities of the adopters, the strength of opinion leaders in diffusion networks, characteristics of the change agent, and organizational factors. [16] In addition, other authors have identified environmental factors, too. Lopata cites seven of them. [7] All of these factor studies suffer several deficiencies: they are static and linear combinations, if combinations at all; there is no priority of factors; there is no time variation of the influence of the factors.

In conducting a literature search for a related paper [15], the author read over 175 references seeking to understand what drives software engineering process adoption. The author believes that many of the factors presented in the literature are actually dissatisfiers, that is, their absence will signal adoption impediments, but their presence is not a sufficient condition for adoption. Presented here are two satisfiers, that is, if the dissatisfiers are addressed then these models positively explain adoption.

1. Two good reasons

The basis of selection for these two reasons is over-simple: They elegantly explain a great deal of otherwise monolithic approaches, such as factor studies that try to identify and isolate the controlling influences on adoption. The two answers below are more dynamic and identify that certain factors are more influential during certain epochs or under certain conditions and not at other times/conditions. Such a contingency style ("What is critical for adoption?" "It depends!") reveals far more than any set of factors that are linearly aligned in an inexorable (or unstated) time sequence. Also, both answers leave plenty of room for human forces, technical details, and organizational/environmental influences, all of which are part of the rich reality of implementing software engineering processes.

1.1. The first model

This model is taken from Repenning [13]. The explanation of process adoption relies on Figure 1, below. The grammar of the diagram was first popularized in Senge [19], where it is called a causal loop diagram. The intuition is that there are three forces that determine whether a new process will be used in practice: normative pressure, reinforcement, and diffusion.

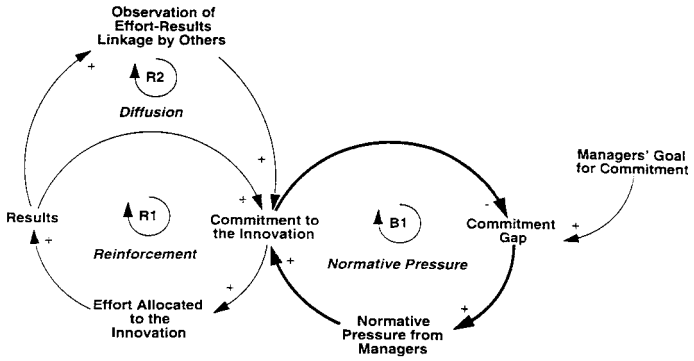


Figure 1. Arrangement of the dynamic forces of implementation. (from [13], pp. 109-127. Reprinted by permission of the Institute for Operations Research and the Management Sciences (INFORMS))

- Normative pressure is that exerted by management to meet expectations, to achieve norms. Managers set goals for commitment to implement the innovation (in this case, process improvement). If the gap between the managers' goal and the current commitment is large enough, then the pressure on those affected is increased to raise their commitment to implement.
- Reinforcement is the process by which the pressure to increase commitment is translated into effort. In this model there is a direct relationship between effort and results, so as effort is increased then positive results are, too.
- Diffusion is something of the flywheel effect in which those affected observe improved results so they, in turn, increase their commitment to implement the improvement innovation.

The explanation -- composed of the (necessarily) linear arrangement of words, sentences, and paragraphs -- gives the appearance that managers' normative intentions might begin the whole process, and then the flow proceeds in the manner described above for the first time through. After that, things can get interesting. For example, Repenning (p. 120) described an instance where the diffusion loop damps the commitment to implement when the results appear to be disproportionately low with respect to the effort allocated.

The simulation model in the title of Repenning's article illustrates the interaction among the three forces. Essentially, the two loops with the R1 and R2 labels tend to amplify effects, because there are + marks all the way around each loop; the one marked B1, where *B* stands for balancing, because it has an odd number of – marks [14], can reduce future commitment as the gap between actual commitment and the managers' goals closes.

Now we can see the ups and downs of implementation:

- When the managers' goals for commitment are not sufficiently different from the current commitment then there will be insufficient pressure to commit going forward.
- Whenever the effort is (too) low, then the results will be low and the commitment will decrease in a vicious cycle.
- Whenever the effort-results linkage observed is (too) low, then others will not be inspired to commit and the effort allocated will be decreased, decreasing the results still more, in a vicious cycle.

Repenning was able to reproduce in his model the situation in which managers set appropriate goals, allocate sufficient effort and then underestimate the delay needed to achieve results, so the commitment is eroded and the results fall off because of the connections among the goal, commitment, effort, and results. With another set of values, Repenning showed that once the flywheel effect of diffusion is in place, due to the long-term positive relationship between effort and results, then normative pressure does not play such an important role, can be removed, and the implementation continues its virtuous cycle.

At the end of the article, Repenning gives advice to managers facing the task of implementation:

1. Do not prepare to implement something new until and unless those who control resources become "fully committed to the effort and patient in the months between adopting" and to having the results motivate further deployment.
2. While seeking to have the results themselves stimulate the flywheel effect, do not do this at all costs. Such a Herculean effort would be seen by future adopters as consuming an effort disproportionate to the results, so that the virtuous cycle would not happen.

The first bit of advice is important because so many authors implore their readers to frame the process improvement implementation as a project, rather like a software project. This would miss the point that planning a software project is by and large a solved problem, while planning human changes, especially by engineers and engineering managers, is not. Accordingly, Repenning's advice can be seen as a case perhaps for *planning* a process improvement as a project, but then do not *implement* it as a project, as it is too difficult to estimate the relationships among the variables.[1]

1.2. Advantages of the first model

There are several reasons that Repenning is a superior source on understanding why new processes are not adopted:

- It has face validity, that is, it tracks what we already know by personal, idiosyncratic experience, and by the experience of others (to be detailed below as part of the literature review)
- It pulls in the characteristics we customarily, perhaps cursorily, associate with implementation success, such as leadership (setting norms and sticking with them), managing change (how improvement is communicated, as in the effort-results link), allocating sufficient resources (effort in this case), rewards, and the need to begin improvement with sufficient energy.
- It takes into account many forces, not just a single one.
- Those forces are arranged in a simple structure that can have a complex, non-linear interaction. Causes may become effects, there can be competition among the forces or they can align, and, therefore, not only success can be explained but so can failure. And the possible ups and downs are illustrated by the model.
- It describes both a process and factors.
- It depends upon and sums up considerable theory. It is not just one person's bright idea.
- Without the insight gained by using the model we are unlikely to succeed on intuition alone.

1.3. The second model

In her article, Markus [8] guides us through the "home grounds" of the two most prevalent arguments about why process innovations are not adopted: either the process (or system of processes) itself is flawed in some technical respect (e.g., hard to use) [4], or the intended targets of the improvement (we humans) have some inherent reason to resist the implementation [17]. That is, there is a system-determined answer and a people-determined answer; the result in both cases is resistance. It is, therefore, the role of the implementer to either restructure the technical aspects of the system or restructure the people aspects (rewards, incentives, span of control, new job titles).

Markus notes that we see this dichotomy in solutions: some solutions address purely technical aspects, such as user involvement in the requirements and design phases, and others address how humans change in response to new processes trying to be introduced. She proposes a third theory, interaction, that does not rely on the assumptions

[1] Mark Paulk frames it differently. Some software projects are planned as discovery activities, iteratively reducing equivocality in the problem, solution, and/or project spaces. Implementation can gainfully be planned and performed this way, in planned cycles that iteratively identify and reduce risk. (Personal communication.)

of the other two. There are two variants of interaction theory:

1. Sociotechnical: it's all one system, and every part interacts with the others [1,5,18].
2. Political: it's about power, who has it, and who loses and gains with the introduction of the new stuff.

In Table 1 Markus frames her insights in terms of resistance. Like any good theory, these three can be used to predict where to look for problems and solutions.

What she finds, and asks us readers to look closely at our own situations for, is that (even) when people- and system-determined problems are addressed and solved, "resistance" remains, but when interaction with the organizational context or power distribution is addressed, then the "resistance" goes away. Accordingly, interaction theory is a better (normative) guide for implementation.

Looking at interaction instead of people or systems implies that a certain kind of information is used as evidence of implementation. That kind of information is not usually valued by us engineers or business people. The logic of using this kind of evidence begins with a worldview or ontology.

	People-Determined	System-Determined	Interaction Theory
Cause of resistance	Factors internal to people and groups	System factors such as technical excellence and ergonomics	Interaction of system and context of use
	Cognitive style Personality traits Human nature	Lack of user-friendliness Poor human factors Inadequate technical design or implementation	*Sociotechnical variant*: Interaction of system with division labor *Political variant*: Interaction of system with distribution of intra-organizational power
Assumptions about purposes of information systems	Purposes of systems are consistent with Rational Theory of Management, can be excluded from further consideration	Purposes of systems are consistent with Rational Theory of Management, can be excluded from further consideration	*Sociotechnical variant*: Systems may have the purpose to change organizational culture, not just workflow *Political variant*: Systems may be intended to change the balance of power
Assumptions about organizations	Organizational goals shared by all participants	Organizational goals shared by all participants	*Sociotechnical variant*: Goals conditioned by history *Political variant*: Goals differ by organizational location; conflict is endemic
Assumptions about resistance	Resistance is attribute of the intended system user; undesirable behavior	Resistance is attribute of the intended system user; undesirable behavior	Resistance is a product of the setting, users, and designers; neither desirable nor undesirable

Table 1. Theories of resistance: underlying assumptions. (from [8], pp. 430-444. (c) 1983 ACM, Inc. Reprinted by permission.)

	People-Determined	System-Determined	Interaction Theory (Political Variant)
Facts needed in real-world case for theory to be applicable	System is resisted, resistors differ from nonresistors on certain personal dimensions	System is resisted, system has technical problems	System is resisted, resistance occurs in the context of political struggles
Predictions derived from theories	Change the people involved, resistance will disappear	Fix technical problems, resistance will disappear	Changing individuals and/or fixing technical features will have little effect on resistance
	Job rotation among resistors and nonresistors	Improve system efficiency Improve data entry	Resistance will persist in spite of time, rotation, and technical improvements Interaction theory can explain other relevant organizational phenomena in addition to resistance

Table 2. Theories of resistance: predictions. (from[8], pp. 430-444. (c) 1983 ACM, Inc. Reprinted by permission.)

Ontologies are basic beliefs about how the world works. One example is positivism, which believes that there is an enduring reality that exists independent of our sensing or perception of it. When we turn our backs on a mountain it is still there! Another example is that the world is socially-constructed, i.e., that we make sense of what we perceive based on how society instructs us to. Each of these two examples also implies epistemology and methodology, that is, what can be known for sure and what methods generate such knowledge. Positivism, sometimes called "normal science," believes in "hard" facts – that is, quantitative measurements – obtained in such a way that the measurements can be obtained by anyone else equipped with the same instruments. Interpretivism, which corresponds to the social construction of reality, seeks to find the patterns that operate in social settings, the collections of phenomena that seem to fit together. In the interpretivist paradigm it is acceptable that the search for those patterns is in a social setting that cannot be repeated, because the environment is not controlled or even controllable, as in a test tube laboratory. Objectivity in this paradigm cannot be obtained. The methods are generally called qualitative [2,10,11,12,20].

The interaction framework espoused by Markus means leaving the methods of normal science (and engineering and commerce) in favor of interpretation, a form of subjective judgment. If we accept the invitation to take into account new kinds of information (namely subjective sources) then we may see things we did not before. But, it is difficult to let go what we think we can know for sure in exchange for learning more about the situation from less of an absolute perspective.

It is worth mentioning that one of the objections of normal science is that social scientists "make up" constructs, such as morale, intelligence, and power, that those constructs do not have an existence independent of their definitions. Abraham [6], a recovering physicist, has argued persuasively that the constructs of classical physics, such as distance, acceleration, and force, to mention but a few, are no less "made up" and do not exist independent of our thoughts about them. That we ascribe measurements to distance, acceleration, and force reify them precisely to the extent that measurements of morale, intelligence, and power do.

One of the popular ways to express that the social construction of reality acts as filter on what we see is the often-cited quip quoted by Karl Weick [21], p. 1. It refers to American baseball, where a ball is thrown (pitched) towards a batter. If the batter does not swing, then a judge (an umpire) calls either "ball" if the trajectory was outside a mythical box between the shoulders of the batter and his knees, or "strike" if it was inside that box. Three umpires were talking. The first said, "I calls them as they is." The second said, "I calls them as I sees them." The third and cleverest umpire said, "They ain't nothin' till I calls them." Later Weick avers that when people say "I'll believe it when I see it," they more likely mean "I'll see it when I believe it." And, quoting another source, "man is an animal suspended in webs of significance he himself has spun." (pp. 134-135)

1.4. Advantages of the second model

Like the first model, this one incorporates other theories [9], so it is not (just) one person's bright idea. It also addresses competing theories that are likely the most prevalent in the implementation literature and practice, so the insights are novel and useful. It also predicts the problems and solutions better than the other two theories. In addition, "resistance" is redefined as natural and a part of any change, not something to be conquered and overcome. And last, it invites us to broaden our computer science-, software

engineering-centric methods for observing and gathering information, something that many implementers feel is necessary to be successful, that somehow trying harder with what we already know how to do is not more effective. [3]

2. Implications

As designers of processes and tools that we want adopted by others, we should understand that there is only so much power in the technical content of our processes and tools. As change agents, that is, implementers of processes and tools, we should understand that the contours of the process and tool are basically dissatisfiers, factors to be overcome, and that we should turn our attention towards the human aspects, especially the collective aspects, of implementation. Power and how our focus shifts over the period of adoption trump technical features every time!

Several examples may help to illustrate this dichotomy. Imagine a software engineering tool that aids component reuse by keeping track of in which programs/classes the components are used and in which version or variant. While this seems innocuous enough, because it appears to be a central repository it would have to fit into an enterprise that is centrally organized. Trying to fit a central repository of component use into an organization that is decentralized would be a challenge, even though we might all agree that the tool is inherently useful. That is, while it has technical merits it cannot be implemented in certain kinds of organizations. Or, imagine a tool that finds errors in static text and is free (such as `lint`). Clearly this is useful, has relative advantage. But it upsets the power structure because it points out defects. What programmer would want to have a list of his/her defects that could be used against him/her – particularly at the height of a career? Yet most advice about implementation says to recruit the opinion leaders, the professionals who are respected precisely because they are at the height of their careers!

What to do? Realize that many tools and processes are point solutions, meant to be inserted into a much, much larger context, one that may not be hospitable. Therefore, point solutions need to be integrated from the start into their larger environments and tested for value in that context.

3. Acknowledgements

The paper this one was based on has benefited from improvements suggested by Eric Busboom, Ray Fleming, Suzanne Garcia, Robert Glass, Watts Humphrey, Philip Johnson, John Kunz, Ray Levitt, Steve Ornburn, Mark Paulk, Shari Lawrence Pfleeger, and John Tittle. I am especially grateful to Marvin Zelkowitz for letting me express some thoughts that had been brewing for a long time.

4. References

[1] Bostrom, R. P., and Heinen, J. S. (1977, September). MIS problems and failures: A socio-technical perspective, part I: the causes. *MIS Quarterly, 1*(3), 17-32.

[2] Burrell, G., and Morgan, G. (1979). "Sociological paradigms and organizational analysis." Heinemann, Portsmouth, NH.

[3] Butler, B., & Gibbons, D. (1998). Power distribution as a catalyst and consequence of decentralized diffusion. *In* "Information systems innovation and diffusion: issues and directions" (T. J. Larsen, and E. McGuire, Eds.), pp. 3-28. Idea, Hershey, PA.

[4] Fitzgerald, B. (1996, January). Formalized systems development methodologies: A critical perspective. *Information Systems Journal, 6*(1), 3-23.

[5] Harris, M. (1996, March). Organizational politics, strategic change and the

evaluation of CAD. *Journal of Information Technology, 11*(1), 51-8.

[6] Kaplan, A. (1964). "The conduct of inquiry: Methodology for behavioral science." Chandler Pub. Co., San Francisco.

[7] Lopata, C. L. (June 1993). *The cooperative implementation of information technology: a process of mutual adaptation*. Unpublished doctoral dissertation, Drexel University, Philadelphia, PA.

[8] Markus, M. L. (1983, June). Power, politics, and MIS implementation. *Communications of the ACM, 26*(8), 430-444 .

[9] Markus, M. L., and Robey, D. (1988, May). Information technology and organizational change: Causal structure in theory and research. *Management Science, 34*(5), 583-598.

[10] McMaster, T., Vidgen, R. T., & Wastell, D. G. (1997). Technology transfer: diffusion or translation? *In* "Facilitating technology transfer through partnership: Learning from practice and research (Proceedings of the IFIP TC8 WG8.6 international working conference on diffusion, adoption and implementation of information technology)" (McMaster, Tom, Mumford, Enid, Swanson, E. Burton, Warboys, Brian, and Wastell, David, Eds.), Ambleside, Cumbria, UK, pp. 64-75. Chapman & Hall, London.

[11] Meyerson, D., and Martin, J. (1987, November). Cultural change: An integration of three different views. *Journal of Management Studies, 24*(6), 623-647.

[12] Orlikowski, W. J., and Baroudi, J. J. (1991, March). Studying information technology in organizations: Research approaches and assumptions. *Information Systems Research, 2*(1), 1-28.

[13] Repenning, N. P. (2002, March-April). A simulation-based approach to understanding the dynamics of innovation implementation. *Organization Science, 13*(2), 109-127.

[14] Richardson, G. P. (1991). "Feedback thought in social science and systems theory." University of Pennsylvania Press, Philadelphia, PA.

[15] Rifkin, S. (2003). Why software processes are not adopted. *In* "Advances in Computers" (M. Zelkowitz, Ed.), vol. 54 ed.. Elsevier, New York.

[16] Rogers, E. M. (1995). "Diffusion of innovations." (4th ed.) The Free Press, New York, NY.

[17] Roth, G., and Kleiner, A. (2000). "Car launch: The human side of managing change." Oxford University Press, New York, NY.

[18] Ryan, T. F. , and Bock, D. B. (1992, November). A socio-technical systems viewpoint to CASE tool adoption. *Journal of Systems Management, 43*(11), 25-9.

[19] Senge, P. M. (1990). "The fifth discipline: The art & practice of the learning organization." Currency Doubleday, New York, NY.

[20] Silva, J., & Backhouse, J. (1997). Becoming part of the furniture: The institutionalization of information systems. *In* "Information systems and qualitative research (Proceedings of the IFIP TC8 WG 8.2 International Conference on Information Systems and Qualitative Research, May 31- June 3, 1997, in Philadelphia, Pennsylvania USA)" (A. S. Lee, J. Liebenau, and J. I. DeGross, Eds.), Chap. 20, pp. 389-414. Chapman & Hall, London.

[21] Weick, K. (1979). "The social psychology of organizing." (2nd) Wiley, New York.

Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE)

Hausi A. Müller and Anke Weber
Department of Computer Science
University of Victoria, Canada
{hausi,anke@cs.uvic.ca}

Ken Wong
Department of Computing Science
University of Alberta, Canada
kenw@cs.ualberta.ca

Abstract

Common office suites are capable, mature, flexible, extensible, and familiar to many developers. For example, they are used daily to browse the Web, produce multimedia documents, prepare presentations, maintain budgets, and to construct Web contents. These Commercial Of-The-Shelf (COTS) software products and middleware-based environments can be extended and leveraged to provide familiar support for software engineering tasks and thus may ease the barriers to adoption. Our hypothesis is that users will more likely adopt tools that work in an environment they use daily and know intimately. That is, tool adoption will be improved if we specifically address the issues of cognitive support and interoperability. In this paper, we describe how we address these issues with our Adoption-Centric Reverse Engineering (ACRE) project and specifically with our tool environment ACRE V1.0.

1. Introduction

Over the past decade, we have directly encountered and experienced the research tool adoption problem in many guises in the process of deploying software reverse engineering and software visualization research tools in industry. Analyzing these problems, we realized that the most critical adoption issues stem from integration and interoperability problems of our tools with respect to the mental models of the developers and their existing development tools.

First of all, the cognitive support afforded by our tools was not compatible with the cognitive support afforded by the existing development tools [10]. Software development tools aid software engineers by participating in their thinking and work. Thus, when we use the term cognitive support, we mean the principles and means by which cognitive software processes are supported or aided by software engineering tools. Working with a set of existing tools, software developers build up cognitive support over time. Leveraging this hard-won cognitive support effectively is critical for their overall productivity and efficiency. Thus, developers easily reject a new tool if it does not jive with their valuable cognitive support model. The problem of inadequate cognitive support in our tools became evident through informal feedback, user studies, and structured tool demonstrations [9] [7] [8].

Second, a single research tool intended to aid software development typically addresses only few development, understanding, or maintenance tasks. Thus, such research tools must interoperate with other tools through integration mechanisms, such as data integration (i.e., so that tools can read and write common data interchange formats, and control integration (i.e., so that one tool can control another), and presentation integration (i.e., so that several tools have a uniform, familiar look-and-feel) [13]. For example, the end-user programming capability [14] through the scripting layer in Rigi [2] allows it to coordinate other tools or to be controlled by other tools. Tool builders have exploited this Rigi feature successfully to create new tool environments (e.g., Dali [17], Bauhaus Rigi [16], or Shimba [15]).

Our main hypothesis is that in order for new tools to be adopted effectively, they must be compatible with both existing users and existing tools. To validate this hypothesis, we are building prototype software engineering tools based on open standards (e.g., Scalable Vector Graphics (SVG) [19], and GXL Graph eXchange Language (GXL) [12], both based on XML) popular office suites (e.g., Microsoft Office XP, Lotus SmartSuite, Sun StarOffice, Adobe Acrobat, and Corel WordPerfect Office), and common middleware technology and plug-in platforms (e.g. scripting languages, Model-Driven Architecture (MDA), IBM Websphere, and Eclipse). Using these, we will conduct industrial case studies and structured tool experiments to validate our hypotheses. The experience gained in this endeavour will be beneficial to both academic research and industrial practice.

Table 1 Meeting RE Requirements with MS Excel, PowerPoint and MS Visio

RE Requirement	Excel	PowerPoint	Visio
Visualize program information artifacts and architecture	Drawing tools	Custom presentations and animation	Templates for diagrams (e.g., UML, Web sites)
Statistical data analysis and metrics	Statistical functions, charts, forecasting		Built-in charts
Re-document system	Report builder; synch with data sources	Custom presentations and animation	Synchronize with data sources
Collaboration features	Protect and share workbook Track changes	Track changes and merge documents	
	Meeting scheduling and sending documents via email		
Robust	Standard functionalities to build upon; e.g. "undo"		
Data-driven and net-centric	Web services, Smart Tags, and Dashboards, Web publishing Integrate with databases (e.g., Access) and MS server environments		
Interoperability	Active X/OLE support Support for XML, SVG objects, and many other objects		
End-user programmable and Office automation	Macro recording, playing, and editing, Scripting with VB script (Com) Add-Ins with VBA, Dynamic Libraries with .Net		
Leverage cognitive support	Popularly adopted and familiar		

2. Extending Common Office Tools and Middleware Technology: ACRE V1.0

Office suites are highly popular platforms that typically offer a number of programmable core functions and applications for document creation, drawing, database storage, spreadsheet, and presentation. Table 1 lists some of the functionalities of MS Office and Visio, which we can exploit for building software reverse engineering and visualization tools on top of these platforms.

Building software development environments using these kinds of commercial products is not a new idea. A cornerstone of the Desert environment is a custom editor, based on the Adobe FrameMaker application to produce source code and architectural documentation [5]. Also, the Visual Design Editor (VDE) is a domain-specific graph editor built on top of Microsoft's PowerPoint application using the Visual Basic scripting language [3].

ACRE V1.0 is the first version of the software evolution environment under development at the University of Victoria as part of our *ACRE (Adoption-Centric Reverse Engineering)* project [5]. It consists of several software visualization engines on top of various office products, including Lotus Notes, Excel, PowerPoint, and Visio. The software engineering tools in our ACRE environment interoperate using the ACRE persistence engine and SVG (Scaleable Vector Graphics). SVG, a W3C XML standard, is an effective solution for smart cross-platform graphics.

In the following subsections, we give short summaries of our implementations. More details can be found on the ACSE Web site [1].

2.1. The ACRE Persistence Engine

Figure 1 illustrates the architecture of the ACRE Persistence Engine, an extensible middleware system upon which we develop software engineering tools [1]. The architecture utilizes international standards and a common data format for third party integration.

The ACRE persistence engine is implemented using the IBM Websphere software platform, the OMG's Model Driven Architecture (MDA), and OTI's universal tool platform Eclipse. We use open standards for network and data exchange services (e.g., Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and Open database Connectivity (ODBC).

The ACRE Persistence Engine supports communication and XML/GXL-based data exchange between the various ACRE clients, which are described in the following subsections.

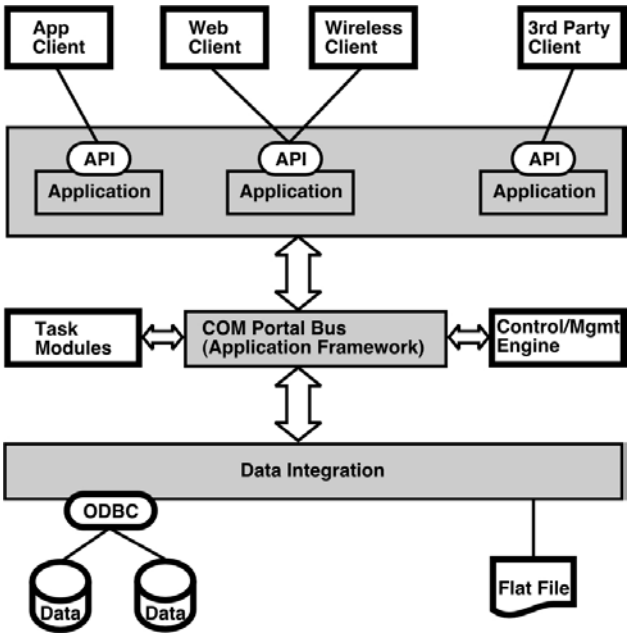


Figure 1: Architecture of the ACRE Persistence Engine

2.2. The ACRE SVG Visualization Engine

The ACRE SVG Visualization Engine (ASVE) is a graph visualization engine for exploring and annotating software artifacts [4]. The user can filter, rearrange, layout, annotate, display, and change the visual characteristics of nodes and arcs to better understand the architecture of the software system.

It is built exclusively with SVG and ECMAScript. ASVE is embeddable into "host" applications such as Web browsers and office tools (e.g., PowerPoint, Excel, or Word). We have also implemented an ASVE visualization interface for LotusNotes.

2.3. Towards a Live User Manual for Software Engineering Documentation

An *ACRE Live Document* is data-driven, interactive, and adapts automatically and intelligently to its context (e.g., its word processor and its reader) [11]. We use Live Documents to overcome selected challenges in software engineering documentation:

- to synchronize code and documentation automatically (e.g., keep diagrams in sync with source code)
- to produce multiple output versions from one source consistently (e.g., for print, online, and audio use)
- to address different audience needs (e.g., user manuals for novice and expert users)
- to explore the system without leaving the document
- to support group collaboration

We have implemented the following Live Document features in Excel:

- We enhanced the documentation capabilities of the Rigi reverse engineering system: Our implementation manages different views on the Rigi graph data and statistics in Excel within one workbook. We also use Excel charts in PowerPoint for advanced presentations features.
- We capture Rigi graphs and display them in Excel, PowerPoint and Visio. We have implemented basic editing functions for Rigi graphs in Excel, PowerPoint, and Visio.

2.4. Leveraging Cognitive Support in Lotus Notes to build ACRE Tools

ACRENotes, developed with the groupware product Lotus Notes, stores both documents and data in Lotus Notes' document database [1]. Documents can easily be selected, browsed, filtered, and categorized. Predefined agents and actions can manage documents as well. By converting reverse engineering data (e.g., from GXL) into these documents, we can maximize the user's capability to manipulate reverse engineering data.

ACRENotes imports Rigi data in Lotus Notes' document format and visualizes it using a Java Swing program and an SVG component, respectively. ACRENotes leverages the data repository, data visualization, end-user programming, and team cooperation features of Lotus Notes.

2.5. An ACRE Metrics Tool in Visio

The ACRE metrics tool in Visio is a welcome addition to the Rigi graph editor. The metrics help the reverse engineer to capture design characteristics of the software system, to understand relationships among attributes, to support software maintenance, and to decide on software modification and reuse.

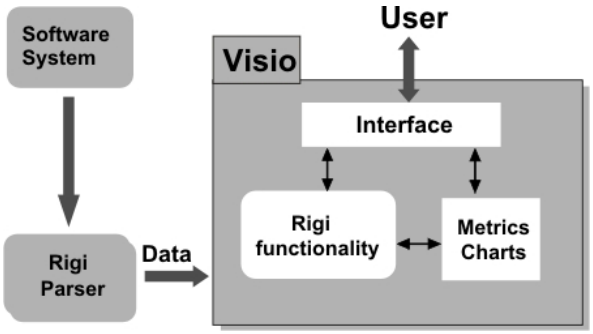


Figure 2: The ACRE Metrics Tool in Visio

Figure 2 illustrates how Rigi is connected to Visio, which displays the results of computing the following object-oriented metrics on the data provided by Rigi:

- LOC (Lines of Code)
- Lines of code (per method)
- NMC (Number of Methods per Class)
- CBO (Coupling Between Objects)
- Number of accessing classes
- RFC (Response For a Class)
- Number of external methods per class

Figure 3 shows an example chart for the NMC metrics in Visio. In the top left corner you can see the customized Rigi toolbar that controls the input of the data and the generation of the Visio charts.

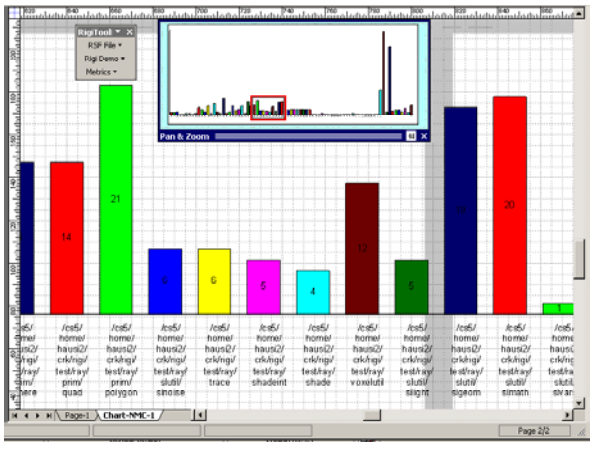


Figure 3: Example NMC Chart in Visio

Visio displays professional looking charts for these metrics. Another useful feature that is difficult to implement from scratch, but comes for free when building on the Visio platform, is the advanced zooming functionality on the chart that the panel in the right upper corner provides.

3. Visio as an adoption-centric platform

Visio provides advanced features for building UML diagrams. These include building static diagrams for Visual C++ and VB code. However, it does not yet consider the relationship between class models. We plan to build on these features to provide an advanced and easy-to-adopt UML editor as a case study for a comparison with professional tools.

Another planned case study is to visualize artifacts to document Web site evolution. Reverse engineering of Web sites requires adequate visualization of Web site maps and Web site architecture. For example, even Adobe GoLive, a standard tool for Web site development, does the visualization of site hierarchies poorly. It visualizes too much information and the hierarchy is difficult to navigate. Furthermore, broken links only appear in one special window and not in the hierarchy visualization itself.

Our target tool for Web site visualization is Visio. Visio can parse Web sites and supports Web site visualization. The visualization is still rudimentary, but the parser seems to be able to visualize applications within Web pages, such as JavaScript, ASP pages, etc. Furthermore, Visio supports many shapes and diagrams. We can therefore implement different visualizations for Web site architectures (e.g., according to different RE problems) and synchronize them by programming Visio. For example, Rigi (implemented in Visio) can be one of these visualizations. Another possibility is UML diagrams for the site architecture based on Visio UML diagrams. In this case, the UML editor would be specifically targeted to Web site evolution. In a further step, it would be worthwhile to integrate the visualization with Excel features (e.g., for metrics on Web sites or statistics). In this context, we will also explore how and if the API's of Macromedia Dreamweaver and Adobe GoLive allow the extension of this tools towards more advanced Web site evolution tools.

4. Evaluation

Using ACRE V1.0, we will conduct industrial case studies and structured tool experiments. Walenstein's PhD thesis [10] includes a survey of the types of phenomena that comprise cognitive support, including external memories, various external structures, and

scaffolding. It also proposes methods for systematically enumerating the cognitive support provided by tools. We will specifically investigate what kind of cognitive support is needed and suited for software engineering tools, and we will examine how to leverage the cognitive support already provided by existing office tools effectively. Walenstein's position paper, contained in these proceedings, illustrates further how cognitive support can be characterized using different factors and how these factors can be used to evaluate and validate cognitive support and in turn adoptability [18].

5. Implementation Experiences

One of the keys for effective MS Office automation is to understand the MS Office object models and the Microsoft terms and technologies (e.g., DCOM, COM, ActiveX) as well as the installation procedures and the scope of packages like Office XP developer. This is even more difficult as the documentation and the relevant Microsoft Web pages are complex and information is hard to find.

In summary, the visualization of Rigi graphs with PowerPoint drawing objects for the nodes and arcs scales poorly. For example, loading a Rigi graph as a PowerPoint drawing needs about three times longer than loading the corresponding SVG plug-in. Once the MS PowerPoint graph is drawn on the slide, changing slides performs normal and efficiently fast in contrast to the behavior of the SVG plug-in.

Implementations on top of MS Office can be ported from one Office tool to another without major implementation changes (e.g., charts use a similar API in Excel and PowerPoint). Programming drawing elements in Visio requires a different programming approach that is based on the master and stencil paradigm in Visio.

We further experimented with Internet Explorer, with Microsoft Word XP, and with Microsoft PowerPoint XP for embedding SVG components, and with Adobe Illustrator 9.0 and 10.0 for creating them. The implementation of complex interactions (e.g., graph filtering) was fast in comparison with the same task done in Tcl/Tk for the Rigi user interface. As SVG is not a high-level language, manual programming is tedious and repetitious. Consequently, automatic generation of SVG files is the appropriate approach for SVG file creation. Loading of SVG components does not (yet) scale very well; all used tools freeze for a recognizable time while loading complex graphs. Once loaded, the graphics perform efficiently in PowerPoint and Internet Explorer but they slow down document behavior, e.g. opening a slide with a SVG component requires the same amount of time for initially loading it freezes the tool for a recognizable time, which is different to common graph formats such as JPG or TIF.

6. Acknowledgements

We would like to thank the entire ACRE/ACSE Team at the University of Victoria for the prototype implementations and documentation. We also would like to thank Jon Pipitone, University of Toronto, for his work on the implementation of the SVG graph visualization engine. This work has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Consortium for Software Engineering (CSER), and the Center for Advanced Studies (CAS), IBM Canada Ltd.

7. References

[1] ACSE Web page, http://www.acse.cs.uvic.ca/

[2] H. A. Müller and K. Klashinsky. "Rigi—A System for Programming-in-the-large," *10th IEEE/ACM International Conference on Software Engineering,* pp. 80-86, April 1988.

[3] N.M. Goldman and R.M. Balzer. "The ISI Visual Design Editor Generator," *IEEE Symposium on Visual Languages (VL '99),* pp. 20-27, September 1999.

[4] H. Kienle, A. Weber and H.A. Müller. "Leveraging SVG in the Rigi Reverse Engineering Tool," *SVG Open Developers Conference,* Zürich, Switzerland, July 15-17, 2002.

[5] H.A. Müller, M.-A. Storey and K. Wong. "Leveraging Cognitive Support and Modern Platforms for Adoption-Centric Reverse Engineering (ACRE)," CSER Research Proposal, November 2001.

[6] S. Reiss. "Simplifying Data Integration: The Design of the Desert Software Development Environment," *18th IEEE/ACM International Conference on Software Engineering (ICSE 1995),* pp. 398-407, May 1996.

[7] S. Sim and M.-A. Storey. "A Structured Demonstration of Program Comprehension Tools," *7th IEEE Working Conference on Reverse Engineering (WCRE 2000),* November 2000.

[8] M.-A. Storey, S. Sim and K.Wong, "A Collaborative Demonstration of Reverse Engineering Tools: The SORTIE Project," http://www.csr.uvic.ca/chisel/collab/

[9] M.-A. Storey, K. Wong and H.A. Müller. "How do Program Understanding Tools Affect how Programmers Understand Programs?" *Journal of Science of Computer Programming,* Vol. 36, No. 2-3, pp. 183-207, March 2000.

[10] A. Walenstein. "Cognitive Support in Software Engineering Tools: A Distributed Cognition Environment," Ph.D. Thesis, Department of Computing Science, Simon Fraser University, May 2002.

[11] A. Weber, H. Kienle and H.A. Müller. "Live Documents with Contextual, Data-Driven Information Components," *Proceedings of ACM SIGDOC 2002,* Toronto, Canada, October 2002.

[12] A. Winter, B. Kullbach and V. Riediger. "An Overview of the GXL Graph Exchange Language," Springer Verlag: S. Diehl (ed.) Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001.

[13] K. Wong, "The Reverse Engineering Notebook," Ph.D. Thesis, Department of Computer Science, University of Victoria, 1999.

[14] S.R. Tilley, K. Wong, M.-A.D. Storey and H.A. Müller. "Programmable Reverse Engineering," *International Journal of Software Engineering and Knowledge Enginering,* Vol. 4, No. 4, pp. 501-520, December 1994.

[15] T. Systä, T. K. Koskimies; and H.A. Müller. "Shimba—An Environment for Reverse Engineering Java Software Systems," Software—Practice and Experience, Vol. 31, No. 4, pp. 371-394, April 2001.

[16] R. Koschke. "Atomic Architectural Component Recovery for Program Understanding and Evolution," Ph.D. Thesis, Institute of Computer Science, University of Stuttgart, Germany, 2000.

[17] S. J. Carrière, S. G. Woods, R. Kazman. "Software Architecture Transformation," *IEEE Working Conference on Reverse Engineering (WCRE '99),* Atlanta Georgia, October 1999. http://www.sei.cmu.edu/ata/products_services/dali.html

[18] A. Walenstein. "Improving Adoptability by Preserving, Leveraging, and Adding Cognitive Support to Existing Tools and Environments," *Proceedings of the Adoption Centric Software Engineering (ACSE 2003),* Portland, Oregon, May 2003.

[19] W3C. Scalable Vector Graphics (SVG) 1.0 Specification, W3C Recommendation, September 2001.

.

Improving Adoptability by Preserving, Leveraging, and Adding Cognitive Support To Existing Tools and Environments

Andrew Walenstein
Software Research Laboratory
Center for Advanced Computer Science
University of Louisiana at Lafayette
walenste@ieee.org

Abstract

Being adoption-centric means focusing research on what technologies would be helpful to real users and trying to ensure that the results are more likely to be adopted. Too little is known about how to improve adoptability. This paper describes preliminary steps towards a framework for understanding methods for injecting innovations in a way that makes the results more likely to be adopted. The framework defines taxonomy of adaptations that tools and users undergo in the face of innovations. It then employs theories of distributed cognition to suggest which potential adaptations would be considered potentially desirable to users because they preserve, leverage, or add cognitive supports. An example is given illustrating how this framework is being used in exploratory design.

1. Introduction

The rate at which practitioners adopt the products of software engineering (SE) tools research suggests that much improvement is possible in making research results available and adoptable. In some cases the lack of adoption may very well be due to the way that software research prototypes are developed. Frequently a simple, stand-alone "demonstration" implementation is developed. Often this is a bare-boned and impoverished environment or tool when compared to the robust, full-featured, and highly usable tools and environments that practitioners normally work with.

An alternative approach to tools research is to employ an "adoption-centric" approach of building innovations as adaptations of the rich tools and environments currently existing in practice. Here, "tools" and "environments" are considered broadly, and would include editors, shells, browsers, word processors, personal information managers, and ordinary software development environments. An adoption-centric approach would perform the tools research with a concern for easy adoption of the tool by some real user community.

Several potential advantages can be offered for this approach. First, reusing an existing environment can make prototype development easier since the researchers do not need to spend time implementing and perfecting common but necessary infrastructure (undo, copy and paste, printing, help, etc.). Simple, bare-boned "toy" tools frequently miss these features or implement them awkwardly. This will normally seriously affect user performance and satisfaction, which will in turn make successful evaluation of the innovation exceedingly difficult. Second, by using an existing toolset one is more likely to find a user population to evaluate the tool on.

Being adoption-centric while adapting existing systems means that close attention will need to be paid to the ways software engineers currently work, and to how innovations can be fit into this work. This point is, in my opinion, the most critical aspect of the approach, and the place where the greatest research benefits can be expected. Being concerned for current work practices grounds the entire research effort in real user needs and situations. In addition, being concerned with how the innovations fit into real work helps assure that the research is in a position to make practical impacts sooner than the 17 or more years that some SE innovations have taken [12]. The adoption-centric approach is therefore not merely another attempt to build extensible development environments or to implement specific SE tools on top of other tools. There is a real concern for making innovations match realistic scenarios, and for introducing innovations to practical environments in ways that are likely to be more readily adopted. Attention to other factors such as marketing and organizational structure may also aid adoption (e.g., Fowler *et. al* [4]), but the prototype implementation is the main adoptability factor under the direct

control of most tools researchers.

Once the above general goals of adoption-centricity are stated, however, the question of how to actually go about achieving them looms large. How and why users adopt new technologies is not well known, and even less is known regarding how, exactly, one can build innovations that are more likely to be adopted.

This paper outlines a general framework for understanding adoption factors and recognizing opportunities for implementing innovations in ways that are more likely to be adopted. There are three main components to the framework. The first component is a taxonomy of types of adaptations, both for tools and users. This taxonomy is presented in Section 2. The second component is an analysis of how to interpret adaptations—and the resistance to such adaptations—in terms of adaptations to distributed cognitive systems. This analysis is presented in Section 3. The third component is a technique for analyzing existing toolsets for opportunities to inject new technologies in a ways that are likely to be adopted. The way this is currently being approached outlined in Section 4. Section 4 also outlines how the framework is being considered in a project relating to software clone detection and copyright violation litigation.

2. Types of adaptation in adoption

Users *adapt* to new tasks and technology. Such user adaptations include learning new concepts, skills, and problem solving techniques or strategies. As Mackay [11] pointed out, users and their environment actually *co*-adapt (also see Fowler *et. al* [4]). Users adapt their tools and environments to better suit their tasks and individual characteristics. Tool adaptations along these lines include setting key bindings, scripting, and programming. Users also adapt their entire information space in order to help solve problems. For instance software engineers implement file naming conventions in part because this makes their browsing and searching tools effective [7].

When any new technology or innovation is adopted by users, it means they adapt again to the changes. It seems likely that these adaptations could be effected in fundamentally different ways. A vocabulary for describing the different forms of adaptation is desirable. This section extends Mackay's analogy by using concepts from biological evolution to understand tool and user adaptation types.

Biological evolution and adaptation

Biological evolution can be seen as an extended process of adaptations to changing conditions. A naive conception of evolution is that it makes steady progress towards organisms of greater complexity and fitness. It is true that some organism features are associated with a history of gradual and incremental refinements of similar-but-less-fit features. For example, Dawkins reconstructed a history of how complicated eyes were built out of a series of additions and refinements of previous structures [2]. Even so, the fossil record also suggests that evolution should not be exclusively characterized as a uniform process of gradual refinement. The late evolutionary theorist Steven Jay Gould conspicuously argued that the evolutionary history is "punctuated" with periods of alternating relative stability and astonishingly rapid and wholesale changes which include radical changes to basic organism design [5]. This sort of distinction in evolutionary progression resonates with certain theories of knowledge acquisition and learning [14] which posit differences between learning by "accretion" and by "restructuring". Accretion occurs when the knowledge can be absorbed with only minor changes to the existing knowledge structures, whereas restructuring is made necessary by concepts and data that cannot be accommodated within the existing structures.

Why do adaptations even occur? Adaptations occur, at least in part, as responses to changes in living environment (e.g., climate). Adaptations in this sense improve the fitness of an organism to some ecosystem. Sometimes these adaptations are specific to particular ecosystems—the organisms become "specialists". An example is the giant panda, is adapted to survive on bamboo shoots and nothing else. In contrast, some organisms are generalists and can survive well in many ecosystems. An example the brown bear, which is omnivorous and ranges very widely.

How do new adaptations arise? One part of the story is simply by design variation through mutations which result in improved fitness to the ecosystem. Another part of the story is that an organism's existing features might be used for additional or new purposes. Gould called this "exaptation" [6]. An example he uses is how feathers may have originally provided warmth, but were eventually a step towards achieving airflight.

Applying the evolution metaphor to technology adoption

It is possible to see biological evolution as a metaphor for user and tool adaptation. Based on the above discussion, three basic contrasts might be helpfully applied to classify user and tool adaptations.

First, user and tool adaptation may be divided into gradual accumulation of design changes, and rapid, wholesale changes. The former is common in the slow evolution of product lines (e.g., creeping product features). Users also gradually adapt by learning different problem solving skills and tool features. Wholesale and rapid changes occur when users adopt radically different tools such as new operating systems, development environments, or office products.

Users are often painfully aware of how they need to adapt to such wholesale changes. Thus a first question for adoption-centric SE is "what type of adoption is being attempted: gradual accretion of localized design variations, or wholesale design changes?" The adoption-centric researcher must know which is being attempted—and which is needed.

Second, the issue of specialization versus generality needs frequently be considered for tool design. Specialized tools may be more fit for certain tasks but require specific learning (user adaptation) and may not "survive" changes in tasks. Task specificity is a common argument for or against various programming languages. An advantage of generalized tool capabilities is that once users learn them they can apply them in many situations. The drawback is that the general capabilities might work less well than the specialized versions, or users may need to do more work, or to customize them. In terms of adoption-centric design, the researcher should likely be encouraged to recognize specialized and generalized capabilities in both tools and users and take advantage of both when opportunities present themselves. For example, in certain work domains, programmers may be specialized to be highly familiar with spreadsheets. This specialized expertise might be exploited by implementing the innovation as an extension to a spreadsheet program. But also note that a spreadsheet itself is general in that it can be applied to many different tasks (compare, for example, a tool that performs fixed calculations).

Third, it may be helpful to distinguish two different classes of adaptations to existing tools, or aspects thereof. The first is by simple design mutation or accretion. For example, adding a call-graph visualization view to a toolset might be termed simple accretion, whereas changing the way error messages are displayed might be considered mutation. The second main class of adaptation type is by exaptation. Tool based exaptation might be said to occur if new uses are made for existing specialized functionality. For example, in Mackay's study [11], mail filtering functionality was used to implement smart filing of messages. In the realm of software development, Bellamy [1] noted that Smalltalk developers would use a cross-referencer as a way of locating semantically-related code. The developers would "tag" class methods as belonging to an application by inserting references to a dummy class. The cross-referencing was therefore not being used to trace down real calls, but to approximate a mechanism for clustering conceptually-related methods from multiple classes.

The adoption-centric researcher will want to be aware of whether changes are being made by mutation, accretion, or exaptation. One reason for wanting to know this, clearly, is that the adaptations made to tools are likely to induce similar types of adaptations to the users' knowledge of how to use the tools. Tool mutation implies that the user must adapt by modifying their skills and mental models for using the affected features. Accretion, on the other hand, is likely to allow simple knowledge accretion by the user. Exaptation is relevant to tool researchers because users may already be skilled in using the tool feature that is being exapted for a different purpose, or they may be able to exapt an existing feature or technique for use in conjunction with the new innovation.

3. Distributed cognition & legacy user systems

I am a *legacy user*. So, in all likelihood, are you and everyone else. My favorite editors are `Emacs` *and* `vi`. This fact might be viewed with considerable disdain by combatants on both sides of the long-running "vi versus Emacs Editor Wars". I use both editors practically every day for writing papers, programs, email, and numerous other activities. Many other and newer editors exist—certainly many specifically tailed for programming. Some of these, perhaps, are even superior to both of `Emacs` or `vi` (at least for some tasks and in some ways), although I may never truly know it. Myself and my computing environment in combination form a *legacy user system*.

I use the term "legacy user" in the way a software maintainer would expect: a legacy user is analogous to a *legacy software system* in SE. This term is intentionally selected. Cognitive science regularly views human minds as computer systems. Learning (i.e., user adaptation) serves to program and maintain the cognitive system.

The term "legacy user system" is no accident either. The cognitive science field of distributed cognition (DC) treats cognition as a computational process distributed between humans and tools (see Hutchins [9], Zhang *et. al* [17]). Thus users in combinations with tools are seen to form DC systems. From this point of view, external artifacts are seen to represent knowledge or cognitive states (goals, intentions, etc.), and both users and computers are seen to process such external knowledge and cognitive states. For instance, Flor *et al.* [3] analyzed programmer pairs from the DC point of view. In their analysis, they likened code scavenging to case-based knowledge use: when code is scavenged, it is copied with appropriate modifications, which is an analogue of schematic abstraction and instantiation. Only instead of occurring "in the head", it occurs in a text editor.

Legacy user systems are also computational systems: *legacy* computational systems. Legacy systems are not necessarily poorly maintained systems. Instead, they are identified by other characteristics: they are typically (1) considered "mission critical" for the organization using them, (2) not implemented using the most up-to-date technologies, although for various reasons it is desirable to bring them into compliance, and (3) poorly documented and understood. These are all characteristics of legacy users systems; the term is apt:

1. The DC system as a whole is critical for effective work. Clearly the user's own mind is mission critical, but just as clearly their normal environments are critical (or else adoption would not be a problem—they would be able to effectively use whatever environment is in front of them).
2. Updates to the legacy DC system is frequently desired and, in the case of adoption-centric research, assumed necessary.
3. The DC systems are almost entirely undocumented. As Hollan *et. al* [8] point out, the roles that artifacts play in cognition are often difficult to recognize. Careful field studies are therefore frequently needed in order to *reverse engineering* and *redocument* legacy DC systems in preparation for reengineering. It is well known that humans generally are unable to articulate how it is they think and act, and they certainly do not come with cognitive design documentation.

The main value in bringing DC theory into the present discussion is that it identifies aspects of legacy user systems which are important for effective distributed cognition.

More specifically, users rely on their external environment to provide *cognitive support*, i.e., to assist or help them in their cognition [16]. This support is partially attributable to the makeup of the tools. For instance most web browsers maintain link visitation history mechanisms, and will display the visitation status of links by rendering non-visited links in a different colour. Those features can support users by acting as external memory: users no longer need to remember where they have been. In other cases the support can be said to be "built up" in the environment through the various adaptations and customizations they make. For instance, the collection and organization of bookmarks is a lasting external memory that users often depend upon. Another example, already mentioned above, is the tagging of methods which was observed by Bellamy. These tags are needed if the programmer is to use the code location tools they are accustomed to. Over time, user and environmental adaptations generate a DC system in which the tools and their features support cognition, and in which the users have the skills, knowledge, and preferences for utilizing them effectively. Users are reluctant to adopt new technologies because doing requires new adaptation (learning), and may destroy the cognitive support built up in their environments, or make it less efficiently usable.

This analysis is helpful because it delves a little deeper into the barriers to adoption. Existing theories of adoption are compatible with this basic analysis (at least, the parts dealing with the adoption factors associated with individuals). For example the so-called "diffusion of innovation" theories [13] posit that individual evaluations of "usefulness", "compatibility" and "ease of use" greatly influence decisions to adopt. But what, precisely, is "usefulness" and "compatibility" and how can it be identified in tools? The answer I am working towards is a partial one, but it is a step in the right direction. Usefulness is a function of the cognitive support provided, and "compatibility" should mean, in part, the retention of built-up cognitive support. To make innovations more adoption-centric, therefore, one needs to reverse engineer and redocument existing legacy DC systems, and then build tools that can reengineer them in ways that retain and build cognitive support.

4. Framework application

The overall aim of the present framework is to help in reengineering existing legacy user systems. Below is a brief outline as to how this might happen in the future. The basic method is to first use the DC ideas to either guess or empirically determine the cognitive infrastructure (i.e., adaptations) users have built up in themselves and their environments. Currently we base this on an inventory of cognitive support possibilities derived from a feature analysis of the environments. Then opportunities are examined for adapting existing features in ways suitable for introducing the new technologies. This general idea is explored below.

Eclipse and clone detection

At the Software Research Laboratory one of our projects is to investigate techniques for detecting software clones, copyright violations, and plagiarism. Software clones are sections of code that are very similar. These commonly occur because of code "scavenging" in which code is copied and then modified to suit local needs. Copyright and plagiarism cases involve finding and verifying the fact that code or design aspects were copied from one code base to another. In each case, code similarities of various types need to be detected and investigated.

Our research is taking an adoption-centric approach to developing and inserting suitable technologies into practices of SE and copyright litigation. This will result in separate tools for software engineers and for legal analysts, but we are planning and developing both sets of innovations on top of three main existing technologies: Eclipse, Microsoft Office tools, and Microsoft Windows[1] These platforms are suitable starting points as our anticipated user base is expected to be familiar with them. In particular, in legal cases the users are expected to use Office tools such as Word, Excel, and PowerPoint in the generation of legal documents and presentations. We expect synergy in our work on both clone detection and copyright violation.

Our research strategy has identified three technological additions that are intended to implement three activities.

[1] Eclipse, Office, and Windows are registered trademarks.

FEATURE	DESCRIPTION	REUSE	MUTATE	ACCRETE	EXAPT
perspectives	customizable/sharable views and visible actions	new projects			
wizards	steps users through common tasks	new tasks			
OLE integ.	editors, views, and toolbar objects can be embeded			embed view	
Task View	users can edit and step through task (i.e., to do) list		new cmnd	auto generate	
Search View	multiple search types, results filtering, sorting			new type	search metadata
Compare View	two files can be compared side by side			new views	

Table 1. Inventory of some Eclipse features and some adaptation possibilities

These additions address activities typical in reverse engineering [15]. First, data must be gathered on where code similarities occur. For this we wish to introduce various automated and semi-automated code comparison technologies. Second, similar items must be classified into clone or non-clone (or copy or non-copy), and then grouped or aggregated into function-relevant clusters. For example, when reengineering a software system the engineer may wish to cluster related clones together so that new modules can be generated by abstracting related and duplicated functionality into a set of related methods. In copyright litigation contexts, the clustering might pertain to collecting together different types of copyright violations (e.g., code copying versus design copying). Third, exploration, visualization, and evaluation must be performed. For instance, the overall distribution of clones across sub-project boundaries may need to be known by project managers. Likewise lawyers will want to see visualizations and analyses of the instances and extent of copying. We are working on novel visualizations for code comparison and system visualizations. We wish to add various automated measurements.

In order to continue forward we need to be able to analyze Eclipse, Office, and Windows so that our innovations can be well matched to these technologies, and can be inserted in a manner that eases barriers to adoption. The remainder of this section describes an approach we are considering for evaluating tools for adaptation possibilities and cognitive support roles. The work is ongoing and preliminary, but the overview below gives a flavour of the type of analyses we are considering. The overview may give others ideas as to how to improve the analysis, and to apply it to their own adoption-centric SE research.

A cursory inventory of Eclipse features yielded a list suitable for determining adaptation possibilities. A subset of this inventory appears in Table 1. This list of features were then examined to see what adaptation mechanisms were provided by Eclipse. These were categorized using the adaptation taxonomy; examples are listed in Table 1. The column "reuse" indicates an instance where a generic feature might be used for new purposes. The entries in these columns indicate ideas about how the adaptations might be made (e.g., adding a new command to the `Task View` mutates it). The inventory and classification generation took about an hour to collect, although we refined this list and its categorizations after various discussions. It is unclear at this time how helpful this exercise has been, although obviously *some* similar type of analysis (perhaps a more informal and haphazard one) would need to be performed if one is to build tool extensions. It may be worth noting, however, that the columns of Table 1 may be helpful in identifying adaptations that are less disruptive to existing cognitive support.

The features in the list were then examined in light of various theories of cognitive support [16]. The aim was to help understand their potential roles in supporting developer cognition. Although this is hardly a substitute for studies of real users (users may not actually use the support, or may have many other types of built-up support not knowable through armchair analyses), it seems to be a prudent first analytic step. The next analytic step is to consider how we might best implement our planned innovations on top of this infrastructure. In this step we expect the theories and models of cognitive support to be helpful, although we have little to report at this point. Nonetheless a flavour of the analysis can be relayed.

We know that current technological limitations make it impossible to automatically and accurately detect all software clones within a system. Thus the user must cooperate with the tools in order to jointly determine which potential clones—i.e., "clone candidates"—should be considered true clones. The classic way of doing this is to have one or more clone detectors generate a list of clone candidates that the user steps through and classifies as clone or non-clone (or copy vs. non-copy in copyright tools). It seems clear that the generic `Task View` functionality can be adapted: the clone detector would generate a clone candidate list in the task list (an accretion of functionality), and the user would need to step through the task list and examine the candidates, deleting ones that are non-clones. This would, in fact, require a mutation (in particular, a specialization) of the task view functionality since the potential clones are clone *pairs* and the `Task View` behaviour would have to be modified so that it browsed to the two clone locations when the user double-clicks on the clone candidate.

The DC support point of view notes that this functionality is an example of distributed planning and plan following [16]. What the clone detector is doing is generating a *plan* for checking the results, which the user (more or less) follows to make those checks. The `Task View` functions as an external memory for the plan, and for where one is in following the plan. Reusing the `Task View` effectively *leverages* existing cognitive support. Once this is realized, opportunities for improving distributed plan following can be explored. For instance, the above envisioned extension forces the user to make a series of decisions about clones. It is likely that ordinary users will not be able to decisively classify clone candidates in the first pass. Said in other words, they will likely have uncertainty in their decisions. It makes sense to try to *add support* for uncertain knowledge management by allowing the uncertainty to be externalized [10]. Otherwise the engineer will need to remember the uncertain clone pairs in order to return to them, if necessary.

Various designs can be entertained for implementing the uncertain knowledge management support. Table 1 provides clues as to which ones might affect adoptability. For instance, the `Task View` might be mutated in order to encode the uncertainty in the classification. It might be preferable, however, to create a more specialized version of the `Task View`. Since there are already several specializations of the `Task View` (the `Compare` and `Search` views are both specialized task steppers), it may be preferable to add an entirely and obviously new view (that is, by accreting similar functionality) that allows clone candidates to be classified with varying degrees of uncertainty. Although we are nowhere near being able to decide what implementation is best, the vocabulary of cognitive support and adaptation taxonomy appears to be helpful in understanding design options and then reasoning about their potential adoptability implications.

5. Summary

In order to achieve the goals of adoption-centric SE research, one must have an idea of what factors affect adoptability and what changes can or should be made to existing environments in order to introduce innovations. The direction presented here is to examine the cognitive support present in target tool environments and look for appropriate support to preserve, leverage, or add. Although progress has at times seemed glacial, the cognitive aspects of adoption resistance appear critical, and my personal feeling is that there is no choice but to continue the difficult and long-term work necessary to understand and address the role of cognitive support in tools and how various adaptations to them affect adoption.

References

[1] R. K. E. Bellamy. Strategy analysis: An approach to psychological analysis of artifacts. In D. J. Gilmore, R. L. Winder, and F. Détienne, editors, *User-Centred Requirements for Software Engineering Environments*, pages 57–67. Springer-Verlag, 1994.

[2] R. Dawkins. *River Out Of Eden: A Darwinian View Of Life*. HarperCollins, 1995.

[3] N. V. Flor and E. L. Hutchins. Analyzing distributed cognition in software teams: A case study of team programming during perfective software maintenance. In J. Koenemann-Bellinveau, T. G. Mohen, and S. P. Robertson, editors, *Empirical Studies of Programmers: Fourth Workshop*, pages 36–64, Norwood, NJ, 1991. Ablex.

[4] P. Fowler and L. Levine. A conceptual framework for software technology transition. Technical Report CMU/SEI-93-TR-31 and ESC-TR-93-317, Software Engineering Institute, Carnegie Mellon University, Dec. 1993.

[5] S. J. Gould and N. Eldgredge. Punctuated equilibria: the tempo and mode of evolution reconsidered. *Paleobiology*, pages 115–151, 1977.

[6] S. J. Gould and E. Vrba. Exaptation – a missing term in the science of form. *Paleobiology*, 8:4–15, 1982.

[7] W. G. Griswold. Coping with crosscutting software changes using information transparency. In A. Yonezawa and S. Matsuoka, editors, *Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[8] J. Hollan, E. Hutchins, and D. Kirsh. Distributed cognition: Toward a new foundation for human–computer interaction research. *ACM Transactions on Computer-Human Interaction*, 7(2):174–196, June 2000.

[9] E. Hutchins. *Cognition in the Wild*. MIT Press, 1995.

[10] J. H. Jahnke and A. Walenstein. Reverse engineering tools as media for imperfect knowledge. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE'2000)*, pages 22–31. IEEE Computer Society Press, 2000.

[11] W. E. Mackay. Responding to cognitive overload: Co-adaptation between users and technology. *Intellectica*, 30(1):177–193, 2000.

[12] S. L. Pfleeger. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software*, 47(2-3):111–124, July 1999.

[13] E. M. Rogers. *Diffusion of Innovations*. The Free Press, New York, NY, 1995.

[14] D. E. Rumelhart and D. A. Norman. Accretion, tuning, and restructuring: three modes of learning. In J. W. Cotton and R. Klatsky, editors, *Semantic Factors in Cognition*. Erlbaum, Hillsdale, NJ, 1978.

[15] S. R. Tilley. The canonical activities of reverse engineering. *Annals of Software Engineering*, 9(1–4):249–271, 2000.

[16] A. Walenstein. *Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework*. PhD thesis, School of Computing Science, Simon Fraser University, May 2002.

[17] J. Zhang and D. A. Norman. Representations in distributed cognitive tasks. *Cognitive Science*, 18:87–122, 1994.

A Lightweight Project-Management Environment for Small Novice Teams

Ying Liu
Department of Computing Science
University of Alberta
Edmonton, T6H 2E8, AB Canada
+1 780 492 3118
yingl@cs.ualberta.ca

Eleni Stroulia
Department of Computing Science
University of Alberta
Edmonton, T6H 2E8, AB Canada
+1 780 492 3520
stroulia@cs.ualberta.ca

ABSTRACT

The success of a software-development project depends on the technical competence of the development team, the quality of the tools it uses, and the project-management decisions it makes during the software lifecycle. New requirements, tight delivery schedules and developer turnaround present the team with challenges that need to be addressed with informed development-plan modifications. Project-management skills are especially difficult to teach. When working on a substantially complex project, students often become too involved with coding to recognize the need for management; and when they do, they frequently lack the necessary information to make a good decision, because, more often than not, they do not have a complete overview of their progress.

In this paper, we describe a lightweight environment for supporting, monitoring and analyzing activities related to software development. This environment integrates a set of tools, including CVS, newsgroups, code analysis and personal-process tools, and uses a browser-accessible Wiki-based user interface as a front end to all the underlying tools. We have just deployed this environment in the context of an undergraduate software-engineering course. We believe that the familiar lightweight user interface will encourage students to use the integrated tools and will improve their overall learning experience, especially in the project-management area. At the same time, it will enable the instructors to monitor the team-development progress and to provide relevant and constructive feedback.

1. MOTIVATION AND BACKGROUND

In addition to the technical competence of the development team, the success of a software-development project also depends on the quality of the tools the team uses, and the project-management decisions it makes during the software lifecycle. New requirements, tight delivery schedules and team-member changes present challenges that need to be addressed with informed development-plan modifications. Such project-management skills are especially difficult to teach. When working on a substantially complex project, students often become too involved with coding to recognize the need for proper tools usage and project management; and when they do, they frequently lack the necessary information to make a good decision, because, more often than not, they do not have a complete view of their progress.

In our WikiDev project, we have been working on developing a lightweight environment integrating a set of basic functionalities for supporting developers, working in small teams, to monitor and reflect upon their process and the code they develop. The specific context is that of supporting student teams in undergraduate project-based software-engineering courses, and enabling instructors to monitor team collaboration, to evaluate work products and to provide salient and timely advice to the teams. Our vision behind WikiDev is to encourage and support reflection in student software-development teams. Our experience has been that a team with a competent project leader, with a solid understanding of the overall project objectives and tasks, is more likely to deal with unforeseen setbacks and complete the development on time. We believe that even teams without such a leader can be equally successful, assuming that the team members take a reflective stance towards their task plan and are always aware of their progress status. Given an accurate task schedule and information representative of the team's actual progress, each individual team member can notice deviations and adjust.

The first important issue we had to address in designing WikiDev was what information to provide to the development team. Such information should be useful for evaluating project progress and supporting project-management decision making. Furthermore, it should be based on data provided by readily available and preferably routinely used tools. At the same time, it should be somehow "richer" than what would be available to the team if they simply used these underlying tools. In principle, the more primary data is available, the more information could potentially be inferred about the project. On the other hand, explicit data collection, when it is not directly useful to the software-development process can be perceived as a burden by the developers; this is especially true with students who do not

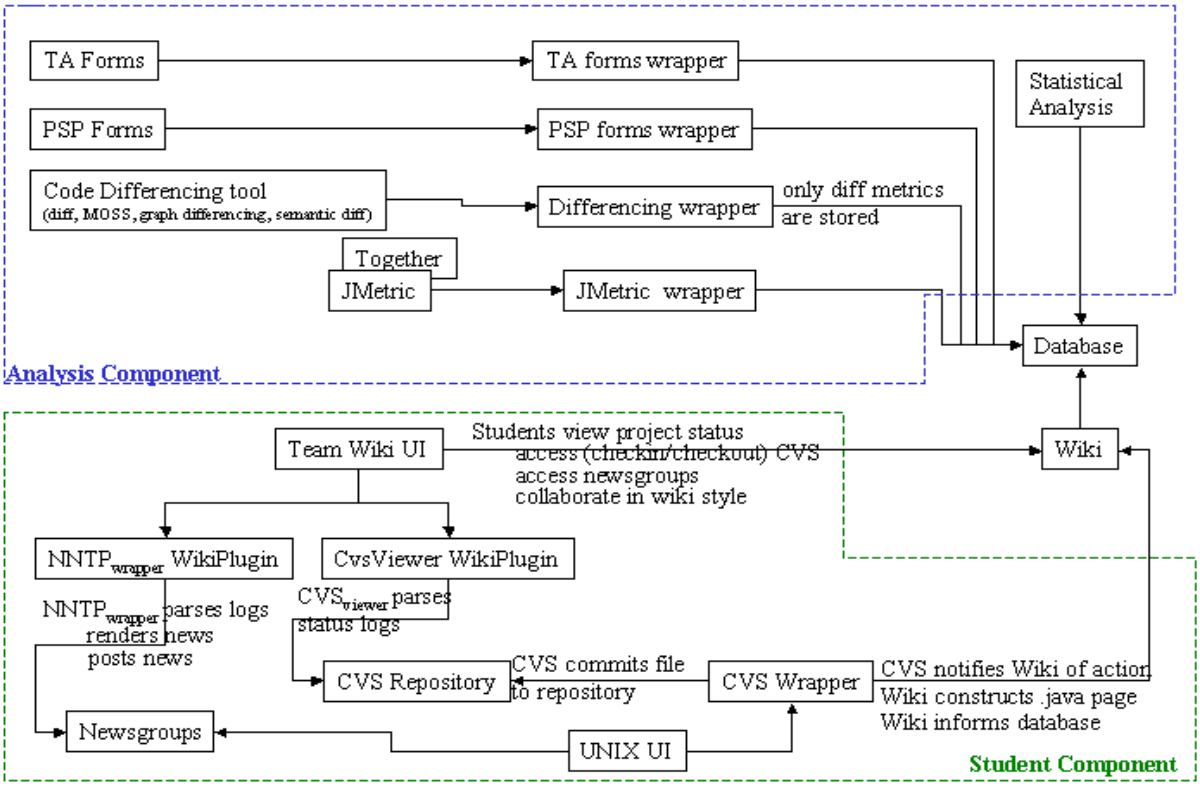


Figure 1: The WikiDev Component Architecture.

necessarily appreciate the need for record keeping for project-management purposes.

We decided that WikiDev should capitalize on CVS, the Concurrent Versioning System, one of the most basic tools frequently used by software teams, even novice ones such as undergraduate computing-science students. As a code repository, CVS already contains multiple versions of the project code that can be analyzed quantitatively, to infer different software metrics, and qualitatively, to extract a model of its high-level design. CVS also maintains a historical log of the team members' activities on the repository. As such, it can be mined to extract information about the collaboration behavior of the team members.

Another regularly used tool for communication within and across teams is the newsgroups. Focused newsgroups, with questions and answers related to a particular technology and/or problem, are a valuable resource for a community of developers and can also be mined to better understand the nature of the difficulties that these developers face.

Another equally important issue was whether or not to develop WikiDev as an extension of a specific IDE. On one hand, it has to be closely integrated to the core development activities, so that it is not perceived as "additional work". It also has to be easy to learn and lightweight to adopt, otherwise it will not be usable in the context of a course term project. We decided against adopting a specific IDE. We have noticed that many students already use their preferred IDE and establishing a single common one would be a challenge. Furthermore, learning an IDE may involve a learning curve that cannot always be accommodated within a course term. Instead, WikiDev adopts Wiki as a lightweight front-end integrating a set of other tools. A Wiki does not support programming per se, and is independent of any programming language and design methodology. Instead, it is a simple web-based whiteboard tool, widely used by communities who wish to communicate information of common interest. Its underlying metaphor is "shareable HTML pages, immediately publishable on the web". This is a quite simple idea and even students who are not already familiar with the concept can easily understand and use the Wiki. In the context of WikiDev, the Wiki simply provides a simple and uniform user interface for accessing the underlying tools and for displaying the collected data and the information inferred from these tools.

The rest of this paper is organized as follows. Section 2 gives a detailed description of the WikiDev architecture and elaborates on its individual components. Section 3 briefly describes the envisioned usage scenarios of WikiDev. Section 4 discusses several related efforts. Finally, section 5 summarizes our work and outlines our plans for the future.

2. ARCHITECTURE OF OUR ENVIRONMENT

Figure 1 diagrammatically depicts the WikiDev component architecture. The browser-accessible Wiki provides the standard user interface for the team developers. All other tools, CVS and the newsgroup among them, are embedded as plug-ins to the Wiki. The data produced by the integrated tools are stored in a

shared database. A set of analysis tools, not directly usable by the teams, have access to the database to analyze the original data and produce additional information of interest that is subsequently delivered to the team through the Wiki.

In the rest of this section, we describe the individual tools integrated in WikiDev and the data they contribute to the WikiDev repository.

Wiki: Wiki is in Ward's original description *"the simplest online database that could possibly work"* [2]. It is a whiteboard-like communication mechanism and provides "open editing". Furthermore, the php-Wiki we adopted for WikiDev is extendible in that it supports the integration of other applications as plugins; control widgets integrated in the browser-accessible user interface can be programmed to invoke other applications. These two capabilities enable a flexible combination of structured interaction, with the plugged-in applications, and unstructured interaction, through the whiteboard.

In WikiDev, each development team has its own area for their project, where all types of related information can be posted, such as meetings agendas and minutes. At the same time various plugins enable the team to invoke functionalities of several other tools, described in the rest of this section, and to view their project-related information produced by the analysis of their project-related data.

CVS: The Concurrent Versioning System [4] (CVS) uses a hierarchical structure to record multiple versions of source-code modules. It enables backtracking, i.e., reverting to earlier versions, when the development has introduced undesired features. It also detects conflicts, when multiple developers modify the same module, and supports the merging of changes when such collisions occur. WikiDev includes as a plugin a browser of the team's CVS repository. Through the repository browser, developers can browse through their code and even use the Wiki search capabilities to search for specific identifiers.

In addition, WikiDev integrates, as a plugin, a browser of the repository operations. CVS keeps a detailed history of its usage, including who performed what operation, when, from where, on which file, from which directory. Based on this data, a lot of valuable information about the team's collaboration can be inferred. For example, it can be inferred whether there are key team members modifying many files, or whether there is a specific file that has been modified by many developers. Statistics, such as the average time lapsing between file modifications, or the average time between a developer's accesses of CVS, or the frequency of conflicts, can also be computed.

We have been encouraging, and even requiring, the use of CVS in our software-engineering courses. Unfortunately, our experience has been that many teams do not use the tool properly. Some use it only in a perfunctory manner to meet course requirements: they simply check their deliverable code in CVS. Yet others exchange code modules among themselves through email and have a single designated person check mature versions into the repository. We believe that, by analyzing their usage of CVS and regularly presenting this information to the team through the WikiDev, they will be more motivated to adopt proper CVS-usage practices.

Newsgroups: Our students use newsgroups extensively, mostly to ask development-related questions that the instructor team and their peers answer. Because of the high message traffic however, some questions get asked repeatedly.

In the context of WikiDev, we plan to integrate the course-related newsgroups so that postings get immediately added to the repository, from where they can be accessed through the Wiki fuzzy-search functionality.

PSP forms: According to our experience, the developers' expectations of their progress deviates, quit often, substantially from their actual project status. Teams may lag behind the schedule from the early stages but only detect it after several weeks. WikiDev integrates a set of forms implementing a simplified PSP [7] instrument to help students obtain a more realistic understanding of their effectiveness.

A full implementation of the entire set of PSP forms is unrealistic in the context of an undergraduate software-engineering course. Instead, we chose to implement a simple defect-recording log associated with the CVS repository, so that students can attach "defect annotations" on code modules of the CVS repository. In addition, a set of developer-specific plugins can be used to keep track of time spent on the project and code-size estimates, and a set of project-specific plugins can track project-plan summaries and improvement proposals.

Based on this information, students could first notice deviations of the actual from the planned status and adjust their development.

Analysis components

The data contributed to the WikiDev repository by the above components, including code modules, CVS history, newsgroup postings, and PSP data, is the subject of further analysis by the WikiDev "analysis components".

Code and Code-evolution Metrics: We have loosely integrated JMetric [6] in WikiDev to quantitatively assess the quality of the project modules. The reports generated by the tool, including the code metrics and their corresponding "advisable" value ranges, are stored in the WikiDev repository and can be viewed as specially structured Wiki pages. In this manner the team can easily notice "illegal" metrics values and may try to improve on the quality of the offending modules.

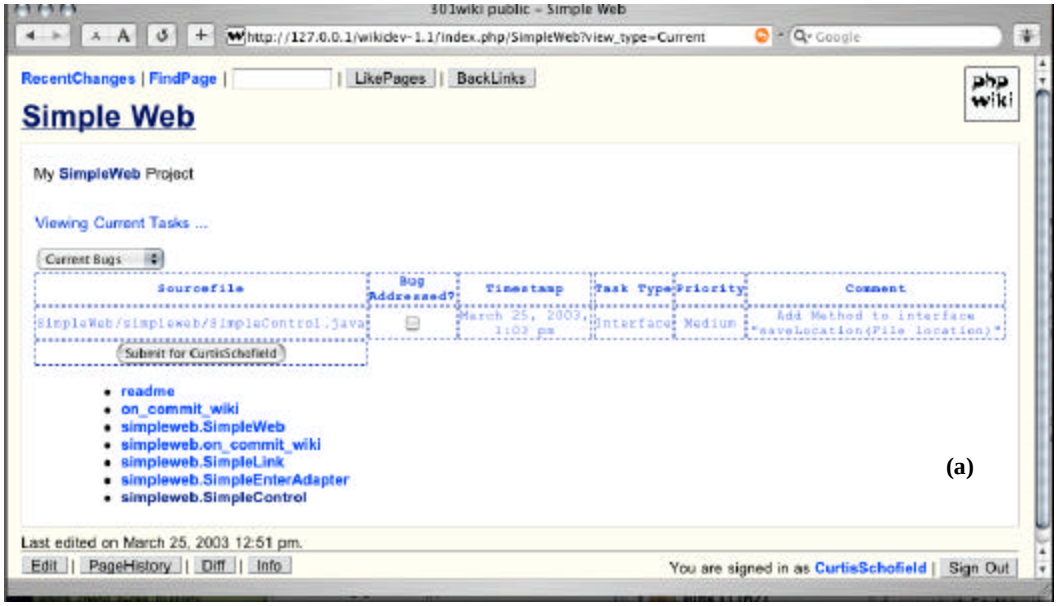
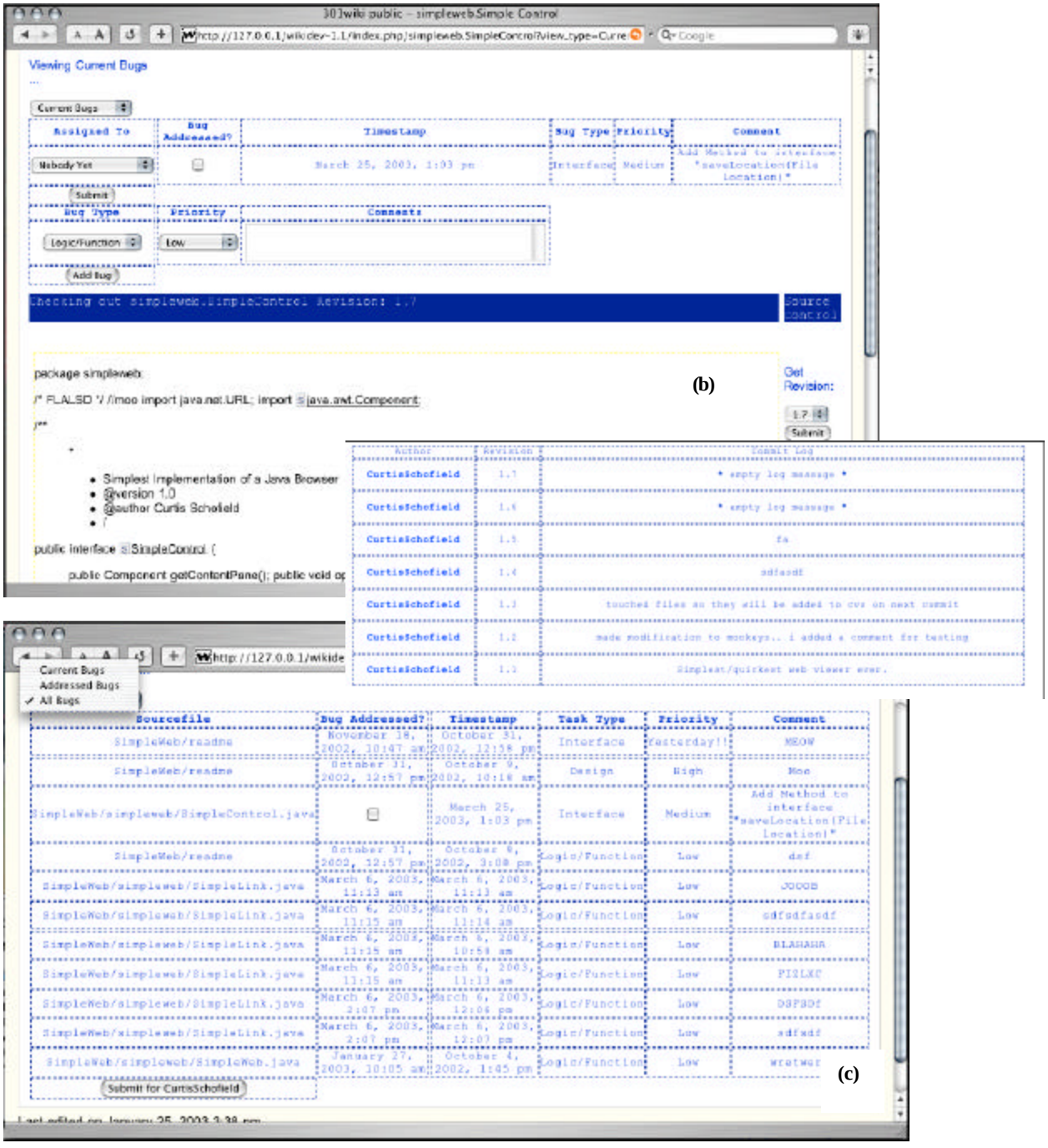


Figure 2: Some WikiDev screenshots.



Furthermore, WikiDev aims at analyzing the coding process itself, by qualifying the nature of change between two subsequent versions of a module in the CVS repository. Focusing on the "modification" operations in the CVS history, the code-evolution analysis plugin compares the original module with the modified one. There are several possible means of comparison and we are currently working on implementing several of them; the simplest is to use the Unix diff on the two versions, where more complex ones may involve the comparison of abstractions of the code such as the Abstract Syntax Trees (AST) [3], or the system dependence graphs (SDG) [1], or the XMI class diagrams. The comparison of different views may provide insights on whether the modified version is a refactored version of the original, or an extension of its functionality, or possibly a debugged version.

Statistical analysis: Finally, WikiDev will also integrate statistical analysis tools to mine interesting patterns in all the above data, such as association rules and sequential patterns [8]. Association rules can uncover information such as "which members usually work at the same time period" for example, where sequential pattern mining might discover that "file A modifications are always preceded by modifications to file B".

This type of analysis may also be employed to correlated patterns in the code-development process and objective assessments of the team's performance, such as grades. Such correlation would shed some light on what behaviors tend to result in successful performance and what not.

3. FUNCTIONALITIES OF THE WIKIDEV ENVIRONMENT

Figure 2 shows the various pages of the WikiDev corresponding to its various functionality plugins. Some WikiDev pages are accessible to the class as a whole and some are accessible to the developers of a particular team.

Team members access a root page for their team-specific WikiDev area (see page (a) in Figure 2). Off this central page are also indexed various unstructured pages that the team may have constructed to maintain information relevant to the project. Furthermore, the "UserTasks" plugin embedded in this page shows to the currently logged in developer his or her tasks.

When a student accesses CVS, this action is recorded by the CVS repository and a notification event is sent to WikiDev. The WikiDev parses the CVS-operation history information and records it its own repository. It also parses the CVS repository structure and generates a page corresponding to each code module in CVS (see page (b) in Figure 2). This page includes a listing of the code module, its revision history in CVS, its associated defect records, and other possible comments from the developers on the module. The defect record of each module is constructed through a special-purpose WikiDev plugin: when a developer faces a problem with a module, he can retrieve the module page in the WikiDev and, using the defect-annotation plugin, he can add a defect record to the module, specifying the nature of the problem, its urgency and who should address it. This defect annotation becomes part of the module WikiDev page. Later, the developer who fixes the defect will check the corrected module in CVS and will edit the defect record to reflect the fact that it has been corrected. Note that both the defect record and the annotations do not affect the module in CVS, only its corresponding WikiDev representation. Through these corresponding pages, developers can use the WikiDev search functions to browse through their code, and look for specific code identifiers or annotations. If they are interested in a particular file, its modification history and the nature of its evolution, developers can focus on the file-specific page.

A user-centric page provides each developer with an overview of their own development contributions (see page (c) in Figure 2). This page shows the history of tasks for a specific developer including any current tasks. Essentially this page is produced by the "UserTasks" plugin – which is also embedded in the Homepage - expanded to show the user's entire history

If a developer has a question or wants to exchange some ideas with his/her peers, he may post a message to the course newsgroups. Instructors, TAs and other students may respond.

4. RESULTS AND FUTURE WORK

In the context of our software-engineering courses, WikiDev is intended as a lightweight "portal" for integrating a set of basic tools that our student developers use and for providing value-added information based on analysis of the collected data. Our objective is to provide sufficient added value on top of the underlying tools so that students are motivated to use them appropriately and frequently, without actually imposing any additional learning curve on them, except from what is required to learn the underlying tools. In some sense, we plan to provide a flexible, extendible "poor man's" IDE.

Of course there are several IDEs that may provide similar and other additional functionalities. To our knowledge none of the available IDEs offer the analysis capabilities we are currently building in WikiDev. We anticipate that the analysis enabled by WikiDev will support the students in their projects and, also, will provide valuable insight to the instructors about how the student teams work. Such insights will, in turn, guide the instructors' feedback to the students and improve their learning experience.

Our initial experimentation with the WikiDev analysis tools has produced some interesting and promising results. The graphs shown in Figure 3 were produced by analyzing the CVS usage behavior of two teams[1]. The two graphs in the lower part of the Figure, (c) and (d), show the number of CVS operations

[1] These graphs were produced by examining the CVS history data of teams before the deployment of WikiDev.

performed by the members of teams A and B during project development. The operation-dense periods are both located in the second half period. However, groupB has a big operation gap from November 6 to November 18; they have hardly accessed CVS between these two dates. All members in groupA used CVS regularly and made almost equal contributions (assuming we can judge that by the number of modifications they did to the CVS repository), but groupB only committed files just before the due date. Moreover student114 performed much fewer CVS operations than the other members of team B; on the other hand, all members of team A seem to have used the CVS with a similar frequency.

More detailed information can be obtained by examining the operation type diagrams, shown in graphs (a) and (b). These two graphs depict the numbers of the various types of CVS operations performed by the team members. The operation types are:

- A: file addition to CVS
- C: merge is necessary between two checked-in versions, but collisions were detected
- F: a module is released
- G: merge is necessary between two checked-in versions, and is successful
- M: file modification
- O: file checkout
- R: file removal
- W: working copy of a file is deleted during update

As evidenced by the number of actual file modifications, the workload of the groupA members is fairly similar and is above the average value for all the students and also higher than that of groupB. Besides, the collision frequency of groupB is higher than average. Based on these graphs we can draw a conclusion that the work habits of groupB are not as good as those of groupA.

WikiDev is deployed for the first time this term and we will soon have the first comprehensive set of data to report on its effectiveness. Furthermore, we are currently re-implementing WikiDev in the context of Eclipse [5], which seems to be widely adopted and is positioned to become the IDE of choice for java developers. We plan to comparative evaluate the two tools, in order to better understand the advantages and disadvantages of adopting an IDE.

5. ACKNOWLEDGMENTS

The authors wish to thank Curtis Schofield for his great development work on WikiDev, and the anonymous reviewers for their insightful feedback. This work was supported by NSERC through a CSER3 grant.

6. REFERENCES

[1] D. Liang, M.J. Harrold: Slicing Objects Using System Dependence Graph, The International Conference on Software Maintenance, Washington, DC, November 1998.

[2] http://Wiki.org/Wiki.cgi?WhatIsWiki.

[3] N. Howarth: Abstract Syntax Tree Design, http://www.ansa.co.uk/ANSATech/95/Primary/155101.pdf

[4] Concurrent Versions System, http://www.cvshome.org

[5] The Eclipse project, http://www.eclipse.org

[6]JMetric, http://www.it.swin.edu.au/projects/jmetric/products/jmetric

[7] H. Watts, Discipline for Software Engineering. Reading, Mass. Addison-Wesley, 1995.

[8] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann publishers, 2000.

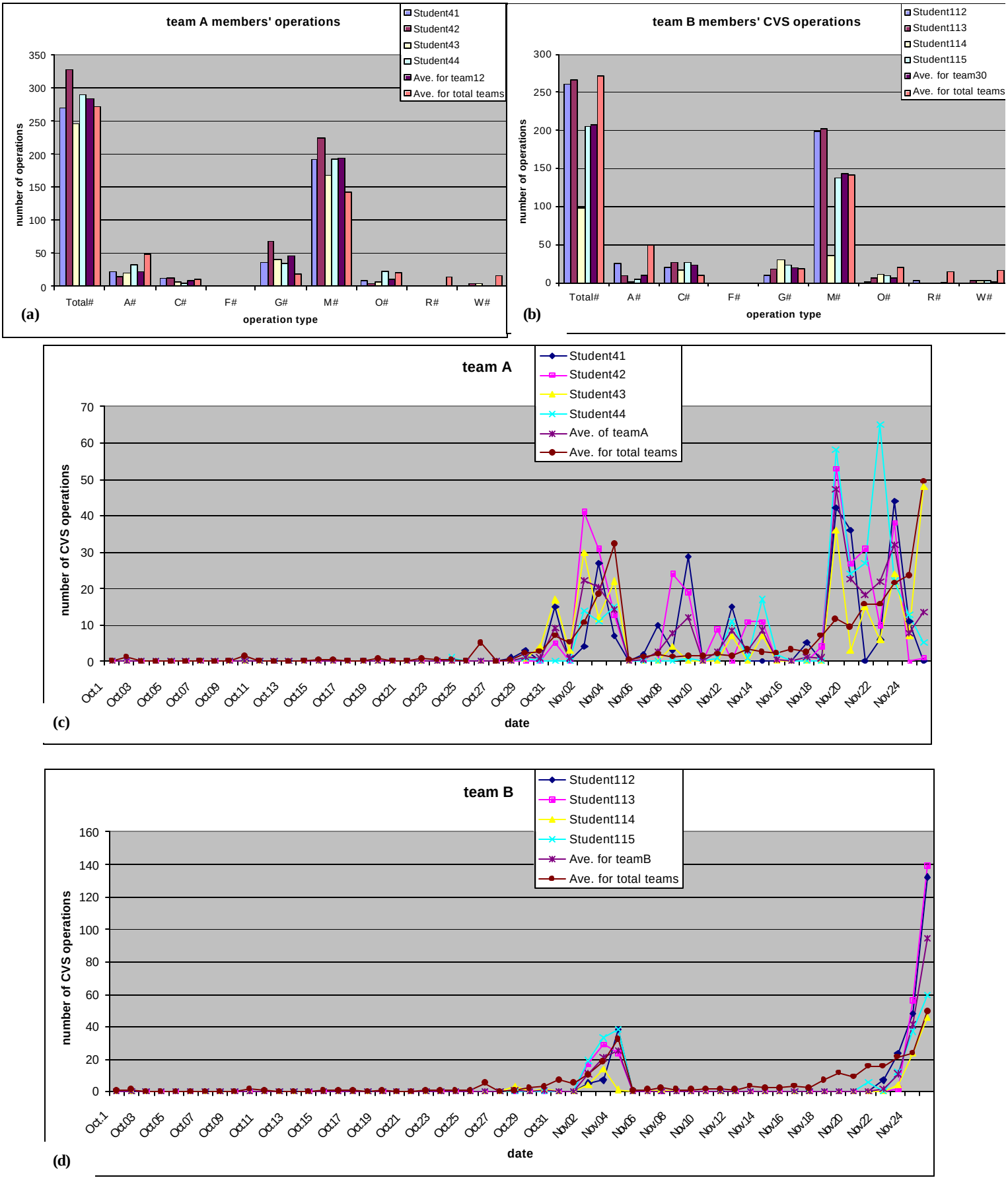


Figure 3: Views on the collaboration process of two teams

Adopting GILD:
An Integrated Learning and Development Environment for Programming

Margaret-Anne Storey[1], Mary Sanseverino, Daniel German, Daniela Damian,
Adrian Damian, Jeff Michaud, Adam Murray, Rob Lintern, James Chisan
Department of Computer Science
University of Victoria

Marin Litoiu
Center for Advanced Studies,
Toronto, IBM

Derek Rayside
Program Analysis Group
Laboratory of Computer Science, MIT

Abstract

This position paper presents GILD – an integrated learning and development environment for programming. The objective of the GILD project is to provide facilities for teaching and learning Java that are tightly integrated with a fully featured, mature and widely adopted development environment. GILD is being designed as a plug-in for Eclipse and takes full advantage of the Eclipse Java development tools. It will also include collaborative support as well as more sophisticated methods for teaching and learning the first principles of a programming language. In this paper we identify the technical and pedagogical aspects that we think will contribute to its adoption by both teachers and students, while discussing the challenges and barriers that we may face.

1. Introduction

Teaching students how to program can be a challenging task. Unfortunately, there are few tools that provide pedagogical support for learning and teaching programming. Fully featured integrated development environments (IDEs) overwhelm novice programmers and do not have all of the features needed to support teaching. The programming pedagogical tools that do exist (such as BlueJ [15] and DrJava [14]) have not seen widespread acceptance. We suspect this is because they tend to offer a minimal or reduced feature set and thus are limited in how long they remain useful to the student programmers. They also lack features that are commonly available in collaborative desktop environments, such as simultaneous document browsing/editing, instant messaging, Web forums etc. Such features are often used in other disciplines to enhance learning.

Currently teachers usually make use of several tools when creating materials for a programming course. They may use an IDE to prepare program samples, a presentation tool to present materials in class, a drawing tool to create pictures, a Web-publishing tool to post course materials, and e-mail to communicate with their students. This leads to a scattering of course materials and related information that is difficult to update and share. Moreover, the current approach of using e-mail for providing programming help outside class time is very tedious and quickly breaks down – for example, students often ask questions about code that the instructor can't see. There exist some tools, such as Blackboard (www.blackboard.com) and WebCT (www.webct.com), that provide a Web portal to the material of a course, but unfortunately these tools are not tailored to teaching and learning programming.

In our experiences, we have found that students will learn programming concepts more quickly if they read, write and test lots of example programs. Unfortunately, instructors rarely have enough resources to provide feedback on numerous programming exercises. Automatic marking techniques are commonly deployed but they are typically custom solutions that are difficult to reuse or extend. Good teaching practices also recommend providing in-class interactive exercises – but with traditional pen and paper media it is impossible to give feedback to every student solution in large classes.

We believe there is a need for an integrated learning and development environment that will expand in usefulness as the programmer's ability and need for training increases. We have begun to develop such an environment to help novice through intermediate programmers learn Java. The GILD (**G**roupware-enabled **I**ntegrated **L**earning and **D**evelopment) environment is being built on top of an existing and well-accepted IDE (integrated development environment) within Eclipse [3].

[1] Contact e-mail: mstorey@uvic.ca; Website for this project: http://gild.cs.uvic.ca

Eclipse is an open-source platform for the creation of highly integrated tools [16].

Our environment will make use of the powerful infrastructure and large number of plug-ins that are provided by Eclipse and the Eclipse community. We are also using other tools and borrowing concepts from Web-based learning tools and collaborative desktop technologies to enhance learning in both co-located and distributed settings. By building on top of a widely adopted and powerful integrated learning environment, while paying careful attention to pedagogical requirements, we believe the GILD environment could be widely adopted for teaching and learning programming.

The rest of the paper is organized as followed. In the next section we detail the adoption goals of our project, in particular emphasizing both technical aspects of the GILD environment as well as the pedagogical requirements that need to be met to ensure adoption. In Section 3, we discuss how we expect both students and instructors will adopt this tool and the expected benefits such a tool can provide. In Section 4 we discuss the adoption challenges that we face with both instructors and students, and from the educational institution's perspective. Finally Section 5 concludes this brief paper and summarizes our current and future work.

2. Research Goals

There are many issues concerning the adoption of a learning environment for programming. There are two sets of users for this environment – teachers and students. We conjecture that for both groups to adopt the environment, it needs to be inexpensive, easy to install, easy to use, and fit in with their existing tools and legacy course information. But more importantly, the tool needs to provide some gains with respect to the pedagogical needs of both groups. In this section, we first discuss some of the technical aspects of GILD and then explore the pedagogical objectives for both teachers and students and how they can be supported by the technical solutions we suggested.

2.1 Technical aspects

2.1.1 Implement GILD as a plug-in for Eclipse

The Eclipse environment and its Java development tools are already seeing widespread adoption. This rapid adoption was anticipated due to Eclipse's open source nature and its extensible architecture. Consequently we decided to build the GILD environment on top of the Eclipse Java Development Tools and Eclipse infrastructure. Moreover we will leverage third party plug-ins that can bring additional benefits to the users of the GILD environment. For example, there are plug-ins to support the automatic generation of UML diagrams (for example, the SlimeUML tool [1]). Generation of UML diagrams can help instructors explain code examples, or can be used by students during code explorations or to show their designs in course assignments. Other plug-ins of interest facilitate pair programming [2] and provide information on code quality [21].

The environment we build will also be available as an open source framework that is extensible (that is, we will create GILD specific extensible points).

2.1.2 Integrate features from existing tools

From our own experiences as teachers and those of our colleagues, we are acutely aware that instructors struggle with integrating and synchronizing information from many different tools. Tool switching and synchronization problems are common to instructors of most fields, and hence the recent rise in popularity of Web-based learning tools [18]. Web-based learning tools provide an integrated environment for preparing and managing course materials. Although such an environment may seem to offer trivial improvements over using a selection of tools, the biggest advantage they offer is that of organizing course materials for both the instructors and students. Note that a key criterion for an effective teacher is to be organized [20].

Moreover, students should be able to use one learning environment for both software development and course material access.

Unfortunately, Web-based learning tools do not help as much as one would initially hope when teaching or taking a programming course. They are not tightly integrated with the development environments that are the cornerstone tool for both instructors and students in a programming course. To address this problem, we propose that key features from a Web-based course management environment should be tightly integrated with a software development environment (as opposed to the alternative approach of creating some programming support and adding that to a Web-based learning tool).

We propose that the User Assistance plug-in for Eclipse be used for authoring, integrating, storing, organizing and presenting Web-based materials. Such materials should be tightly coupled with the programming examples in the Java development tool repository. We are also exploring how collaborative support (currently being added to Eclipse for other projects) can be leveraged in the GILD environment.

The integrated approach we advocate would reduce the need for both students and teachers to be constantly switching between tools when interacting with a course.

2.1.3 Provide a customizable environment for learning and teaching

One size definitely does not fit all when it comes to teaching and learning programming. All instructors and students will have very different needs when using such an environment. Instructors have very different styles of teaching, and may choose to use only a subset of the features offered. The features they use in their course will also depend on the level of the course being taught.

We have noticed that even students in a first year programming course tend to be at very different skill levels and consequently the more advanced students will choose not to use the simple tools that are often advocated at universities, as they are too limited. For example, at the University of Victoria, the TextPad tool [17] is recommended in first year, as it is very simple and easy to learn for novice programmers. However, such a tool is clearly very limited and will be rejected by the more savvy students.

The instructor and the students have to be able to configure the environment so that it is suitable for the varied levels of the students. Eclipse offers many advanced features, such as "code assist" and refactoring, which may be overwhelming for novice programmers but desirable for the more advanced student. Customization of the features in the Eclipse JDT can be achieved in part through the use of its UI features (natures, perspectives and views), thus meeting the diverse needs of instructors and students while supporting changing needs over time for both groups.

2.2 Pedagogical objectives

2.2.1 Novice programmers should read, write and test lots of code

It is generally accepted that novice programmers should have lots of experience reading, writing and testing programs in order to learn. Unfortunately current teaching tools tend to place the students (and indeed the instructors) outside the domain of the development environment and instead trap them in a Web browser or in a presentation tool.

The GILD environment should instead position students and teachers within the development environment providing easy access to relevant executable program examples and other course materials.

We are using the existing version control systems in Eclipse (such as CVS) for storing examples and exercises selected by the instructor. Students will be able to check code and assignments in and out using these facilities. We will also integrate facilities to allow automatic deployment, submission and marking of assignments. Such facilities will enable students to do more programming (which is the best way to learn how to program).

Over time, a library of code examples, course notes and tutorials can be created by leveraging and extending the features of the version control repositories and the help system in Eclipse.

2.2.2 Present syntactically and semantically correct code to students

Many novice programmers struggle when learning the basics of a new programming language. Their knowledge is very fragile, and seemingly innocent mistakes in an instructor's snippet of code, can cause students much grief. These mistakes are common when the program examples and code snippets are taken outside the development environment. By keeping such examples grounded within a development environment, the instructors can more easily correct syntactic mistakes as they occur in real time. Eclipse provides 'eager parsing' and 'code assist' features that can also be used to help the students learn from their own mistakes and may promote more exploration on the part of the students. We are also exploring these facilities to see if they lend themselves to customization. This would allow instructors to tailor messages to emphasize topics of relevance.

2.2.3 Assign Interactive Exercises in the classroom

Many universities have wired classrooms or laboratories where each student sits at an individual computer in a networked environment. In such an environment, a tool should provide support for the instructors and students to write, annotate and run code in real time–passing control from one to another as required. In addition, students should be able to submit answers that can be marked automatically and direct questions to the class for discussion as they arise.

We are also exploring how these objectives can be met through existing Eclipse plug-ins that support pair programming such as SanGam [2] and the collaborative support that is being added to Eclipse.

2.2.4 Provide support for Communication and Just-in-time Training

Web-based learning tools are in part also popular because of the collaborative support they provide. Communication mechanisms such as forums, instant messaging and e-mail are used frequently. Such facilities take student interaction beyond the classroom and enhance the learning experiences of the students. Students can learn from and help one another when such facilities

are available. Without these, many students express isolation in a university environment and have only the instructor to approach when they have problems with course material. Moreover, interactions with instructors can be limited to either office hours or to e-mails, which tend not to be very expressive.

We advocate that the students should be able to interact with the instructors and other students both in real-time and asynchronously by asking questions and receiving replies that are positioned *within the context of their code*. When the students and instructor are not co-located but are working synchronously, collaborative support features such as simultaneous code editing and instant messaging will move the interactions between the student and instructor out of the e-mail world and back into the development world where these interactions should take place (see www.groove.net for an example of such a general-purpose collaborative environment).

As the instructor receives questions about tricky parts of the assignment, he or she can insert links to related code examples and other hints that will provide "just-in-time" training for the more complex exercises. Furthermore, pair-programming techniques have been used successfully in many introductory programming courses [19]. Simultaneous code editing combined with instant messaging will enable students and the instructor to collaboratively author code and improve their learning experiences.

Eclipse already has some infrastructure that we believe will be useful for helping us provide this support–such as extensible markers and decorators.

3. GILD's Adoption

There are many reasons that lead us to believe that GILD will be adopted. We list these reasons from four perspectives -- that of instructors, students, post-secondary institutions, and the Eclipse community -- while recognizing that these claims have yet to be validated.

3.1 Instructor Perspective:

Repository of course content. Using GILD we hope to achieve a closer integration of course materials (notes, pictures, animations, etc) with executable program examples. Typically, course content that one finds on the Web, or borrows from colleagues, is incomplete or lacks documentation. In particular, standalone programs are not often explicitly tied to course material or learning objectives and it takes a professor contemplating reuse of the material much time to figure out if the material matches their needs. By integrating such programs more closely with course material (linked by learning objectives) we hope to lead to more reusable content and programs. The reuse of lecture materials is very attractive to individual instructors and department administrations.

Fewer tools required when teaching. Such an environment could alleviate the need for switching and synchronization of materials between tools and hence lead to more adoption.

3.2 Student Perspective:

Popularity of Eclipse. Eclipse is already widely used in industry. Students will likely see the value of learning it, as they can apply their knowledge later when their education is finished.

Free. Students have very limited resources so cost will have a big impact on whether they adopt a tool or not. Gild will be free when used for academic purposes.

Collaborative support. Many students feel isolated from other students and indeed from the professor in a course. We conjecture that collaborative support would lead to more adoption of this approach.

Interactive learning support. Learning how to program is a very dynamic activity. As instructors of these courses, we have noted that the students that write lots of code and actively engage with the material do much better than students who take a more passive approach. Unfortunately, our current tools support the passive approach rather than a dynamic environment for code exploration and experiences. We believe students will embrace the opportunities to do more programming if they are presented to them in an easy to access manner.

3.3 Post-Secondary Institution Perspective:

Platform independence. GILD will run on any platform in which Eclipse runs. This includes Microsoft® Windows®, Mac OS, and various flavours of UNIX®. Moreover, GILD will co-exist with other applications on these machines, as students, faculty and staff use them for many purposes.

Free. GILD, and all subsequent updates, will be free to post-secondary institutions. These institutions are under constant budget constraints, so even a small fee could be a break point. As well, licensing issues can be easily dealt with. Students will also be able to use GILD free of charge on their own machines.

Easy to deploy and maintain. Typically, post-secondary institutions have limited system personnel and teaching assistance resources. At University of Victoria, we currently use TextPad because it is easy to deploy and maintain. It is also easy to train teaching assistants and computer consultants on. However, TextPad is limited in that it is neither a learning nor a teaching environment.

Our goal is to make GILD a tool that is easy to deploy and maintain, easy to learn and use, and actively supports various learning and teaching strategies.

3.4 Eclipse Community Perspective:

Community oriented. We will build on top of other research and expect other research groups may wish to build on top of our work. We also expect several communities to flourish around GILD's repository. Instructors and students will be able to share their content and their experiences using GILD.

Open source. By making GILD open source we expect two main outcomes. First, it will be free for anybody to use. Second, other projects can reuse GILD or part of it in order to provide functionality or products beyond the original goals of the project.

Extensible. Our objective is to create an architecture that permits the customization of GILD to specific environments, for example, teaching C/C++.

4. Adoption Challenges

As in many areas of technologically influenced change, the adoption challenges for GILD are classic. They include the following: infrastructural capability, staffing and training issues, and, perhaps most importantly, attitudinal differences in the potential adopter community.

Infrastructural capability: in the post-secondary education community, a committee often determines changes in technological infrastructure for teaching. Good teaching/learning rationales have to be provided to these committees before any technology that "pushes the infrastructure envelope" is adopted. Typically, those responsible for day-to-day teaching infrastructure are the first to recognize the benefits of such upgrades, and are often looking for "good causes" to support their requests. Of course, any infrastructural change must be manageable for the target institutions. Moreover, most post-secondary institutions have commitments, both formal and informal, to supporting widely varying technologies within the same infrastructure. Technologies that need to "take over" existing infrastructure are often not successful in this milieu. GILD should balance as small an infrastructural change footprint as possible while providing exceptional teaching and learning possibilities.

Staffing/training issues: The typical post-secondary computer help desk is a hive of activity – especially when assignments are due. Staff are usually run off their feet. Therefore, to be readily adopted, new technology must be robust enough to run without much intervention by skilled staff. However, in GILD's case, we do want users to ask questions, but about content, not operation. Therefore, front-line staff have to be trained to use GILD and be kept up-to-date on the types of questions to expect from users. Moreover, GILD will need to be easy to install and run on users' home machines. We could anticipate that some demands may be reduced on teaching assistance staff, as it will be easier for students to help one another 24/7. As instructors, we have noted a big decrease in student questions when we provide facilities to support their interactions outside the classroom.

Attitudinal differences: In most post-secondary institutions, the resources used to teach a given course are very much influenced by the preferences of the individual instructor. Any new learning and teaching technology would have to easily interface with the majority of the resources already used by the instructor. Failure to do so would be a major barrier to adoption. Certainly many instructors want to improve their teaching methods, but almost every instructor has already spent a great deal of time preparing and testing material. The GILD system must be able to integrate this legacy material and provide instructors with new ways of using and building on it.

Other challenges: We have yet to discover what other challenges and barriers to adoption there remain with respect to the Gild tool. We look forward to feedback at the Adoption Centric Software Engineering workshop on people's opinions and insights about our proposed work. For a tool to be adopted, it must fill a need and provide advantages that outweigh the disadvantages from adopting such a tool. Do the needs we identified resonate with others in software engineering? Does it seem likely that we can overcome the challenges and reduce the potential adoption barriers we identified? And are there other significant challenges that we may face that we have not yet considered?

5. Conclusions

GILD is an integrated learning and development environment for programming. Our goals for this environment are to improve the experiences of students' learning and professors' teaching Java programming. Besides the more general adoption challenges, GILD faces challenges of providing gains with respect to the pedagogical needs of both students and teachers. In this paper we described a project in which we intend to overcome these adoption challenges by making use of the powerful infrastructure and large number of plug-ins available in Eclipse, as well as by thoroughly researching the pedagogical needs that such an environment provides.

Our project is in its early stages, where our challenges include the rigorous definition of the technological as well as pedagogical needs of the intended GILD users. We are embarking on a big effort to collect requirements about how an integrated learning and development environment can be used for teaching programming. We will consider scenarios of how such a tool could be used

and document these. We are currently striving to provide technical solutions to the identified needs, as well as creating a research environment in which the adoption barriers are well-understood and addressed through proven research methods. The user requirements will be defined through iterative prototyping with intended categories of students and teachers, while the environment will be user tested following its development. We also intend to address adoption by organizing workshops and training with GILD and observe its use in classrooms for continued improvement and removal of adoption barriers.

Developed through well-identified research methods, we expect the GILD project to provide a powerful tool to convey our research practices to other disciplines and to advance research in our community. It will foster continued collaborations within the Eclipse community as well as among researchers in the area of adoption-centric software engineering.

References

[1] Slime UML plugin,
http://www.mvmsoft.de/content/plugins/slime/slime.htm
[2] Pair Programming Plug-in,
http://sourceforge.net/projects/sangam
[3]Eclipse Overview,
http://www.eclipse.org/whitepapers/eclipse-overview.pdf
[4] Eclipse Website: www.eclipse.org
[5] Eclipse perspectives,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.user/concepts/concepts-4.htm
[6] Eclipse Views,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.user/concepts/concepts-5.htm
[7] Natures,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.isv/guide/resAdv_natures.htm
[8] Refactoring,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.jdt.doc.user/reference/ref-115.htm
[9] CVS,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.user/reference/ref-47.htm
[10] Code assist,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.jdt.doc.user/reference/ref-143.htm
[11] Eclipse Markers,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.isv/guide/resAdv_markers.htm
[12] Team Decorators,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.isv/guide/team_ui_decorators.htm
[13] Help System,
http://dev.eclipse.org:8080/help/content/help:/org.eclipse.platform.doc.user/tasks/tasks-1g.htm
[14] DrJava: A lightweight pedagogic environment for Java, Eric Allen, Robert Cartwright, and Brian Stoler September 7, 2001, presented at SIGCSE 2002
[15] David J. Barnes & Michael Kölling, Objects First with Java A Practical Introduction using BlueJ Prentice Hall / Pearson Education, 2003, ISBN 0-13-044929-6
[16] Eclipse FAQ,
http://www.eclipse.org/eclipse/faq/eclipse-faq.html
[17] TextPad, http://www.textpad.com/
[18] Evaluating the usability of Web-based learning tools, M.-A. Storey, B. Phillips, M. Maczewski and M. Wang, Special issue on Evaluation of Learning Technologies in Higher Education, Guest Editor: Grainne Conole, *Educational Technology & Society 5 (3) 2002, ISSN 1436-4522*
[19] In support of student pair-programming , Laurie Williams, Richard L. Upchurch , Technical Symposium on Computer Science Education, Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education 2001, Charlotte, North Carolina, United States
[20] Effective Teaching Behaviors in the College Classroom. Harry G. Murray in Effective Teaching in Higher Education: Research and Practice. Editors: Raymond Perry, John Smart. Agathon Press, New York. 1997, pgs. 171-204
[21] Eclipse Metrics Plug-in,
http://www.teaminabox.co.uk/downloads/metrics/

Trademarks

An Authoring Framework for Live Documents: Collaborative Writing with Infinite Persistent Annotated Change Tracking (ImPACT)

Adam Murray
University of Ottawa
amurray@site.uottawa.ca

Jeff Michaud
University of Victoria
jmichaud@uvic.ca

Timothy C. Lethbridge
University of Ottawa
tcl@site.uottawa.ca

Abstract

Live Documents make information manipulation and retrieval interactive and collaborative for readers. However, there is little support for authoring of such documents. In this paper, we lay the groundwork for an authoring framework, and provide a feature set derived from social aspects of conversation. Collaborative Writing with Infinite Persistent Annotated Change Tracking (ImPACT) allows and support many features for authoring of live documents including but not limited to transparent version control management, concurrent authoring, and advanced visualizations. These features should be adoptable since they can be built into programs such as word processors that users are familiar with.

1. Introduction

Why are conversations great? They are dynamic and fluid; participants' ideas grow, evolve, and flow from one topic to another. Documents, on the other hand, are not like that. We believe incorporating the interactivity and creativity that occurs in conversations into live documents lays the groundwork for an authoring framework. We suggest that a feature set that includes collaborative writing support and infinite persistent annotated change tracking are the best way to achieve this.

In this paper, we propose the foundation for an authoring framework for live documents, as deemed necessary in [8]. Since live documents are aimed to achieve a high level of awareness and interactivity, it seems natural to choose a highly interactive medium that is infused with awareness to provide guidance for our framework. We also believe that incorporating aspects of conversation will help promote the use of live documents.

We take the opportunity to reflect on the origins of documents and conversations in the social context of the real world, and to bring this knowledge into the digital context of live documents. In particular, we note that a document's persistence is fundamental to its ability to acutely convey information. We also note that conversations have differing strengths rooted in their dynamic and interactive natures. Wanting the dynamic nature of conversations with the persistence quality of documents leads us to establishing the features we believe necessary for an authoring framework for live documents: enhanced collaborative authoring support and Infinite Persistent Annotated Change Tracking (ImPACT).

The remainder of this paper is structured as follows. In Section 2, we provide the theoretical groundwork for our authoring framework by exploring and contrasting elements of documents, live documents, social translucence and conversations. In Section 3, we introduce the essential feature set and requirements for both collaborative writing and ImPACT while noting the effects to the current live documents theory. And finally, Section 4 provides our conclusions and future work.

2. Theoretical Groundwork: From Documents to Conversations

2.1. Documents

Documents have a history almost as long as the history of the human race. In this time they have taken many forms, most recently in a digital manner. Documents are an instrument for human communication. They are typically used as a platform for mutual understanding or consensus of meaning. A document is created for specific purposes, audiences, and uses.

A document comprises three essential types of information: data (the essence of the document, the content and the meaning), format (appearance), and structure (the parts of the document and the relationships between them). Structure and format contribute to absorbing the essence of a document.

Documents are typically persistent, enabling them to be easily shared, searched, copied, reused, etc. The notion of persistence is central to our authoring framework. We argue that a document's evolution history must be persistent in order to support multiple parties collaborating in authoring a document.

2.2. Live Documents

A live document aims to improve on conventional document's ability to convey information to its readers. Mockus, Hibino, and Graves describe live documents as interactive documents with embedded, contextual information visualization components [5]. Their approach involves extending web pages using simple visualization components in order to facilitate collaboration among project members.

Weber et al. [8] greatly expand on the work described above. The main idea of keeping documentation "in sync" is discussed, and reinforced with a set of requirements for live documents. In particular, they introduce the notion of a live document *sensing* its context, and *adapting* its contents to the recognized environment. The following are Weber *et al*'s requirements; a live document must:

R1: Have a state
R2: Manage state automatically

R3: Support reuse
R4: Adapt intelligently to the presentation
R5: Adapt intelligently to the reader
R6: Support advanced visualization
R7: Support contextual search and navigation
R8: Support scripting

These requirements focus upon elements to enrich the reader's experience. A noted gap in the live document research is the need for an authoring framework [8], which we aim to initiate with this work. Weber *et al*. suggest that the framework should allow users to use traditional authoring tools while at the same time allow meaningful interactions with live documents.

2.3. Social Translucence

An authoring framework for live documents is not complete without supporting multiple authors; as such, we believe leveraging technology to incorporate social aspects of conversation that can improve how authors interact essential. Enabling multiple authors to participate in the process of writing a document is a topic within the research area of computer supported collaborative work. Theories and tools in this domain have been developed to enable co-located and remotely located users to effectively collaborate [1][2][6]. Visualization techniques are used to make each participant aware of each other's presence and each other's actions. Conflict management protocols and access control are put into place to handle problems and to facilitate cooperation.

To ensure that a collaborative environment enables interactions possible in face-to-face social situations, the following attributes must be achieved with the visualizations, protocols:

- Awareness of other participants
- Awareness of the constraints of the chosen protocols and controls.
- Accountability of participants for their actions.

The combination of the enhanced awareness and the accountability requirement has been described and labeled "social translucence" by Erickson and Kellogg in [3]. To adapt their work to our goal of making collaborative document writing socially transparent, a tailored approach to awareness is appropriate.

We believe that awareness should be addressed in three different areas: awareness of presence, awareness of actions, and awareness of mood/attitude.

- **Presence**: Whether the mode of collaboration is asynchronous or synchronous, participants should be made aware of when and where in the document the other participants are (or have been) interacting.
- **Actions**: Document editing events and others (such as which references were consulted) can be made accessible and visualized as needed.
- **Mood/Attitude**: A format is needed to convey the feedback normally found in facial and body gestures. A symbolic representation is likely best. We propose the use of something similar to "emoticons" [7] so that authors can quickly and efficiently convey their moods and attitudes towards segments of the document. We suggest a colouring mechanism, similar to applying a font, with meanings such as:
  - I {mildly / strongly} {agree / disagree} with this
  - I think this is {well / poorly} expressed
  - This is a {key / critical / unimportant} point

When we use the term 'colouring', however, we do not necessarily mean that actual colours should be used to indicate mood and attitude; other user interface coding schemes could also be used.

2.4. Conversations

Social translucence suggests incorporating social aspects of the real world, such as elements of conversations, into a digital context, such as our authoring framework. Thus, in this section we continue our theoretical groundwork with an examination of interesting aspects of conversation. Conversations have many relevant parallels with documents.

First, the primary (and essential) purpose of a conversation is similar to that of a document: "It is through conversation that we create, develop, validate, and share knowledge." [3, p. 67]. Many of our first skills as children are taught with the aid of conversation, including the skills of reading and writing.

Second, conversations, like documents, can be highly structured. The structure of a conversation varies according to the number of participants, the pattern of participation (such as the degree of synchrony), the duration of the discussions and the number of discussions or topics covered [3].

Conversations, however, have some differences from documents that can make them a superior format for exchanging knowledge. Firstly, all aspects of conversation are directly affected by social factors placed upon the participants. For instance, some people are inclined to be polite and not dominate a long conversation and allow other participants to speak.

The second advantage conversation has over documents is that their process is dynamic. As a conversation proceeds, the participants are continuously interpreting the dialog, verifying that they have been understood, and offering new contributions where they feel it is appropriate [8].

Another benefit conversation has is that it produces continuous feedback on what has been shared between the participants. Often this takes the form of verbal responses. However, in situations where the participants are collocated facial expressions and body language are also forms (voluntary or involuntary) of feedback. Our idea of extended emoticons, discussed in the last section, can help to provide this feedback.

It is important to note, however, that document persistence, as discussed in Section 2.1, is an advantage over conversations. Conversations usually only remain in

the memories of the participants and typically the process of retelling factual information learned in conversation is prone to mistakes and errors.

2.5. Conversational Live Documents

For our authoring framework to allow live documents to adapt intelligently to the writer, it must integrate the strengths of documents and conversations, as discussed in the previous sub-sections. Doing so will allow live documents to inherit some of the benefits obtained when knowledge is exchanged through conversation. At the time of authoring, operations such as brainstorming, composition, and editing can all benefit from conversation.

Live documents can also realize a set of benefits by applying the concept of persistence to conversation. This will take the form of a history of the document that will detail how it has evolved from its inception to its current state. Being able to discover how an idea (or paragraph, or title, or diagram) evolved has the potential to provide a richer understanding of the live document.

The question that now becomes apparent is how can live documents inherit the advantages of conversation while still realizing the benefits of persistence? We believe the former can be achieved by making the process of editing live documents socially translucent and the latter by ensuring a system of universal access and infinite persistent annotated change tracking. We discuss these essential features for our framework in the next section.

3. Collaborative Writing with Infinite Persistent Annotated Change Tracking (ImPACT)

In our framework a live document must adapt intelligently to, and facilitate collaboration among, its authors. The live document maintains and uses knowledge about the authors and their interactions both with each other and with the document. Similar conversational interactions of this type are present in a Wiki [9]; however, our work contributes further as we envision truly concurrent collaboration and advanced visualization, among other features, nested in middleware, such as Microsoft Word.

In this section, we expand on the theoretical groundwork described in Section 2 and state the essential features of collaborative writing and ImPACT for our authoring framework. Some of the features we feel ImPACT allows or support include: Transparent Version Control Management, State-based Document Traversal, Document Variants (Branching), Powerful Search Capabilities, Undo Across Sessions, Concurrent Authoring, and Advanced Visualization. Use of our framework for authoring live documents has the potential to enrich how live documents fulfill the existing requirements outlined by Weber et al. in [8]; hence, where pertinent, we note how features relate to the requirements we summarized in section 2.2.

3.1 ImPACT

In many simple document-editing environments, saving or closing a document removes all history of changes. Modern word processors, such as Microsoft Word, however, allow the time-stamped tracking of both changes and annotations (comments) contributed by multiple authors. However, if one author wants to 'accept' or approve the changes of another, he or she can only do so by removing the record of those changes. Keeping a complete history even after changes are accepted allows much more flexibility for everyone interacting with the evolving document; little research has been performed on this concept, which we call Infinite Persistent Annotated Change Tracking (ImPACT). This notion opens up the possibility for much additional functionality, which we discuss in the following sub-sections.

3.2 Transparent Version Control Management

Enabling collaborative writing by utilizing a version control management system is a topic that has been previously explored. In [1], Byon et al. have established that a version control system, which records changes in documents at a fine granularity, is the most beneficial. We believe supporting the finest level of granularity is important, though hierarchical decomposition of granular levels is also useful for a reader. For example, a user can review changes by character, by sentence, by section, and so on. The version control management system must not burden the users; hence, we propose a transparent version control system. We imagine authors working on personal documents with their collective modifications being stored at a master source. We explore the notion of branching further in the next section.

3.3 State-based Document Traversal

Each time a document is changed, we say it enters a new state. State-based document traversal would allow two capabilities: 1) Switching an artifact to a different state, which may be a previous one, or one reached by a different path of state transitions (e.g. omitting a particular set of earlier changes). 2) Interacting with the history of events (state transitions), including annotations (e.g. comments) applied to particular events, document elements or to the events or states themselves. Furthermore, replaying some path through the document's state space, e.g. to better understand the reasons certain changes were made is another interesting feature to be examined.

3.4 Document Variants (Branching)

In addition to working with copies of a master document, authors could have the option of working with their own document variants distinct from the master (or mainline) branch. A variant represents a distinct set of

states, and a distinct path through those states; the set of variants, plus the mainline, form a tree or lattice. Variants allow for multiple writing strategies such as incorporating versions of a document with rationales, or multiple ways of expressing different concepts. Authors can edit any of the branches and merge sub-trees from each (i.e. turning the tree of editing states into a lattice).

3.5 Powerful Search Capabilities

The environment we envision would permit sophisticated searches through the document content as well as through all the meta-information such as knowledge of authors and their actions, annotations, state space, version branching, etc. Semi-automatic cross-referencing document elements and events to other relevant artifacts (such as emails or meeting notes) is another useful idea (an idea explored in [4]). This feature extends the notion of R7 (contextual search and navigation).

3.6 Undo Across Sessions

Undo simply involves rewinding in a stack-popping fashion from a current state, back through a path to an earlier state. The notion of persistent change tracking is much more powerful since instead of merely providing a stack, it provides random access to changes. However, stack-based functions are still extremely useful as a special case.

3.7 Concurrent Authoring

Other authors/readers should be able to see the evolution of the changes in real time, but may choose not to (although constant awareness of who is making changes should be provided). Chat facilities could be used to help coordinate efforts and quickly discuss issues between authors. These conversations can also be recorded and incorporated into the document's history.

3.8 Advanced Visualization

Tools could display a visualization based upon historical attributes such as which sections of the document are new, old, and/or changed most often. Also, patterns of interaction within the document can be displayed, and automatically analysed; these patterns are akin to conversational patterns.

Document readers could display an overview of the state space (as an FSM, perhaps simplified), or the conversation sequence (perhaps showing annotations only with a highly abstract view of the rest of the document). In either case, readers or authors could access parts and/or states of the document by clicking on elements. Also, in such overviews, the recentness of changes should be visible, in addition to the sections currently being worked on. Advanced visualizations directly correspond to R6 from section 2.2.

4. Conclusions

Live documents present an innovative way for presenting information, and supporting collaborative work for knowledge workers. We contribute to previous research with an authoring framework for live documents supporting features such as collaborative writing and Infinite Persistent Annotated Change Tracking. We believe this area of research is highly pertinent, and requires detailed study. We also believe that live documents with the features we have described should be readily adopted by users since we propose merely enhancing and merging widely used technologies found in today's word processors, chat programs, etc.

5. References

[1] G.L. Byon, K.H. Chang, N.H. Narayanan, "An integrated approach to version control management in computer supported collaborative writing" *In Proceedings of the 36th annual conference on Southeast regional conference*, p, 34-43, 1998.

[2] P. Dourish and V. Belloti, "Awareness and Coordination in Shared Work Spaces" in *Proceedings of the ACM Conference on Computer-Supported Cooperative Work CSCW'92*, p 107-114, Toronto, Canada, 1992.

[3] T. Erickson and W.A. Kellogg. "Social Translucence: An Approach to Designing Systems that Mesh with Social Processes." In *Transactions on Computer-Human Interaction*. Vol. 7, No. 1, pp 59-83. New York: ACM Press, 2000.

[4] S. Minneman, *et al.*. A confederation of tools for capturing and accessing collaborative activity. *In the third ACM International Conference on Multimedia*, p. 523-534, 1995.

[5] A. Mockus et al. A Web-based approach to interactive visualization in context. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pp. 181-188, 2000.

[6] W. Reinhard, J. Schweitzer, and G. Volksen "CSCW Tools: Concepts and Architectures" in IEEE Computer, Vol 27, Iss 5, p 28-36, 1994.

[7] K. Rivera, N.J. Cooke, and J.A. Bauhs "The effects of emotional icons on remote communication" in the *Conference companion on Human factors in computing systems*, p 99-100, Vancouver Canada, 1996.

[8] A. Weber, H. Kienle, H. Muller. 2002. Live Documents with Contextual, Data-Driven Information Components, *In Proc. of SIGDOC 2002*, Toronto, Canada, Oct. 20-23, 2002.

[9] http://www.wiki.org/

Evaluating the Eclipse Platform as a Composition Environment

Chris Lüer
School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3430, USA
chl@ics.uci.edu

Abstract

Eclipse is a popular open-source software development workbench, and is suitable for the integration of experimental research tools. It includes a state-of-the-art plug-in technology, and so it can be considered as a platform for reusable components, or composition environment. We evaluate the Eclipse platform from this point of view, and discuss strengths and weaknesses of its plug-in technology.

1. Introduction

The Eclipse Platform [1,5] is an open-source workbench for the integration of software development tools. Because of its extensibility and its industrial-quality user interface, Eclipse has been used as a framework for software engineering research projects [4,7]. Its advanced plug-in technology allows the concurrent deployment of a large number of independently developed components; this makes Eclipse a *composition environment* for development tools. In previous work, we have surveyed composition environments, including software architecture environments and visual programming environments [6]. In this work, we will evaluate how Eclipse compares when it is considered as a composition environment, and we will give recommendations as to how the platform could be improved in this respect.

Eclipse consists out of a core platform, and numerous plug-ins. The standard configuration of the Eclipse workbench is, indeed, mostly realized as collection of plug-ins. Each plug-in has a file system directory of its own, in which its code and a manifest file are located. The manifest is an XML file that provides information about the plug-in. When the Eclipse platform is started up, all the plug-in manifests are read, and the associated plug-ins are hooked up with the system. For example, a plug-in might define an additional toolbar button, and after the manifest has been read, this toolbar button will be created. However, the code of the plug-in is not loaded and executed until the button is actually pressed. The Eclipse Equinox project [3] is concerned with exploring new plug-in technologies for Eclipse; however, it is still in the early planning stages.

Web directories list over 200 existing Eclipse plug-ins in varying stages of completion [2]; many of these are experimental in nature. While most plug-ins are integrated with other plug-ins only through the common user interface provided by the platform and through the underlying file system, tighter integration among plug-ins is possible. The integration technologies provided by Eclipse are in no way restricted to user interface integration.

2. Strengths of Eclipse

Multi-level architectures. Most plug-in systems only provide for one level of plug-ins: components can be plugged into the framework, but there is no generic way to allow plug-ins to be extended by plug-ins of their own. In Eclipse, though, much of the environment itself is realized in the form of plug-ins to the Eclipse core platform; and many other plug-ins are secondary to those primary plug-ins. Any plug-in can define extension points that other plug-ins can connect to. As a consequences, tool architectures can have many levels.

Explicit ports. Each plug-in can define any number of extension points, to which other plug-ins can connect themselves. Extension points are specified in the manifest file of a plug-in. They function as specifications of optional requirements of a component, or requirement ports. The client (i.e., the component defining the extension point) declares that it can support any extension that adheres to a given interface. By making extension points explicit, it is easy to see where and how components may be plugged into a system.

Self-description. Eclipse manifest files provide description of plug-ins. The information given is partly redundant to the information in the code, and partly extends it. Redundant information is given for performance optimization; it allows the platform to be aware of the essential

properties of a plug-in before it has loaded its Java classes. Non-redundant information includes the definition of extension points as requirement ports, version numbers, and user interface information.

Containers to encapsulate components. Eclipse puts effort into effectively encapsulating plug-ins from each other. A plug-in can access another plug-in only if it is declared as "required" in its manifest. To make this possible, each Eclipse plug-in has a Java class loader of its own; this class loader can verify whether an attempted access to another class is legal according to the manifest. The class loaders thus realize a container concept – each component runs in a container of its own that regulates communication with other components. In this way, Eclipse manages to avoid unintended or undocumented dependencies between plug-ins.

3. Weaknesses of Eclipse

Strict requirements are not possible. All Eclipse plug-ins are optional: they can extend the functionality of existing components, but are never required. It may be desirable, however, to strictly require the presence of certain components. For example, a word processing component may require a language library that includes functions such as hyphenation and spell-checking, which depend on the natural language employed. A word processor without such a component does not make much sense; on the other hand, it should be possible to replace this component independently from the rest of the word processor. In this case, the plug-in does not constitute an optional extension of the system, but a required component that has multiple, alternative implementations. Eclipse does not support such required components.

No explicit connectors. The Eclipse plug-in architecture is based on the assumption that each plug-in extends specific extension points in specific components. It is not possible to have alternative implementations of the same functionality, since every component that provides a certain functionality (i.e., connecting to a given extension point) will be hooked up to the system. Architectural connectors can be used to mediate between components. With explicit connectors, it would be possible to have two plug-ins provide alternative functionalities, and use only one of them. As an example, a word processor might have two alternative text layout components: one that is fast and imprecise, and one that is slow and precise. It makes no sense to use both layout components at the same time; instead, one should be chosen over the other. A connector between the word processor component and one of the two possible layout components would encapsulate the selection.

Other benefits of connectors are known from the literature. For example, connectors can be used to adapt components that are not exactly compatible to each other. An adaptation is a property of the relationship between two components, and not a property of either one, and thus it should not be encoded in either of the components, but in a connector.

Whether strict requirements and connectors should in fact be added to the Eclipse platform is a question of the priorities of its users. On the negative side, both of these features would add some conceptual overhead. However, we believe that they could be added to the existing platform in a comparatively non-intrusive manner, and without breaking the backwards-compatibility of existing plug-ins. These features would greatly add to the architectural expressiveness of the platform, and would enable it to be used with more diverse configurations of components.

4. Conclusions

The Eclipse Platform is a state-of-the-art plug-in system for the domain of software development tools. It goes beyond most plug-in frameworks in incorporating component-based technologies such as ports, run-time containers, and self-description. This makes it a perfect candidate for integration of experimental tools, such as those typically developed at research institutions.

Additional advantages of Eclipse as a platform for research prototypes are its extensible user interface and the fact that its source code is open. User interface design is often not a focus in the development of experimental tools; thus, such tools can benefit greatly from the easily extensible, industrial-quality graphical user interface of Eclipse. Open source code has many advantages to researchers; among the most important ones are modifiability of the product, increased trust in its correctness, and the assurance that the product will not suddenly disappear from the marketplace.

We have pointed out the strengths and weaknesses of Eclipse as a composition environment. While we believe that its strengths make Eclipse a beneficial addition to most software engineering research projects, extending the Eclipse platform in the ways suggested may make it even more powerful.

References

[1] Eclipse Platform Technical Overview. http://www.eclipse.org/whitepapers/eclipse-overview.pdf. 2001.

[2] Eclipse Plugins. http://eclipse-plugins.2y.net/.
[3] Eclipse Projects: Equinox. http://www.eclipse.org/equinox/index.html.
[4] Eclipse Research Community. http://www.eclipse.org/technology/research.html.
[5] Eclipse.org. http;//www.eclipse.org/.
[6] Lüer, C., and van der Hoek, A. Composition Environments for Deployable Software Components. Technical Report 02-18, Dept. of Information and Computer Science, University of California, Irvine, 2002.
[7] Rayside, D., Litoiu, M., Storey, M.-A., and Best, C. Integrating Shrimp with the IBM Websphere Studio Workbench. Proceedings of the 9th NRC/IBM Centre for Advanced Studies Conference (CASCON'01) Toronto, 2001. 79-93.

Matching Multiple COTS: Can We Achieve a Happy Marriage?

Carina Alves, Anthony Finkelstein
Department of Computer Science
University College London
{c.alves, a.finkelstein@cs.ucl.ac.uk}

Abstract

In this position paper we investigate the challenges that arise when selecting multiple COTS. We analyse the matching between customers requirements and COTS packages and propose an approach to identify possible mismatches. In particular, a range of conflicts can arise from these mismatches. Therefore, effective conflict management strategies are needed to support the selection of COTS packages. Finally, we present a research agenda to address problems of multiple COTS selection.

1. Introduction

Integrating COTS (Commercial-Off-The-Shelf) packages instead of developing systems from scratch is a promising approach to improve the state of the practice in software engineering [4]. When buying COTS products, customers can take the advantage of acquiring products that have been tested many times by other users with consequent improvement in software quality [16]. Moreover, the system productivity can be increased, as these packages are currently available in the marketplace. According to [6], the selection of one package usually depends on other packages to be selected in order to support a complete functionality. We argue that integrating multiple COTS involves many challenges and risks. In particular, a number of dependencies and conflicts can occur among the packages as well as among the package capabilities and the customer requirements. It is also very likely that selected packages have to interoperate with bespoke systems, which involves extra constraints over the selection process.

In previous works [2][1] we have developed a goal driven approach to manage conflicts that can occur between a single package and users requirements. We have showed that COTS selection demands some form of inexact matching between features and requirements, for example: there may be requirements not satisfied by any available package, requirements satisfied by some joint packages, requirements partially satisfied, features not initially requested but that can be helpful, irrelevant features or even unwanted features. Moreover, there may be cases where critical requirements cannot be entirely satisfied without considerable product adaptation and others where these requirements must be compromised to accept products limitations. In this context, it is necessary to engage in an extensive process of requirements negotiation [14] in which the requirements of the organization are balanced against the capabilities of the package.

In this paper we investigate the problems that may arise when selecting multiple COTS packages. In these situations, we have to cope with the situation where several products from different vendors are being evaluated to satisfy distinct and specific functionalities. In addition, we have to ensure interoperability and compatibility between them. Our main hypothesis is that in order to successfully develop COTS-based systems, conflicts have to be properly managed. We envisage an approach where developers can identify conflicts, analyse it, explore potential resolutions and examine the associated risks of proposed resolutions.

The remainder of this paper is structured as follows. In section 2, we describe a mail server selection example based on [5] to motivate the discussions presented herein. Section 3 describes some relevant aspects of COTS evaluation. In Section 4, we describe our proposal to support the matching process. Section 5 describes the conflict management. Finally, Section 6 discusses some open issues.

2. A Case Study

Organizations need messaging solutions that enable their employees to maximize collaboration and communication efficiency. There are a number of packages available in the market that support an integrated messaging environment with functionalities such as:

- Messaging-based applications, such as workflow and information tracking;
- Real-time collaboration so that virtual teams can meet and work together across geographic and organizational boundaries;

- Different options of mail clients from a single server giving users more flexibility.

Note that, in order to keep competitive advantages, large vendors usually provide mail packages with quite similar functionalities. Therefore, selecting the "best" mail package requires a careful assessment of features that help differentiating candidate products.

In the following sections, we explore the evaluation of mail server packages and how they satisfy a set of requirements. We have chosen the two leading packages, Microsoft Exchange 2000 [12] and Lotus Notes/Domino 5 [11] to evaluate. In particular, the former is primarily a messaging system that includes improved application development capability, while the latter is mainly an application development environment that includes robust messaging capability. Therefore, if the main organization goal is to develop custom collaboration applications, Notes/Domino might be favored. Note, however, that as side effect the use of that package is more pervasive in the organization than Exchange.

3. Evaluating Packages

The first task of any software development is specifying users requirements. However, there are some fundamental differences between the traditional requirements engineering process and the COTS-based one. When defining requirements for COTS-based systems, the requirements specification does not need to be complete. Instead, initial incomplete requirements can be progressively refined and detailed as soon as products are identified. High-level needs are identified using typical elicitation techniques, such as interviews and use cases. In our goal-driven approach, these needs are called goals that represent requirements in a higher level of abstraction (Lamsweerde [10] gives a complete discussion about goal-driven requirements engineering). From these goals, possible COTS candidates can be identified in the marketplace. In our case study, we consider the following goals that the mail server package should satisfy:

M1. Ensure and communicate message delivery.
M2. Ensure fast message delivery.
M3. Support development of collaborative applications.
M4. Support protection against external attacks.
M5. Provide installation and administrative facilities.

The evaluation of each package is performed based on well how they satisfy the stated goals. Note that goals have degrees of satisfaction instead of a one-to-one basis (i.e. satisfied or not). Consider, for example, that one mail server partially supports the goal M1 "*ensure and communicate message delivery*", allowing users to configure number of delivery retries, but not providing message delivery notification which is also a desirable attribute to fully achieve the stated goal. Therefore, it is necessary to perform a complex matching between goals and COTS packages.

4. Matching Process

In our approach, the matching between goals and COTS consists of a compliance-checking mechanism, as we have to evaluate whether goals are sufficiently achieved in terms of features. We have proposed a set of matching patterns to classify the matching in a more formal way (see [1] for a full discussion).

Fulfill – The matching is complete as the feature fully satisfies the requested goal.

Differ – This situation is the most complex and probably the most common case. There is a partial mapping between goals and features. However, the feature does not fully satisfy the goal.

Fail – This situation occurs when packages provide extra features not requested by customers.

Extend – In this case the evaluated products cannot satisfy a specific goal.

Unknown – Available information is insufficient to classify the matching.

Note that the matching proposed here does not necessarily correspond to the veracity of products capabilities and limitations. We have invented a matching scenario to explain some important aspects of how conflicts can arise from mismatches. Table 1 shows a hypothetic matching between mail server packages and goals. In particular, mismatches can arise from situations with patterns *differ, fail* and *extend.* However, mismatching situations do not necessarily originate a real conflict as some potentially conflicting situations can be accepted without major consequences. The challenging task is then to separate simple conflicts from major ones that can compromise the success of the final system.

Table 1. Example of matching patterns associating mail servers features with customer requirements

Goal	Matching between goals and features	
	Exchange 2000	Lotus/Domino
M1	Fulfill	Differ
M2	Differ	Differ
M3	Differ	Fulfill
M4	Fail	Fail
M5	Differ	Fail

In order to properly deal with mismatches, we propose a desirability attribute to be attached to each goal, representing the importance of a particular goal to be satisfied. Therefore, evaluators can make well-justified decisions based on the desirability of goals. We

propose a five level desirability: <5-*very high, 4-high, 3-medium, 2-low, 1-very low*>. Suppose, for example, that goal M4 has desirability 5. During the evaluation process, we identified that M4 is refined into anti-virus and anti-spam facilities. However, according to Table 1 both messaging packages *fail* supporting that requirement. This is a typical case where a potential conflict can arise. Another mismatch that can lead to conflicts occur due to the fact that Lotus/Domino, which is so far the preferred package, *fail* in supporting M5. As Exchange does not fully satisfy M5, evaluators should explore alternatives to overcome that limitation. In the next Section we analyse these potential conflicts and investigate how possible resolutions can be originated.

5. Conflict Management

The conflict management involves the following processes: understanding the nature of conflicts; analysing the causes of conflicts and other involved factors; and finally, exploring potential resolutions that might be a compromise among these concerns. These resolutions are generated from the combination of interacting and interdependent issues.

In order to tackle the conflicts identified due to the fact that requirements M4 and M5 that are not properly satisfied by mail server packages, we propose the following scenario that comprises the selection of the mail server together with anti-virus, anti-spam and administrative tools as a way to fully achieve those requirements (see Figure 1).

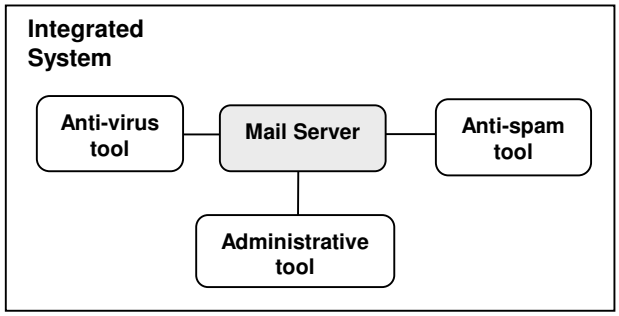


Figure 1. Mail Server Integrated System

We argue that when evaluating several COTS, the core package should shape the selection of the peripheral ones. Therefore, the mail servers will influence the selection of the other tools. Note that such tools are usually compatible with a specific mail server package or run under a particular platform. Therefore, when evaluating one package, developers have to take into account if other tools are well suited or not.

The evaluation of the peripheral tools should also start with the definition of goals, for the Anti-virus tool, the goals include:

V1. Detect and disinfect viruses incoming and outgoing massages.
V2. Support Continuous update for new virus to be detected.
V3. Allow multiple scheduled jobs to be configured for automatic virus scanning.
V4. Allow automatic scan of mailboxes.

The defined goals have to be assessed against the anti-virus features so that we can verify whether these goals are met or not. Similarly, we have to identify the goals to conduct the evaluation of the anti-spam and administrative packages. We have found in the market some tools that cover anti-virus and anti-spam functionalities that might be also considered as selection choices. In fact, the integrated system will reflect the combination of products that best meets customers requirements. On the other hand, evaluators have to judge if it is feasible acquiring all the peripheral packages that means extra costs and integration efforts or accepting mail servers limitations. Therefore, it is necessary to perform a continuous negotiation process.

When selecting multiple COTS, package features have to be matched against the goals for that particular package, which we have illustrated some examples of conflicts that may arise in case of mismatches. Moreover, we have also to take into account the conflicts that may arise during the integration of COTS packages, where issues like interdependencies between packages should be examined. Figure 2 illustrates these interactions, which is the starting point for examining a number of open questions.

6. Discussion

The problems examined in the previous sections represent a significant challenge for software engineers. More precisely the following points might be addressed.

- **Interoperability –** One of the key issues in multiple COTS-based systems is that packages have to interoperate. Some integration elements can be use with the purpose of effectively interconnecting COTS packages. In particular, wrappers may be used for adapting COTS; these are pieces of code

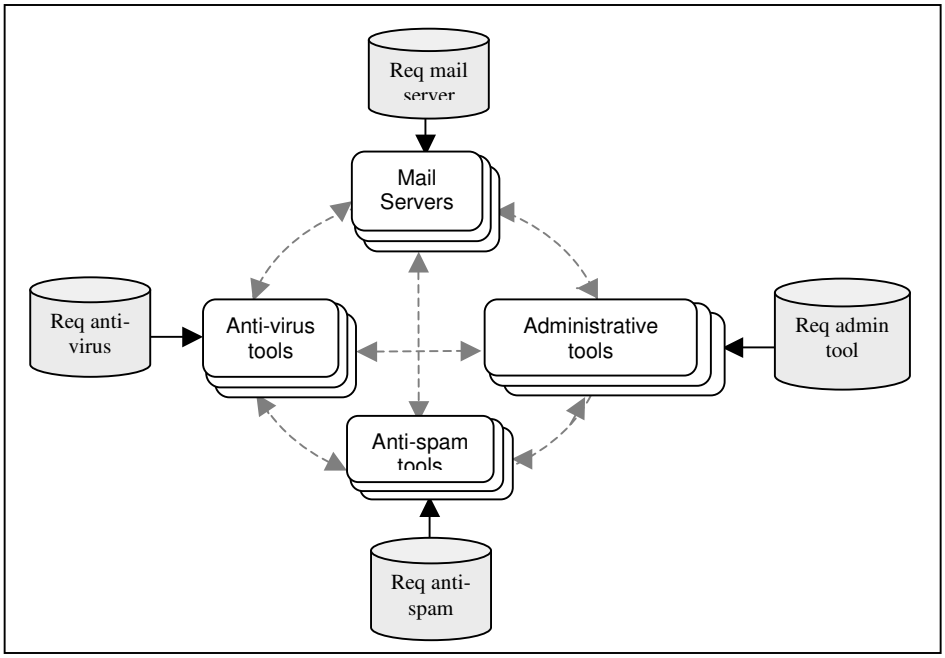


Figure 2. Matching interdependencies among different packages as well as between individual COTS features and customers requirements.

custom built in order to isolate the unwanted functionalities of the COTS package from other components of the system. Glue code can also be used in order to coordinate packages interactions [8].

- **Interdependency –** It is very likely to find interdependencies between COTS packages. For instance, there may be cases where a package requires a particular architecture to run properly; other cases where one package requires other specific package to achieve the desired capabilities or that one package does not work together with some packages. Franch [6] highlights the importance of characterizing dependencies among COTS components that can affect the behavior of the architectural design. We are currently working on how the selection of a specific package configuration (i.e. set of packages interconnected) influences the satisfaction of overall requirements. One potential technique to apply is visualization to explicit display interactions between COTS packages.

- **Decision-Making –** COTS selection requires a careful decision-making of multiple criteria. In particular, selecting the optimal individual package does not imply the optimal joint selection. The successful selection should reflect a balancing between what is wanted and what is possible to meet. Moreover, users have to be prepared to accept commitments. A number of quantitative decision-making methods have been used to support the selection of COTS, like the AHP [15] that has been widely applied. However, the main drawback of such techniques is that they assume a predefined and fixed set of requirements as evaluation criteria. On the other hand, we believe that a more qualitative approach is needed, in which requirements are evaluated in terms of levels of satisfaction rather than a Boolean basis. To achieve that, we have proposed a goal-based framework [1] to support the decision-making of selecting a single COTS, further work is required to investigate the multiple COTS decision-making. We have also investigated the use of heuristics similar to the ones proposed in [9], to support the resolution of conflicts.

- **Evolution –** COTS packages are continuously changing to keep competitive advantages. Therefore, evolution is an unavoidable characteristic of such systems and must be considered as an intrinsic part of the development process. In fact, it is not ensured that new versions of packages will be compatible with other packages previously integrated. Moreover, replacing an unsuitable package can result in several inconsistencies and extra expenses to redesign the system. To cope with that problem, we need some risk analysis techniques to predict the impact of how a future package substitution can affect the other integrated packages.

- **Uncertainty –** Because of the uncertain nature of COTS features, the evaluation of some quality attributes can be difficult to be measured at the time packages are being evaluated, for example the level

of interoperability between the package and other system might only be known after the product has been integrated. Besides that, some vendors can hide functions as a way to warrant their intellectual property, what makes the understanding of packages capabilities a complicated task to be performed. We aim at investigating how artificial intelligence techniques like Bayesian Networks could be applied to deal with uncertain information of COTS packages.

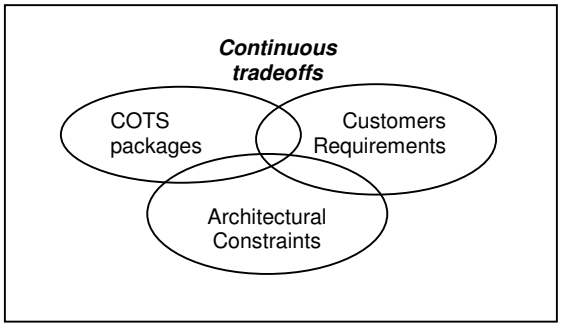


Figure 3 – Multiple COTS systems require tradeoffs among COTS packages, customers requirements and architectural constraints.

- **Architectural Constraints –** Last but not least, the selection of COTS packages requires simultaneous tradeoffs among potentially conflicting issues: COTS packages, customers requirements and architectural constraints [13] (see Figure 3). Multiple COTS integration usually brings several inconsistent architectures frameworks that have to be represented within a single system. This problem has been identified as architectural mismatches. Garlan [7] has identified four categories of mismatches between conflicting architectural components, they are assumptions about: the nature of the components, the nature of the connectors, the global architectural structure, and the construction process. These architectural mismatches have been identified as a fundamental obstacle to COTS-based development.

Acknowledgements. We would like to give special thanks to Xavier Franch and Juan Carvallo for the valuable discussions about the mail server case study. This work is partially supported by CAPES grant – Brazil.

10. References

[1] Alves, C. Finkelstein, A. "A Goal Driven Approach to Manage Conflicts in COTS Selection*" In International Journal on Software Engineering and Knowledge Engineering*. (in submission).

[2] Alves, C. Finkelstein, A. "Negotiating Requirements for COTS Selection" In 8th *International Workshop on Requirements Engineering: Foundation for Software Quality.* 2002.

[3] Easterbrook, S. Beck, E. Goodlet, J. Plowman, L. Sharples, M. Wood, C. "A Survey of Empirical Studies of Conflict" In S. M. Easterbrook (ed) *CSCW: Cooperation or Conflict?* London: Springer-Verlag. 1993.

[4] Finkelstein, A., Spanoudakis, G., Ryan, M., "Software Package Requirements and Procurement." In *8th International Workshop on Software Specification and Design.* 1996.

[5] Franch, X. Carvalo, J. Defining a Quality Model for Mail Servers. In *2nd International Conference on Component-Based Software Systems.*

[6] Franch, X. Maiden, N. "Modelling Component Dependencies their Inform their Selection" 2nd International Conference on Component Based Software Systems. 2003.

[7] Garlan, D. Allen, R. and Ockerbloom, J. "Architectural Mismatch - Why it is hard to build systems out of existing parts" Proceedings of the 17th International Conference on Software Engineering, April, 1995.

[8] Iribarne, L. Troya, J. Vallecillo, A. "Selecting Software Components with Multiple Interfaces" Euromicro 2002.

[9] Lamsweede, A. Dairmont, R. Letier, E. "Managing Conflicts in Goal-Driven Requirements Engineering" *IEEE Transactions on Software Engineering*. 24(11). 1998.

[10] Lamsweerde, A. "Goal-Oriented Requirements Engineering: A Guided Tour." Invited mini-tutorial paper 5th IEEE International Symposium on Requirements Engineering. 2001.

[11] Lotus Domino. http://www.lotus.com/products/r5web.nsf/webhome/nr5serverhp-new

[12] Microsoft Exchange Server. http://www.microsoft.com/exchange.

[13] Ncube, C., Maiden, N. A., "PORE: Procurement-Oriented Requirements Engineering Method for the Component-Based Systems Engineering Development Paradigm." In *International Workshop on Component-Based Software Engineering.* 1999.

[14] Robinson, W. "Negotiation Behaviour During Requirements Specification" *12th Conference on Software Engineering*. 1990.

[15] Saaty, T. "The Analytic Hierarchy Process" New York: McGraw-Hill. 1990.

[16] Wallnau, K. Hissam, S. Seacord, R. "Building Systems from Commercial Components" *SEI Series in Software Engineering*. Addison Wesley. 2002.

Integrating a Tool into Multiple Different IDEs

Lutz Prechelt and Matthias Peter
abaXX Technology AG, Stuttgart
lutz.prechelt|matthias.peter@abaxx.de

Abstract

abaXX Technology produces component-based platform products that help in the construction of Web-based systems, in particular process portals, using Java2 Enterprise Edition (J2EE) technology. Most parts of these products are API-based and hence require support by appropriate construction tools. Much support is available in leading J2EE IDEs, but some specialized tools have to be provided in addition. Since the products are platform-independent, the tools should work in many different IDEs, too.

This position paper shortly describes the issues encountered in designing one of these tools in such a way that it is portable to both Eclipse 2.0 and IntelliJ IDEA 3.0 (and possibly others as well).

1. The starting point: Web UI Framework, Vanilla Portal, PortalBuilder

This section describes the context of the *parts.xml editor* tool discussed in the paper. However, most of the issues can be understood even if you skip this section.

The Web UI Framework is one of abaXX' J2EE component products. It consists of the base framework (similar to Jakarta Struts [2]) for implementing a Model-View-Controller design style, a powerful tag library, a Parts framework for hierarchically modular UI construction and configuration, and the corresponding runtime system.

The Parts framework defines the notion of Part, a fragment of a UI dialog page having its own view, controller, and model. Parts appear to be somewhat similar to Portlets, but in fact they are much more lightweight, can be arbitrarily nested by using containers (CompositePart) with dynamically controlled layout, and can have their look-and-feel be centrally modified by Decorators.

The UI structure of a portal is defined in a file called parts.xml; see an excerpt in Figure 6.

Along with the Web UI Framework we ship the Vanilla Portal, an basic portal frame containing a few generic reusable Parts and predefining the directory structure, naming conventions etc., thus making setting up a new portal development project quick and easy.

On top of the Vanilla Portal comes the third major element, the PortalBuilder: an application for interactively modifying a live portal on the Parts level. The whole portal (see Figure 1) is switched into 'edit mode' (see Figure 2) and then Parts can be introduced, removed, moved, and (re)configured. One can even introduce new (not yet implemented) Parts, will immediately get to see a dummy representation, and can then add actual views and controllers incrementally. During all of these activities, the full functionality of the portal proper is always visible and available for use.

2. The tool and the integration goals

During the implementation phase of a Part (writing the view JSP, controller class, and model bean), one would not normally want to work with the PortalBuilder, but rather with an IDE. Nevertheless, some of the functionality of the PortalBuilder is relevant then, too – namely, entering, reviewing, and modifying parts.xml parameters for the given Part, its parent Part (container), and children Parts, if any.

For simplifying this task, we offer a specialized tool, the *parts.xml editor* (see Figure 4) that allows for generating and editing these entries and that ensures their syntactic and semantic integrity. For maximum benefit, the parts.xml editor needs to be integrated into the IDE.

The following integration between parts.xml editor and IDE would be nice:

(1) User starts the editor from within the IDE.
(2) Editor recognizes which parts.xml is relevant and where to find it; editor loads and saves it.
(3) Editor understands where to find any resource mentioned in the parts.xml (JSP, controller class, model class, decorator, layout, etc.); can make the IDE load and show/edit any of these.
(4) Editor can create lists of candidate resources in any of the various categories from (3) and offer them in selection lists.

(5) Editor makes semantic checks of internal consistency of Parts descriptions. (Note that this does not really require integration.)
(6) Editor makes semantic checks of consistency between Part description and the resources mentioned therein, such as (in increasing order of complexity): JSP exists, controller class exists, controller class is indeed a controller class, all events declared in Part description are fired somewhere, etc.

So far, we have implemented (1), (2), (3), (5), and some of (4), but only basic elements of (6).

3. Integration issues

We have currently implemented the parts.xml editor for three different contexts:

- The IntelliJ IDEA 3.0 IDE ([4], see Figure 3)
- The Eclipse 2.0 IDE ([3], see Figure 4)
- The abaXX Workflow Modeler 3.2 tool (see Figure 5)

The Workflow Modeler is an editor that manipulates process descriptions for the abaXX Workflow Engine, a process execution component that integrates process control logic, calls to business logic, and GUI page flow.

3.1. GUI issues

The GUIs of different IDEs are neither technically nor conceptually identical (or sufficiently similar).

For example, the parts.xml editor tool is programmed in Java Swing (Java Foundation Classes), which is fine for IntelliJ IDEA, because Swing is both its native technical GUI platform as well as its standard look and feel. The same is true for the Workflow Modeler.

For Eclipse, however, Swing is foreign: Eclipse, although also based on Java, is built using a special, native GUI library. While the look-and-feel issues arising out of this can be overcome, at least for a tool as simple as the parts.xml editor, the technical integration becomes a problem: The parts.xml editor cannot easily be shown as a fully integrated subwindow of an Eclipse session, but appears as a separate window on top.

For more advanced tools, these problems will become worse.

3.2. Semantic integration issues

The repository structure and services of different IDEs are very different.

With respect to the integration wish list from Section 2, this means that all functions that require advanced access to the IDEs fact base are difficult to design in a portable fashion. They essentially have to be re-done for each new IDE.

For example, while the file-based integration functions such as most of (3) and simplified versions of (4) can be ported reasonably well, the advanced parts of (6) dig so deep into the IDEs internal model of Java programs that their design will invariably be very different for different IDEs. In some cases (IDEA may be one of them) it may even be impossible to provide this integration, because too little of the respective functionality of the IDE is documented and exposed to the tool developer.

3.3. Conceptual issues

Underlying all of the above technicalities is a much more fundamental problem. Different IDEs approach their problem in conceptually different ways.

For instance, while IntelliJ IDEA follows mostly a rather pragmatic approach, working in a file-level, text-based manner wherever possible, other tools that are more inclined towards a Modeling/CASE Tool kind of approach will not just have different technical mechanisms inside, but will require an add-on tool to have a totally different form and approach in order to get a good conceptual fit. In the case of the parts.xml editor this could mean for instance to have visual representation (and direct manipulation) of the inheritance relationships and event relationships between parts, rather than just a parts tree and attribute table.

4. Conclusion

Integrating a development tool tightly and adequately into more than one kind of IDE is a difficult task. The standardization that would be required for making this task easier is not currently in place, neither on a technical level (APIs, GUI look, basic GUI feel), nor, much more importantly, on a conceptual level (repository structure, service architecture, overall presentation and operation styles).

5. References

[1] abaXX Technology AG, “abaXX.components”, *http://www.abaxx.com*.

[2] Apache Software Foundation, “Jakarta Struts”, *http://jakarta.apache.org/struts/index.html*.

[3] eclipse.org, “Eclipse”, *http://www.eclipse.org*.

[4] JetBrains Inc., “IntelliJ IDEA”, *http://www.intellij.com/idea/*.

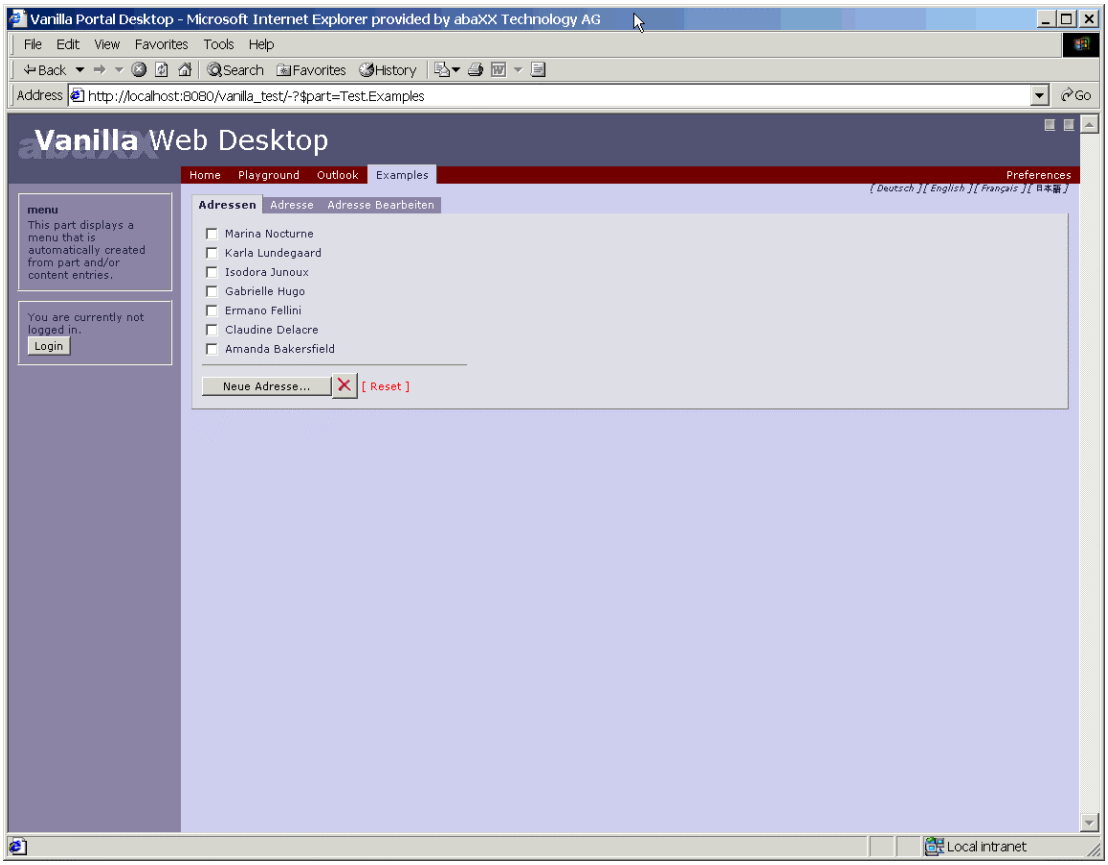


Figure 1: A small portal with 5 Parts: banner, menubar, empty menu, login, and a mini-application

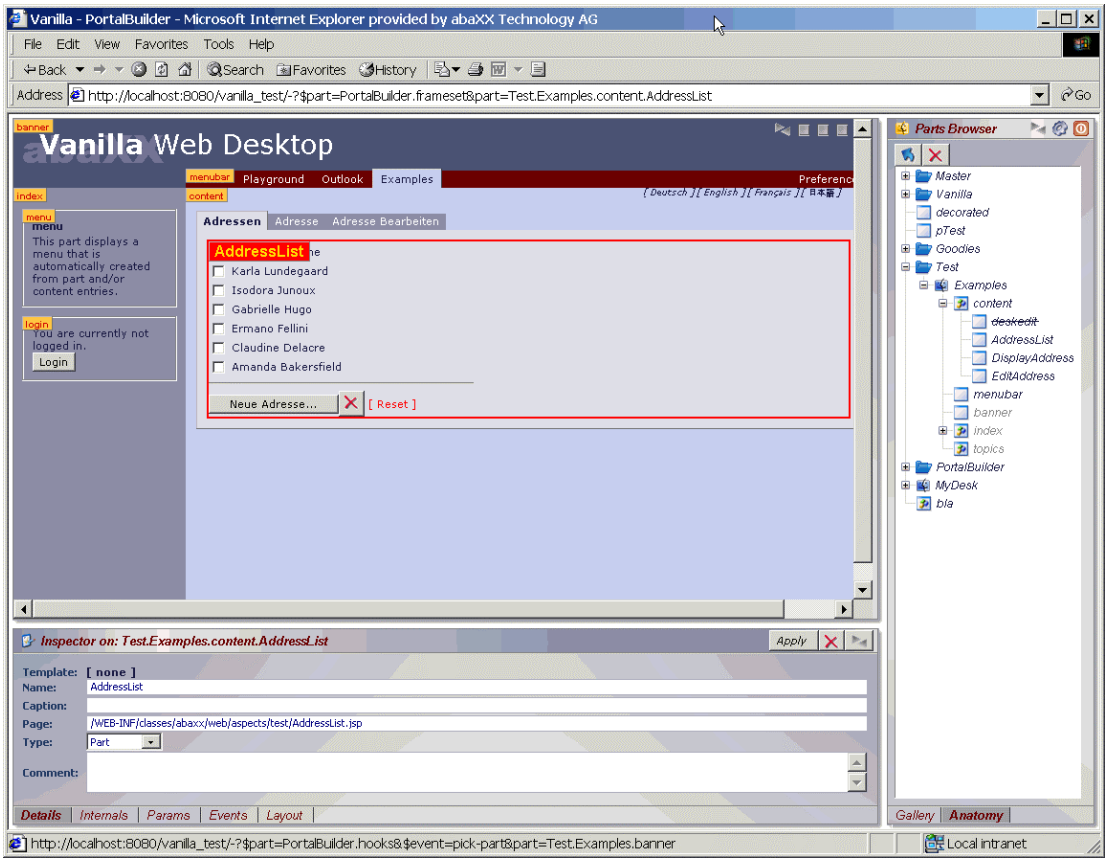


Figure 2: The same portal in PortalBuilder mode. The PartsBrowser shows some of the portal's Parts hierarchy.

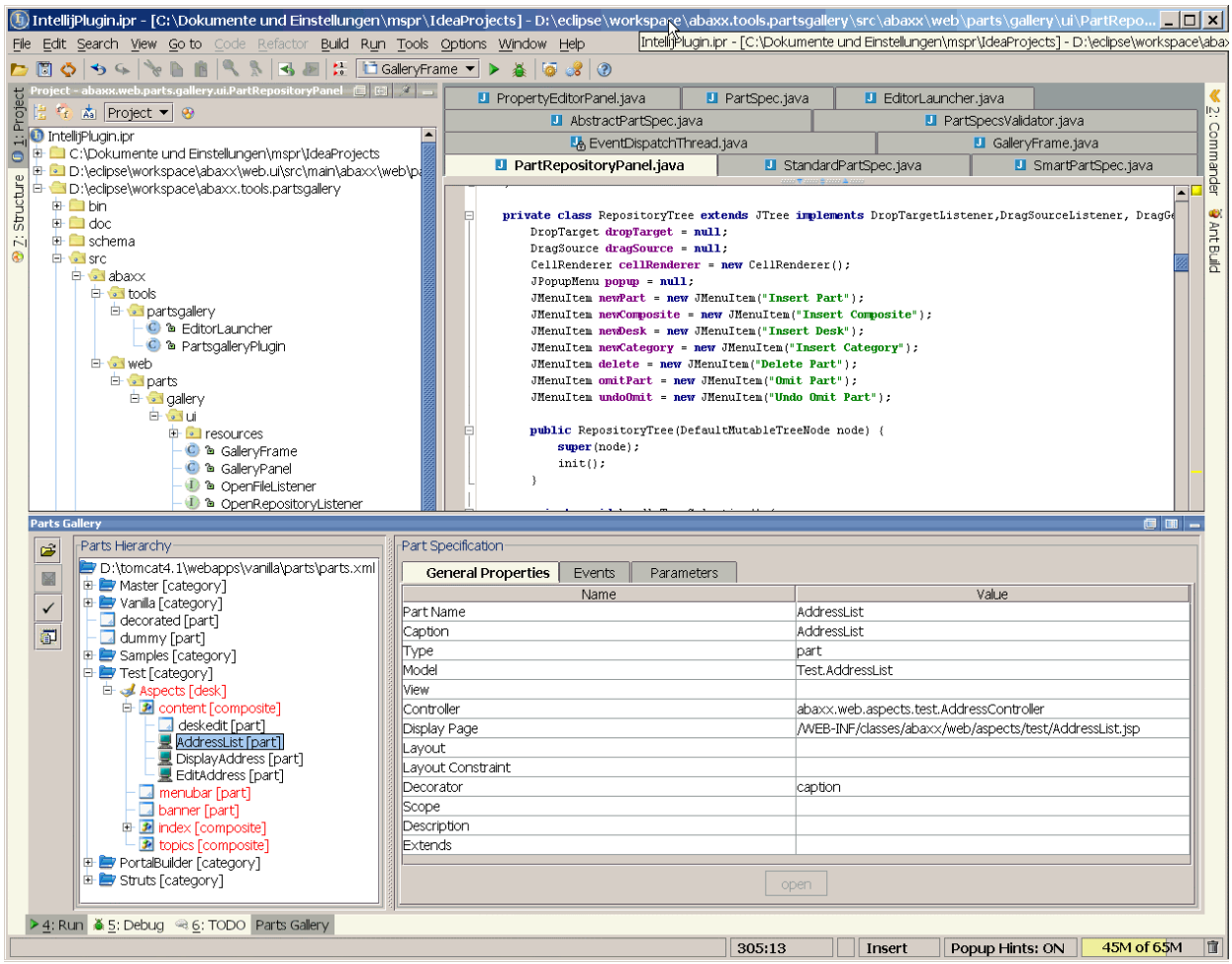


Figure 3: The parts.xml editor tool window within the IntelliJ IDEA IDE.

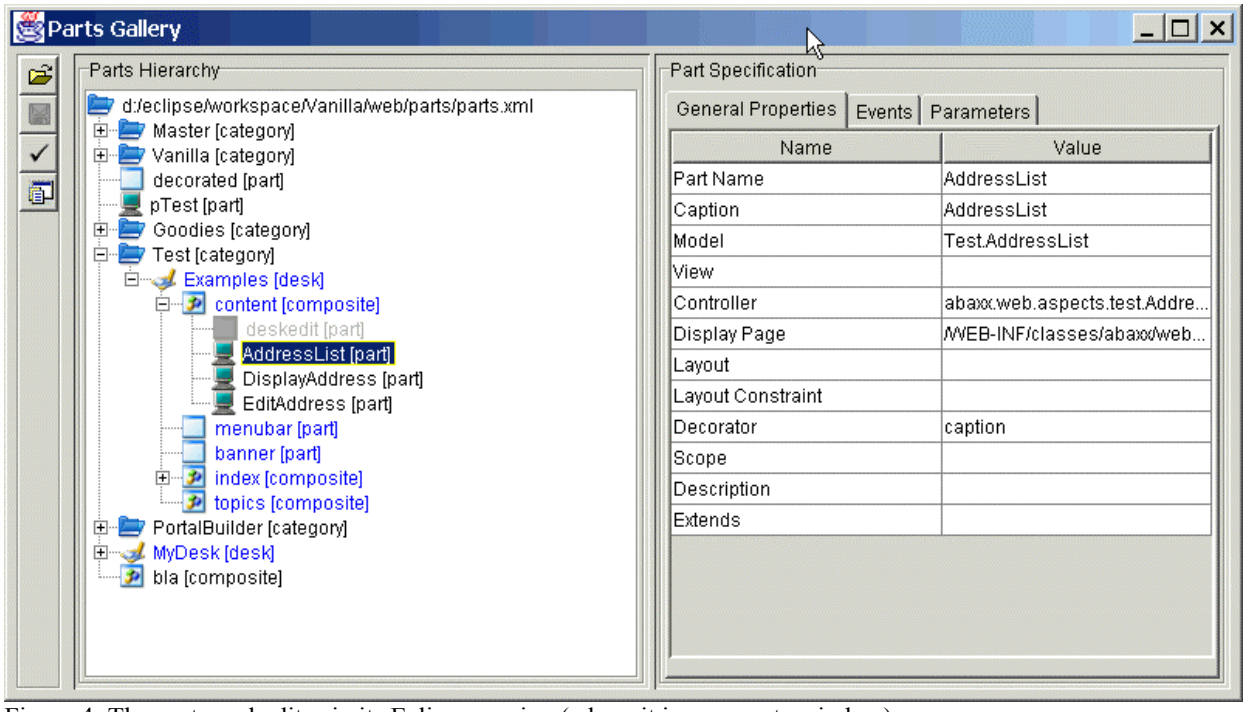


Figure 4: The parts.xml editor in its Eclipse version (where it is a separate window)

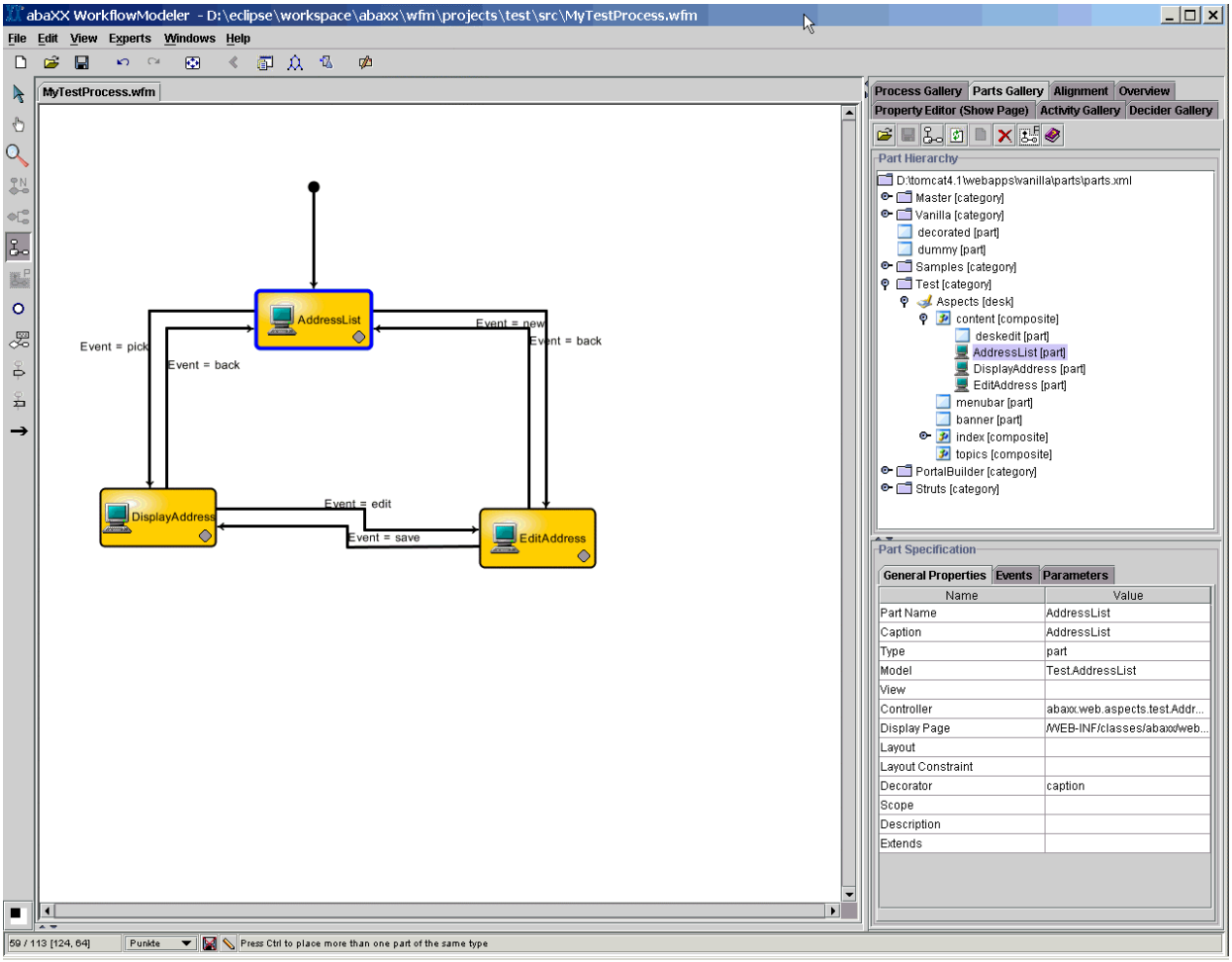


Figure 5: The parts.xml editor as a plugin to the abaXX Worflow Modeler.

```
<desk name="Examples">
        <theme>
                <styles>
                        .portal-content, .menubar a.selected { background-color: #ccccee; }
                        .portal-banner { color: #aaaaff; }
                </styles>
        </theme>
        <content layout="tabbed-switch" visual="abaxx.web.parts.CompositePart">
                <part name="deskedit" omit="true" />
                <part name="AddressList" controller="abaxx.web.aspects.test.AddressController"
                  model="Test.AddressList" url="/WEB-INF/classes/abaxx/web/aspects/test/AddressList.jsp"
                  decorator="caption">
                        <event name="pick" target="DisplayAddress" />
                        <event name="new" target="EditAddress&gt;create" type="redirect" />
                        <event name="delete" />
                        <event name="reset" />
                </part>
                <part name="DisplayAddress" controller="abaxx.web.aspects.test.AddressController"
                  model="Test.Address" url="/WEB-INF/classes/abaxx/web/aspects/test/DisplayAddress.jsp"
                  decorator="caption">
                        <event name="edit" target="EditAddress" />
                        <event name="back" target="AddressList" flags="populate,validate" />
                </part>
        [...]
```

Figure 6: Excerpt from the parts.xml file

Hosted Services for Advanced V&V Technologies: An Approach to Achieving Adoption without the Woes of Usage[1]

Position Paper for ACSE 2003

Lawrence Z. Markosian
QSS Group, Inc.
lzmarkosian@email.arc.nasa.gov

Owen O'Malley
QSS Group, Inc.
owen@email.arc.nasa.gov

John Penix
NASA Ames Research Center
John.J.Penix@nasa.gov

William A. Brew

Abstract

Attempts to achieve widespread use of software verification tools have been notably unsuccessful. Even "straightforward", classic, and potentially effective verification tools such as lint-like tools face limits on their acceptance. These limits are imposed by the expertise required for applying the tools and interpreting the results, the high false positive rate of many verification tools, and the need to integrate the tools into development environments. The barriers are even greater for more complex advanced technologies such as model checking.

Web-hosted services for advanced verification technologies may mitigate these problems by centralizing tool expertise.

The possible benefits of this approach include eliminating the need for software developer expertise in tool application and results filtering, and improving integration with other development tools.

1. Introduction

Software engineering tool developers face numerous obstacles in getting their tools adopted. Some of these obstacles are listed in the Call for Papers for this Workshop. Others include the cost of the infrastructure for maintaining and applying the tools, and the difficulty of interpreting and filtering the results. Integration with other tools pose additional barriers. The approaches suggested in the CFP represent ways to address these problems.

Our focus in this paper is on adoption of verification tools, specifically, program analysis and simulation tools such as static analyzers and model checkers. These tools are likely to require significant expertise in their use, for reasons that we discuss in the next section. In addition, these tools generally require a greater effort to integrate than "front end" tools such as design tools and compilers. Therefore, for verification tools, a more radical approach may be needed to ensure adoption.

[1] The research on model checking described in this report was performed at NASA Ames Research Center's Automated Software Engineering group and is funded by NASA's Engineering for Complex Systems program. The experience reported regarding other technologies and tools is based on the authors' professional experience developing and applying them in a variety of organizations.

Hosted application service providers may provide an effective way, in appropriate markets, to dramatically lower these acceptance barriers.

2. The problem

The Intelligent Software Engineering Tools team at NASA Ames Research Center (ARC) develops advanced verification tools based on source code model checking technology[1], an ongoing research area in the Automated Software Engineering group at ARC. Our target languages are Java, C and C++. This position paper is based in part on our current work and in part on our previous experience at NASA and elsewhere building or applying a variety of commercial verification tools. These tools include PolySpace[2], Flexelint[3], and Y2K defect detection/ remediation tools as well as research prototypes such as Java PathFinder[1] and ESC/Java[4]. Effective use of many of these tools faces similar problems.

In our experience, specialized expertise is required for effective use and adoption of verification tools. Integration issues also impede adoption.

2.1 Kinds of knowledge required for effective use of verification tools

Effective use of verification tools includes detection of real defects; a low false-positive rate; the ability to triage reported defects; and a high confidence level in the results. Effective use of model checking tools requires a mental model of their operation to interpret the output, tune the model checker's operation, and identify the root cause of defects that it reports. This knowledge is largely application-independent.

In addition to a mental model of tool operation, effective use for our target languages, C, C++ and Java, may require expert knowledge of the semantics of language operators in order to evaluate a defect report. In our experience, C/C++ programmers may not have the level of understanding of language semantics necessary to interpret tool output. As is the case with tool expertise, this knowledge is application-independent.

Effective use of verification tools may also require application-specific knowledge. Static analyzers may be unable to conclude that an operation may produce an exception, because the range of possible values of variables cannot be derived from the source code. This knowledge is often provided in the form of formal specifications or design information.

All of these considerations are prior to root cause analysis, which imposes further demands on the user, if the defect reports are to be actionable. Once the defect is well-understood, confirmed as real, with high confidence, then application-specific knowledge *may* be important in the remediation task.

Our experience with defect detection tools based on static analysis suggests that some of these problem exist even for lint-like tools, which have been available for 20+ years. These tools have such a high false positive rate that programmers are reluctant to apply them: they cannot filter the output efficiently, nor are they motivated to spend time on what they view as "busy work".

2.2 Integration of verification tools

Effective use of verification tools also requires that the tools be well-integrated into the development environment. Static analysis tools can, in principle, be run at compile or build time. Thus they hold forth the promise of early defect detection if they are well integrated into the development process.

Verification tools generally require a greater effort to integrate than "front end" tools such as design tools and compilers, since the verification tools must report defects in a way that supports in-the-loop evaluation and remediation or other action.

Tools that have a high false positive rate, if they are to be integrated at all, require a well-defined filtering process —some combination of automated post-processing and human filtering. Our experience is that developing an efficient filtering process requires extensive experience with the tool and its use in a particular development environment; the distillation of this experience must be retained as enterprise knowledge in a training system because of the high turnover rate of reviewer personnel.

Advanced verification tools themselves are likely to be "niche market" tools, since their range of applicability is limited, and hence the resources available for integration and maintenance will be limited compared to, for example, the resources available for integrating and maintaining a new compiler.

3. Hosted Verification Services

We have argued that effective use of verification tools requires three kinds of knowledge: a mental model of the tool's operation; knowledge of the semantics of the target language; and application-specific knowledge. To the degree that the first two kinds of knowledge dominate, it makes sense to centralize that expertise and even to hide it from users. One way to do this is to provide web-hosted verification services.

Usage scenario. In one scenario for hosted verification services, a developer checks her successfully-compiled source files into the host server's configuration management system (CMS). The nightly build is run on the development team's network, which accesses the files

from the host's CMS. The build transcript is written to the CMS server. Following the build, a configuration analysis tool, which is resident on a server at the hosting service, analyzes the build transcript to determine what files need to be analyzed and how (what compiler and compilation options were used, etc.) Some of the options specific to the verifier have been preset by the service provider's tool experts based on prior customer input about the application or about the current build. These options may include decisions about what defects are of interest to the customer, how much explanation of the defects is to be provided, and the highest priority modules for verification.

The verification tools use the configuration analysis data to initialize the verification options. Then they proceed to analyze the application. Tool operation is monitored by the service provider and human interventions are made as necessary—for example, to make tradeoffs between runtime and completeness of the analysis, or to focus on specific execution paths. The verification tools may need to be run repeatedly, with different settings, to obtain the desired results. Verification output is then filtered by an automated filtering system and may be presented to human reviewers for final filtering. The human-filtered output is directed to other facilities that are provided at the hosting site, such as an issue tracking system and test case generator. Ideally the data are available to the user when she logs on in the morning.

Commercial models. The model for this scenario is not far from existing commercial application service providers (ASPs) of software development tools and services.

For example, DevX and Merant provide hosted issue management tools and services. VA Software and Collabnet provide tools and services for hosted configuration management and collaborative development. SoftGear focuses on testing. Other organizations provide source code inspection services. The degree of user access to the "tools", and what capabilities are provided by ASPs, varies. In some cases the user directly accesses a tool (such as a configuration management system) using a browser interface, and the services include maintenance of the tools and the platform. In other cases the user may simply make submissions (for example, an application to be tested) and later accesses a database for the results—the ASP provides a service based on a combination of tools and human expertise.

Lessons from Y2K verification. Our experience with the Y2K problem suggests the effectiveness of a service-based approach to verification. Solving the Y2K problem for Cobol required extensive program analysis, as well as the ability to understand specific Y2K errors and remediate them systematically. This was beyond the capability of most Cobol programmers, particularly given the time constraints and the massive volume of source code to be examined. Advanced program analysis tools based on alias analysis and program slicing largely automated the analysis, but required a good mental model on the part of the user in order to tune their operation and interpret the results. Our experience was that training Cobol programmers in the required concepts, such as parsing, alias analysis, reaching definitions, evidence and confidence levels, built-in heuristics, and remediation strategies was broadly ineffective. Intensive process and tool training in a "factory" enabled a high throughput whether the goal was independent verification or error detection and remediation.

Application-specific knowledge was largely unnecessary[2]: the most important consideration was obtaining a complete, consistent set of source and copy books (include files).

4. Success and Risk Factors

We are not advocating hosted verification as a panacea. We have already indicated the basic precondition for hosted verification services—dominance of the importance of tool knowledge over application knowledge, or the superior ability of the tool to acquire application knowledge. Specific additional factors within NASA enhance the prospects for hosted verification services there. There are risks also, both within NASA and in a broader context.

Intellectual property & security issues. Commercially, intellectual property issues coupled with security concerns produce a reluctance in some markets to allow source code offsite. This consideration is largely absent within NASA; there are security issues (such as ITAR) but these are addressable through existing procedures. And, as we discuss below, NASA already has an internal software verification facility.

Need for verification. In addition, there is a recognition within NASA that software complexity, particularly for autonomous vehicles, is increasing rapidly, and that advanced software V&V are enabling technologies for autonomous space exploration. NASA ARC has for years been conducting research in software V&V as well as other approaches for reducing defects in autonomous applications. ARC does research and some tool development; it does not provide verification services.

However, NASA does have an organization dedicated to V&V, the Independent Verification and Validation

[2] More accurately, Y2K defect analysis and remediation required *extensive* application knowledge, but this *never* favored the application expert: the toolset *became* the application expert in the course of analyzing the application.

Facility. Its charter includes "identifying system and software risks to improve software quality and safety". We view the IV&V Facility as is a possible natural entry point for hosting verification services within NASA.

Support for software development process improvement. There is also an increasing recognition within NASA that detailed data and metrics should be collected on high-assurance software engineering projects. A hosted development environment, with specialized V&V tools, provides a platform for obtaining such data and evaluating the effectiveness of the development process, including the individual V&V tools. It offers the possibility of obtaining fine-grained enough data on individual V&V tools that NASA can model the return on investment of each tool when used at various points in the development lifecycle and feed this data into risk assessment tools such as [6].

Integration with other tools. The same environment that hosts V&V services should also host other development tools—at the very least, CMS and issue tracking tools. This does not completely address the integration issues mentioned earlier, since builds and testing, for example, are usually not conducted in the hosting environment, and there is a wide range of development environments. However, hosting CMS and issue tracking may provide a synergistic environment—the CMS can provide a complete, consistent configuration for verification; and verification output can be directed to the issue tracking system and possibly the test environment. An integration risk is that there is a large range of development environments, and it may be difficult to integrate closely with the build process.

Turn-around time. Integration into the development environment also suggests that the verification results are available quickly enough to be actionable before the next build. Human intervention in the verification process, which we have argued is necessary to apply verification tools and filter the results, may preclude a rapid enough response—for example, when builds are done on a daily basis. One strategy for mitigating this risk is to provide several levels of verification, where the "deeper" (and more time-consuming) verification levels are reserved for high-assurance applications that are more tolerant of turn-around time. Another strategy is to optimize the verification insertion points in the customer's development cycle—for example, with respect to milestones such as start of integration testing, certification, alpha and beta release, etc. Incremental verification should also improve turnaround time.

One of our research goals is to determine the right lifecycle insertion points for verification tools, especially when several verification and validation tools are used together.

Incremental verification. A related risk is inability to support "incremental" verification on successive builds. The human effort required for verification should be approximately proportional to the amount of "new" code. Certainly this is the customer's perception. A combination of engineering and research may be needed to address this for large applications with frequent builds.

5. Conclusion

We need to understand how the verification tools we are developing at ARC can overcome barriers to deployment within NASA, and how they are best integrated into the development process.

Web-hosted verification services may provide an opportunity for verification technology to gain acceptance in NASA and elsewhere. One of the barriers to the use of verification tools is the expertise required to apply them effectively, which dominates the application expertise required on the part of the user.

Benefits of web-hosted verification services may also include better integration into the development lifecycle; better integration with other software development tools; and the ability to obtain fine-grained performance data for evaluating the effectiveness of particular tools in various contexts.

Risks include difficulty of providing application-specific knowledge to assist the tool; inability of the hosting site to model the application development site; inadequate turn-around time; and the inability to support incremental verification of large applications. In the commercial environment, intellectual property and security concerns may limit acceptance.

References

[1] W. Visser, K. Havelund, G. Brat, S. Park. "Model Checking Programs", *Proceedings of the 15th International Conference on Automated Software Engineering (ASE)*, Grenoble, France, September 2000.

[2] PolySpace is a trademark of PolySpace, Inc. http://www.polyspace.com

[3] Flexelint is a trademark of Gimpel Software, Inc. http://www.gimpel.com/

[4] http://research.compaq.com/SRC/esc/

[5] CodeWizard is a trademark of Parasoft, Inc. http://www.parasoft.com

[6] Raffo, D., and Kellner, M. I., "Predicting the Impact of Potential Process Changes: A Quantitative Approach to Process Modeling," *Elements of Software Process Assessment and Improvement*, IEEE Computer Society Press, 1999

[7] Cousot, P. "Abstract Interpretation: Achievements and Perspectives." http://www.polyspace.com/docs/Abstract_Interpretation_P_Cousot.pdf

An experiment in facilitating adoption of an integrated software development environment

Lech Krzanik and Mikko Nurmi
University of Oulu, Department of Information Processing Science, Oulu, Finland
Lech.Krzanik@oulu.fi

Abstract

An integrated software development environment is considered that supports processes of feasibility analysis, requirements engineering, and design, for selected application domains such as user interface development for consumer products. We improve adoption of the environment in three ways. First, we improve the environment's user interface by adding common COTS products such as office suites or imaging packages using common middleware. Next we provide feedback to users' decisions which is referring to past experience on similar projects. Finally, we simplify the underlying software models by assuming that the target software is developed incrementally with frequent delivery steps - so that simple local approximations are possible. The environment's user interface is simpler and more intuitive, and understandability of the produced artifacts improved. The tools can use straightforward software representations with commonsense operations on them. To assure that such a setting is valid and to minimize wrong developer decisions caused by oversimplified modes of interaction with the tools we additionally introduce mechanisms for monitoring validity of developers' decisions. A typical developer's reaction to the warning messages would be selecting an alternative requirement or design decision, or decreasing the incremental delivery step for the software process.

Keywords: Software tool extensions, COTS, tool adoption, end-user participative development, incremental delivery.[1]

1. Introduction

A new practice is adopted by a group (e.g., an organizational unit) or an individual when it is routinely used, e.g., for the business purpose. On the commitment curve [2] adoption and institutionalization is preceded by contact, awareness, understanding, and trial use. All these stages can be facilitated by a transparent, common-sense access to the practice implemented with simple software tool extensions to commonly used products. Such a solution can be shared by many stakeholders participating in the practice maturation - the entire process from initial idea to the widespread, self-optimized use. This is particularly important as most technology developers do not explicitly define their stakeholders, much less obtain their commitment. The approach is likely to support the facilitator factors of adoption [3], such as prior positive experience, incremental change, and clear need. On the other hand the approach is likely to neutralize inhibitors such as high cost, psychological hurdles, and extensive training.

An integrated software development environment (IDE) is considered that supports processes of feasibility analysis, requirements engineering, and design, for selected application domains such as user interface (UI) development for consumer products. We improve adoption of the environment in three ways. First, we improve the environment's user interface by adding common COTS products such as office suites or imaging packages using common middleware. Next we provide feedback to users' decisions which is referring to past experience on similar projects. Finally, we simplify the underlying software models by assuming that the target software is developed incrementally with frequent delivery steps - so that simple local approximations are possible. The environment's user interface is simpler and more intuitive, and understandability of the produced artifacts improved. The tools can use straightforward software representations with commonsense operations on them. To assure that such a setting is valid and to minimize wrong developer decisions caused by oversimplified modes of interaction with the tools we additionally introduce mechanisms for monitoring validity of developers' decisions. A typical developer's reaction to the warning messages would be selecting an alternative requirement or design decision, or decreasing the incremental delivery step for the software process.

[1] This is an extended and changed version of the paper presented at the Workshop on Adoption-Centric Software Engineering ACSE'2003.

The above research resulted from the following practical situation. A company A provided for company B the first version of a specialized IDE, oriented toward component-oriented embedded UI software engineering for consumer products. Introduction of the system in B took longer than expected and it didn't reach the expected level of use. It was then decided that in parallel with the work on *Version 2*, a *Release 1B* would be developed addressing specially the adoption issue from the technical side. Interviews were conducted with current and potential users, which resulted in a set of proposed solutions:

1. Address major important user groups first
2. Address major important practices and issues first:
   - Simplify the developer's tasks addressed: Convert the overall development process into a product line-type of development [5], and consider first the application engineering only that performs assembly from available components
   - Focus on application quality and cost
3. Introduce more intuitive user interaction with the tools
   - Intuitive interaction in nominal situations, including evaluating user decisions and explaining development state transitions
   - Guidance in exceptional situations, including evaluations of user decisions and explaining the reasons for entering an exception
4. Exploit common end-user COTS software with familiar user interfaces
5. Provide references to other uses of same software components. Make references to experience data about previous or other similar systems and components
6. Use analogies on how similar tasks are performed in non-software domains
   - Component assembly for various consumer product lines.

These recommendations were selectively integrated into the *Release 1B* development plan with the characteristics described above. *Release 1B* had schedule of several months, and resources negotiable and design agreed with *Version 2*. Results to be later refined within *Version 2*. To create adequate user feedback, this was an end-user delivery, not a prototype. The incremental technology used was mostly COTS software for faster and less expensive deployment. It included an office suite and smart virtual modeling COTS software as well as databases, EJB server, etc. components.

Generally *Release 1B* can be characterized as follows:

- End-user participative
- Incremental delivery of software
- Emphasizing the role of nonfunctional requirements
- Including developer decision validity monitoring based on various measures of tolerances, uncertainties, risks, etc., of requirements and technology (components)
- An opportunistic approach, not necessarily globally optimal
- It is estimated that the tool can provide valuable information to stakeholders and support their decisions in early requirements engineering stages.

For larger delivery steps the validity monitoring system may indicate uncertainty or risk evaluations at unacceptable level, and possibly return a warning of invalid results, or may indicate no solution at all. The transparent, common-sense management of nonfunctional attributes (properties) facilitates adequate user responses, e.g., regarding selection of an alternative requirements or design decision (e.g., another component), or decreasing the incremental delivery step.

2. An example tool

Multi-stakeholder distributed systems are characterized with personalized and time-dependent views of stakeholder requirements. In such systems validation becomes problematic because individual stakeholders tend to be only aware of their local operation. The proposed tool is intended to support multi-stakeholder-participative requirements engineering. End-user understandability of local goal statements is as important as the precision of global validation of the specifications.

The example tool extends a web browser and uses a number of common middleware technologies (for instance, COM to support user interface implementation and EJB on the server side with component repositories). It addresses configuring new versions and variants of an existing product for new stakeholders. The requirements process makes use of a conventional function-and-attribute approach. Various usability engineering techniques are used [6, 7]. It is assumed that the differences between the stakeholders' system requirements are small or the products are developed incrementally with small incremental steps. A procedure is introduced for controlling the extent of "delivery deltas" between the stakeholders or the successive deliveries. The build process is component-oriented. Reuse of previously applied components - locally or for other stakeholders and systems - is preferred. A dedicated

requirements support is used to produce a mapping between the requirements space, where the difference measure is defined, and the implementation space where the components are integrated. A sample screenshot from the tool is demonstrated in Figure 1.

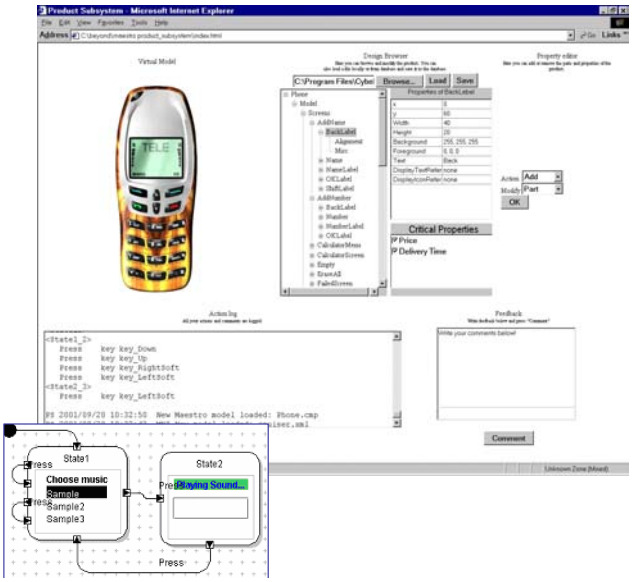


Figure 1. A property manager for incremental delivery of a multi-stakeholder system (handset skin graphics courtesy of Cybelius Software Oy).

The system allows for integrating software and non-software components according to the assumed property model. Where applicable, the visual models are functional, that is, they can react with predefined behaviors of user interaction. An advanced version of property management involves product-oriented decision support including product assessment, comparison, and delivery planning.

The approach is based on the following assumptions:

- New requirements specifications are based on existing, evaluated implementations for the same or different stakeholders[2]. Stakeholder feedback is an integral part of the evaluations. References to pre-existing implementations are the starting point for requirements validation. Successful validation requires that the difference between new specification and the referenced implementations (the delivery delta) is kept small. A requirement specification support tool controls that difference.
- Specifications are structured to facilitate goal setting, references to previous results, evaluations, and delivery delta planning and monitoring.

The method is generally suitable for:

- Medium (days to weeks) and longer scale (months and more) ephemeral requirements. The applicability depends on the component binding practice
- Incremental requirements engineering for evolutionary systems development
- Product family development (both domain and application engineering)
- Volatile requirements
- Technology changes.

The tool's decision process is outlined in Figure 2. The requirements specifications consist of behavioral, qualitative functions and nonfunctional quantitative attributes representing system qualities and resources. The attributes must be measurable, with a defined measurement unit, the test conditions, etc. A set of target values indicates goal preferences and boundaries. The functions and attributes include references to existing implementations to support requirements validation. If the attribute references do not use same measurement definitions they must provide for appropriate measurement conversions. They also have to include respective nominal and boundary values.

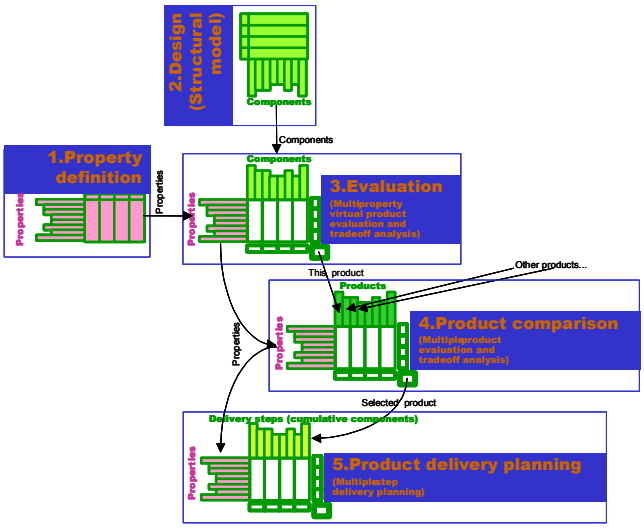


Figure 2. The tool's decision process.

[2] For creating the initial implementation a similar method may be used for incremental system development rather then creating a new stakeholder variant.

To justify the target attribute values, sets of benchmarks are provided based on comprehensive classes of references. Qualifiers such as stakeholder category, deployment time, location, authority, legal constraints, etc., provide for additional information. The above function-and-attribute approach to requirements and component specification is not new; it has been used in a number of recommendations regarding scenarios, architecture evaluation [1], etc. The innovative element of this work is the delivery delta control mechanism, which is used to assure validity of new specifications.

3. Validation of the approach (informal)

The users interviewed after *Release 1B* indicated that the solutions provided a critical necessary contribution to facilitating tools adoption. Two aspects were emphasized in particular: Using the experience data in decision validation and the familiar and user friendly interface utilizing an office suite, and the virtual modeling COTS software..

The results of *Release 1B* were then refined on the main project with expectation that the adoption of *Version 2*, incorporating *Release 1B,* would be improved considerably. At the time of writing this paper the final results were not yet known.

4. Problems and solutions

There are three classes of problems encountered in implementing the method:

- Delivery delta minimization
- Delivery re-composition
- Multiple stakeholder coordination.

In principle, the method as proposed is only valid if the delivery deltas are sufficiently small. Any major deviation from available implementations must be specially investigated, which entails introduction of costly prototype benchmarks. A delivery delta minimization procedure is used to assure that the deltas remain valid at acceptable cost. Various performance criteria may be used. In general the result is not globally optimal. In practice this corresponds to such deployment policies as “smallest useful deliverable” (“smallest” in terms of delivery deltas). Appropriate feasible sets of candidate alternatives are derived the interval specifications of targets and benchmarks for requirements and technology (components) with tolerances, uncertainties, and risks.

It is likely that the two last problems can only be efficiently solved if we have a satisfactory solution for the first one. Delivery re-composition determines the structure, functions and attributes of the new delivery, rebuilt from available components according to the changed specification with delivery deltas. The proposed approach is based on a set of predefined attribute aggregation models. The models are selected when providing attribute definitions. The approach is end-user participative, opportunistic, and not necessarily globally optimal. It is estimated that it can provide valuable information to stakeholders in early stages of requirements engineering. For larger deltas the user may get uncertainty evaluations at unacceptable level, and possibly a warning of invalid results, or no solution at all. Critically important are stakeholder-oriented representations of the delivery step control, re-composition, and stakeholder consensus search. Feature graphs are proposed with emphasis on user-friendly representation of attributes. There may be function- and attribute-oriented feature graphs. An automated conversion between the two is considered.

5. Conclusion

The paper outlines selected intermediate results of the work in progress. The final conclusions will be published separately [4]. The results confirm that in selected cases of simple software tool extensions it is possible to provide a satisfactory commonsense interface facilitating adoption of software engineering practices. We presented one such example, with these characteristics:

- End-user participative
- Incremental delivery of software
- Emphasizing the role of nonfunctional requirements
- Including developer decision validity monitoring based on various measures of tolerances, uncertainties, risks, etc., of requirements and technology (components)
- An opportunistic approach, not necessarily globally optimal
- It is estimated that the tool can provide valuable information to stakeholders and support their decisions in early requirements engineering stages.

6. Acknowledgments

This work was partly supported by the ITEA projects BEYOND and AMBIENCE.

References

[1] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice*. Addison-Wesley, 1997.

[2] Daryl Conner and Robert Patterson, Building commitment to organizational change, *Training and Development J.* Vol. 18, No. 30, April 1982.

[3] Everett M. Rogers, Diffusion of Innovations. Simon and Schuster, 1995.

[4] "Evolutionary Delivery of Configurations for Ubiquitous Applications", *AMBIENCE Report*. To appear, 2003.

[5] Paul Clements and Linda Northrop, *Software Product Lines. Practices and Principles*. Addison-Wesley, 2002

[6] Xristine Faulkner, *Usability Engineering*. Macmillan Press Ltd., 2000.

[7] Jakob Nielsen, *Usability Engineering*. AP Professional, 1994.

Tool Adoption Issues in a Very Large Software Company

Jean-Marie Favre Jacky Estublier Remy Sanlaville

Adele Team, Laboratoire LSR-IMAG
University of Grenoble, France
http://www-adele.imag.fr

Abstract

Tool adoption is a major issue in software engineering. In the last decades many ideas and tools have been developed by academics but only a few have had a direct impact on software industry. This paper describes the major issues in tool adoption and presents some technological approaches to cope with these issues. The focus is then on adoption-in-the-large. The results of a ten-years collaboration between the LSR laboratory and Dassault Systèmes are presented. Dassault Systèmes is one of the major software companies in Europe. Two scenarii in tool adoption are described. The first one describes the successful adoption of a configuration management tool, the second one describes adoption issues related with a reverse architecting tool.

1. Introduction

Industry often says “no thanks” to software engineering (SE) research, in particular when tools are proposed [1]. Technology maturation and adoption is known to be a very long process [2][3]. Nevertheless; it is still disappointing to see that while many SE tools are developed, their adoption in software industry is quite an exception. In fact, while researchers concentrate on designing and building *tools*, industry is looking for *solutions*. Even when a tool is very close to a solution, it is actually very hard to get this tool used in industry. The “last mile” [4] is a very difficult step in the technology transfer process. It is indeed a crucial one.

To cope with this problem, a new research trend called *adoption centric software engineering* (ACSE) aims to address the adoption issue in the first place. Successes and failures in software engineering adoption should be studied. Innovative ways to ease adoption must be found.

This paper describes a case study led over the last decade in a very large software company. It describes tool adoption successes and failures in the context of a collaboration between the LSR-IMAG research laboratory and the Dassault Systèmes (DS) company. DS is one of the largest software editors in Europe. This paper shows how the size of the company can lead to specific problems that are unlikely to occur in small or medium-size companies.

The paper is structured as following. Section 2 describes the main issues in tool adoption. Section 3 presents differents ways to cope with technical issues in tool integration. Section 4 shortly describes the main charactertics of Dassault Systèmes which constitutes the context of the case study. Section 5 describes a positive scenario in which a configuration management tool is adopted. Section 6 describes problems related to the adoption of an architecting tool. A discussion is provided in Section 7 and finally Section 8 concludes the paper.

2. Issues in software engineering tool adoption

Over the last decades, Software Engineering (SE) research has produced many methods and tools. To evaluate the actual impact of this body of research on software industry, a large study called IMPACT is being held at the international level. This study shows that the topic of tool adoption is quite complex.

Even when SE tools are quite close to solutions and are based on well-defined concepts, there are still important barriers to their actual adoption. Many factors should be taken into account :

- *Scalability issues*. Very often, a large number of unexpected problems are discovered when applying good tools at a large scale. This includes not only the size of the software but also the size of the company. While a tool could perfectly suit the needs of a single user, its use by hundreds of software developers may unveil new issues.
- *Usability issues*. Many software engineering tools focus on functionality, not on usability. However the user interface plays a very important role. Software engineers are under time pressure. They will not use a tool if they can’t get easily the result they need from the tool.
- *Tool integration issues*. It is unlikely for a tool to be adopted if it is not tightly integrated with the tools already in use. Data, control and interface integration are required.

- *Process integration issues.* A very good tool will not be adopted if it does not fit well in the development process of the company. This is especially true in companies with well defined and strict software processes.
- *Customization issues*. Large companies often define their own set of concepts and rules. Customization could be of paramount importance at the company level. A systematic way to use tools over the whole company should be enforced. In some situations customization is also a desired property at the end-user level.
- *Deployment issues*. In a very large company deploying a tool could be a real issue, especially since different teams may use different platforms, different tools, etc.
- *Administration issues*. Once installed, some complex tools require a great amount of administration. This could include tasks such as backups, error recovery, creation of new users, new projects, etc. The amount of work and the skills required could be a serious barrier to the adoption of complex tools.
- *Evolution and continuity issues*. Just like any other software, a SE tool has to evolve to meet the company evolving requirements. A company will not invest time and money in a tool if not convinced that the tool will be maintained and enhanced for years. Universities are hardly convincing on that aspect.
- *Training issues*. The introduction of any new tool implies a learning process. Except for very simple tools, learning new concepts and advanced features requires a full training program. This is a very serious issue in large companies since learning represents a temporary loss of productivity with uncertain return on investment.
- *Strategical issues*. At the level of the company, many factors could also hamper the adoption of a particular tool. This might occur for instance to avoid dependencies towards a particular organisation or tool vendor, licensing problems, or any other political issues.

Introducing a new tool in a company represents indeed a risk. In many cases, it is actually quite difficult to get effective managers and software engineers involvement.

3. Tool integration

Many issues listed above are intimately linked to the organisation and the strategy of the company. Researchers should avoid trying to address directly these issues because it is very unlikely for them to have the appropriated skills and knowledge. It is extremely hard for a researcher to have an impact on the company major decisions.

To have an actual impact, ACSE research should therefore concentrate on those parts that can be controlled in a research environment. It is therefore better to focus on technology rather than strategy.

3.1. Integration to existing tool sets

One way to ease the adoption of a tool is to integrate it into the set of tools already used in the company. Currently they are at least 4 main groups of tools used in almost any company producing software:

- *Web-based and communication tools*. Web browsers and mailers are everywhere. They are used by every actor in the company. They play a very important role since all communications rely on these tools.
- *Integrated Development Environments* (IDE). These tools are now widely accepted among the developer community. They are primary tools for most software engineers.
- *Office Tool Suite*. Software engineers and managers share in common the use of text editors like Word, FrameMaker or WordPerfect. They also use tools like PowerPoint. Documents are exchanged using standard formats such as HTML, Postscript, PDF, etc. Managers deal with metrics and other indicators using spreadsheets and bar chart generators.
- *Modelling Tools*. Tools supporting analysis and design are becoming increasingly popular. This is in part due to the success of the UML standard in software industry. There are now numerous UML workbenches such as Rose, Objecteering, Together, etc. These environments support not only the drawing of models, but they also integrate code generators, documentation generators, metrics, and many other tools. Modeling environments still have to be adopted at large but there is no doubt that there is an increasing interest in UML and its associated tools.

Users are more likely to adopt a tool that works in the same environment they use on a daily basis. This means that SE tools should be integrated to the existing set of tools.

3.2. Integration Levels

To be effective, different kinds of integration [5] should be supported between SE tools and existing tools:

- *Data integration*. Data consumed and produced by SE tools should be shared with other tools. For instance, the result of a metric or profiling tool should be easily exportable to a spreadsheet for further manipulations. It should also be very easy to insert an architectural view into an existing document, to publish it on the web or to send it via email for further annotations.

- *Control integration*. It should be possible to call a tool from another one. For instance, it should be possible to call the functions of a metric tool from an IDE, and to display the result through a visualisation tool.
- *User-interface integration*. All tools should be accessible from a consistent user-interface with a common look and feel. The popularity of today IDEs is in part due to the fact that many programming tasks can be performed easily through a unique user interface.

Work on tool integration is far from new. The different kinds of integration were identified in the 80's. Interoperability techniques such as RPC and Corba take their roots from then. In the 80's and 90's a large amount of effort was dedicated to Computer Assisted Software Engineering (CASE).

In particular, many research projects focused on syntax-directed environments. In such environments syntax-based descriptions played a central role in supporting both data integration and control integration. These environments have not been adopted widely, though many concepts and techniques they have been introduced, form the basis of modern IDEs. It is interesting to note that many techniques initially developed in the CASE field have been successfully applied to other domains. This includes for example office suites which present nowadays many of the desirable features of CASE tools such as mutliple sychnornized views . In fact, one of the mistakes in the CASE vision might have been to believe that the CASE market was large enough per se to support the development and adoption of a rather immature technology.

3.3. Approaches to SE tool integration

It appears now clearly that the market is driven by web technologies, office suites, and to a lesser extent by IDEs and UML modelling tools. New and innovative SE tools must be integrated into these existing suites. The other way around is very unlikely.

The move from Field [6] to Desert [7] is representative of this shift. Field is one of the precursors of CASE environments with its strong emphasis on control integration. Desert, its successor, seeks to integrate programming facilities into the FrameMaker text editor. More recently a few attempts have been made to use PowerPoint as a design tool. These approaches provide good illustrations of Adoption Centric Software Engineering.

Integrating new tools in a proprietary toolset could be far from obvious. Fortunately, a lot of improvements have been made over the last years, making the integration of SE tools possible. This results from efforts made both by tool vendors and by standardization bodies

- *Standard exchange formats*. Different de-facto standards are widely used to exchange documents such as RTF, MIF, HTML or PDF. The XML standard will clearly have a major impact on data integration. Specialized formats such as GXL [8] could be useful to improve interoperability between research tools dealing with graphs. Nevertheless, there is currently no indicator that it will be adopted by software industry.
- *Standard schemes*: Specifying a file format is not enough to ensure data integration. It is also necessary to specify the schema or the meta model used to represent data. Fortunately these concepts are well supported by current technologies such as XML and the OMG' Model Driven Architecture (MDA) [9][10]. For instance MDA provides XMI to represent and exchange UML models and meta models. It also includes some standard meta models for different application domains including the Software Process Engineering Metamodel (SPEM) and the Common Warehouse Metamodels (CWM).
- *Standard APIs.* Using standard exchange formats and standard schemes enables data integration but not control integration. One way to cope with this issue is the publication of APIs and in some case the standardization of the APIs. Most of the tools described in Section 3.1 provide APIs and include a "developer kit". The MDA approach is based on the MOF standard [11] which describes how to generate a standard API for each particular meta model.
- *Scripting languages*. Using APIs can be quite complex. Scripting languages and macros offer a much cheaper alternative for customization and automation of common tasks. Nowadays, almost all commercial environments include some sort of scripting capabilities, although most of the time the scripting languages they provide are proprietary. For instance office tool suites typically include extensions of the Basic language. UML workbenches like Rose or Objecteering include proprietary scripting languages that support the manipulation of UML models and the addition of new features.
- *Plugin and component technologies*. Most environments also support the concept of "plugin". which enables the addition of new features in predefined points of extension. More generally, a large amount of research is devoted to component technologies including for instance Microsoft' COM, Sun JavaBeans and Entreprise Java Beans, Corba CCM, Microsoft .NET, etc. While these technologies are not specifically oriented towards the development of SE tools, component technologies will certainly play an increasing role in the future. It is interesting to

notice that the COM component model was originally designed in the context of an office tool suite to enable the inclusion of "live documents" in other documents (e.g. the inclusion of a spreadsheet in a word processor document). These kind of technologies obviously present a strong interest to embed software engineering tools or views in other documents. With respect to the MDA approach, a new research trend tries to define the notion of MDA components.

- *Standard infrastructures*. While component technologies enable the integration and assembly of new tools, they does not ensure per se a strong consistency between the applications being built. For instance good user-interface integration is not possible without some standard rules. Fortunately, the emergence of very open infrastructures such as Eclipse [13] can cope with these issues.

As pointed out before, tool integration has been a primary objective of CASE research. In the 80's and 90's the problem was to integrate together SE tools like editors, debuggers, profilers, etc. This objective has been met, but mostly in situations in which the tools are built by a single tool vendor, or by a small set of tool vendors with very close partnerships.

In the context of ACSE, the problem has changed: it seems now necessary to be able to integrate an arbitrary SE tool to the suites already used in a given company. Fortunately, the generalization of the technologies described above leads to new opportunities. In fact, for each set of tools described in Section 3.1, (web-based tools, office tools, IDEs and modeling tools), efforts have been undertaken to improve their openness. Web-based and office tools are increasingly based on XML technologies, Eclipse could have a strong impact on IDEs, MDA could help the interoperability between modelling tools. One important approach in ACSE is then to use these emerging standards in order to facilitate the adoption of SE tools.

3.4. Summary

There are many barriers to tool adoption and the "last mile" is always a difficult step. Some issues are due to the organisation and strategy of the company itself and should not be addressed directly by researchers. Others approaches are related to the technology used. A major trend in ACSE is to use industrial standards as a basis for the development of innovative SE tools. Though promising this approach will not solve all the problems. The following case study shows that SE tool adoption is a real challenge, especially in large software companies in which there is a real shift from adoption-in-the-small to adoption-in-the-large.

4. Case study in a large software company

In the 80's a Software Configuration Management (SCM) tool called Adele was developed by our team at the University of Grenoble. This tool was very generic and it included many innovative ideas. Adele was adopted by different companies including Matra and Sextant. The expertise gained with industrial partners has resulted in a long and thight collaboration between the LSR-IMAG laboratory and Dassault Systèmes [14][16]. While the first part of the collaboration has been dedicated to configuration managment, the second part has been devoted to software architecture. The rest of this paper uses the whole collaboration as a case study to summarize our experience in SE tool adoption over the last decade.

4.1. The Dassault Systèmes (DS) company

Dassault Systèmes (DS) is one of the largest software editors in Europe. DS is also the world leader on CAD/ CAM with more than 19 000 clients and 180 000 seats. DS constitutes a very interesting context for a case study because of the size of the company, and the architecture of its software.

The company is indeed very large: 1000 engineers are working simultaneously on the same software product. CATIA is one of the main software products with more than 5 MLOC. The requirements on CATIA software architecture are also very strong, especially since many customers around the world contribute to the development of the CATIA product line. CATIA is sold to companies that have an important know-how in their respective domains. Boeing, for example, is a specialist in plane construction and owns many software tools and rules. DS customers must be capable of adapting CATIA, integrating their own functions into existing DS applications. These extensions constitute a significant part of the software. Boeing alone is said to have developed more lines for CATIA adaptation and extension than DS for CATIA itself.

Actually DS and its partners constitute a large virtual software factory in which thousands of software engineers collaborate to the development and the evolution of a very complex software product line. From the software engineering perspective this implies very strong needs both in configuration management and in software architecture.

The collaboration between the Adele team and DS has been centred around these two themes. Section 5 describes the process leading to the adoption of the Adele SCM tool, while Section 6 describes the difficulties we met in deploying the OMVT, architecting tool.

But let us first review how tools are usually introduced in DS since it may be representative of a typical organisation for adoption-in-the-large.

4.2. Tooling support in large software companies

In small companies tool adoption issues are mostly related to individual software engineers adopting individual tools. This could be refered as adoption-in-the-small. This contrast with tool adoption-in-the-large that takes place in very large companies and that requires a much more complex organisation. This are various reasons for that:

- The collaboration between hundreds of software engineers is possible only if the company has a rather well defined process, the tools playing an important part in that process.
- Scalability issues could be so huge that only few tools in the market, if any, fit the company needs. Tool evaluation is indeed a very important issue.
- Large companies often have specific needs related with their process and their culture. Customizing existing tools could be a complex yet essential task. Developing new tools is sometimes necessary.
- Many issues that can be solved rather easily in the context of small companies, can lead to very complex problems in large companies. This includes for instance deployment on hundreds of machines, learning programs over thousands of developers, administration of hundreds of projects and thousands of user accounts, etc.

Obviously programmers must not be in charge of managing SE tools; they must concentrate on their jobs, that is developing software. In large software company this usually leads to the existence of a *Tool Support Team* (TST). The TST is in charge of all activities related with tool support including tool evaluation, customization, integration, deployment, administration, learning, support, etc. Most of the time, this team is in charge to evaluate commercial tools. In some occasions some TSTs develop tools internally.

In such a context tool adoption is not only restricted to end-user adoption (that is adoption by software engineers). A tool will be adopted only if it meets the needs of three kind of actors:

- *Managers*. They define the strategy of the company at various levels. Without their agreement a tool will not be included in the company toolset. Researchers have to convince them of the actual benefits that the tool is supposed to bring. And this should not be in technical terms but it terms of actual benefits.
- *TST members*. They deal with all technical aspects of tools. Most of the time, failures in tool adoption will happen at that level, because this team is in charge of evaluating the tool. Scalability issues, deployment issues, integration issues, etc. will be discovered here. Researchers must collaborate closely with the TST during the tool adoption phase.
- *Software engineers*. Ultimately software engineers are those who use the tool. Some usability issue could appear at this level because software engineers may have different habits in performing their development activities. The pressure on them to use a given tool could be important or not. Typically a company will oblige all software engineers to use a critical tool such as a configuration management tool. It will be much less strict on second-class tool like a browser for instance.

This organisation makes tool adoption even more difficult. Researchers should face tool adoption by managers, TSTs and end-users. They should cooperate therefore with many different actors in the company, and this at different point in times. This could be quite difficult, especially since researchers are usually neither aware of the precise organisation of the company not of the exact role of each person they meet. In each situation, the discourse should be adapted to the concerns of the interlocutor.

The situation is even more complex in an organisation such as Dassault Systèmes, because of the various partners constituting the virtual software factory. A tool used in the company, could be later included in the development kit delivered with the product sold. For instance, the Adele tool, after being adopted by DS, was include in the CATIA toolset and delivered to customers such as Boeing. The next sections describe two scenarii of tool adoption in the context of Dassault Système.

5. A successful story in configuration management

All adoptions of the Adele SCM tool [15] started at the initiative of the company. This was also the case of DS. This company really needed a configuration management system to manage the parallel developement of CATIA by hundreds of software engineers. Actually, before the first contact with our team, DS first asked other users of Adele what they thought of the product and the support. They checked many other informations, technical ones but also, if not essentially, non technical ones. Such precautions are natural and common. An SCM tool is a critical tool, involving a large training, with deep influence on the software process, and even in the software structure. Any failure of the tool can have dramatic consequences for the company, ranging from few hours of unavailability, to loss of sources, or delivery of inconsistent products. The investment is heavy, the choice risky. Evolution and continuity of the tool are of paramount importance in this context. Any large company needs to be convinced the product will live for long. It was improbable a tiny academic team could satisfy these requirements.

DS took the risk to rely on a research team. Actually, this is mainly because they had to. On one side, their analysis showed Adele was the only system capable to satisfy their requirements, mostly because of its flexibility and its capability to adapt to their very unusual process and characteristics. Solving the customization issue was indeed considered as one of the major issues. On the other side DS had to select an SCM tool. The evaluation of existing tools showed that commercial tools were not well-suited to fit their uncommon size and requirements. During the evaluation period DS crashed almost all evaluated tools. Scalability issues in existing commercial tools was therefore another argument to invest on Adele.

Having a tool they could tailor to their needs, and having a team capable of making it evolve in their direction was a very strong point. It is surprising to consider that we were the only ones (apart from the other tool vendors) who tried to discourage them to use Adele, arguing that it was not designed for such a huge software and large software team.

Indeed, the first full scale test of the tool was almost a disaster; efficiency being well below requirements. Scalability was also a major concern. In this first phase the tool was also clearly rejected by software engineers because they were feeling that the tool was not helping them in their work. Usability was also a major issue. A number of problems had to be fixed in a panic mode, because the tool was already used at large. Then, major parts of the tool were redesigned and reimplemented to cope with scalability and usability issues. This was done quite quickly and efficiently. This ability to make the tool evolve was probably the main factor that convinced the managers to continue the experience. The product was under control and the relation between the TST and the research team was good.

The situation thus improved, to reach an acceptable level in which the acceptance by end-users was relatively good. In parallel, different customizations were experimented by the TST. The tool proved to be flexible enough to provide solutions for all new requirements. One of the main features of Adele is that it included an event-based system that enabled to attach a trigger and a reaction to any arbitrary event raised during the software process. The final version of the parametrized system was used for years. It included about 10 000 triggers describing the complete development process of the whole company. It is probably the world's largest trigger-based industrial application.

A new version of the system was later developed from scratch by the TST. The goal was to retain the exact same features, but with an order of magnitude in size of software to be supported, and in efficiency. Of course that new version has no flexibility nor genericity at all, but it meets all other requirements.

6. A missed opportunity in software architecture

The collaboration continued after this successful story in configuration management. In the mid 90's, Dassault Systèmes decided to move its products from Fortran to an object oriented design with C++. This huge redevelopment effort succeeded at the beginning of 1999 with a commercial release of CATIA V5. CATIA is made of appoximately 50 000 C++ classes and more than 8 000 components.

In order to develop this complex product line, DS decided to create its own framework. The OM component model is a fundamental part of this infrastructure. This component model is similar to Microsoft's COM but includes more powerful constructs.

Just like other component technologies, the OM is quite difficult to understand and to teach. Using the OM requires experienced and skillful engineers. It introduces new conceptual entities programmers are unfamiliar with (e.g. OM components, bases, extensions, delegations, etc.). Moreover these concepts are described using low-level mechanisms like C++ entities, naming conventions, macros in source code, specific implementation patterns, etc. An additional issue is due to the lack of centralized description for these concepts. Due to concurrent development within the virtual software factory, information about a single OM entity is often spread among a large set of different files, that may come from different companies. Though the introduction of the OM component technology was a major step towards the definition of the CATIA product line, managing this complexity quickly became a difficult task.

The collaboration between Adele-DS concentrated therefore on software architecture. The initial goal of was to review which recent advances in software architecture were applicable in the context of DS. A survey of existing ADLs was conducted to see if one of them could be adopted to describe the architecture of CATIA. None of the existing ADLs appeared to be satisfactory [20].

First of all, there were serious customization issues. Some concepts like the OM inheritance between OM components could not be described in the existing ADLs. Handling all the specificity of the OM was an essential requirement. In fact, most ADLs introduce high level concepts such as connectors, but this concept for instance was not perceived as being useful within the company.

It also appeared that the primary objectives of many ADLs did not fit the requirements of DS. Many ADLs focus on the verification or simulation of interaction protocols between components. Though this is an important topic this approach does no scale up and describing the behaviour of all components was not considered as feasible.

Finally ADLs are tailored to fit in a forward engineering process, but they usually do not take into account existing software. It makes no sense in a large company to maintain an architectural description if the link to source code is not maintained. In fact DS, just like almost every other company, is essentialy code-centric. Moving the an architecture-centric approach was not even considered since it would require a major risk with an absolutely unclear return on investment.

To cope with this problem, we defined an architectural notation for the OM concepts, but took a reverse engineering approach. The architecture had to be extracted from the code, not the other way around. This process had to be fairly automatic to show immediate benefits. We developed a specific tool called OMVT [22] to explore CATIA at the architectural level. This tool offers different architectural views and analysis features with a custom graphical syntax. A special emphasis was put on the scalability and usability of this tool, providing for instance a clean and easy to use interface.

Various demonstrations were performed to show the tool to different actors in the company. DS software engineers were very positive: for the first time they were “seeing” the CATIA architecture. Comments were very encouraging: "it is very promising for a controlled definition of components"; "it provides both global and detailed views … not available today without cumbersome browsing of many files …" [21]. Although the OMVT tool is close to DS' requirements and has positive assessments, it is still not adopted in the company. The "last mile" is indeed a difficult step.

7. Discussion

Two different scenarii have been described: the successful adoption of Adele configuration management tool, and the issues in deploying the OMVT in the company. Actually, a closer look on these examples reveals that there are at least two main categories of tools: *critical tools* and *non-critical tools*. This characteristic obviously has an impact on its adoption by the company.

7.1. Adoption of critical tools

SCM tools are critical for large companies like DS. Adele was adopted because solving SCM issues was urgently needed and because no other tools with similar features were available on the market. In fact the adoption process was quite long. The TST ultimately implemented a specialized version of the tool from scratch.

The size at DS clearly plays a very important role. Almost all tools, including popular and effective commercial tools, suffer from serious issues when applied at large at DS. DS has to tailor existing tools to their needs. This often implies special partnerships with tool vendors. These partnerships are fundamental in the adoption of a given tool.

In fact, if a tool is critical for a company, the company will deploy considerable energy in getting the tool and customizing it. If the company is large enough, developing an in-house tool is sometimes considered. One of the main qualities of Adele were its genericity and flexibility. They enabled DS to define and implement their own developement process. The integration with existing tools in the company was not considered a primary issue because the TST could handle these problems. In fact, existing tools were integrated into the ADELE environment, which reflects the fact that SCM tools are always critical ones in large companies.

7.2. Adoption of non critical tools

Tools that do not have a direct and immediate impact on source code are usually not considered as critical ones. They are plenty of ways to avoid using a tool in a company.

Reverse engineering and architectural tools might fall in the “non critical tool” category (except when a major reengineering effort is needed, but this is not common).

It should be recognized for instance that, CATIA developement and evolution is very successful even though DS does not have a clear vision of the full architecture of CATIA, at least in the ADL sense of the term. Communication among highly skilled teams reveals itself to be very effective in practice. Though the OMVT could represent a helpful tool, until now it has not been adopted.

Although it is quite difficult to determine which are the most important barriers to adoption of OMVT, it is clear that it is not currently a priority in the strategy of the company. Let us review however which technical issues should have received more attention in an ACSE perspective:

- *Deployment issues*. Deploying a tool at DS is clearly complex. Though the OMVT tool is quite simple to install on one computer, more attention should have been dedicated to automatic deployment. Another alternative would have been to integrate the OMVT in the intranet of the company since this solution would not require any kind of installation on client machines. This kind of solution is widely used at DS for other tools. The intranet is quite rich in terms of SE data.
- *Integration issues*. Communication is of paramount importance at DS. Providing architectural views of CATIA as live documents would have also greatly helped in the diffusion of the architectural notation and the corresponding concepts. An architectural view that cannot be annotated and shared among various

software engineers is not really useful in practice. Similarly an important aspect would have been to integrate the OMVT as a plugin in the DS toolset.

- *Customization issues*. Demonstrations and interviews among various actors of the company revealed very different needs. Many enhancements and customizations were required. We implemented 22 view points of software architecture, but though the tool is based on a clean object-oriented framework, adding and customizing new features still required significant developement efforts. The ease of customization would have been a strong point, especially since just like in the case of Adele, the needs of the company are not clearly identified.
- *Evolution and continuity issues*. The evolution of CATIA V5 infrastructure is continuous. To continue to be useful, the OMVT should have been capable of evolving accordingly. In fact new architectural concepts are sometimes introduced while some others are removed. For instance, some interviews revealed a strong interest in taking into account the concepts of the "Feature Modeller," that is additional layer built later on top of the OM.

Actually, one of the major problem we have faced is that we haven't been able to achieve major TST involvement. No TST engineer has been assigned to the deployment, administration, support and evolution of the OMVT. Such involvement is a prerequisite for the adoption of a tool. Once again, since such kind of tools is not considered as being critical, their use is not considered as a top priority.

8. Conclusion

Adoption Centric Software Engineering addresses a very important issue in software engineering: how to cope with the fact that research produces many tools but that, most of the time, these tools are not adopted by the software industry. A large list of issues in tool adoption can be established. Some barriers are related to the organization and strategy of the company. Others are related to technological aspects.

Researchers should not expect to change the way companies are making business. Research provides tools while industry is looking for solutions to support their strategies. This constitutes indeed a very big gap.

Research should rather concentrate on concrete and technological aspects, especially since an actual impact is possible there. Tool integration is an important issue. It is however not clear if it always constitutes the most important barrier to tool adoption. Much ACSE research focuses on this issue and relies on the use of industry standards to improve the chance of adoption. This is a promising way, but this approach should not be considered in isolation.

In fact, the case study described in this paper reveals that the size of a company plays an important role in tool adoption. Adoption-in-the-small and adoption-in-the-large are quite different. End user, that is software engineers, are the natural targets in small companies. They will decide if the tool is worth it or not. In a large company, the tool adoption process is much more different. At least three different parties have to be convinced: the managers, the tool support team, and the software engineers. Even if the software engineers would not adopt naturally a tool, tool adoption could be decided at the managing level and implemented by the tool support team. The distinction between critical tools and non critical ones is important. If a tool is considered as critical, a large company will spend a vast amount of energy in deploying it in the whole company. In a company like DS, it is illusory to believe that a research prototype will be adopted as is. A close collaboration with the tool support team is a key to success.

Our experience also suggests that to be adopted in a large company a research tool should be generic and should support a high level of customization, at least in the initial phase. Actually a rewarding collaboration schema with a tool support team would be to help them to define their own requirements by using a generic tool. If the company is large enough and the results of the experiments are convincing enough, the company might implement later a specific tool using a full engineering power. Industrial strength tools and research tools do not play in the same category.

Our current research includes the design of G^{SEE} [23][24], a *G*eneric *S*oftware *E*xploration *E*nvironment. This is a meta-model-driven environment. It enables a very fast integration of various kind of data sources and tools as well as the interactive definition of new architectural views.

Our work in component-based software engineering is also applied in this context. One of our goals is to build a component-based framework that enable the TST to build interactively they own environements by assembling and customizing very generic components. We do not refer here to component programming, but on the contrary to component assembly. The first approach still requires programming skills and a non trivial development effort. By contrast, component assembly refers to the *interactive* assembly and customization of existing components via an assembly tool. This tool could provide significant help to support these tasks by using for instance wizard-style dialogs and very interactive prototyping.

We believe that this kind of tool could considerably improve the relationships between research teams and the tool support teams. Research team would bring the generic framework, the tool support team would bring their knowledge about the company. The utltimate goal is to help large software companies to assemble their own SE tool suited to their specific needs.

9. References

[1] A. M. Davis, "*Why Industry Often Says 'No Thanks' to Research*", IEEE Software, November. 1992

[2] L.B.S. Raccoon, "*Fifty Years of Progress in Software Engineering*", ACM Sigsoft, Software Engineering Notes, Vol 22, No 1, pp 88-104, January 1997

[3] S.T. Redwine, W.E. Riddle, "*Software Technology Maturation*", 8th International Conference on Software Engineering, pp 189-200, August 1985

[4] H. Muller, "*Adoption Centric Software Engineering*", presentation at the Dagstul seminar on Software Architecture Reconstruction and Modeling, Feburary 2003

[5] A. Wasserman, "*Tool Integration in Software Engineering Environments*", LNCS 467, Springer-Verlag, pages 138–150, 1990.

[6] S.P. Reiss, "*Interacting with the FIELD Environment*", Software – Practice and Experience, 20(S1), 1990.

[7] S.P. Reiss, "*The Desert environment*", Brown University http://www.cs.brown.edu/software/desert

[8] R. Holt, A. Winter, A. Schürr, S. Sim, "GXL: Towards a Standard Exchange Format", 7th orking Conference on Reverse Engineering, November 2000

[9] OMG, "*MDA: the OMG Model Driven Architecture*", http://www.omg.org/mda/

[10] OMG, "*Model Driven Architecture - A Technical Perspective*", ormsc/01-07-01, 2001

[11] OMG, "Meta Object Facilities (MOF) Specification, Version 1.4", April 2002

[12] OMG, "Common Warehouse Metamodel (CWM) Specification, v1.1", March 2003

[13] http://www.eclipse.org

[14] Dassault Systèmes, http://www.3ds.com/

[15] J. Estublier, R. Casallas, "The Adele Software Configuration Manager", book chapter in *Trends in Software*. J. Wiley and Sons, 1994

[16] J. Estublier, J.M. Favre, R. Sanlaville, "*An Industrial Experience with Dassault Systèmes' Component Model*", Book chapter in Builiding Reliable Component-Based Systems, I. Crnkovic, M. Larsson editors, Archtech House publishers, 2002

[17] C. Hofmeister, R. Nord and D. Soni. *Applied Software Architecture. Addison-Wesley Publisher*, 2000.

[18] IEEE Architecture Working Group. "*IEEE Recommended Practice for Architectural Description of Software-Intensive Systems"* . IEEE Std 1471-2000, October 2000.

[19] P. B. Kruchten. The 4+1 view model of architecture. IEEE Software, 12(6): 42--50, November 1995.

[20] Y. Ledru, R. Sanlaville, J Estublier, "*Defining an Architecture Description Language for Dassault Systèmes*", 4th International Software Architecture Workshop, 2000.

[21] R. Sanlaville, "*Software Architecture: An Industrial Case Study within Dassault Systèmes*", PhD dissertation in french, Univeristy of Grenoble, 2002

[22] J.M.Favre, F. Duclos, J. Estublier, R. Sanlaville, J.J. Auffret, "*Reverse Engineering a Large Component-based Software Product*", European Conf. on Software Maintenance and Reengineering, CSMR'2001

[23] J.M. Favre, "*A New Approach to Software Exploration: Back-packing with* G^{SEE}", European Conference on Software Maintenance and Reengineering, CSMR'2002

[24] J.M. Favre, "G^{SEE}*: a Generic Software Exploration Environment*", 9th International Workshop on Program Comprehension, IWPC'2001

A Visual Language in Visio: First Experiences

Holger M. Kienle and Jens H. Jahnke
University of Victoria, Canada
{kienle,jens}@cs.uvic.ca

Abstract

It takes a lot of effort to go from the conceptual design for a new software engineering technique to its implementation and industrial adoption. Many such efforts fail and some for all the wrong reasons, e.g., good concept but awkward/immature user interface. Modern software engineering methods are oftentimes visual in nature. Developing mature "bread & butter" functionality for such tools requires significant effort in addition *to the actual conceptual advancement the inventor wants to make. As a consequence, this bread & butter functionality is oftentimes immature or awkward to use in research tool prototypes. Unfortunately, such problems can seriously impede industrial adoption of a new technique. We propose a software development approach that leverages widely-used, shrink-wrapped office tools by building visual tools on top of them. We believe that tool implementations that follow this approach have many desirable features from the user and adoption point of view. The approach is explored with a case study that develops* microSynergy*—a graphical development environment for the rapid construction of distributed embedded microcontroller software—on top of Microsoft Visio.*

1. Introduction

Potential reasons why some promising new software engineering tools remain lab orphans are, for example, users' unfamiliarity, difficult installation, non-intuitive user interface, unfavorable learning curve, and poor interoperability with existing development tools and practices.

Tools that are based on visual approaches pose additional adoption challenges. Examples of visual tools are (domain-specific) visual languages in the programming domain and graph-based exploration and editing tools in the reverse engineering domain.

Visual approaches often force developers to abandon parts of their existing tool infrastructure *in addition* to learning new paradigms. Understandably, developers are reluctant to adopt a new visual tool (whose impact they cannot yet assess) if it means that they have to abandon their familiar productivity tools. Furthermore, advanced graphical editing lacks universal paradigms which means that knowledge gained from one tool cannot readily transfered to another one [1].

While a visual approach can result in a more intuitive user interaction, the oftentimes idiosyncratic GUIs of research tools pose a significant adoption challenge. How can visual tools offer GUIs that are familiar to users? We propose a software development approach that leverages widely-used, shrink-wrapped office tools by building visual tools on top of them. Office tools such as Microsoft Visio and PowerPoint already offer general functionality that is expected of visual languages. For example, visual elements can be created and deleted; manipulated; connected; copied and pasted; and saved and loaded. Developers are already intimately familiar with this functionality because they use them on a daily basis to produce pictures for documentation and presentations. Both Visio and PowerPoint are highly programmable and customizable. Thus, domain-specific behavior (e.g., the particular look of visual elements and how they are allowed to interact) can be crafted on top of the existing, general functionality.

We illustrate our proposed approach with a case study of a visual tool in the embedded micro-controllers domain, called *microSynergy*, which we are currently implementing on top of Microsoft Visio.

Section 2 briefly describes *microSynergy* in general and its requirements. Section 3 discusses our preliminary experiences of implementing *microSynergy* with Visio. Section 4 closes with some conclusions.

2. *microSynergy*

microSynergy [3] is a graphical development environment for the rapid construction of distributed embedded micro-controller software. In industry, such applications are often developed from scratch and implemented using low-level programming languages like assembler or C. The software is often tightly coupled with specific hardware architectures, allowing little reuse and hindering interoperability.

This results in applications that are often only maintainable by programming experts who have an understanding of both the target hardware and the application domain.

microSynergy allows to visually specify the interaction logic for communication among multiple micro-controllers with the Specification and Description Language (SDL) [6]. Being developed by the International Telecommunications Union (ITU), SDL initially was intended to serve as a specification language for telecommunication systems. Today, it is increasingly adopted for other application areas. It has a standardized graphical representation with entities such as blocks, processes, procedures, signal and type declarations, inputs, outputs, conditions, variables, states, and transitions. A major advantage of SDL over alternative specification languages such as, for instance, the Unified Modeling Language (UML), is its formally defined and unambiguous semantics of each entity.

Research in *microSynergy* has been carried out in tight collaboration with Intec Automation Inc., a company in the area of embedded systems [2]. One of the research goals identified by Intec was the development of a tool that

- smoothly integrates into their current tool infrastructure and work processes.
- offers a steep learning curve[1] thus optimizing training costs and raising productivity.

These issues are closely related to tool adoption. A tool that does not integrate well into an existing infrastructure and process is less likely to be adopted. The same holds if the tool requires a significant up-front investment from the user.

Users of *microSynergy* are, for example, engineers that have no knowledge of general-purpose programming languages. However, these engineers are often familiar with the SDL formalism. Therefor, we chose a visual modeling paradigm based on SDL. *microSynergy*'s visual programming approach has the benefit that engineers manipulate graphical SDL objects with which they are already familiar. Since engineers operate in their familiar domain, we can expect the work product to be more easily understood and maintained in comparison to other (textual) representations that lack closeness to the problem domain.

3. Visio *microSynergy*

We analyzed office tools with regard to their suitability in providing a development environment for a *microSynergy* interface with desirable usability features. We just started our implementation with Visio and report on preliminary results. Figure 1 shows a screenshot.

[1] Learning curve researchers use *steep* to denote the desired behavior; this is opposite to the popular use of the term that views a steep learning curve as bad [4].

We decided to choose Microsoft Visio 2002 [7] because it is highly customizable and offers a robust user interface to build upon. Furthermore, it has a large user base and is commonly found in industry. In fact, our industrial partner, Intec, is using Visio.

Visio can be easily customized for different domains as nicely illustrated by the applications that Visio already offers: Web maps to visualize the components of Web sites, ER diagrams to model databases, electrical engineering diagrams for industrial control systems etc.

A customized application for a specific domain can, for example, offer

- customized stencils that contain the visual elements of the domain.
- additional toolbars, accelerators and menus.
- additional menu entries in a visual element's context menu.
- custom properties for visual elements.
- windows that contain hierarchical views ("model explorer") and domain-specific error messages.
- help features as an extension to the Visio help systems.

Visio exposes a VisualBasic API to access and analyze a document. All GUI elements (e.g., window, page, shape, and selection objects) are represented in the Visio object model [8]. Thus, it should be possible to seamlessly integrate domain-specific functionality on top of Visio.

Visio promotes tool adoption by leveraging:

a familiar GUI: The user interacts with a familiar environment and paradigm. Application knowledge has been typically built up by the user over years. Since the user is already familiar with the standard functionality, he can concentrate on learning new functionality (incrementally).

tool interoperability: Other office tools such as Word, PowerPoint, and Excel interoperate with Visio via cut-and-paste and (file-based) import/export facilities. For example, a SDL drawing in Visio can be easily imported in Word for documentation and PowerPoint for presentation purposes. Thus, users can be more productive in their daily work. Visio can output drawings, templates, and stencils in XML encoding. Thus, standard XML tools can be used to pre- and post-process a document.

customization and personalization: Office applications often have fine-grained customization features, especially for the graphical user interface. For example, GUI elements in Visio can be (interactively) repositioned and hidden.

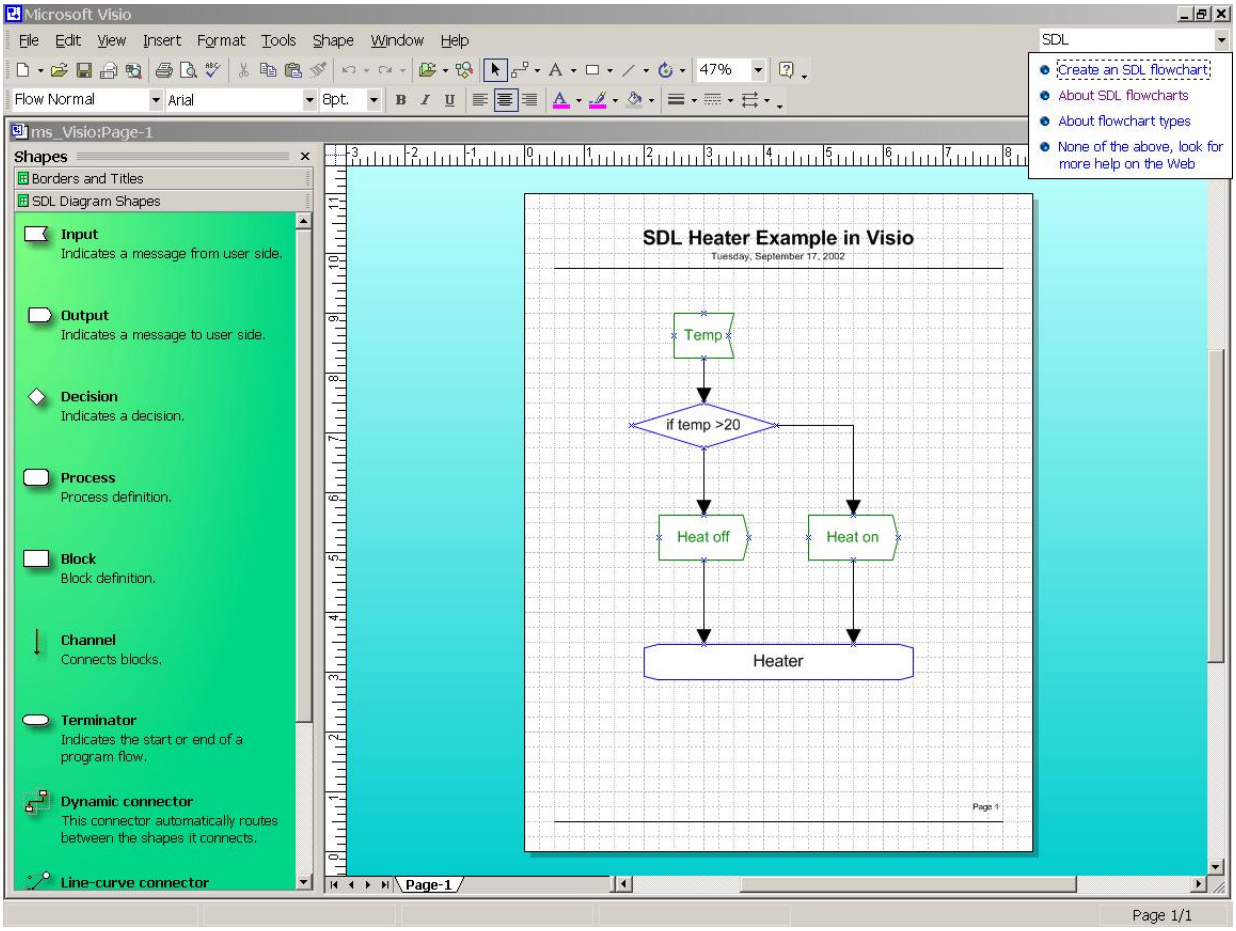


Figure 1. Screenshot of Visio *microSynergy*

tool support: Popular tools come with a large infrastructure that provides useful information to the user. For example, (online) publications discuss how to use a tool most effectively. Mailing lists and discussion forums help troubleshoot users' problems.

Tools that have a small target audience—which is typically the case for research tools—especially lack in the above areas. Since these tools focus often on the "proof of concept" and have limited financial resources and manpower, their GUIs provide only rudimentary functionality and does not support sophisticated customization or scripting. Since the user base is small, few experienced users exists (sometimes these users are mostly the tool's developers) that can offer help via mailing lists or newsgroups. Documentation is unprofessional, outdated, or non-existent.

3.1. Experiences

Visio keeps a group of graphical elements in a stencil (see "Shapes" window on the left in Figure 1). For SDL, we could reuse several SDL shapes (*masters* in Visio terminology) that were already part of Visio. We had to modify Visio's SDL "Output" master (see "Heat off" and "Heat on" in Figure 1) because its orientation was wrong. We used the master editor to change the shape and the master icon editor to change the shapes appearance on the stencil. We reused the SDL "Process" master (see "Heater"in Figure 1) from another SDL editor [5].

We found that domain-specific (GUI) behavior can be conveniently implemented with VisualBasic scripting. It was helpful having had prior experience in scripting other Microsoft office tools; they share many concepts and code can often ported with no or minor modifications. Since a programmer can incrementally add and immediately afterwards test the code, it is natural to build the application with rapid prototyping. VisualBasic offers a full development environment (with editor and debugger) that is tightly integrated with Visio. Scripting languages are a common mechanism to provide extensibility in office tools. Thus, power-users can look "under the hood" of an implementa-

tion and modify it to better fit their needs.

An important customization feature in Visio are custom properties. They are essentially key/value pairs that can be used to associate data (e.g., strings, numbers, boolean values, and lists) with visual elements. Custom properties are automatically saved and loaded as part of a Visio document. Thus, in our case, no code for persistence had to be written and the user can use the standard save and load functionality.

4. Conclusions

We believe that idiosyncratic GUIs along with unfamiliar concepts and paradigms are the primary adoption challenges that have to be overcome for visual tools. Our main hypothesis is that users will more likely use visual tools that are integrated into an environment that they use daily and know intimately.

A promising approach to achieve this goal is building these tools as extensions of shrink-wrapped office applications. Shrink-wrapped applications such as Visio can provide strong support for generic editor functionality, but offer quite limited support otherwise. Scripting can be used quite effectively to rapidly implement (domain-specific) functionality on top.

As discussed, Visio *microSynergy* has many desirable features from the user and adoption point of view. Our experiences give a first indication that this development approach can indeed help tool adoption. However, formal studies with user experiments are needed to confirm this hypothesis.

Acknowledgments

The *microSynergy* project has been jointly funded by Intec Automation Inc. and the Advanced Systems Institute of British Columbia.

References

[1] W. J. Hansen. The 1994 visual languages comparison. *IEEE Symposium on Visual Languages*, pages 90–97, Oct. 1994.

[2] Intec Automation Microcontrollers & Data Acquisition Systems. `http://www.steroidmicros.com/`.

[3] J. Jahnke. Engineering component-based net-centric systems for embedded applications. *ESEC/FSE 2001*, Sept. 2001.

[4] C. F. Kemerer. How the learning curve affects CASE adoption. *IEEE Software*, 9(3):23–28, May 1992.

[5] SDL-2000 and MSC-2000 Visio Add-on. `http://www.pherber.com/share/sdl/index.html`.

[6] Telelogic Tau. `http://www.telelogic.com/products/tau/sdl/index.cfm`.

[7] Microsoft Visio. `http://www.microsoft.com/office/visio/`.

[8] G. Wideman. *Visio 2002 Developer's Survival Pack*. Trafford, 2001.

Challenges Faced in Adopting Automated Standards Enforcement Tools

T. K. Shivaprasad
Tata Consultancy Services
t.krishnamurthy@usa-tcs.com

Vipul Shah
Tata Consultancy Services
v.shah@usa-tcs.com

Abstract

A good software development process requires programs to adhere to well-defined programming standards. Strict and comprehensive conformance checking of a program leads to detection of potential errors and unsafe uses, early in the software development life cycle. This, in turn, leads to a reduction in time spent on testing and also contributes to the improved quality of the program. In this position paper we describe the tool Assent, built at Tata Research Development and Design Centre (TRDDC) Pune, India. Assent uses a powerful specification-driven mechanism to generate standards checking tools. Assent uses a static semantic analyzer which makes use of global data flow analysis to detect potentially dangerous constructs in software. We present a case study, on a legacy billing application for the telecom industry, to demonstrate benefits to the user. We also describe the challenges faced in adopting the tool in practice.

1. Introduction

Identifying defects early in the software development lifecycle is very important for any application and especially so for mission-critical systems. Failure to detect defects could lead to loss of revenue and possibly loss of life. There have been a number of instances in the past where systems have failed resulting in disasters like the: "Ariane-5 mishap" [1, 2], the "Space shuttle Columbia mishap in 1981" [3] and the "AT&T Service failure" [10, 11]. In most of these cases, the problems were attributed to software defects.

A good software development process would require programs to be written using a well-defined programming standard. Programming standards are usually equated to uniform naming conventions and coding style. Though these are important, good programming practices contribute more to reducing defects and ensuring better quality of the application system. There are other sets of rules, we would like to term as "semantic" rules, non-conformance to which can lead to program crashes. These would include rules like "do not use un-initialized variables" or "do not de-reference null pointers". Programming standards should therefore comprise good programming practices, semantic rules, naming conventions and coding style.

Our experience in working with a large number of organizations across industry verticals like insurance, finance, banking, and telecommunications, among others, has been that there is usually a lack of well-documented programming standards. In most cases, programming standards may exist, but they may not be current. Even if we assume that standards are documented and available for use, how do we address the larger problem of finding skilled reviewers? How do we ensure completeness and consistency of review with respect to the laid down standards? It is manually impossible to apply each and every rule from the standard on every line of code!

In practice, people use their experience and look out for most commonly occurring defects. One of the problems with code review is that there is usually a lack of focus. Reviewers try to address too many things — functionality, performance, ensuring modularity and re-usability and adherence to good programming practices.

An automated standards checking tool can therefore be of great value to programmers. It can not only ensure completeness and consistency with respect to the laid-down standards, but also enable the reviewers to focus their review on other aspects.

The remainder of the paper is organized as follows. Section 2 provides a brief description of the automated standards checking tool developed by TRDDC, the research division of Tata Consultancy Services (TCS). Section 3 presents a case study on Assent and illustrates the results of using Assent. Section 4 lists the challenges faced while trying to get the tool adopted by developers. Finally, section 5 provides the conclusion of the paper.

2. Assent

Assent [5] is a tool that can automatically detect potentially dangerous constructs in software. Assent uses a powerful specification-driven mechanism to generate standards checking tools. Programming standard rules are defined as rules in a high level functional specification language. The specification is then automatically translated into a standards checking tool by the Assent framework. This ensures that a standards checking tool can be built quickly and at a far lower cost than by using conventional tool building techniques.

Program analysis, in particular, data-flow analysis [4, 8], has found many new and interesting applications in practically all aspects of software engineering over the last few years. Assent uses a static semantic analyzer which makes use of global data flow analysis [9]. Static analysis provides information on "reaching definitions", "used variables propagation" and "alias analysis in the presence of pointers and parameter bindings" [7] which can help identify "def-use and use-def chains", "use of uninitialized variables", and "possible null pointer de-referencing", among other things. The global data flow analysis technique provides Assent an ability to check programs for conformance to semantic rules.

We illustrate the specification language through the following examples:

Example 1: Syntactic rule. This rule doesn't use any flow-based analysis:

RULE: There shall be no un-used variable (variable declared but not used) in the program

USES: crossRef() // crossRef , for a variable x, gives all usage points of x in the program

RULE_ENTITY: All variables

RULE_BODY:

let

vars = entities(program, VARIABLE); // collect all the variables in the program

in

{ x <- vars | IsEmpty(crossRef(x))}; // report all the variables whose usage set is empty.

END

Example 2: Rule using data flow analysis

RULE: 'for' loop counter should not be modified within body of for

USES: reaches() which implements "reaching definitions";

RULE_ENTITY: All 'for' statements

RULE_BODY:

let

for_incr = incrExpr (for_stmt); //take the increment expression in 'for'.

incr_vars = usedVars(for_incr); // collect the used variables in the increment expression

Reaches = Filter (reaches(for_incr,true), incr_vars) // get the reaching definitions at the incr. expression node. Filter to get the definitions for the identifiers used in increment expression

for_AST = entities (body(for_stmt), ASTNODE); // get all the statements (nodes) in the body of the 'for' statement

in

iSEmpty (for_AST . Reaches) // take the intersection of the two sets – for_AST and Reaches. If the set is not empty then the definition is in the body of the 'for'.

END

The Assent tool can be integrated into a normal SDLC, making automatic standards checking an integral part of the development process. Assent has been successfully deployed in building standards checking tools for different standards, including MISRA-C [6] and Java standards.

The Assent framework of standards checking tools addresses a wide variety of programming standards and programming languages. Assent will automatically ensure that programs adhere to the defined standard. Benefits of using tools like Assent in review process are:

- Automated inspections
- Cutting down on code inspection time
- Defects are caught early in the life cycle
- Reduced cost of software development
- Improved quality of standards checking and hence better product

3. Case study

A module of 54K lines of C code, from a legacy billing application for the telecom industry, was chosen to demonstrate the capabilities of Assent. The aim was to evaluate the capabilities of a standards checking tool in improving productivity and delivering better quality systems.

Since the system was under maintenance, Assent was not expected to find any major problems. Surprisingly, Assent detected about 1411 possible violations. Of these, 39% of the violations were possible defects that could cause the application to crash, if a particular path in the program was executed. Non-adherence to certain good programming practices also led to incorrect functionality. The detection of such a large number of undetected fatal errors, from only a small section of the code that had been

in production for over 10 years, was a matter of grave concern.

The effort to manually review 54KLOC of code would be over 1 person month. Since complete and consistent application of coding standards is an impossible task manually, the manual process would have been able to find only a subset of all the non-conformances. Using Assent, we were able to complete the task in 1.5 hours — *a tremendous reduction in review time!*

3.1. Results

The programs were checked for conformance to standards using Assent and the non conformances were verified. 1411 non-conformances were detected by the tool. Out of the 25 rules defined for the standard, 11 rules were violated in the code. It took Assent 1.5 hours to check the conformance of the code to the given standards.

The 1411 non-conformances reported would not necessarily translate to an equivalent number of defects. As the programs were not executed, and therefore conditions not evaluated, a small number of the non-conformances detected would be false negatives.

Table 1 below depicts the non-conformances reported by Assent.

Table 1. Non-conformances reported by Assent

Defects that can cause programs to crash	39%
Structured Programming	30%
Good programming practices	19%
False Negatives	12%

3.1.1 Structured programming. Table 2 below depicts the non-conformances reported by Assent that are related to structured programming.

Table 2. Structured programming

Goto should not be used	54%
Null statements should be placed on a separate line	43%
Every switch should have a default case	3%

The rule "Null statements should be placed on a separate line" is useful in detecting unintentional statement terminators. Avoiding Goto in programs is a very well documented feature in programming.

3.1.2 Good programming practices. Table 3 below depicts the non-conformances reported by Assent that are related to good programming practices.

Table 3. Good programming practices

Identifier should not have the same name in different scopes	55%
If/While expression should not contain assignments	28%
For loop counter should not be modified within for body	9%
Every function should have a prototype visible at the call point	5%
Type mismatch	3%

Some of the rules violated in this category are:

- The 'for' loop counter should not be modified within the 'for' body, as it can lead to the for loop executing an incorrect number of times.
- Conditional statements should not have assignments. This may lead to confusion. In C, one of the most common errors is usage of '=' in an conditional statement instead of '=='.

3.1.3 Defects that cause programs to crash. Table 4 below depicts the defects in the code reported by Assent that can cause programs to crash.

Table 4. Fatal defects

Possible null pointer dereference	41%
Use of uninitialized variables	35%
Possible null pointer dereference - False Negatives	24%

Some of the rules violated in this category are:

- Usage of un-initialized variables. If the value of variables that have not been initialized is undefined, then the use of incorrect values may lead to incorrect behavior or may cause programs to crash.
- The Null pointer should not be dereferenced. Pointers that have a null value when dereferenced cause the application to crash.

4. Challenges with large scale adoption

From the case study, it is evident that standards checking tools like Assent aid in improving productivity and delivering better quality systems. Tools ensure maximum coverage of the code and list the non-conformances with respect to the specified standards. To prevent critical applications from crashing, it is imperative that tools like Assent be integrated into a normal SDLC, making automatic standards checking an integral part of the development process.

One would assume that the next logical step would be for users to deploy it as a part of the process. We faced

the following practical problems when we tried to get the tool adopted.

- Large number of non-conformances reported by the tool: Developers are not prepared for the number of violations reported by a standards checking tool. The manual process can only detect a fraction of the possible defects in the code due to the inherent limitation of the process.

  Code reviewers and developers are used to these numbers and large number of non-conformances, though valid, turns them away from the tool. Large numbers of violations in the code are not taken very kindly by developers as it raises doubts about their abilities.
- Programming standards: The practice commonly followed while defining programming standards for any organization, is to collate from existing standards and form an all encompassing standard. This results in a number of rules unnecessarily being made a part of the standard. Since the programming standards are used as a guide and not put completely in practice, it does not impact the manual process, but results in a lot of noise, in terms of violations, in an automated process.
- Severity levels: This leads to another interesting observation. It is a practice to assign severity to each non-conformance. What may have been perceived as a major severity defect while defining the standard, may no longer be considered major after checking the code for conformance to the standard.

  Though this provides a good opportunity to revisit the standards and tune them, it is easier said than done and developers shy away from it.
- False negatives: Since the technique employed to perform standards checking is static, the results are conservative. As programs are not executed, governing conditions which ensure correct execution are not evaluated, resulting in a number of false negatives.
- Performance: Global data flow analysis is known to be slow, as it iterates over the program graphs. Developers expect a standards checking tool to perform in real time like a compiler and are not willing to wait for minutes or hours, as the case may be. For large programs, performance becomes an issue and though we recommend that the tool be run in either single file analysis mode or batch mode for an entire application, this is not acceptable to developers.
- How do I fix it?: It has been our experience that once the tool reports a non-conformance, developers do not have a clue as to what the violations mean and what should be done to fix them. The rationale behind defining a rule is not clear. We need to impart training on programming standards, before providing training on the tool itself.
- Integration into process: Standards checking tools should integrate seamlessly into the development process. The tools should either integrate into the visual environment, or into the make files, as the case may be. We found that in the case of our tool, users had to re-specify the include paths and defines. Until all the include paths and defines were given to the tool, it would not function. A numbers of developers gave up after initial failures.
- Language dialects: Although ANSI/ISO standards have been defined for C/C++, we have found to our dismay that most compilers differ in their implementation. They support extensions that are non-standard. In the case of C++, we have yet to come across a compiler that does not deviate in its interpretation of the standard, especially with respect to language features like templates. This is a tool builder's nightmare, as all the dialects cannot possibly be supported.
- Modify/Customize rules: Developers perceive that they may need to modify the rules or add new rules over a period of time. It is not very easy for the developers to do so as the specification languages are either too complex or too simple, so that they end up writing a lot of code.

5. Conclusion

Adhering to good programming practices not only improves maintainability, but also improves the quality of the code by reducing the number of defects that get injected into the system. But, conforming to programming standards is not easy.

Some of the problems discussed in the earlier section can be addressed to some extent by providing adequate training to developers. What are programming standards? Why are they needed? What is the rationale behind each rule? Answers to these should be highlighted in the training.

The need for automated standards checking tools exists and tools have failed to fulfill the potential. The tools need to also provide clues and hints to the developers and possibly auto correct as many non-conformances as possible. A channel to incorporate developer feedback to improve the standards should also be set up. Proliferation of standards checking tools would then be easier.

6. References

[1] European Space Agency, *EAS/CNES Joint Press Release Ariane 501*, No. 33-96, 23 July 1996.

[2] European Space Agency, *Ariane 5, Flight 501 Failure*, Board of Inquiry Report, 19 July 1996.

[3] Excerpt from the Case Study of the Space Shuttle Primary Control System Communications of the ACM 27(9), September 1984, p. 886.

[4] A.V Aho, R. Sethi, and J.D Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, Mass., 1986.

[5] A. Sreenivas and V. Shah. Assent: An Automatic Specification-driven Standards Enforcement Tool. Technical report, Tata Research Development and Design Centre, Pune, India, 2000 - 01.

[6] The Motor Industry Software Reliability Association. Guidelines for the use of the C language in Vehicle Based Software. The Motor Industry Research Association, Warwickshire, UK, April 1998.

[7] M. Emami. A practical interprocedural alias analysis for an optimizing parallelizing C compiler. PhD thesis, School of Computer Science, McGill University, August 1993.

[8] M.S. Hecht. Flow Analysis of Computer Programs. Elsevier, North-Holland, 1977.

[9] H. Pande and A. Sreenivas. Darpan: A program analysis framework. Technical report, Tata Research Development and Design Centre, Pune, India, 1998.

[10] Ghost in the machine, Time magazine, 1990.

[11] Risks Digest, Vol. 9, Issue 61.

On the Security Risks of Not Adopting Hostile Data Stream Testing Techniques

Alan Jorgensen
Department of Computer Sciences
Florida Institute of Technology
aj@se.fit.edu

Scott Tilley
Department of Computer Sciences
Florida Institute of Technology
stilley@cs.fit.edu

Abstract

Security in all its forms plays an increasingly important role in many aspects of our lives. Given the importance of software applications to the modern information-driven economy, latent errors in popular commercial applications should no longer be considered mere annoyances, but as potentially serious security risks. Sample data from two recent case studies are used to illustrate the scale of the problem. This paper argues that techniques such as hostile data stream testing be adopted as standard practice by the software engineering community.

Keywords: security, testing, buffer overrun, best practice

1. Introduction

It is an unfortunate fact that many software developers fail to properly constrain the activities of input, output, storage, and computation [15][6]. It is also an unfortunate fact that software testers often fail to test for these failures, resulting in a plethora of security breaches initiated by buffer overruns. A search of the CERT/CC Web site [11] using terms such as "buffer", "buffer overrun", and "buffer overflow" yields an alarming number of hits (nearly 1,000 as of March 2003). Unfortunately, as shown in Figure 1, things don't seem to be getting any better.

Buffer overruns are a major source of security breaches for users (and providers) of the Internet. A typical breach involves sending a carefully crafted overlong data string such that program control is appropriated and redirected to the data string itself. The data string is crafted to contain hostile code that performs undesirable actions such as, in the case of a computer virus, retransmitting the hostile data stream to other computers. Once hostile code has been executed, a large variety of insidious behaviors may take place.

One of the most dramatic instances of the havoc caused by buffer overrun happened in January 2003 [13]. Branded the "SQL Slammer" worm, it exploited known vulnerabilities in Microsoft SQL 2000 servers. It caused widespread outages on the Internet due to the extremely large amount of network traffic it created while it scanned for vulnerable hosts. Microsoft issued a patch for this problem in the summer of 2002, but many system administrators failed to apply it to their computers (including machines at Microsoft itself). Fortunately, there was no malicious payload associated with the SQL Slammer worm; if there had; the damage to the Internet worldwide could have been extremely significant.

The next section overviews a technique, called hostile data stream testing, that can be used to proactively identify buffer overflow errors in software. Section 3 offers some examples of the use of this technique on common applications. Section 4 discusses some recommendations on the adoption of testing techniques to expose security vulnerabilities. Finally, Section 5 summarizes the paper and outlines possible avenues for further work.

2. Hostile Data Stream Testing

A program storing data outside of the area reserved for that data creates a buffer overrun. Typically this involves storing a sequence of data greater in length than the storage area (buffer) reserved for that data. Storing data in an inappropriate place usually causes the software to enter states unanticipated by the developer and consequently the behavior of software after an arbitrary buffer overrun is unpredictable.

Steganography is the embedding of a hidden message within another message. In the context of data transmitted over the Internet, data included within that transmission that serves a purpose other than the original purpose of the data transmission is steganographic data. An example of this kind of data would be hostile code embedded in what would otherwise be informational (inactive) data.

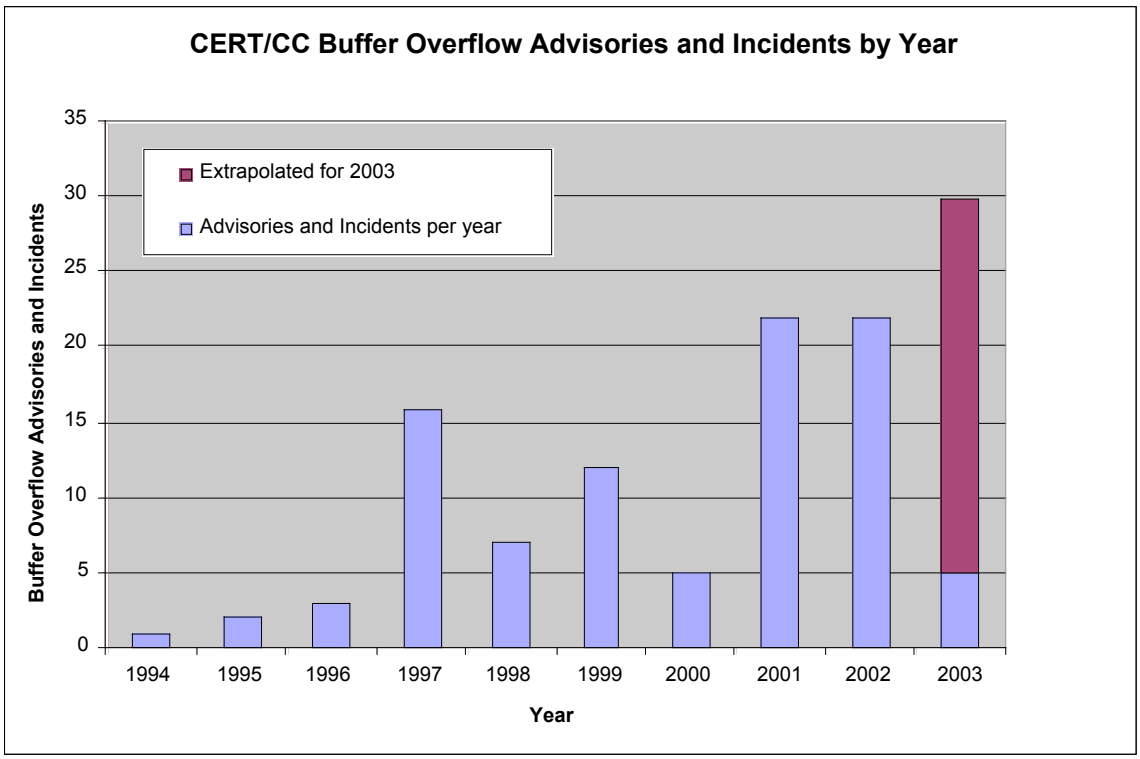


Figure 1: CERT/CC Buffer Overflow Summary

This class of security failure can be avoided by the identification and elimination of the defective code that performs buffer overruns. Historically, buffer overruns have been located by manual examination of source code. Though manual or automated code inspection is a useful and (sometimes) effective way to locate buffer overrun coding defects [2][14], in today's development environment of large, complex programs and libraries, the source code may not be available for review. Moreover, coding errors may be sufficiently subtle so as to elude even the most experienced programmer.

A black box testing method for identifying buffer overruns is described in [5]. This testing technique applies randomly deformed data streams to the application under test. This technique also provides a broader testing capability, however, and includes the ability to detect steganographic possibilities (places in data streams where information can be hidden without detection by the application processing that data stream).

Random data stream deformation is the process of taking a valid data stream and deforming that data stream in a manner such that the data stream is no longer valid. Three types of file deformation are used in the technique: lexical, syntactic, and semantic. Lexical deformations involve changing a valid lexical element to an invalid lexical element. An example of a lexical deformation is replacing a printable character with a non-printable character. In practice, the method is extended to include any character replacement.

Syntactic deformations involve changing valid syntactic elements to lexically correct but invalid syntactic elements. An example of a syntactic deformation is replacing a left parenthesis with a space character. (Previously, this definition was extended to include long string insertions. However, long string attacks now appear to be fourth technique for data stream deformation.)

Semantic deformations involve changing valid semantic elements to syntactically correct but semantically invalid elements. An example of a semantic deformation is changing the representation of a number to the representation of a different number that is invalid in that context.

A system for performing randomly malformed data stream testing system was recently developed at the Florida Institute of Technology. It is an automated solution to the buffer overrun detection problem that has been applied to commercial client application software. The next section discusses two case studies that apply this tool to widely used Internet applications.

3. Examples

To illustrate the pervasive nature of latent security bugs in common software applications, we present brief summaries of two case studies. The hostile data stream testing technique outlined in Section 2 was used in both cases. The number of errors identified using this method was somewhat surprising, since both applications under scrutiny have been widely deployed for a number of years.

3.1 Example 1

The first case study looked at a popular application for viewing Portable Document Files (PDF) [1]. This application is generally considered to be "trusted" software, coming as it does from a well-known company with a long and successful history. The PDF viewer is a ubiquitous application that runs on multiple platforms, and as such made an excellent choice for analysis.

The study applied 141,306 unique test cases. There were four recorded categories of failure, which were characterized as (1) application failure, (2) infinite loop, (3) failure-to-respond, and (4) steganographic. The tests revealed eleven distinct indications of buffer overrun, numerous program lock-ups, and four steganographic possibilities. Over all the test cases, there were 426 recorded cases of the first three types (which are severe application failures). Other than the uniqueness of the test cases, failure-to-respond failures were not classified to determine uniqueness of the failure.

In an infinite loop failure, the application is still running, but it is no longer possible to communicate with it. In contrast, failure-to-respond failures occur when the application is stalled and no longer responds to external stimulus. Both types are failures share characteristics that are similar to a denial-of-service attack on a network.

Steganographic failures seemed to be the result of the application performing incomplete parsing of the input data file. This leads to possible steganographical opportunities such as (1) comments, (2) "Document Property" objects, and (3) data after the "End of File" indicator ("%%EOF").

These application failures can legitimately be considered to be security vulnerabilities.

3.2 Example 2

The second case study looked at a popular browser plug-in and standalone application for viewing animation on the Web. Like the first case study, this program is nearly ubiquitous; it enjoys the largest penetration of any browser plug-in. Therefore, any flaws identified in it would have considerable implication for a very large number of users.

There were over 650,000 test cases that were run against this application. Each test case took (on average) less than 10 seconds, resulting in about 75 machine-days of testing on a standard personal computer running Microsoft Windows 2000. From these tests, more than 30 distinct symptoms of buffer overrun were discovered. There were also several hundred failure-to-respond stalls, but they have not as yet been further classified except by gross symptom (requiring hardware reboot, 100% CPU utilization, etc.)

The analysis of the data from this case study is still in progress. Nevertheless, preliminary results suggest that there are a large number of latent bugs in the application. If even a fraction of these bugs can be exploited as security vulnerabilities, then the problem is quite severe indeed. (The term "exploited" seems to have a broad range of meanings, from simple denial of service because the application crashes or stalls, to the most severe that allows preemption of control of the processor with root user privileges.)

4. Recommendations

The current situation appears to be one where developers do not properly defend against buffer overflow attacks in their code, and where testers do not properly exercise the application to bring latent buffer overflow errors to the attention of the developers. In this context, we offer three recommendations that would begin to address this problem.

4.1 Improve the Quality of Software

This is not a glib statement. It has long been a goal of the software engineering community to improve the state-of-the-practice. However, many potential improvements have been discarded in favor of practices that shrink the time to market for commercial software applications – often at the expense of product quality. When the quality attribute in question is security, we do not feel that this is a valid tradeoff.

Security flaws in modern software systems should no longer be treated as mere annoyances, but as the high-level risks that they truly are. It is an old anecdote that consumers would never put up with the number of bugs found in personal computers (and the software that runs on them) in any other appliance or mainstream product. The software engineering community as a whole must be

held more accountable for the quality of the applications they produce.

There are known engineering techniques for defending against buffer overrun errors. In general, adopting a defensive programming style would go a long way to solving this problem. Unfortunately, such coding idioms are only suggestions; they must become the standard way of doing business. It must become a uniform requirement that software that processes data from external sources shall examine that data for validity prior to any other use. In addition, we must adopt more thorough testing techniques to catch these types of errors before the code goes into production.

4.2 Adopt Hostile Data Stream Testing

As software testing becomes ever more integrated with software development, close attention must be paid to testing for potential hostile attacks. The old paradigm, "Garbage In, Garbage Out" is not acceptable and produces software vulnerable to security attacks. We propose a new paradigm, "Garbage In, Apology Out."

The testing technique outlined in Section 2 and described in [5] has proven to be a reliable way of detecting buffer overrun errors. Moreover, the two case studies outlined in Section 3 illustrate the great need for such testing – particularly in mature and commonly used applications. What remains to be done is for these techniques to be adopted by the community-at-large.

However, even if the techniques were adopted, the problem will not be solved until software vendors become more responsive to bug reports, particularly security-related bug reports that have the potential for causing so much harm.

4.3 Improve the Bug Reporting Mechanism

The security errors located in the two products outlined in Section 3 were reported to the respective vendors. In the first case study, the vendor completely ignored these reports. Due to the lack of response, the bug was then reported to the CERT/CC, who promised a 45-day response. This is in keeping with their stated policy [10]:

> "Effective October 9, 2000, the CERT Coordination Center will follow a new policy with respect to the disclosure of vulnerability information. All vulnerabilities reported to the CERT/CC will be disclosed to the public 45 days after the initial report, regardless of the existence or availability of patches or workarounds from affected vendors. Extenuating circumstances, such as active exploitation, threats of an especially serious (or trivial) nature, or situations that require changes to an established standard may result in earlier or later disclosure. Disclosures made by the CERT/CC will include credit to the reporter unless otherwise requested by the reporter. We will apprise any affected vendors of our publication plans, and negotiate alternate publication schedules with the affected vendors when required."

The PDF viewer problem was reported to CERT/CC on April 26, 2002. The buffer overrun reported in that disclosure is the one identified in [5] that allows storage of specified data in a specified location. As of the time of writing (March 2003), this defect still exists in the current release of that product.

In the second case study, the vendor did fix the critical problem, but did not publicly announce that the new release was in fact a security fix for several months. (Nor did they provide any credit to those who identified and reported the problem to them.)

Perhaps what is needed is an accepted protocol for reporting and fixing security vulnerabilities that would be adopted by vendors, security testers, and customers (such as the Federal Government). Ideally an independent body would enforce this protocol. For now we'd settle for it becoming the recommended best practice.

5. Summary

Latent security vulnerabilities pose a serious potential risk to all users. Sadly, such vulnerabilities seem omnipresent in typical application software. Given the widespread use of applications that suffer from these flaws, the implications for the user community as a whole are disheartening.

Fortunately, there are ways to combat the problem. The technique discussed in this paper, using hostile data stream testing, brings to the fore many of the hidden bugs in software that are due to buffer overrun errors. Since this testing technique is highly automated, it can be incorporated into today's regular development processes with relatively little negative impact.

There are adoption challenges to be addressed before this type of testing becomes common practice. Certainly the typical technology transition issues inherent with any new tool should be properly managed [8] [12]. However, the risks of not adopting hostile data stream testing pose an even greater problem.

As this paper was "going to press", an article appeared over the newswire concerning a newly discovered vulnerability in the widely used Sendmail

program [7] [9]. The flaw appears to be another instance of a buffer overrun problem that arises when the mail program parses an overly long header. This could allow an attacker to send a specially formatted email that could take control of the mail server and execute a malicious program. The bug has been present in the Sendmail code for 15 years, even though the code has been manually inspected many times by many different people.

The final irony is that the flaw appears in a Sendmail security function.

References

[1] Adobe, Inc. "Adobe PDF". Online at http://www.adobe.com/products/acrobat/adobepdf.html.

[2] Aleph One. "Smashing The Stack For Fun And Profit." *Phrack 49, Volume Seven, File 14,* November 1996.

[3] Costello, S. "McAfee: New virus is first to infect image files." *ComputerWorld*, June 13, 2002. Online at http://www.computerworld.com/securitytopics/security/virus/story/0,10801,71968,00.html.

[4] Costello, S. "Users question JPEG virus; McAfee stands firm." *ComputerWorld*, June 24, 2002. Online at http://www.computerworld.com/securitytopics/security/story/0,10801,72220,00.html.

[5] Jorgensen, A. "Testing with Hostile Data Streams." *ACM Software Engineering Notes*, March 2003.

[6] Jorgensen, A. *Software Design Based on Operational Modes*. Ph.D. dissertation, Florida Institute of Technology, 1999.

[7] Lemos, R. "Companies Mobilize to Patch Sendmail." *C|Net News.com*, January 27, 2003. Online at http://news.com.com/2100-1009-990802.html?part=dht&tag=ntop.

[8] SEI Technology Adoption program. Online at http://www.sei.cmu.edu/adopting/adopting.html.

[9] Software Engineering Institute CERT/CC. "CERT Advisory CA-2003-07 Remote Buffer Overflow in Sendmail." Carnegie Mellon University, 2003. Online at http://www.cert.org/advisories/CA-2003-07.html.

[10] Software Engineering Institute CERT/CC. "The CERT/CC Vulnerability Disclosure Policy" Carnegie Mellon University, 2003. Online at http://www.kb.cert.org/vuls/html/disclosure.

[11] Software Engineering Institute. *CERT Coordination Center*. Carnegie Mellon University. Online at http://www.cert.org/.

[12] Tilley, S.; Huang, S.; and Payne, T. "On the Challenges of Adopting ROTS Software." *Proceedings of the 3rd International Workshop on Adoption-Centric Software Engineering* (ACSE 2003: May 9, 2003; Portland, OR).

[13] Vamosi, R. "SQL Slammer Slows Internet Traffic." *C|Net News.com*, January 27, 2003. Online at http://www.cnet.com/software/0-7760531-8-20820927-1.html.

[14] Wagner, D., J. Foster, E. Brewer, and A. Aiken. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities." *Proceedings of the Year 2000 Network and Distributed System Security Symposium,* pp. 3-17, San Diego, Calif.ifornia, February 2000.

[15] Whittaker, J. and Jorgensen, A. "Why Software Fails." *ACM Software Engineering Notes*, July 1999.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	June 2003	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
3rd International Workshop on Adoption-Centric Software Engineering	F19628-00-C-0003

6. AUTHOR(S)

editors: Robert Balzer, Jens-Holger Jahnke, Marin Litoiu, Hausi A. Müller, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, Kenny Wong, Anke Weber

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2003-SR-004

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This report contains a set of papers that were presented at the Third International Workshop on Adoption-centric Software Engineering (ACSE). The papers focused on overcoming barriers to adopting research tools. Such barriers include the user's lack of familiarity with the tools, the mismatch between the tools and the users' cognitive models, a lack of interface maturity, limited tool scalability, poor interoperability and limited support for complex software engineering development tasks. The workshop papers explored innovative approaches to the adoption of software engineering tools and practices in particular by embedding them with middleware products and other commonly available commercial products.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software architecture, software architecture design, software architectural tactics, quality attribute scenarios	120

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL