Software Engineering Institute

Proceedings of the Workshop on Software Engineering Foundations for End-User Programming (SEEUP 2009)

Len Bass
Grace A. Lewis
Brad Myers
Dennis B. Smith

November 2009

SPECIAL REPORT
CMU/SEI-2009-SR-015

Research, Technology, and System Solutions (RTSS) Program

http://www.sei.cmu.edu

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

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.

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 FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications section of our website (http://www.sei.cmu.edu/publications/).

Table of Contents

List of Figures

List of Tables

Acknowledgments

There are many people that we would like to thank for making this workshop a success.

First of all, we would like to thank the authors and presenters that shared their work with us.

- Margaret Burnett, Oregon State University, USA
- Andrew J. Ko, University of Washington, USA
- Harold Schellekens, SAP BusinessObjects, Canada
- Mark Vigder, National Research Council, Canada
- Xingliang Yu, Chinese Academy of Sciences, China

Next we would like to thank the workshop attendees who contributed to the lively discussion.

- Thomas Aschauer, University of Salzburg, Austria
- Nat Ayewah, University of Maryland at College Park, USA
- Yvonne Dittrich, IT University of Copenhagen, Denmark
- Suzanne Garcia, Software Engineering Institute, USA
- Seth Holloway, University of Texas at Austin, USA
- Holger Kienle, University of Victoria, Canada
- Daniel Kulesz, Stuttgart University, Germany
- Hausi Muller, University of Victoria, Canada
- Gregory Nain, INRIA Rennes, France
- Karl Reed, University of Milano, Italy
- Chris Scaffidi, Oregon State University, USA
- Joao Sousa, George Mason University, USA
- Kathleen Stolee, University of Nebraska at Lincoln, USA
- Kerim Yildirim, Kocaeli University, Turkey

Finally, we thank our program committee, which made sure we had a set of high quality papers to frame our workshop.

- Margaret Burnett, Oregon State University, USA
- Steven Clarke, Microsoft Research, UK
- Sebastian Elbaum, University of Nebraska, USA
- Martin Erwig, Oregon State University, USA
- Mary Beth Rosson, Pennsylvania State University, USA
- Gregg Rothermel, University of Nebraska, USA
- Janice Singer, National Research Council, Canada

Abstract

The Workshop on Software Engineering Foundations for End-User Programming (SEEUP) was held at the 31st International Conference on Software Engineering (ICSE) in Vancouver, British Columbia on May 23, 2009.

This workshop discussed end-user programming with a specific focus on the software engineering that is required to make it a more disciplined process, while still hiding the complexities of greater discipline from the end user. Speakers covered how to understand the problems and needs of the real end users of end-user programming. The discussion focused on the software engineering and supporting technology that would have to be in place to address these problems and needs.

1 Workshop Introduction

We define *end-user programmers* as people who write programs not as their primary job function but in support of achieving their main goal, which is something else, such as accounting, designing a web page, doing office work, performing scientific research, or creating entertainment products. End-user programming (EUP), their activities in writing programs, has been around for a long time, in the form of shell scripts and Microsoft Excel spreadsheets that allow users to automate tasks specific to their needs.

However, since the Internet gained wider use and more particularly with the recent explosion in the availability of web technologies, end users now have more ways (such as JavaScript and Flash) to author programs, share them with others, and use programs created by others. Given the appropriate tools, end users can construct applications simply by using a set of drag-and-drop operations that pull together capabilities from different sources.

To fully realize the substantial potential benefits of EUP, software engineering discipline needs to be in place to enable the flexibility it provides and protect against the problems that might arise from that flexibility. For example, EUP through the Internet has vastly increased the use of shared code and shared data; the accompanying risk is that users are more exposed to code and data that is of poor quality or that might be malicious. This example is not just hypothetical: Businesses are seeing and understanding the impact of errors in end-user programs on their processes. The concern about safeguards for EUP has become known as End-User Software Engineering (EUSE).

To discuss EUSE, we organized a workshop on May 23, 2009 at the 31st International Conference on Software Engineering (ICSE 2009). Our "Workshop on Software Engineering Foundations for End-User Programming, SEEUP 2009" is also known as "WEUSE V: The 5th Workshop on End-User Software Engineering," to associate it with workshops held at previous ICSE and Computer Human Interaction (CHI) conferences and also at Dagstuhl, Germany.

1.1 Workshop Goal

The goal of the SEEUP 2009 workshop was to discuss EUP with a focus on how software engineering practice can be used in ways that make it more disciplined without burdening end users with the complexities of the greater discipline. Specifically, the intent is to understand the problems and needs of the real user in EUP.

We set the themes of the workshop to include discussions of:

- the range of EUP approaches
- specific EUP approaches that have the potential to provide significant benefits
- mechanisms to prevent malicious use of end-user capabilities while not inhibiting legitimate use
- an adoption path for EUP based on a disciplined software engineering foundation, highlighting potentials and limitations of end-user programming

1.2 Discussion

EUP has a broad range of application:

- Accounting (e.g., writing spreadsheet formulas and macros)
- Analysis using MatLab (a leading high-level language for computationally intensive tasks)
- Creation of web pages
- Recording of macros in Microsoft Word
- Automation of office tasks
- Creation of reports and forms in business software (e.g., SAP programming)
- "Programming" of appliances such as VCRs, remote controls, and microwave ovens
- Scientific research
- Authoring of educational software
- Creation of e-mail filters
- Configuration of synthesizers by musicians
- Entertainment (e.g., creation of behaviors in The Sims games)
- Web 2.0: mashups and end-user created content

This widespread use is reflected in the variety of other names by which EUP is known and the areas to which it is related. The term End-User Development (EUD) seems to be preferred in Europe; it encompasses other aspects of development besides programming, such as end-user design, testing, and documentation. The area of Domain-Specific Languages (DSLs) is highly related to EUP, since most DSLs are aimed at EUPs; languages for EUP are sometimes called scripts or macros. The concepts of end-user *tailoring* and *radical customization* are related to EUP, as is *visual programming* (the use of graphics to help with programming tasks). Another related area is programming-by-example (PBE) which is also called programming-by-demonstration (PBD). Finally, we have also heard the term *rapid application development* (RAD) used for systems aimed at EUP.

The National Science Foundation (NSF) reports that there are about 6 million scientists and engineers in the U.S., most of whom do some programming as part of their jobs [10]. Also, two NSF workshops determined that end-user software is in need of serious attention [2]. Our own research found that there are about 3 million professional programmers in the United States, but more than 12 million people say they do programming at work. Another 55 million use spreadsheets and databases and thus may also be considered to be doing programming [12].

Unfortunately, given how much EUP is being done, errors are pervasive in software created by end users. When the software that end users create is not dependable, the people whose retirement funds, credit histories, e-business revenues, and even health and safety rely on decisions made using that software can face serious consequences. For example, a Texas oil firm lost millions of dollars in an acquisition deal through spreadsheet errors [11]. There are many other examples of spreadsheet errors causing financial losses at http://eusesconsortium.org/euperrors/. Errors in web page programming and even in programming e-mail filters also cause people annoyance and harm.

A growing group of researchers is trying to address this problem. Two recent large collaborative efforts, for instance, have produced a number of promising results in this area (e.g., *End-User Development* [13]):

- in the U. S., the EUSES Consortium (http://eusesconsortium.org/)
- in Europe, the Network of Excellence on End-User Development (http://giove.cnuce.cnr.it/eud-net.htm)

Also, special Interest Group meetings at CHI'2004 [5], CHI'2005 [6], CHI'2007 [8], CHI'2008 [7], CHI'2009 [9] and the WEUSE series of workshops at ICSE'2005 [4], CHI'2006 [3], Dagstuhl 2007 (see www.dagstuhl.de/07081) and ICSE'2008 [1], very successfully brought together researchers and companies interested in this topic.

We feel that EUSE is a multi-disciplinary problem needing software engineering research, programming language research, education research, end-user programming research, and human-computer interaction (HCI) research of all types. Therefore, the SEEUP 2009 workshop encouraged interested people to work together and to connect researchers and companies with EUP products.

References

[1] R. Abraham, M. Burnett, and M. Shaw, eds., *Proceedings of the Fourth Workshop on End-User Software Engineering (WEUSE IV)*. New York: ACM, 2008.

[2] B. Boehm, and V. Basili, "Gaining intellectual control of software development," *Computer* vol. 33, no. 5, pp. 27-33, 2000.

[3] M. M. Burnett, et al., "The Next Step: From End-User Programming to End-User Software Engineering (WEUSE II)," *CHI'2006'06 extended abstracts on Human factors in computing systems*, Montreal, Canada, pp. 1699-1702, Apr. 2006.

[4] S. Elbaum and G. Rothermel, eds. *Proceedings of the First Workshop on End-User Software Engineering: WEUSE 2005*. http://www.cse.unl.edu/~grother/weuse/weuse-proceedings.pdf (accessed November 15, 2009).

[5] B. A. Myers, and M. Burnett, "End-Users Creating Effective Software (Special Interest Group Meeting Abstract)," *Extended Abstracts of the 2004 Conference on Human Factors in Computing Systems, CHI'2004*, Vienna, Austria, pp. 1592-1593, Apr. 2004.

[6] B. A. Myers, M. Burnett, and M. B. Rosson, "End Users Creating Effective Software. (Special Interest Group)," *Extended Abstracts Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI'2005,* Portland, OR (USA), pp. 2047-2048, Apr. 2005.

[7] B. A. Myers, et al., "End User Software Engineering: CHI'2008 Special Interest Group Meeting," *Extended Abstracts Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI'2008*, Florence, Italy, pp. 2371-2374, Apr. 2008.

[8] B. A. Myers, et al. "End User Software Engineering: CHI'2007 Special Interest Group Meeting," *Extended Abstracts Proceedings of the 2007 Conference on Human Factors in Computing Systems, CHI'2007*, San Jose, CA (USA), pp. 2125-2128, Apr. 2007.

[9] B. A. Myers, et al., "End User Software Engineering: CHI'2009 Special Interest Group Meeting," *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI'2009, Extended Abstracts Volume*, Boston, MA (USA), pp. 2731-2734, Apr. 2009.

[10] National Science Board, Science and Engineering Indicators 2006. National Science Foundation volume 1: NSB 06-01; volume 2: NSB 06-01A, 2006. Arlington, VA. http://www.nsf.gov/statistics/seind06/ (accessed November 15, 2009).

[11] R. Panko, "Finding spreadsheet errors: Most spreadsheet models have design flaws that may lead to long-term miscalculation," *Information Week*, pp. 100, May 29, 1995.

[12] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the Numbers of End Users and End User Programmers," *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, Dallas, TX (USA), pp. 207-214, Sept. 2005.

[13] H. Lieberman, F. Paterno, and V. Wulf eds., *End-User Development*. 2006, Springer: Dordrecht, The Netherlands.

2 Invited Talk: The State of the Art in End-User Software Engineering

Andrew J. Ko, The Information School, University of Washington, ajko@u.washington.edu

From the first digital computer programs in the 1940's to today's rapidly growing software industry, computer programming has become a technical skill of millions. As this profession has grown, however, a second, perhaps more powerful trend has begun to take shape. According to statistics from the U.S. Bureau of Labor and Statistics, by 2012 in the United States there will be fewer than 3 million professional programmers, but more than 55 million people using spreadsheets and databases at work, many writing formulas and queries to support their job [1]. There are also millions designing websites with Javascript, writing simulations in MATLAB [2], prototyping user interfaces in Flash [3], and using countless other platforms to support their work and hobbies. Computer programming, almost as much as computer use, is becoming a widespread, pervasive practice.

What makes these "end-user programmers" different from their professional counterparts is their goals: professionals are paid to ship and maintain software over time; end users, in contrast, write programs to support some goal in their own domains of expertise. End-user programmers might be secretaries, accountants, children [4], teachers [5], interaction designers [3], scientists [6] or anyone else who finds themselves writing programs to support their work or hobbies. Programming experience is an independent concern. For example, despite their considerable programming skills, many system administrators view programming as only a means to keeping a network and other services online [7]. The same is true of many research scientists [8], [6].

Despite their differences in priorities from professional developers, end-user programmers face many of the same software engineering challenges. For example, they must choose which APIs, libraries, and functions to use [9]. Because their programs contain errors [10], they test, verify and debug their programs. They also face critical consequences to failure. For example, a Texas oil firm lost millions of dollars in an acquisition deal through error in a spreadsheet formula [11]. The consequences are not just financial. Web applications created by small-business owners to promote their businesses do just the opposite if they contain bad links or pages that display incorrectly, resulting in loss of revenue and credibility [12]. Software resources configured by end users to monitor non-critical medical conditions can cause unnecessary pain or discomfort for users who rely on them [13].

Because of these quality issues, researchers have begun to study end-user programming practices and invent new kinds of technologies that collaborate with end users to improve software quality. This research area is called end-user software engineering (EUSE). This topic is distinct from related topics in end-user development in its focus on *software quality*. For example, there have been prior surveys of novice programming environments [14], discussing systems that either help students acquire computing skills or enable the creation of computational artifacts; while quality is a concern in these contexts, this work focuses largely on learning goals. There have also been surveys on end-user programming [15], [16], but these focus on the *construction* of programs to

support other goals, but not on engineering activities peripheral to construction, such as requirements, specifications, reuse, testing, and debugging.

In this invited talk, I explore prior work on end-user software engineering and identify several themes. In particular, I make the distinction between people who create programs for other people to use (typically called *professional programmers*) and people who create programs for their own use (typically called *end-user programmers*). I then explore five kinds of software engineering activities, identifying several differences between these communities:

- *Requirements*. In end-user programming, requirements are often emergent, tend to involve automating things, and they come from the user themselves. There is an interesting continuum between how many different requirements a program must satisfy: the more complex and diverse the requirements, the more end users must attend to software engineering goals.
- *Design and specifications*. End-user programmers rarely need to explicitly define designs or specifications, primarily because their requirements come from themselves. However, many systems have found ways of rewarding the creation of explicit specifications. For example, some systems support a particular design process (such as web site prototyping tools). Others raise the level of abstraction of the programming language, making it more like a specification language. Others still enable users to make intermediate specification languages, such as spreadsheet templates, and then get correctness guarantees.
- *Reuse*. Most of what end-user programmers do is reuse code and APIs, but there is a wide range of kinds of reuse. It may be at the level of copying and pasting code, parameterizing a form, creating rules, or choosing from a list of existing primitives or API calls. Systems have supported this variety of reuse in several domains.
- *Testing and Verification*. End-user programmers rarely have the incentive to explicitly test and verify their programs and they often are overconfident in their program's correctness. Systems have found ways to provide immediate feedback about correctness to combat this overconfidence. For example, spreadsheet systems have provided ways for users to confirm or reject output values and then see the impact of their decision on the overall spreadsheet correctness.
- *Debugging*. The central challenge in debugging is that guessing and checking what caused an incorrect output or behavior requires people to have a deeper understanding of the dynamic dependencies in the execution of a program. This typically requires more training than is necessary to write the program in the first place. Systems have provided ways for users to trace backwards from faulty output, to aid in the exploration of dynamic runtime dependencies.

In addition to the trends above, I explore several cross-cutting issues. For example, the choice to use any of the above systems usually involves an implicit assessment of the cost, risk, and potential reward of using the tool. Alan Blackwell's Attention Investment model [17] is a helpful way of evaluating these. This model has been converted into a design strategy called Surprise, Explain, Reward [18] in which the system creates some surprise (such as a conflict between the reasoning of the user and the system), explains the potential for a reward (such as improved program correctness), and then provides the reward (such as an increase the program's testedness).

I conclude the talk by arguing that most of the above techniques allow users to structure their work in a way that is consistent with their goals by providing incremental, immediate feedback for a range of software engineering techniques. Because feedback is incremental, users can proceed in whatever order makes sense for their work and still get useful feedback from the system. The immediacy of feedback is crucial to overcome end-user programmers lack of a disciplined and rigorous software engineering practice. This incremental, immediate feedback design strategy has the potential to generalize to many end-user software engineering tools.

References

[1] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the Numbers of End Users and End User Programmers," *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005, pp. 207-214.

[2] N. Gully, "Improving the Quality of Contributed Software on the MATLAB File Exchange," *The Next Step: From End-User Programming to End-User Software Engineering (WEUSE II Workshop at CHI'2006)*, Montreal, Québec, Canada. [Online] Available: http://eusesconsortium.org/weuseii/proceedings.php (accessed: October 31, 2009).

[3] B. Myers, S. Park, Y. Nakano, G. Mueller, and A. J. Ko, "How designers design and program interactive behaviors," *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing - Volume 00*, 2008, pp. 177-184 .

[4] M. Petre, and A. F. Blackwell, "Children as unwitting end-user programmers," *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, 2007, pp. 239-242.

[5] S. Wiedenbeck, "Facilitators and inhibitors of end-user development by teachers in a school environment." *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005, pp. 215-222.

[6] J. Segal, "Some problems of professional end user developers." *Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, 2007, pp. 111-118.

[7] R. Barrett, E. Kandogan, P. P. Maglio, E. M. Haber, L. A. Takayama, and M. Prabaker, "Field studies of computer system administrators: analysis of system management tools and practices," *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work*, 2004, pp. 388-395.

[8] J. Carver, R. Kendall, S. Squires, and D. Post, "Software engineering environments for scientific and engineering software: a series of case studies," *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 550-559.

[9] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*, 2004, pp. 199-206.

[10] R. Panko, "What we know about spreadsheet errors," *Journal of End User Computing*, vol. 2, pp. 15-21, 1998.

[11] R. Panko, "Finding spreadsheet errors: most spreadsheet models have design flaws that may lead to long-term miscalculation," *Information Week*, p. 100, May 29,1995.

[12] M. B. Rosson, J. Ballin, and J. Rode, "Who, what, and how: A survey of informal and professional web developers," *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, 2005, pp. 199-206.

[13] E. Orrick, "GE Healthcare Integrated IT Solutions, Centricity," *The Next Step: From End-User Programming to End-User Software Engineering (WEUSE II Workshop at CHI'2006)*, Montreal, Québec, Canada. [Online] Available: http://eusesconsortium.org/weuseii/proceedings.php. (accessed: October 31, 2009).

[14] C. Kelleher and R. Pausch, "Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers," *ACM Computing Surveys*, vol. 37, issue 2, pp. 83-137, 2005.

[15] A. Sutcliffe and N. Mehandjiev, "End-user development," *Communications of the ACM*, vol. 47, issue 9, pp. 31-32, 2004.

[16] H. Lieberman, F. Paterno, and V. Wulf, eds. *End-User Development*. 2006, Springer: Dordrecht, The Netherlands.

[17] A. F. Blackwell, "First steps in programming: A rationale for attention investment models," *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 2002, pp. 2-10.

[18] A. Wilson. M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel, "Harnessing curiosity to increase correctness in end-user programming," *Proceedings of the ACM CHI'2003 Human Factors in Computing Systems Conference*, 2003, pp. 305-312.

3 Extended Abstracts of Workshop Papers Published in ICSE Proceedings

The workshop accepted three papers for presentation at the workshop and publication in the official workshop proceedings. These proceedings are available in the ACM Portal at http://portal.acm.org/toc.cfm?id=1569137&coll=portal&dl=ACM&type=proceeding&idx=SERIES402&part=series&WantType=Proceedings&title=ICSE&CFID=48200530&CFTOKEN=64163928.

Authors of accepted papers were asked to submit extended abstracts for publication in these post-proceedings. The first paper by Burnett et. al. was part of the introduction and conceptual foundations for the workshop. These authors submitted an enhanced version of the paper based on input from the workshop. The second paper by Yu et. al and the third paper by Vigder are examples of the developer view of end-user programming. These are extended abstracts of the papers that appear in the official workshop proceedings.

3.1 End-User Software Engineering: A Distributed Cognition Perspective

Margaret Burnett, Christopher Bogart, Jill Cao, Valentina Grigoreanu, Todd Kulesza, Joseph Lawrance (Oregon State University, USA)

End-user programmers may not be aware of many software engineering practices that would add greater discipline to their efforts, and even if they are aware of them, these practices may seem too costly (in terms of time) to use. Without taking advantage of at least some of these practices, the software these end users create seems likely to continue to be less reliable than it could be. We are working on several ways of lowering both the perceived and actual costs of systematic software engineering practices, and on making their benefits more visible and immediate. Our approach is to leverage the user's cognitive effort through the use of distributed cognition, in which the system and user together reason systematically about the program the end user is creating or modifying. This paper demonstrates this concept with a few of our past efforts, and then presents three of our current efforts in this direction.

Introduction

A recent workshop on end-user software engineering focused on how to encourage end-user programmers to engage in the software engineering that is required to make end-user programming a more disciplined process, while still shielding the end user from the complexities of greater discipline (http://www.sei.cmu.edu/interoperability/research/approaches/seeup2009.cfm). This is an interesting issue from the perspective of the user's time and priorities. What might cause end-user programmers to become "more disciplined," and how would this impact their cost-benefit trade-offs of investing the time to do so versus the time/trouble they might save by doing so?

Our position is that we do not expect end-user programmers to voluntarily elect to become more disciplined unless doing so either is perceived by users to have obvious pay-offs given their own priorities or is so low in cost, they can afford to become more disciplined without worrying about the time cost.

To keep the cost of discipline low, we believe end-user software engineering must be a collaboration between the system and the user[1]. The system has two roles: (1) to pay some of the cost of adding discipline and (2) to make clear low-cost steps the user can perform to take advantage of that discipline and the benefits of doing so. We hypothesize that, if the user perceives reasonably low costs and useful benefits, the disciplined approaches suggested by the system will often seem more attractive than ad-hoc approaches, and users will follow them. Our previous work along these lines empirically supports this hypothesis.

Distributed cognition "extends the reach of what is considered cognitive beyond the individual to encompass interactions between people and with resources and materials in the environment [8]." In system-user collaborations to support the direction we have just described, by definition, the user does some of the reasoning and the system does some of the reasoning. The system's contribution to this reasoning may be simple, such as helping users remember judgments they have made so far; mid-level, such as reasoning about priorities as to which issues the user should consider next; or complex, such as performing static or dynamic analysis of source code to deduce possible errors.

Thus, instead of trying to build systems that solve this type of problem: "*What can the system figure out automatically so that users need not think too hard?"*, our distributed cognition perspective is that the problem statement becomes: "*How can end-user software engineering tools help end users think?"*

The rest of this paper provides examples as to how reasoning by end-user programmers can become more disciplined with the addition of distributed cognition support to end-user software development environments.

Examples from our Previous Work in End-User Software Engineering

Our *What You See Is What You Test* (WYSIWYT) methodology for testing spreadsheets [3, 16] demonstrates how distributed cognition can augment end users' abilities to use more disciplined approaches. In the case of WYSIWYT, the increased discipline is in testing and debugging.

With WYSIWYT, as a user incrementally develops a spreadsheet, he or she can also test that spreadsheet incrementally yet systematically. The basic idea is that, at any point in the process of developing the spreadsheet, the user can validate any value that he or she notices is correct. Behind the scenes, these validations are used to measure the quality of testing in terms of a test adequacy criterion. These measurements are communicated by visual decorations to reflect the new "testedness" state of the spreadsheet, to encourage users to direct their testing effort to the cells the system has systematically identified as needing the most attention.

For example, suppose that a teacher is creating a student grades spreadsheet, as in Figure 1. During this process, whenever the teacher notices that a value in a cell is correct, she can check it off ("validate" it). The result of the teacher's validation action is that the colors of the validated cell's borders become more blue, indicating that data dependencies between the validated cell and cells it references have been exercised in producing the validated values.

[1] Unless there are professional software developers involved to provide the discipline, as in the work of Fischer and Giaccardi [6] and Costabile et al. [5].

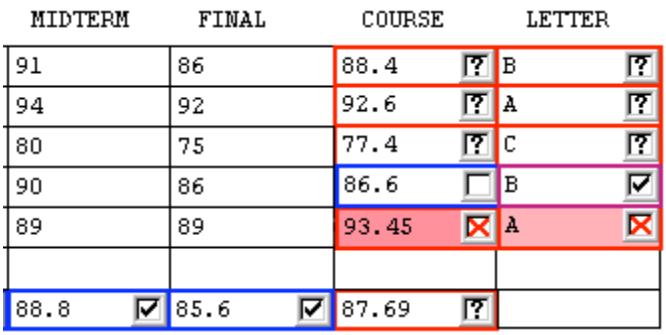


Figure 3-1: WYSIWYT supports systematic testing for end users, to help the user test and debug spreadsheet formulas [16].

A red border means untested, a blue border means tested, and shades of purple (i.e., between red and blue) mean partially tested. From these border colors, the teacher is kept informed of which areas of the spreadsheet are tested and to what extent. Thus, in the figure, row 4's Letter cell's border is partially blue (purple), because some of the dependencies ending at that cell have now been tested. Testing results also flow upstream against dataflow to other cells whose formulas have been used in producing a validated value. In our example, all dependencies ending in row 4's Course cell have now been exercised, so that cell's border is now blue.

The border colors support distributed cognition by remembering (and figuring out and updating) a "things to test" list for the teacher. This distributed cognition allows the teacher to test in a more disciplined way than she might otherwise do, because it constructs its things-to-test statuses using a formal test adequacy criterion (du-adequacy) that the user is not likely to know about.

The checkboxes further support distributed cognition by remembering for the user the specifics of testing that was done. Here the checkmark reminds the teacher that a cell's value has been validated under current inputs. As with the border colors, the distributed cognition goes further than just remembering things done directly—it also manages the "things tested" set by changing the contents of the checkboxes when circumstances change. For example, an empty checkbox indicates that the cell's value was validated, but the value was different than the one currently on display. Finally, the system helps the teacher manage her testing strategy by showing a question mark where validating the cell would increase testedness.

Checkmarks and border colors assist cognition about things to test and things tested successfully. There is also a fault localization functionality for things tested *un*successfully. For example, suppose our teacher notices that row 5's Letter grade is erroneous, which she indicates by X'ing it out instead of checking it off. Row 5's Course average is obviously also erroneous, so she X's that one too. As Figure 1 shows, both cells now contain pink interiors, but Course is darker than Letter because Course contributed to two incorrect values (its own and Letter's) whereas Letter contributed to only its own. These colorings are another example of distributed cognition. They make precise the teacher's reasoning/recollection about cell formulas that could have contributed to the bad value and direct her attention to the most implicated of these, thereby encouraging her to systematically consider all the possible culprits in priority order to find the ones that need fixing.

Recall our hypothesis from Section 1 that it is necessary to keep the cost of discipline low. WYSIWYT gives us a vehicle for considering the cost of discipline: Just why *would* a user whose

interests are simply to get their spreadsheet results as efficiently as possible choose to spend extra time learning about these unusual new checkmarks, let alone think carefully about values and whether they should be checked off?

To succeed at enticing the user to use discipline, we require a strategy that can motivate these users to make use of software engineering devices, can provide the just-in-time support they need to effectively follow up on this interest, and will not require the user to spend undue time on these devices.

We call our strategy for enticing the user down this path Surprise-Explain-Reward [21]. The strategy attempts to first arouse users' curiosity about the software engineering devices through surprise, and to then encourage them, through explanations and rewards, to follow through with appropriate actions. This strategy has its roots in three areas of research: research about curiosity [13], Blackwell's model of attention investment [2], and minimalist learning theory [4].

The red borders and the checkboxes in each cell, both of which are unusual for spreadsheets, are therefore intended to surprise the user, to arouse curiosity [13]. These surprises are non-intrusive: users are not forced to attend to them if they view other matters to be more worthy of their time. However, if they become curious about these features, users can ask the colors or checkboxes to explain themselves at a very low cost, simply by hovering over them with their mouse. Thus, the surprise component delivers the user to the explain component.

The explain component is also very low in cost. In its simplest form, it explains the object in a tool tip. For example, if the user hovers over a checkbox that has not yet been checked off, the tool tip says: *"If this value is right, √ it; if it's wrong, X it. This testing helps you find errors."* Thus, it explains the semantics very briefly, gives just enough information for the user to succeed at going down this path, and gives a hint of the reward to the task at hand, as per the implications of attention investment and minimalist learning theory [2], [4].

The main reward is finding errors, which is achieved by checking values off and X'ing them out to narrow down the most likely locations of formula errors. A secondary reward is a "well tested" (high coverage) spreadsheet, which at least shows evidence of having fairly thoroughly looked for errors. To help achieve testing coverage, question marks point out where more decisions about values will make progress (cause more coverage under the hood, cause more color changes on the surface), and the progress bar at the top shows overall coverage/testedness so far. Our empirical work has shown that these devices are both motivating and that they lead to more effectiveness [3], [17].

Current Research Directions

More disciplined debugging of machine-learned programs

The recent increase in machine learning's presence in a variety of desktop applications has led to a new kind of program that needs debugging by the end user: programs written (learned) by machines. Since these learned programs are created by observing the user's data and reside on the user's machine, the only person present to fix them if they go wrong is the user.

Traditional methods for end users to improve the logic of machine-learned programs have been restricted to relabeling the output of these programs. For example, imagine a movie recommenda-

tion system that uses machine learning techniques to make intelligent recommendations based on a user's previously viewed movies. This system allows the user to label the suggestions by marking each as something they are either interested in or not interested in. Such debugging, however, is ad hoc. The user can neither know how many recommendations to label before the system will improve nor know how far-reaching an observed improvement is, and therefore, cannot plan or be strategic: the entire process is based solely on the luck of just the right inputs arriving in a timely way.

We are working on a more disciplined approach to debugging machine-learned programs to remove this complete reliance on fortuitous inputs arriving. In our approach, end users directly fix the logic of a learned program that has gone wrong, and the machine keeps them apprised of the greater impacts of their fixes. Two keys to our approach are (a) an interactive explanation of the learned program's logic and (b) a machine-learning algorithm that is capable of accepting and integrating user-provided feedback. The approach is more disciplined than traditional approaches in that it removes much of the element of luck in the arrival of suitable inputs, and also employs distributed cognition devices to enable users to predict the impacts of their logic changes.

We have designed and implemented our first prototype [10] aimed at meeting these goals. The domain we used for this prototype was automatically filing the user's incoming email messages to the correct folder. Our approach was inspired by the Whyline [9]. In our variant of why-oriented debugging, users debug through viewing and changing answers to "why will" and "why won't" questions. Examples of the questions and manipulable answers in our prototype are shown in Figure 2. As soon as the user manipulates these answers, the system not only updates the result of the input they are working with (in this case, the particular email message), but also apprises them of how other messages in their email system would be categorized differently given these changes, in essence running a full regression test.

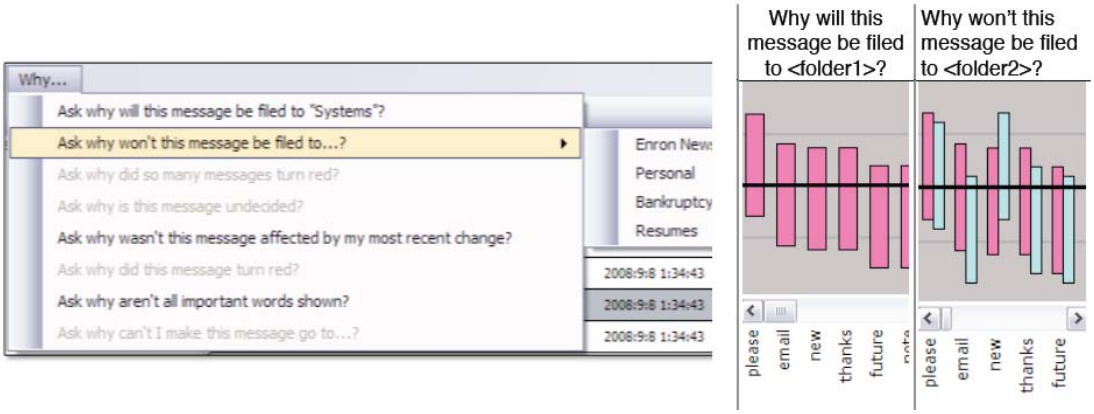


Figure 3-2: An interactive visualization for showing end users the relative importance of different words to a machine-learned program's decision-making. For example, the word "email" is fairly neutral in classifying messages to folder1 (pink), but is forbidden for folder2 messages (blue). Users are able to drag any bar up or down to explicitly change the logic of the learned program [10].

Using this prototype, we analyzed barriers end users faced when attempting to debug using such an approach [10]. The most common obstacle for users was determining which sections of the learned program's logic they should modify to achieve the desired behavior changes on an ongo-

ing basis. The complete set of barriers uncovered is being used to inform the design of our continuing work in enabling end users to debug programs that are learned from that particular user's data.

Strategies as agents of discipline for males and females

Given their lack of formal training in software engineering, end-user programmers who attempt to reason about their programs' correctness are likely to do so in an ad-hoc way rather than using a systematic strategy. Strategy refers to a reasoned plan or method for achieving a specific goal. It involves intent, but the intent may change during the task. Until recently, little has been known about the strategies end-user programmers employ in reasoning about and debugging their programs. We have been working to help close this gap, and to devise ways to better support end-user programmers' strategic efforts to reason about program correctness.

The WYSIWYT approach described in Section 2 promotes debugging strategies based on testing. One problem with testing-based strategies is that they do not seem to be equally attractive to male and female end-user programmers. In a recent study, we found males both preferred testing-based strategies more, and were more effective with them, than females [18]. This was also the case for dataflow strategies (Figure 3). On the other hand, the same study showed that code (formula) inspection strategies were more effective for females than for males.

Gender differences in approaches to end-user software development have also been reported in debugging feature usage [1] and in end-user web programming [15].

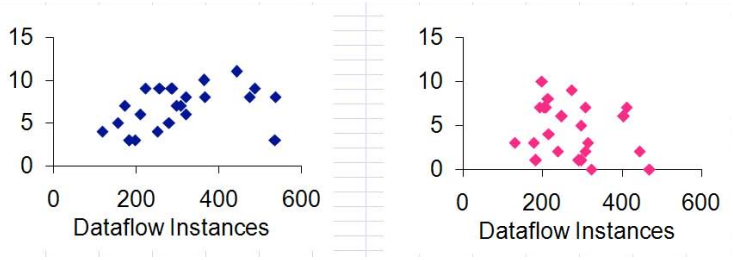


Figure 3-3: Correlation between total bugs fixed and number of dataflow following instances. Left: male (significant), right: female (not significant) [18].

Gender differences in approaches to end-user software development have also been reported in debugging feature usage [1] and in end-user web programming [15].

In fact, of the eight debugging strategies we learned about in our study of spreadsheet work—Testing, Code Inspection, Specification Following, Dataflow, To-Do Listing, Color Following, Formula Fixing, and Spatial—seven (all but Spatial) had gender differences in ties to success at fixing spreadsheet formula errors [18]. In a follow-up study on strategies employed by a different population at the border between end-user programmers and professional developers, namely IT professionals debugging system administration scripts, the results on what debugging strategies were used were nearly the same, with a few additions due to differences in resources and paradigm [7]. The resulting list of ten end-user debugging strategies is shown in Table 1 [7].

Table 3-1: Strategies in Finding and Fixing Bugs

Strategy	Definition
Direct Matches	
Testing	Trying out different values to evaluate the resulting values.
Code Inspection	Examining code to determine its correctness.
Specification Checking	Comparing descriptions of what the script should do with the script's code.
Dataflow	Following data dependencies.
Spatial	Following the spatial layout of the code.
Generalized Matches	
Feedback Following	Using system-generated feedback to guide their efforts.
To-Do Listing	Indicating explicitly the suspiciousness of code (or lack of suspiciousness).
New Strategies	
Control Flow	Following the flow of control (the sequence in which instructions are executed).
Help	Getting help from people or resources.
Proceed as in Prior Experience	Recognizing a situation (correctly or not) as one experienced before, and using that prior experience as a blueprint of next steps to take.

We are now beginning to explore how to explicitly support strategies that seem particularly attractive to one or the other gender, but are not yet well supported in end-user software development environments. For example, females' most effective strategies, namely Code Inspection, To-Do Listing, and Specification Checking, are not supported in spreadsheet software [18]. One example of an approach to supporting code inspection would be adding an "inspectedness" state to cell formulas, similar to the "testedness" state supported by WYSIWYT. This way, distributed cognition in the system environment could help users track which *formulas* have been inspected and judged correct or incorrect (the spreadsheet auditing product described in [19] shows one possible approach for this).

In the study of IT professionals' debugging, one interesting way in which females used Code Inspection effectively was by looking up examples of similar formulas to *fix* errors in the spreadsheet, after already having *found* the error [7]. This is a repurposing of code inspection for debugging purposes that has little support in debugging tools for end-user programmers. An idea along these lines that we are exploring is that part of the cognitive effort of searching for and memorizing related formulas could be reduced by offloading to the software the task of finding related formulas and displaying them (external memory).

Thus, the goal of this work is to encourage the use of disciplined, strategy-based problem solving by end-user programmers through distributed cognition approaches that support a variety of strategies. We hypothesize that such support will increase the discipline used in end-user programmers' problem-solving and, as a result, will increase male and female end-user developers' productivity and success at debugging their programs.

How Information Foraging Theory can Inform Tools to Promote Discipline

The theme of this paper is that distributed cognition can help promote discipline for end-user programmers, but beyond this basic point, it would be helpful to developers of tools for end-user programmers to have guidance that is more concrete and prescriptive. We believe information foraging theory can provide such guidance.

Information foraging theory models a person's search for information as a hunt for *prey*, guided by *scent*. The *prey* is the information they are seeking. The *scent* is the user's estimate of relevance, which the user derives from cues in the environment. The hunt for relevant information then proceeds from location to location within that environment, each time following the most salient cue. Thus the *topology* of that information space and the scent of the cues predict how well the user will be able to navigate to the most needed information. Information foraging theory was proposed as a general model of information seeking [14], but has primarily been applied to web browsing. We have been researching the applicability of this theory to people's information-seeking behavior in software maintenance. In our studies to date on professional programmers working in Java, information foraging theory predicted people's navigation behavior when debugging as well as the aggregate human wisdom of a dozen programmers, and it was also effective at picking the right locations to focus on when debugging [11, 12].

Because cues are externalizations of scent (relevance), following cues is a strategic way to eliminate large portions of the code that must be considered in tracking down a bug. Cues can be found in both the GUI and in the software artifacts themselves. For example, variable names, component names, pictorial icons that seem to represent the relevant functionalities, are all cues. Tool feedback, such as the highlighted cells of WYSIWYT's fault localization, is the system's way of enhancing distributed cognition about where to navigate. The user can also enhance this distributed cognition through what Pirolli and Card term *enrichment*.

Pirolli defines enrichment as the extra work an information seeker does to enhance the information density or topology of the space they are working in. Pirolli [14] studied an analyst flipping through a pile of magazines, cutting out articles to examine more closely later. The information density of this smaller pile of articles would make later information seeking go more quickly. Many software engineering tools and practices are about enrichment. Professional developers comment code, draw UML diagrams, write specifications, document changes, and link these all together to create an information topology that allows developers to more quickly get to useful information about the program. Some enrichment is informal as well, such as maintaining "to do" lists and sketching diagrams. These little notes and diagrams users create in a working space are in essence a new user-defined patch whose purpose is to help the user process information quickly. In these ways, enrichment adds to distributed cognition.

These constructs of information foraging theory—scent, cues, topology, and enrichment—thus suggest a design strategy for tools aimed at encouraging discipline in end-user programmers. First, identify what questions the tool is trying to support. Then, given these questions, choose the scent (form of relevance) that the tool will support in order to answer them. Then choose a topology and cues to allow following that scent.

For example, if the question being supported is "Why did…", the relevant scent could be the trail of dynamic state, the cues could be "invoked" edges between each called function/component as well as their names, and the topology could connect these cues to the function's states at the time they were called so that the user can step along the call sequence with each function's details (as in the Whyline [9]).

Topology and choice of cues are interdependent. The topology needs to allow a user to navigate to the relevant information called out by the cues, and the cues (and thus distributed cognition) need to emanate scent to attract the user down appropriate paths in the topology. Finally, the system

should allow the user to easily enrich the cues and topology to further enhance their working environment's distributed cognition.

Tools based on information foraging theory could also promote program maintainability. For example, tools based on information foraging theory could evaluate and suggest improvements to words, labels, pictures, and explicit connections in programs and their associated artifacts, so that cues in these artifacts would emanate stronger and more precise scent.

In the service of reuse, tools based on information foraging theory could serve as distributed cognition elements connected to cues, topology, or enrichment, to help people get answers to reuse questions [20] such as: (1) I know generally what I want; which components are relevant? (2) There is a component that I've used before from this repository, but I forget the name and several other details of it; where is it? (3) This component is not quite what I need; which other components are similar?

As these examples show, information foraging theory provides a basic foundation from which design ideas can be derived on how to promote disciplined approaches to navigation-oriented needs in end-user software development.

Conclusion

As we have shown, tools based on distributed cognition can promote more disciplined behavior by end-user programmers. Distributed cognition works because it allows the system to contribute part of the reasoning and memory, so that users do not have to manipulate all the relevant information in their own heads alone in order to follow disciplined approaches. That is, end-user software engineering approaches based on distributed cognition do not aim to *remove* users' need to reason systematically about their software; rather, they aim to *enhance* users' ability to reason systematically about their software.

Our empirical results over the years have provided encouraging evidence that this approach can not only encourage software engineering discipline, but can do so in a way that tears down some barriers that, in current tools, seem to disproportionately target female end-user programmers. One key to reaping these benefits is keeping the costs of using these tools low, and another is keeping the benefits to the targeted users high, so that users' perception of the costs/benefits/risks involved will make them *want* to use these devices.

Acknowledgements

This work was supported in part by an IBM Faculty Award, by the Air Force Office of Scientific Research, and by NSF ITR-0325273, IIS-0803487, and IIS-0917366. We are grateful to our colleagues in the EUSES Consortium for helpful discussions, feedback, and ideas, and to Laura Beckwith, Rachel Bellamy, Curt Cook, Paul ElRif, Xiaoli Fern, Mark Fisher, Cory Kissinger, Andrew Ko, Vaishnavi Narayanan, Ian Oberst, Kyle Rector, Stephen Perona, Gregg Rothermel, Joseph Ruthruff, Amber Shinsel, Simone Stumpf, Neeraja Subrahmaniyan, Susan Wiedenbeck, Weng-Keen Wong, and our other former students and colleagues who contributed to this work.

References

[1] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, "Effectiveness of End-User Debugging Software Features: Are There Gender Issues?" *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2005, pp. 869-878.

[2] A. F. Blackwell, "First steps in programming: A rationale for attention investment models," *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 2002, pp. 2-10.

[3] M. Burnett, C. Cook., and G. Rothermel, "End-User Software Engineering," *Communications of the ACM*, vol. 47, issue 9, pp. 53-58, Sept. 2004.

[4] J. M. Carroll, and M. B. Rosson, "Paradox of the Active User," in *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, J. M. Carroll, Ed., Cambridge, Massachusetts (USA): MIT Press, 1987, pp. 80-111.

[5] M. F. Costabile, D. Fogli, P. Mussio, and A. Piccinno, "End-User Development: The Software Shaping Workshop Approach," in *End-User Development*, Lieberman, H., Paterno, F., and Wulf, V., eds, Springer: Dordrecht, The Netherlands, 2006, pp. 183-205.

[6] G. Fischer, and E. Giaccardi, "Meta-Design: A Framework for the Future of End-User Development," in *End-User Development*, Lieberman, H., Paterno, F., and Wulf, V., eds, Springer: Dordrecht, The Netherlands, 2006, pp. 427-457.

[7] V. Grigoreanu, J. Brundage, E. Bahna, M. Burnett, P. ElRif, and J. Snover, "Males' and Females' Script Debugging Strategies," *International Symposium on End-User Development, Siegen, Germany, published as Lecture Notes in Computer Science 5435*,V. Pipek et al., eds., Springer-Verlag, 2009, pp. 205-224.

[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*, vol. 7, pp. 174-196, 2000.

[9] A. Ko, and B. Myers, "Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior," *Proceedings of the 2004 ACM Conference on Human Factors in Computing Systems (CHI'2004)*, 2004, pp. 151-158.

[10] T. Kulesza, W.-K. Wong, S. Stumpf, S. Perona, R. White, M. Burnett, I. Oberst, and A. Ko, "Fixing the Program My Computer Learned: Barriers for End Users, Challenges for the Machine," *Proceedings of the 13th International Conference on Intelligent User Interfaces*, 2009, pp. 187-196.

[11] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks," *Proceedings of the 12th International Conference on Intelligent User Interfaces*, 2008, pp. 1323-1332.

[12] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Can Information Foraging Pick the Fix? A Field Study," *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'08)*, 2008, pp. 57-64.

[13] G. Lowenstein, "The Psychology of Curiosity," *Psychological Bulletin*, vol. 116, no. 1, pp. 75-98,1994.

[14] P. Pirolli and S. Card, "Information Foraging," *Psychological Review* 106, pp. 643-675, 1999.

[15] M. B. Rosson, H. Sinha, M. Bhattacharya, and D. Zhao, "Design Planning in End-User Web Development," *Proceedings of the 2007 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'07)*, 2007, pp. 189-196.

[16] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov, "A Methodology for Testing Spreadsheets," *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 1, pp. 110-147, 2001.

[17] J. Ruthruff, A. Phalgune, L. Beckwith, M. Burnett, and C. Cook, "Rewarding Good Behavior: End-User Debugging and Rewards," *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'04)*, 2004, pp. 115-122.

[18] N. Subrahmaniyan, L. Beckwith, V. Grigoreanu, M. Burnett, S. Wiedenbeck, V. Narayanan, K. Bucht, R. Drummond, and X. Fern, "Testing vs. Code Inspection vs. What Else? Male and Female End Users' Debugging Strategies," *Proceedings of the 2008 ACM Conference on Human Factors in Computing Systems (CHI'2008)*, 2008, pp. 617-626.

[19] N. Subrahmaniyan, M. Burnett, and C. Bogart, "Software Visualization for End-User Programmers: Trial Period Obstacles," *Proceedings of the 4th ACM Symposium on Software Visualization, 2008*, pp. 135-144.

[20] R. Walpole and M. Burnett, "Supporting Reuse of Evolving Visual Code," *Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*. 1997, pp. 68-75.

[21] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. "Harnessing Curiosity to Increase Correctness in End-User Programming," *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2003, pp. 305-312.

3.2 Extending the Boundary of Spreadsheet Programming: Lessons Learned from Chinese Governmental Projects

Xingliang Yu, Jing Li, Hua Zhong (Chinese Academy of Sciences, China)

Solving business problems through programming by end users themselves is not only a problem of great concern of academics, but also the dream of the end-user. Having the ability to write programs without professional developers will go a long way toward liberating the end-user. Our research concerns the end-user development (EUD) issues in Chinese e-government projects, including how to enable end users to create programs, how to expand the scope of problem-resolving by end-user programming, and how to improve the quality of end user programming.

The Chinese style table occupies an important position in e-government application in China. Its complexity exists in the management of data, presentation format, computing rules, process of analysis and so on. EUD is deemed a promising measure to resolve these problems. But it's heavily dependent on a language that the end users can accept.

Spreadsheet language is considered to be a very successful EUD language, Excel and other products have demonstrated the success. But the limitation of a spreadsheet system is also very obvious. For example, a user must be responsible for all the problems of analysis and modeling; Excel and the like can only handle a limited number of smaller tables; the analysis and the calculation in practice often need more complex process to complete, which are all very difficult to express and execute in Excel. To this end, we broke down the spreadsheet into templates, sheet and calculations, and in accordance with the principle of separating presentation layer, logic layer and data layer from each other, we model the issue of tabular data analysis. Based on the model, we designed the EUD-enabled tabular data analysis language, which is called ESL. A formula in ESL is no longer bound with a single cell of spreadsheet, and there are several enhancements in the expression of range reference; thus, end users can use simple formulae to complete large-scale and complex calculations. At the same time, the data of tables is organized into three-dimensional array by the dimension of table, row and column (T, R, C), and with the conjunction of the business dimension, tabular data can also be organized into higher-dimensional data cube, so that the tabular data can be processed with a similar way of data warehouse.

Due to the lack of professional software engineering processes and tools, the effectiveness of EUD is of continuing concern. However, typical EUD activities are usually sporadic and small-scale, so experimental studies related to EUD effectiveness are difficult; in particular, it is very difficult to accumulate a certain amount of experimental data. This study selected the development of Web-Plugins in the social networking site (SNS) as a typical EUD activity, and conducted research on how to carry out testing activities, standardize the development process. Experimental data and the analysis show that the testing process in EUD is very important, and based on appropriate tools and environment, many measures of the traditional software engineering, quality assurance methods and techniques can be used to improve the effectiveness of the EUD, such as configuration management, testing process, and test management.

To date, research on EUD can be divided into theoretical studies, general studies and the domain-oriented research on EUD. This study is a domain-oriented research on EUD in e-government research. As far as possible, we use all possible means to provide programming ability to end users. In addition to previously mentioned techniques and methods, the study summarizes experience and the lessons learned in Chinese e-government projects.

3.3 End-User Software Development in a Scientific Organization

Mark Vigder (National Research Council, Canada)

Introduction

Scientific and engineering research organizations are often required to develop and maintain complex software systems. They use software for controlling experiments, analyzing data, and embodying research results. Because of the complexity and nature of the domain, the software engineering is often performed by scientists and engineers rather than by trained software professionals. Although the end-users have been able to produce and evolve sophisticated software applications, the lack of software engineering discipline can result in a number of problems related to the software engineering process.

In order to address these issues, we studied two different research institutes within our organization (the National Research Council (NRC) of Canada), and studied how they use software to control experiments and analyze data. Based on this study, we developed a software framework to assist end-users in developing and configuring the software used for these activities.

Background

In order to understand the end-user software engineering issues within the scientific research domain, we worked with two research institutes of NRC Canada: the Institute for Aerospace research (IAR); and the Institute for Ocean Technology (IOT). NRC-IOT performs experiments on scale models of marine structures (ship hulls, offshore oil platforms, etc.) Models are placed within a tank with different conditions of waves, ice, etc. Sensors on the model are used during the experiment to determine the models characteristics under different conditions.

The group within NRC-IAR with which we worked performed experiments on gas turbine engines. Sensors attached to the engine measured its operating characteristics under different conditions.

In both organizations, custom software was frequently developed in order to analyze data during and after data acquisition. The software development process usually took the form of finding a piece of software within the organization that was similar to the requirements, and modifying it to meet the specific needs of the current experiment. Modifications took different forms and included changing parameters within the source code or adding specialized functions for filtering data. Within NRC-IOT this work was sometimes performed by software engineers and sometimes by end-users; within NRC-IAR the software development was performed entirely by end-users.

In studying the software development within these two organizations, the following issues were noted:

- End-users were often spending significant amounts of time making changes to the software for each experiment. In many cases these changes were small and could be represented as a selection among parameters.
- Many of the activities performed by users during the data acquisition and analysis could be easily automated.
- End-users made use of complex domain knowledge when developing software. The end-users with whom we were dealing were people involved in research science. They had detailed knowledge of their domain that was often difficult to communicate to software professionals who were not trained within the domain [1].
- Tasks that were conceptually simple, such as re-running an analysis with different parameter settings, were often complex to do in practice. Much of the difficulty was due to the fact that the task required the end user to modify code. Even simple modifications were often difficult and error prone.
- Software variants proliferate in an uncontrolled manner. End-users tend to find a piece of code that most closely meets their requirements, and then clone and modify it. This is usually done in an uncontrolled manner leading to many variants of each piece of software with no configuration management.

- Reproducibility of results is important. When results are produced for publication and dissemination, it is important that it is known exactly how they were produced and any analyses can be re-performed at a later date and will give identical results. Note that although this is an important requirement, it is one that was not met by many organizations to which we talked.

A Framework for End-User Programming

The primary product of this work is a framework that allows end users to more easily tailor and control software. An application framework was developed with the following characteristics:

- Different software tools, specific to the domain, could be integrated into the framework. The tools provided services such as data analysis, report generation, etc. A wrapper could be used, if necessary, to provide a consistent interface to the end-user, and to facilitate interoperability between the different tools.
- Parameterized workflows of the organization are described using a simple textual template mechanism. The templates describe how the software tools are integrated to perform the organizational workflows. Although requiring some programming knowledge to create the templates, end-users with some programming capability were able to create the templates with minimal training.
- Automatic GUI generation from the parameterized workflows allows end-users without any programming experience to instantiate and execute workflows.
- The framework provided a set of basic services for research organizations. Plug-ins can be used to customize the framework for specific research domains.

Software engineers, not the end users of our domain, performed the actual tool integration. However, once the integration was completed, end users were able to create the workflow templates. Through a GUI automatically generated from the templates end-users instantiate the workflows providing the actual parameters to be used.

Conclusions

After deployment, a formal interview-based evaluation of the software was performed [2]. The evaluation included both end-users and professional programmers who were using and maintaining the system. The evaluation found that ease of use and reduced training time were major benefits. There was also a reduction in the number of software variants produced that reduced, though did not solve, the configuration management issues. End-users could seamlessly use different software packages that had been integrated into the framework.

References

[1] J. Segal, “When Software Engineers Met Research Scientists: A Case Study,” *Empirical Software Engineering Journal*, vol. 10, pp. 517-536, October 2005.

[2] M. R. Vigder, N. G. Vinson, J. Singer, D. Stewart, K. Mews, “Supporting the Everyday Work of Scientists: Automating Scientific Workflows,” *IEEE Software*, vol. 25, pp. 52-58, July/August, 2008.

4 Invited Talk: Using Crystal Reports: Examples of Richly Formatted Report Creation by Non-Developers

Harold Schellekens (SAP BusinessObjects, Canada)

4.1 Context

Crystal Reports is a very pervasive and popular reporting solution that provides a significant part of the BusinessObjects division of SAP's revenue. BusinessObjects is recognized as the top Business Intelligence (BI) vendor, with 14.2% of the market share and over $1 billion in revenue; 19.2% when combined with SAP BI[2].

BusinessObjects customers tell us (and show us) that they can do almost anything in Crystal Reports. However, they wish Crystal Reports were easier to learn and use, especially for non-developers. The rationale for this invited talk is to explore the synergy between making Crystal Reports easier for non-developers and the EUP theme of this workshop.

4.2 Crystal Reports

Crystal Reports is a software suite for designing interactive reports and connecting them to virtually any data source. Reports created using Crystal Reports are

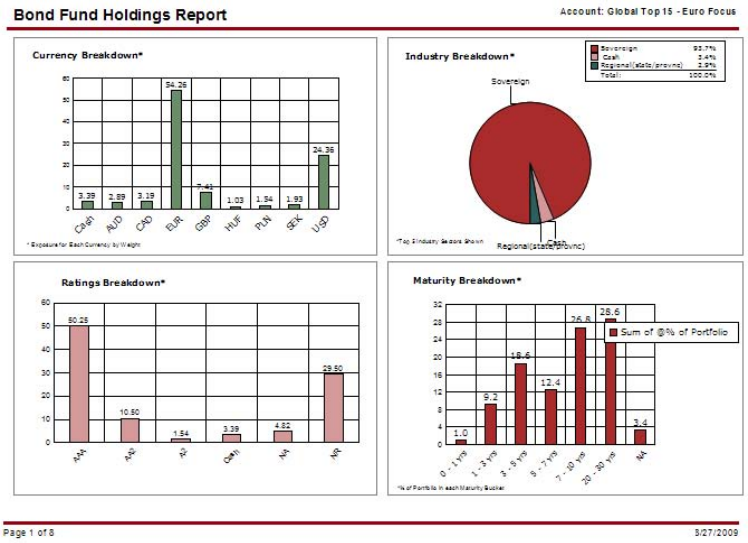
- formatted
- printable
- structured
- personalized
- scalable and high-performing
- interactive
- standalone or embedded into applications

Examples of reports created using Crystal Reports are shown in Figure 4-1. Reports are designed and created using Crystal Reports Designer and can viewed and navigated in a variety of *viewers* that are available for different platforms and technologies. Figure 4-2 shows a screenshot of the free desktop viewer.

4.3 Crystal Reports and End-User Programming

Crystal Reports provides for very rich formatting using an extensive, powerful formula language. A simple example of row banding is presented in Figure 4-3. In this example, the code on the bottom says that if a record number is odd then its background should be green, otherwise it should have no color. There are also multiple helpers for less technical end users with less programming experience.

[2] From IDC Market Analysis. 2007.



Income Statement

For the Month Ended Dec 1999

(In Thousands)

	December				QTD				YTD			
	2000 Act	%	1999 Act	%	2000 Act	%	1999 Act	%	2000 Act	%	1999 Act	%
REVENUE												
Gross Sales	$256,751	111.5 %	$241,978	111.7 %	$256,751	111.5 %	$241,978	111.7 %	$2,040,012	110.8 %	$1,952,721	111.4 %
Cash Discount	(1,039)	(0.4%)	(1,511)	(0.6%)	(1,039)	(0.4%)	(1,511)	(0.6%)	(10,312)	(0.5%)	(11,793)	(0.6%)
Trade Discount	(20,433)	(8.0%)	(18,475)	(7.6%)	(20,433)	(8.0%)	(18,475)	(7.6%)	(149,805)	(7.3%)	(140,454)	(7.2%)
Other Adjustment	(8,115)	(3.2%)	(8,383)	(3.5%)	(8,115)	(3.2%)	(8,383)	(3.5%)	(59,178)	(2.9%)	(70,460)	(3.6%)
Discounts	(29,587)	(11.5%)	(28,369)	(11.7%)	(29,587)	(11.5%)	(28,369)	(11.7%)	(219,296)	(10.8%)	(222,706)	(11.4%)
Net Sales	227,164	100.0 %	213,609	100.0 %	227,164	100.0 %	213,609	100.0 %	1,820,716	100.0 %	1,730,014	100.0 %
Net Licensee Rev	3	0.0 %	0	0.0 %	3	0.0 %	0	0.0 %	3	0.0 %	0	0.0 %
Net Revenue	$227,168	0.0 %	$213,609	0.0 %	$227,168	0.0 %	$213,609	0.0 %	$1,820,719	0.0 %	$1,730,014	0.0 %
COST OF SALES												
Cost @ Standard	$133,030	58.6 %	$125,313	58.7 %	$133,030	58.6 %	$125,313	58.7 %	$1,073,586	59.0 %	$1,023,799	59.2 %
PPV	(1,053)	(0.5%)	(1,338)	(0.6%)	(1,053)	(0.5%)	(1,338)	(0.6%)	(2,092)	(0.1%)	(5,894)	(0.3%)
Manufacturing	0	0.0 %	0	0.0 %	0	0.0 %	0	0.0 %	0	0.0 %	0	0.0 %
Net Quality Cost	3,686	1.6 %	2,473	1.2 %	3,686	1.6 %	2,473	1.2 %	11,459	0.6 %	11,872	0.7 %
Direct Ship Allw	1,162	0.5 %	985	0.5 %	1,162	0.5 %	985	0.5 %	8,150	0.5 %	7,224	0.4 %
PPV F Air Ft>Std	902	0.4 %	781	0.4 %	902	0.4 %	781	0.4 %	16,364	0.9 %	6,484	0.4 %
Royalties	2,376	1.1 %	4,046	1.9 %	2,376	1.1 %	4,046	1.9 %	20,311	1.1 %	20,028	1.2 %
Warehousg & Dist	3,457	1.5 %	2,660	1.3 %	3,457	1.5 %	2,660	1.3 %	21,230	1.2 %	21,355	1.2 %
R&D	3,330	1.5 %	3,346	1.6 %	3,330	1.5 %	3,346	1.6 %	23,314	1.3 %	24,369	1.4 %
Other COGS	1,039	0.5 %	870	0.4 %	1,039	0.5 %	870	0.4 %	14,079	0.8 %	2,398	0.1 %
Cost of Sales	$147,928	65.1 %	$139,134	65.1 %	$147,928	65.1 %	$139,134	65.1 %	$1,186,401	65.2 %	$1,111,634	64.3 %
Gross Profit	$79,239	34.9 %	$74,474	34.9 %	$79,239	34.9 %	$74,474	34.9 %	$634,319	34.8 %	$618,381	35.7 %
Pricing Margin %		50.0 %		50.0 %		50.0 %		50.0 %		50.0 %		50.0 %
GM % on Net Rev		30.0 %		30.0 %		30.0 %		30.0 %		30.0 %		40.0 %
Revenue Units		7.015		6.775		7.015		6.775		57.333		54.446
Licensee Units		4				4				4		

Figure 4-1: Examples of Reports Created Using Crystal Reports

One of Crystal Reports' goals as a product is to make the formula language easier and more accessible to people not trained formally as developers, without diminishing the power and flexibility of the product. As with any commercial product, it is difficult to have a product that satisfies every single user. What follows are some examples of decisions and tradeoffs regarding Crystal Reports features that are related to EUP.

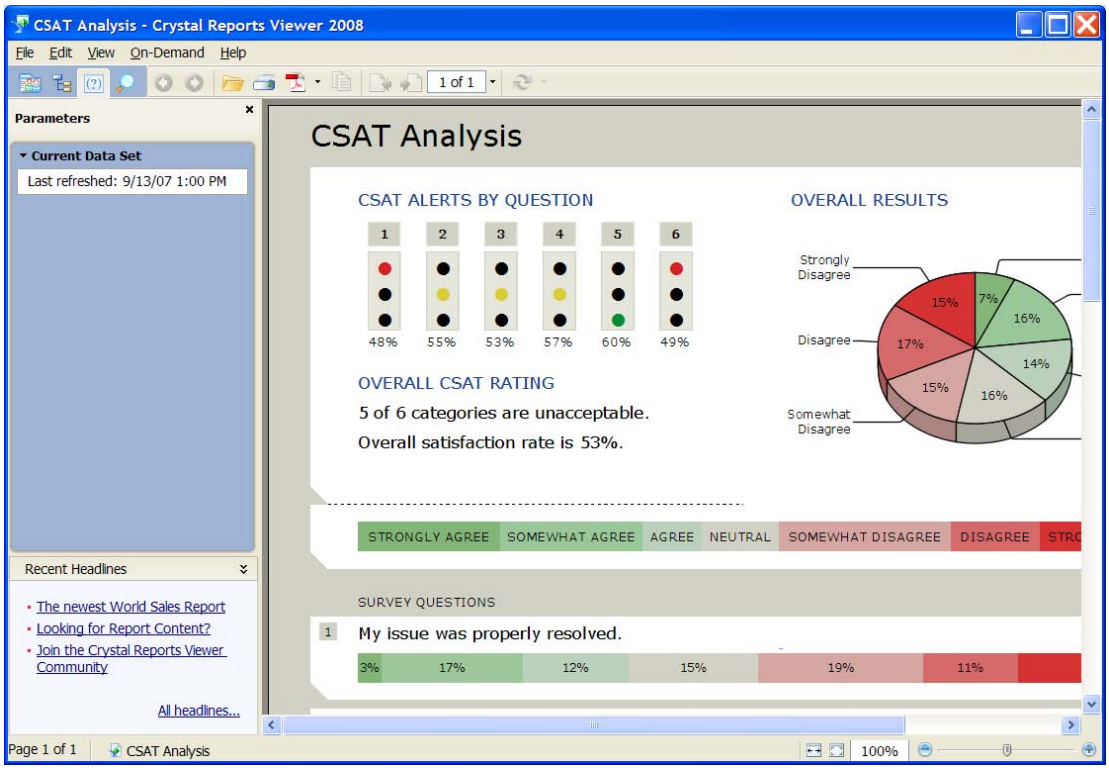


Figure 4-2: Screenshot of Crystal Reports Free Desktop Viewer

4.3.1 Formula Language vs. Helpers

As mentioned earlier, in addition to the formula language, Crystal Reports has multiple helpers for end users who are less technical. However, from a product development perspective, it would be an overwhelming task to create helpers for every formula language construct. This creates several dilemmas because the goal is to make the product easier to use without diminishing its power and flexibility.

- Should the formula language be easier or should there be more helpers in the product?
- Which features should have helpers?
- Where is the formula language necessary?
- What is the right balance?

Research to find out the answers to these questions would be of great value.

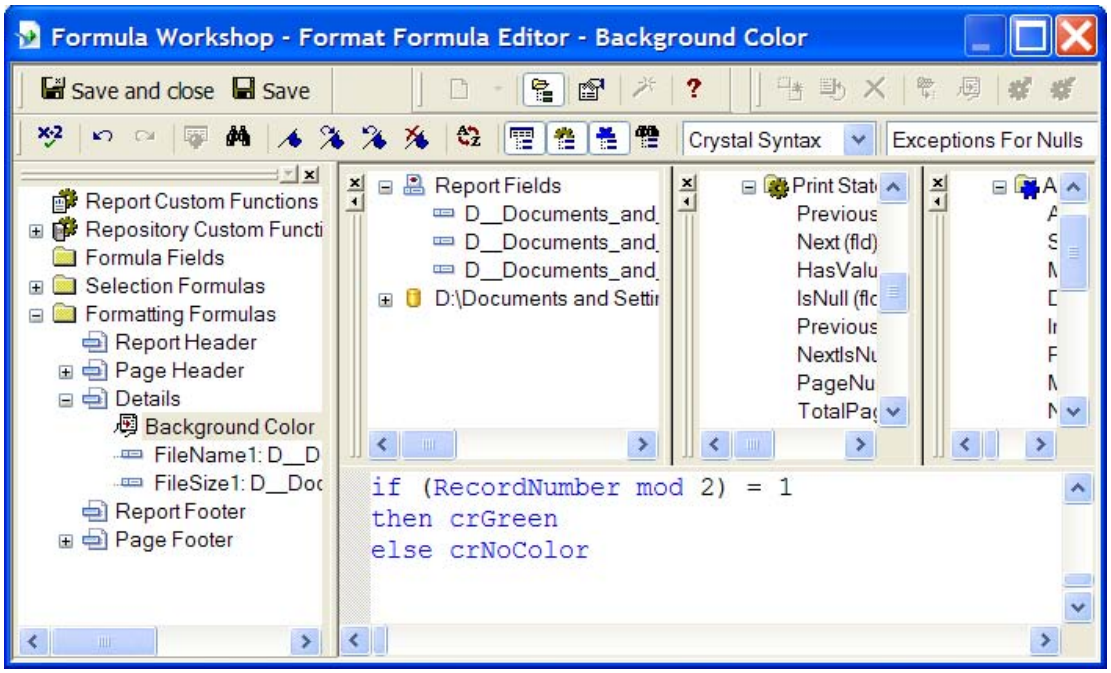


Figure 4-3: Example of Crystal Report's Formula Language

4.3.2 Data Abstraction

Another aspect that contributes to the power of Crystal Reports is that end users have direct access to data with no abstraction between the data and the person creating the report. The complexity of the source data (especially in application data) is a major problem. Semantic models could help with this problem, but someone still has to model the information.

- How could semantic models be introduced as a feature into a tool such as Crystal Reports?
- What is the learning curve on the end user?
- How are data semantics entered into the tool? How are they represented?
- What is the overhead for the data modeler?

4.3.3 Programming Best Practices

Crystal Reports has powerful features such as sub-reports, which means that there can be reports embedded inside other reports. If a sub-report is placed inside a report header, it executes once; if placed in the details, it executes once for each row. This simple explanation shows the effect that such decisions place on the performance of a report.

- How can programming best practices be introduced into tools such as Crystal Reports?
- How are users notified of potential problems?
- When does notification become an annoyance for more experienced end users?
- What is the right balance?

4.3.4 Templates

Templates are another example of a feature that could be useful for Crystal Reports users. Creating a set of templates of benefit for the greatest number of users would require studies to see

which reports are the most created by users. A simple technique could be to discover reports published on the web, find those that are the most common, and set up a template repository. This repository could be bundled with the product or be set up as an online repository.

4.4 Final Thoughts

As a commercial vendor, it is a challenge for SAP BusinessObjects to determine the features that go into each release of a product. There is always a tradeoff because more features create a bigger product that is harder to maintain and requires more resources to run.

As a representative of the developer view in this workshop, we are a community that would greatly benefit from best practices for conducting user studies and usability testing in the domain of EUP that can help us build better products.

5 Workshop Summary

The SEEUP workshop was held at the 33rd International Conference on Software Engineering (ICSE) in Vancouver, British Columbia on May 23, 2009. Twenty-three participants from both the research and industry contributed to the discussions.

The workshop was divided into five parts:

- introduction to EUP state of practice
- the developer perspective of EUP
- the end-user perspective of EUP
- discussion of selected topics
- workshop review

The following five subsections summarize the discussions on these topics. Copies of the presentations are available on the SEEUP website [14].

5.1 Introduction to EUP State of Practice

The introduction consisted of three talks that outlined the current state of the practice and set the tone for the rest of the day.

The first talk, by Brad Myers of Carnegie Mellon [14], described previous EUSE workshops and introduced the major themes of the SEEUP workshop. This talk defined the scope of EUP, traced its importance, and noted the previous work that has impacted it. Myers referred to two NSF workshops that determined the need for serious attention to the end user [2]. He cited research findings showing that while there are about 3 million professional programmers in the U. S., over 12 million people say they do programming at work, and over 55 million people use spreadsheets and databases at work and thus may also be considered to be doing programming [12]. The NSF reports that there are about 6 million scientists and engineers in the U. S., most of whom do programming as part of their jobs [10].

A problem that needs serious attention, Myers noted, arises from the pervasiveness of errors in software created by end users. These errors result in serious consequences for the people whose retirement funds, credit histories, e-business revenues, and even health and safety rely on decisions that are made based on that software. For example, a Texas oil firm lost millions of dollars in an acquisition deal because of spreadsheet errors [11]. These problems are symptoms of the increased use of shared code and shared data, the potentially poor quality of that data, the lack of appropriate discipline on in the EUP, the lack of testing, and the failure to properly address security.

EUSE is a multi-disciplinary problem needing software engineering research, programming language research, education research, end-user programming research, and HCI research of all types. Two recent large collaborative efforts, one in the U. S. (the EUSES Consortium http://eusesconsortium.org/) and the other in Europe (the Network of Excellence on End-User Development, http://giove.cnuce.cnr.it/eud-net.htm), have produced promising results in this area

(see, e.g., *End User Development* [13]). Special Interest Group meetings at CHI'2004 [5], CHI'2005 [6], CHI'2007 [8], CHI'2008 [7], and CHI'2009 [9], as well as the WEUSE series of workshops at ICSE'2005 [4], CHI'2006 [3], Dagstuhl 2007 (see www.dagstuhl.de/07081), and ICSE'2008 [1] successfully brought together researchers and companies interested in this topic.

The second talk, by Andrew Ko of the University of Washington, was on "The State of the Art in End-User Software Engineering." This talk identified the pervasiveness and importance of EUP. It reviewed how EUP is beginning to be more disciplined across the software life cycle and identified areas of continued challenge. EUP, although more informal, does face requirements, reuse, specifications, testing, and debugging problems. One primary concern is to inculcate the need to focus systematically on quality and to build more automated tools to support life cycle activities. Discussion topics included

- Requirements—End users write programs for their own use; they usually do not develop formal requirements. The drivers for changes to those programs come from the end users themselves, and often involve ways to achieve greater automation.
- Design and specification—These elements tend to be emergent, because end users do not see value in making them explicit. Techniques that have been successful in focusing greater attention on specification include designing DSLs to replace lower-level languages (e.g., Yahoo pipes), so that the specification becomes the language itself, and supporting design exploration (such as Newman's Denim).
- Reuse—End user programmers in general do not have the same cultural aversion to reuse that some professional programmers do. Reuse occurs by finding code on the web and in repositories and by customizing templates and changing APIs. There is a need to evaluate how inputs, APIs, and outputs are changed when code is reused.
- Testing—End users tend to be overconfident about testing and verification because they do not have the experience that professional programmers have in seeing the variety of ways that programs can fail. Techniques that provide immediate feedback tend to be the most effective, such as visualizing test data and automated consistency checking.
- Debugging—This function is difficult for end-user programmers. Effective techniques support automated reasoning backward from output and suggesting potential solutions.

Margaret Burnett of Oregon State University gave the third talk on "End-User Software Engineering and External Cognition" [15]. Burnett's talk started with the premise that for EUP to become more disciplined, programmers need to see a perceived payoff at a low cost. One mechanism to accomplish this is distributed cognition, which is cognition beyond the individual to encompass interactions between people, resources, materials, and environment. This talk reviewed ways to help end users recognize the perceived cost and benefits across a greater set of such interactions. Some current research in this area includes

- "What you see is what you test (WYSIWYT)": This kind of tool tracks progress of "things-to-test," finds possible culprits for errors, prioritizes them, and provides information for a programmer to make value correctness judgments.
- Debugging learned programs to allow users to ask questions of machine-learned programs (such as e-mail filters): The explanations provided by the system represent "code" that users can correct.

- Debugging via information foraging: Empirical studies are showing that information foraging is a significant activity in debugging. Current tools are providing clues on where to look for relevant information.

5.2 The Developer Perspective of EUP

Two papers were presented in this session. Xingliang Yu of the Chinese Academy of Sciences presented a paper entitled "Extending the Boundary of Spreadsheet Programming: Lessons Learned from Chinese Governmental Projects" [15]. Mark Vigder, NRC Canada, presented a paper entitled "End-User Software Development in a Scientific Organization" [15].

Xingliang Yu provided a case study of how spreadsheets are used to enable information sharing in education across multiple levels of government. The project described was to integrate different types of data and different report formats into a coherent form that meets objectives at the local, province, and national levels. The primary lessons learned are that

- Mature spreadsheet applications provide significant leverage.
- Editing for specific needs provides additional impact.
- Dictionaries of common terms enable greater interoperability.
- Codifying a language (ESL—EUD-enabled Spreadsheet Language) facilitates use by diverse end users.

Vigder's paper provided a case study of end-user scientists who develop scientific programs to support their own needs as well as the needs of their teams. These end users have limited software knowledge and there is very little software engineering support for such processes as testing, configuration management, or versioning. NRC has developed a general model of the workflow for such applications that includes such activities as converting data, selecting relevant data, and filtering and transforming data. Simple tools have been provided to support many of these tasks that involve selecting the appropriate workflow and inserting an appropriate set of parameters. This has led to the following preliminary results:

- Because a GUI is automatically generated, many "programming" operations are now done at the GUI level.
- A single version of a program is maintained and only parameters at invocation are different. This minimizes the error-prone practice of "cloning and modifying."
- The automation of tedious workflows has been simplified because of standard tool integration mechanisms and standard domain-relevant data structures.
- Off-the-shelf tools have been integrated into workflows.

5.3 The End-User Perspective of EUP

Harold Schellekens of SAP BusinessObjects, presented an invited talk on "Using Crystal Reports: Examples of Richly-Formatted Report Creation by Non-Developers" [14]. The Crystal Reports tool enables business users to develop highly adaptive, customized reports from multiple data sources. Based on customer input, SAP is simplifying the Crystal Reports designer tool to make it more accessible for people who are not professional software developers. The simplifying is taking the form of more buttons for formulas, better context-sensitive help, analysis of usage data from web accesses to understand the types of reports that are most commonly used, and a greater

number of templates for development. Other areas of research and improvement include developing more wizards to make the use of formulas more accessible, analysis of more advanced workflows, design based on experience of the report designer, and additional usability testing.

5.4 Discussion of Selected Topics

The talks on the state of EUO, the developer perspective, and the end-user perspective led to a significant amount of discussion. Because of the variety of interests, attendees were asked to brainstorm on topics they would like to discuss in more detail as part of the workshop. The full list of topics, in no particular order, is

- critical steps for instilling software engineering discipline in the scientific community
- experiences and examples of multi-user creation of end-user engineered software
- how to effectively advocate for the "goodness" or "first-class citizenship" of EUSE as part of software engineering
- implications of some end-user programmed software being part of a "human-mediated" service versus other software being treated more as a product
- prevalence and implications of "programming by example" in EUP
- how to shape EUP frameworks to produce better software
  - automated software quality measurement
  - heuristics to improve user awareness
- enabling and motivating better specification of end-user needs in EUP-created software products
- inherent difficulty in EUP for different products
- open research questions in EUSE
- the joy of problem solving and its implications for EUSE
- lessons learned from EUSE for professional software engineers
- implications of distributed cognition for EUSE

From this list, the participants selected two topics for more detailed discussion: (1) multi-user creation of end-user engineered software and (2) how to shape EUP frameworks to produce better software. These discussions are summarized in the following sections.

5.4.1 Multi-User Creation of End-User Engineered Software

Pervasive (or ubiquitous) computing has triggered the idea of multiple people contributing to an end-user program. EUP emerged with a single-user focus, but there is an increasing trend toward multi-user creation. For example, an assisted-living program may monitor a set of patients and prompt for follow-up with house calls in case of emergencies. Later, a different set of users might develop a number of variations to this program. As those changes make the program more complex, challenges of configuration management, stakeholder conflict and control, social and legal concerns such as privacy, and quality attributes such as safety and security need to be considered. Multiple end-user creation can also be seen in collaborative prototyping, Wiki creation, cooperative tailoring, and search tool tailoring.

Challenges of multi-user EUP include

- understanding the current state of a program
- the nature of the relationship between end users
- traceability of who made specific changes
- training
- greater need for validation
- solving conflicts and clashes in multi-user creation environments
- more languages for collaborative environments that could exploit computation

Tools are needed to supporting these collaborative environments in the areas of

- versioning, to indicate who made changes and why they were made
- collaboration, for requirements management

These tools need to be focused on intuitive functionality because end users do not have professional software engineering backgrounds.

5.4.2 How to Shape EUP Frameworks to Produce Better Software

The discussion on shaping the overall development framework revolved around the question of whether development frameworks can be controlled and constrained.

EUP poses a significant challenge because it is very domain specific. For example, the needs of the scientific users reported by Vigder differ from those of the spreadsheet users reported by Xingliang Yu and from those the business users that Crystal Reports targets. Empirical and descriptive studies, such as the ones reported by Margaret Burnett, help with understanding how end users develop software in specific domains.

As a counter to the domain-specificity of EUP, workshop participants suggested the identification of principles within domains that can be generalized across domains through techniques such as parameterization. One tool mentioned for capturing common interactive behaviors that do not need to be re-coded by each programmer is Adobe Flash Catalyst to [16]. Another possible answer discussed is the customization of domain-specific methods and languages.

An option discussed for shaping EUP frameworks to produce better software is to have gain more knowledge about what people need to do and to automate this behavior as much as possible. A problem noted is that most tools relying on automation make assumptions not often confirmed by reality.

Discussion of automation led into the social side of EUP: If too much is automated, the joy of problem solving goes away. An ideal mix would be to let end users think about what really matters (where they can be creative) and use automation to relieve them of having to think about the obvious via automation. From a tool perspective, this notion translates into determining the appropriate level of expressiveness in languages and tools.

5.5 Workshop Review

Grace Lewis from Software Engineering Institute led a discussion of the major workshop themes and conclusions [14]. An overall conclusion is that EUP accounts for a significant amount of activity. Furthermore, because its impact is growing, the need for instilling software engineering discipline into EUP is reaching a critical stage.

Common questions and themes that emerged throughout the day included

- EUP targets: who are the right people to target as EUP adopters? When does it not make sense to introduce EUP?
- Adoption: How to get end users to use tools?
- Tool characteristics: What tool characteristics appeal to end users in a specific domain? Are there common aspects of tools in different domains that can be generalized?
- Adaptation of tools: Tools are tailored for different levels of expertise—what is the delta between levels of expertise? How do we reduce the delta? How does the delta change by introducing EUP?
- What are the points along the continuum from professional programming to EUP?
- Will skills change when the prevalent demographic groups become "digital natives?"
- How can work from other fields be better integrated with EUP?
- How can we enable end users to more effectively do their activities without requiring a four-year degree?
- What are appropriate processes for different types of EUP situations?

5.6 Next Steps

The workshop concluded that EUSE is a multi-disciplinary problem needing software engineering research, programming language research, education research, end-user programming research, and HCI research. Based on the workshop discussions, there was a consensus that the next edition of WEUSE should be planned for 2010 at an appropriate venue.

References

[1] R. Abraham, M. Burnett, and M. Shaw, eds., *Proceedings of the Fourth Workshop on End-User Software Engineering (WEUSE IV)*. New York: ACM, 2008.

[2] B. Boehm, and V. Basili, "Gaining intellectual control of software development," *Computer* vol. 33, no. 5, pp. 27-33, 2000.

[3] M. M. Burnett, et al., "The Next Step: From End-User Programming to End-User Software Engineering (WEUSE II)," *CHI'2006 extended abstracts on Human factors in computing systems*, Montreal, Canada, pp. 1699-1702, Apr. 2006.

[4] S. Elbaum and G. Rothermel, eds. *Proceedings of the First Workshop on End-User Software Engineering: WEUSE 2005*. http://www.cse.unl.edu/~grother/weuse/weuse-proceedings.pdf (accessed November 15, 2009).

[5] B. A. Myers, and M. Burnett, "End-Users Creating Effective Software (Special Interest Group Meeting Abstract)," *Extended Abstracts of the 2004 Conference on Human Factors in Computing Systems, CHI'2004*, Vienna, Austria, pp. 1592-1593, Apr. 2004.

[6] B. A. Myers, M. Burnett, and M. B. Rosson, "End Users Creating Effective Software. (Special Interest Group)," *Extended Abstracts Proceedings of the 2005 Conference on Human Factors in Computing Systems, CHI'2005*, Portland, OR (USA), pp. 2047-2048, Apr. 2005.

[7] B. A. Myers, et al., "End User Software Engineering: CHI'2008 Special Interest Group Meeting," *Extended Abstracts Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI'2008*, Florence, Italy, pp. 2371-2374, Apr. 2008.

[8] B. A. Myers, et al. "End User Software Engineering: CHI'2007 Special Interest Group Meeting," *Extended Abstracts Proceedings of the 2007 Conference on Human Factors in Computing Systems, CHI'2007*, San Jose, CA (USA), pp. 2125-2128, Apr. 2007.

[9] B. A. Myers, et al., "End User Software Engineering: CHI'2009 Special Interest Group Meeting," *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI'2009, Extended Abstracts Volume*, Boston, MA (USA), pp. 2731-2734, Apr. 2009.

[10] National Science Board, Science and Engineering Indicators 2006. National Science Foundation volume 1: NSB 06-01; volume 2: NSB 06-01A, 2006. Arlington, VA. http://www.nsf.gov/statistics/seind06/ (accessed November 15, 2009).

[11] R. Panko, "Finding spreadsheet errors: Most spreadsheet models have design flaws that may lead to long-term miscalculation," *Information Week*, pp. 100, May 29, 1995.

[12] C. Scaffidi, M. Shaw, and B. Myers, "Estimating the Numbers of End Users and End User Programmers," *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, Dallas, TX (USA), pp. 207-214, Sept. 2005.

[13] H. Lieberman, F. Paterno, and V. Wulf eds., *End-User Development*. 2006, Springer: Dordrecht, The Netherlands.

[14] International Conference on Software Engineering. "Workshop on Software Engineering Foundations for End-User Programming (EUP)," *International Conference on Software Engineering*, 2009 [Online]. Available: http://www.sei.cmu.edu/interoperability/research/approaches/seeup2009.cfm ([accessed: October 31, 2009).

[15] Grace Lewis, Dennis Smith, Len Bass, and Brad Myers, "Report of the Workshop on Software Engineering Foundations for End-User Programming," *ACM SIGSOFT Software Engineering Notes*, vol. 34, pp. 51-54, Sept. 2009.

[16] Adobe Labs, "Adobe Flash Catalyst," *Adobe Labs*, 2009. [Online]. Available: http://labs.adobe.com/technologies/flashcatalyst/ [accessed. October 31, 2009].

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	November 2009	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Proceedings of the Workshop on Software Engineering Foundations for End-User Programming (SEEUP 2009)	FA8721-05-C-0003

6. AUTHOR(S)

Len Bass, Grace A. Lewis, Brad Myers, and Dennis B. Smith

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2009-SR-015

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)

The Workshop on Software Engineering Foundations for End-User Programming (SEEUP) was held at the 31st International Conference on Software Engineering (ICSE) in Vancouver, British Columbia on May 23, 2009.

This workshop discussed end-user programming with a specific focus on the software engineering that is required to make it a more disciplined process, while still hiding the complexities of greater discipline from the end user. Speakers covered how to understand the problems and needs of the real end users of end-user programming. The discussion focused on the software engineering and supporting technology that would have to be in place to address these problems and needs.

14. SUBJECT TERMS	15. NUMBER OF PAGES
EUP, EUSE, end-user programming, end-user software engineering, SEEUP	46

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