

The Next Step: From End-User Programming to End-User Software Engineering



WEUSE II Workshop at CHI 2006
Montréal, Quebec, Canada, April 23, 2006
Hyatt Regency Hotel
9:00 am – 6:00 pm

Organized by:

Margaret Burnett, Oregon State University

Brad Myers, Carnegie Mellon University

Mary Beth Rosson, Pennsylvania State University

Susan Wiedenbeck, Drexel University

Extended Abstract:

The Next Step: From End-User Programming to End-User Software Engineering

Margaret Burnett

Elec. Engr. & Computer Science
Oregon State University
Corvallis, OR 97331 USA
burnett@eecs.oregonstate.edu

Mary Beth Rosson

Information Sciences & Technology
Pennsylvania State University
University State College, PA 16802
mrosson@psu.edu

Brad Myers

Human-Computer Interaction Inst.
Carnegie Mellon University
Pittsburgh, PA 15213 USA
bam@cs.cmu.edu

Susan Wiedenbeck

Information Science & Technology
Drexel University
Philadelphia, PA 19104 USA
Susan.Wiedenbeck@cis.drexel.edu

Abstract

Is it possible to bring the benefits of rigorous software engineering methodologies to end users? End users create software when they use spreadsheet systems, web authoring tools and graphical languages, when they write educational simulations, spreadsheets, and dynamic e-business web applications. Unfortunately, however, errors are pervasive in end-user software, and the resulting impact is sometimes enormous. A growing number of researchers and developers are working on ways to make the software created by end users more reliable. This workshop brings together researchers who are addressing this topic with industry representatives who are deploying end-user programming applications, to facilitate sharing of real-world problems and solutions.

Keywords

End-User Software Engineering, Testing, Empirical Studies of Programming, Psychology of Programming, Programming by Demonstration.

ACM Classification Keywords

D.2.5 Testing and Debugging; H.1.2 User/Machine Systems—*Software psychology.*

Copyright is held by the author/owner(s).

CHI 2006, April 22–27, 2006, Montreal, Canada.

ACM 1-xxxxxxxxxxxxxxxxxx.

Introduction

There has been considerable work in empowering end users to be able to write their own programs, and as a result, users are indeed doing so. The “programming” systems used by these end users include spreadsheet systems, web authoring tools, and graphical languages for demonstrating the desired behavior of educational simulations. Using such systems, end users create software, in forms such as educational simulations, spreadsheets, and dynamic e-business web applications.

Unfortunately, however, errors are pervasive in this software, and the resulting impact is sometimes enormous. When the software is not dependable, there can be serious consequences for the people whose retirement funds, credit histories, e-business revenues, and even health and safety rely on decisions made based on that software. Such problems are ubiquitous in spreadsheets [6], open resource coalitions [7] and dynamic web applications [8]. Two recent NSF workshops have determined that end-user software is in need of serious attention [1].

Researchers have begun to join together into a subarea of “end-user software engineering,” to develop and investigate technologies aimed at this problem. We have already demonstrated some interesting progress in tools and techniques in this area.

Special interest group (SIG) meetings at CHI’04 and CHI’05 have successfully brought together these researchers with many others in the CHI community who are concerned about the user interfaces and reliability of software and software tools. At CHI’06, this second **Workshop on End-User Software Engineering**

(WEUSE II) builds upon the interest expressed by these participants and those who attended WEUSE I at the ICSE’05 conference. We plan to organize follow-up events (WEUSE III, ...) at future CHI, ICSE, and related venues as well.

Example Technologies

There is a tremendous range of technologies that can be brought to bear on this problem. This section highlights a number that are being developed by the organizers and their collaborators in the EUSES Consortium¹, and we expect to find out about others at the workshop.

Traditional methods and tools for addressing software development and dependability problems for professional programmers are usually not suitable for end-user programmers. Rather, we envision systems that create software in collaboration with those users, in a software development paradigm that combines traditionally separate functions – blending specification, design, implementation, component integration, debugging, testing, and maintenance into tightly integrated, highly interactive environments. These environments employ new, incremental, feedback devices supported by analysis and inferential reasoning to help the user reason about the dependability of their software as they work with it, in a manner that respects the user’s problem-solving directions to an

¹ The EUSES Consortium (End Users Shaping Effective Software) consists of researchers from Oregon State University, Carnegie Mellon University, Drexel University, Pennsylvania State University, University of Nebraska, and Cambridge University. See <http://eusesconsortium.org>.

extent unprecedented in existing software development environments.

The *End-User Software Engineering* project at Oregon State University aims to improve the reliability of software produced by end-user programmers in general, and by spreadsheet users in particular. Some results have included “What You See Is What You Test” (WYSIWYT) integrated with fault localization and with assertions for end-user programmers [2], and semi-automated detection of erroneous combinations of units in spreadsheets [3]. A recent emphasis has been on how to interest users in end-user software engineering devices without detrimentally interrupting their problem-solving efforts [9].

The *Natural Programming Project* at Carnegie Mellon University is investigating a variety of techniques around the idea of applying computer-human interaction principles to the design of programming languages and environments. In 2004, we reported on the “WhyLine,” a debugging tool that helped end users find bugs in 1/8 the time, and increased programmer productivity by about 40% [5]. Current work is looking at more effective tools for supporting the editing and construction of code [4] and for users’ investigations of new SDKs.

Penn State researchers in the *Informal Learning in Software Construction* project are studying real world situations and communities that can motivate and aid non-programmers in learning and using end-user programming tools. Prior work characterized the problems of public school teachers learning to build visual simulations, and designed minimalist training materials and reusable code to serve as scaffolding

[11]. Recent research is studying the mental models of web software construction held by sophisticated end users, and is using these results to develop a tool for building simple web applications [10].

Researchers at Drexel University are studying cognitive and social factors that may affect end users’ acceptance of end-user programming tools and their effectiveness in using them. Research on school teachers has investigated strategies that teachers use in programming [13] and has identified facilitators and inhibitors to end-user programming in the school setting [12]. Current research in collaboration with researchers at Oregon State University is focusing on the effect of culture and gender on success in end-user programming.

Researchers in end-user software engineering are working on a variety of other approaches as well. Among them are new surveys of end-user programmers in real organizations, fault detection through statistical methods and through program analysis, pedagogical methods to encourage a quality-control culture for users of technology, and motivational and attention allocation issues for end-user programmers.

Workshop Goals

The workshop’s goals are: (1) to generally share information and raise awareness among researchers already in this area with researchers in the related areas of Empirical Studies of Programming and Psychology of Programming, and with practitioners interested in current and future techniques that can be embodied in tools and development processes; and (2) to concretely match end-user software engineering

problems in industry with potential solutions drawn from new and emerging research findings. One outcome of the first goal, in addition to shared knowledge, will be the groundwork for a new collaborative effort, involving interested attendees at the workshop, for a survey paper on the state of end-user software engineering research. At the CHI'04 and CHI'05 SIGs and ICSE'05 workshop, initial categorizations of existing research and the problem space began to emerge, and these will form as a starting point for this workshop.

We hope to also make one or more matches resulting in future collaborations that apply research findings to problems that industrial participants would like to solve.

References

- [1] Boehm, B. and Basili, V., "Gaining Intellectual Control of Software Development." *Computer*, 2000. 33(5): pp. 27-33.
- [2] Burnett, M., Cook, C., and Rothermel, G., "End-User Software Engineering," *Communications of the ACM*, 2004. 47(9): pp. 53-58.
- [3] Erwig, M. and Burnett, M. "Adding Apples and Oranges," *Fourth International Symposium on Practical Aspects of Declarative Languages*. 2002.
- [4] Ko, A.J., Aung, H.H., and Myers, B.A. "Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing," *Extended Abstracts CHI'2005: Human Factors in Computing Systems*. Portland, OR, April 2-7, 2005. pp. 1557-1560.
- [5] Ko, A.J. and Myers, B.A. "Designing the Whyline, a Debugging Interface for Asking Why and Why Not Questions About Runtime Failures," *CHI'2004: Human Factors in Computing Systems*. 2004. Vienna, Austria: pp. 151-158.
- [6] Panko, R., "Finding Spreadsheet Errors: Most Spreadsheet Models Have Design Flaws That May Lead to Long-Term Miscalculation." *Information Week*, 1995. p. 100.
- [7] Raz, O. and Shaw, M. "An Approach to Preserving Sufficient Correctness in Open Resource Coalitions," *10th International Workshop on Software Specification and Design*. 2000.
- [8] Ricca, F. and Tonella, P. "Analysis and Testing of Web Applications," *International Conference on Software Engineering*. 2001. pp. 25-34.
- [9] Robertson, T., Prabhakararao, S., Burnett, M., Cook, C., Ruthruff, J., Beckwith, L., and Phalgune, A. "Impact of Interruption Style on End-User Debugging," *CHI 2004: Human Factors in Computing Systems*. 2004. Vienna, Austria: pp. 287-294.
- [10] Rode, J. and Rosson, M.B. "Programming at Runtime: Requirements and Paradigms for Nonprogrammer Web Application Development," *IEEE Symposium on Human-Centric Computing Languages and Environments*. 2003.
- [11] Rosson, M.B. and Seals, C. "Teachers as Simulation Programmers: Minimalist Learning and Reuse," *CHI'2001: Human Factors in Computing Systems*. 2001. Seattle, WA: pp. 237-244. .
- [12] Wiedenbeck, S. "Facilitators and inhibitors of end-user development by teachers in a school environment." *IEEE Symposia on Visual Languages and Human-Centric Computing*, 2005, pp. 215-222.
- [13] Wiedenbeck, S. and Engebretson, A. "Comprehension strategies of end-user programmers in an event driven application." *IEEE Symposia on Visual Languages and Human-Centric Computing*, 2004, pp. 207-214.

The Next Step: From End-User Programming to End-User Software Engineering



WEUSE II Workshop at CHI 2006, Montréal, Quebec, Canada, April 23, 2006

**Hyatt Regency Hotel
9:00 am - 6:00 pm**

Online Proceedings

Papers:

End-User Software Engineering in the Real World:

1. Improving the Quality of Contributed Software on the MATLAB File Exchange
Ned Gulley
2. GE Healthcare Integrated IT Solutions, Centricity
Erika Orrick
3. Adobe/Macromedia Flash
Jen deHaan
4. End-User Software Engineering for System Administrators
Allen Cypher, Eben Haber, Eser Kandogan

End-User Software Engineering Research:

5. End-User Development in Small and Medium Enterprises: Research and Development Issues
Matthias Betz, Jan Heß, Volkmar Pipek, Markus Rohde, Volker Wulf
6. Gender in Domestic Programming: from Bricolage to Séances d'Essayage
Alan F. Blackwell
7. Games Programs Play: Obstacles to Data Reuse
Chris Scaffidi, Mary Shaw, Brad Myers
8. End User Software Engineering: Auditing the Invisible
Joshua B. Gross
9. End-User Programming Productivity Tools
Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Jeffrey Stylos

10. Toward Sharing Reasoning to Improve Fault Localization in Spreadsheets
Joseph Lawrance, Margaret Burnett, Robin Abraham and Martin Erwig
11. End-User Software Engineering in Natural Language
Henry Lieberman, Hugo Liu, Ying Li
12. Abstractions for End Users
Michael Toomim
13. End-User Programming at the University of Washington
Daniel S. Weld, Pedro Domingos, Raphael Hoffman, Sumit Sanghai

CHI 2006 Montreal
WEUSE workshop (<http://eusesconsortium.org/weuse/>)
April 23, 2006

Position paper by Ned Gulley

Improving the Quality of Contributed Software on the MATLAB File Exchange

MATLAB (which is incidentally one of the languages mentioned in the Call for Participation for this workshop) is a technical computing language that enjoys wide usage among scientists and engineers around the world. Typically, these people are not trained as programmers, and they almost never describe what they are doing as programming. They would say they are simply problem-solving, and they are often under great pressure to produce results quickly. As a result, MATLAB is often used to arrive at "quick and dirty" solutions. This kind of usage is emblematic of end-user programming.

In order to help our user community, we at The MathWorks have created a programming archive called the MATLAB Central File Exchange (<http://www.mathworks.com/matlabcentral/fileexchange/>), where people can freely upload and download MATLAB programs (known as M-files). After four years of operation, there are now 4057 files in 20 top level categories available for free download. Files are added at the rate of 100 per month, and file downloads regularly exceed 7500 per day. We have therefore succeeded at the most important part of building a community site: drawing a crowd.

Unfortunately, many of these files are poorly written (uncommented spaghetti code) or poorly motivated (homework problems of no general interest). We are always looking at ways to improve the quality of the code we host on our site. But how does one best go about this? The fundamental problem we face with this site is how to improve the overall value of this site without falling into the following traps:

- low barriers to participation lead to a proliferation of worthless code
- high barriers to participation drive away people away and impoverish the site
- high cost of maintenance leads to too much work for us

Over the past four years, we have generated a great deal of real-world data and hands-on experience about how to run a code repository. In the context of how to improve the value of the site, I plan on discussing the tradeoffs associated with

- comments for files
- numeric ratings for files (1-5, where 1 = "nominate for deletion")
- download counters
- metrics and reports pages
- author reputation ranking systems
- submission guidelines: high and low barriers
- "collaborate with me" flags
- "Pick of the Week" blog
- wiki-mediated file review team made up of community members

- removal of low quality submissions

For each case above, we have real data on what happened in response to various experiments we have tried.

As we grow, we need to work with our users to see that the File Exchange is meeting their needs. Do people want the File Exchange to eventually be like Boost, a vetted, high-quality library of code? Do we want to make SourceForge our model, in which we would provide development environments to many people? Or do we simply want to give people a place to put heaps of code and let a consensual voting process determine what's good? Finally, would we ever want to provide a File Exchange-in-a-Box product to those companies who are prevented from sharing their code publicly?

My background

Originally coming from a background in aerospace engineering and aircraft control design, I have been working as a software developer for The MathWorks since 1991. Since 2001 I have been leading the MATLAB Central web community team. We are continually innovating to provide our customers with a vibrant, valuable, and personally rewarding resource for developing MATLAB-based code.

Ned Gulley
gulley@mathworks.com
January 13, 2006

Erika Orrick
User-Centered Design Engineer
GE Healthcare Integrated IT Solutions, Centricity Practice Solutions

Position Paper for the CHI 2006 Workshop on End-User Software Engineering

With the recent federal government push towards pay for performance and other initiatives that lend themselves to eHealthcare, physician interest in electronic medical record (EMR) systems is growing. Currently, less than 20% of ambulatory physician clinics use any type of EMR system. One of the biggest obstacles to adoption is clinician resistance to an interruption in their normal routine. For example, many of the physicians currently practicing were taught to document a patient visit using a “SOAP” (Subjective, Objective, Assessment, Plan) note. Physicians expect to document these observations in longhand, and, more importantly, want to be able to read them back in longhand. Many feel a computer will not be able to accommodate this. In an attempt to address this, MedicalLogic (now part of GE Healthcare) developed a markup/programming language, Medical Expression Language (MEL) that allows users to develop clinical content forms that gather input using standard form elements and generate output in a number of formats including bulleted lists and longhand.

GE Healthcare provides a number of clinical content forms to customers when they purchase the Centricity Physician Office EMR product. The specific forms provided is currently undergoing some revision, but, in general, all customers are provided a generic set of forms that will be used in most practices. These forms include those for recording vital signs, patient histories, etc. Additionally, we produce in-house and resell specialty and condition-specific forms including dermatology and diabetes management as two examples. Each of these forms provides a combination of point-and-click, free text entry, and voice-activated entry for clinicians to document a patient’s condition. The information from these clinical content forms is stored in a database, where it can be referenced more easily than a traditional paper chart, both on an individual patient basis and in aggregate.

Although we sell many forms for customer use, many clinics choose to build and/or customize their own. To accommodate this, we have built an Encounter Form Editor that allows the user to place form elements and write custom MEL functions to gather input and generate output in exactly the way their practice prefers. Unfortunately, our tool has not been substantially updated in several revisions. It lacks the ability for the user to visualize the form they are working on without having to load it into the actual EMR system. Placing and editing individual form elements is an unnecessarily complex process that does not allow the user to see all properties of the element at once. We also do not provide any guidance with common MEL queries that we find many clinicians’ offices use, even though we have easy access to this information on a well-used mailing list. Debugging a MEL function often requires switching between the Editor and the EMR program multiple times. If selected for this workshop, I will be able to bring our Encounter Form Editor as well as several clinical content forms to demonstrate some of these issues.

Information overload and usability on the clinical content forms is a key priority for GE Healthcare this year both for our in-house forms and for those that are developed by our customers and VARs. I think there is a lot to be examined in the tools and processes we provide to them to determine how much of the poor usability of these forms is the end-user programming tools we provide and how much is lack of usability knowledge on the part of the user. There is a great deal to be gained in patient safety and clinician efficiency with the use of EMR systems, but not if we cannot reliably enter the data.

Jen deHaan
Sr Technical Writer
Adobe Systems Incorporated
601 Townsend St, MB#269
San Francisco, CA 94103 USA
415.832.7443
jdehaan@adobe.com

I am a Sr. Technical Writer at Adobe, working with the Flash team to create documentation and instructional media for our software releases. The following outline describes my background in software and education, and the current challenges our team faces with helping our customers learn how to use Flash and write ActionScript.

I have a BFA in developmental art (art education) from the University of Calgary, where I focused on both learning how to teach art to a variety of individuals, such as children and challenged students. After university, I attended Vancouver Film School and graduated with top honors in New Media. The course focused on using software to create web, video, audio, 3d, and animated content.

Before graduating from Vancouver Film School, I began writing third-party technical books for large publishers such as Macromedia Press and Wiley, primarily on Macromedia Flash. To date, I have authored or co-authored over a dozen books, and contributed to and technically edited many others. I have also written on Macromedia ColdFusion and Dreamweaver, digital video, and Adobe Creative Suite. I consider my teaching experience and education beneficial to writing these technical publications, and the documentation I write today.

While living in Canada, I ran a small freelance web design and development business specializing in designing and creating Flash content. I also ran a local Macromedia User Group, which held meetings to discuss using software, design, and web development.

Macromedia (now Adobe) hired me in October 2004 because of my experience creating Flash content and writing technical books. I am currently the lead writer for ActionScript content, which means that I design, develop, write, technically review, and oversee the completion of large sets of documentation. I continually provide technical feedback on other sections of Flash and ActionScript documentation, to improve both the technical accuracy and usability of our documentation for our target audiences.

In addition to documentation, I also create sample applications for Flash, help moderate the LiveDocs web site that lets users comment on documentation (<http://livedocs.macromedia.com>), participate with beta software testing, run a web log (<http://weblogs.macromedia.com/dehaan>), and write articles for the Macromedia/Adobe web site (<http://www.macromedia.com/devnet>). All of these activities lead to regular customer interaction, which allows me to gather feedback about how we can improve documentation, what resources users need to learn Flash, and determine what parts of the documentation or a tutorial leads to user difficulties.

Outside of work, I continue to run my Flash forum (<http://www.flash8forums.com>), where I can help users and gather more feedback about the software. One reason I run the forum is to learn more about our users to improve the

documentation and instructional media. The forum is excellent at helping me determine what difficulties Flash users face, the kinds of applications they build, common questions, and figure out different ways people learn Flash.

Flash has always been difficult to learn; it has a steep learning curve, robust programming language, and complex user interface. Users must first figure out how to use the authoring tool, which involves a complex workflow, the concept of a timeline, and many other features. Users also face a programming language (ActionScript) to accomplish many tasks, which has quirks and changes with each Flash Player update. Our documentation team must keep on top of the many changes in the authoring tool and the sometimes revolutionary changes in ActionScript.

In addition to this, Flash has a variety of users that range from artistic designers to enterprise-level developers. The documentation team needs to create instructional media for this wide audience, who might have novice to highly developed skills in Flash. Such a wide audience means that our team has complex decisions to make when creating and targeting our media. For example, the documentation needs to remember that many designers are terrified of programming, but will often need to do so to meet their goals. Similarly, application programmers might need to use design tools. In both use cases, readers are often wary, frightened, and insecure and we need to accommodate their needs adequately.

Flash is a part of my daily life, and the people who use Flash are regularly a part of both my job and my free time. My interaction with our users helps me improve my skills and knowledge for teaching Flash to our customers through better documentation, sample files, and tutorials. I hope that this insight might be useful at your workshop, which sounds very interesting and valuable to me as a documentation writer and creator of instructional media.

Regards,
Jen deHaan

Jen deHaan
Sr Technical Writer
Adobe Systems Incorporated
601 Townsend St, MB#269
San Francisco, CA 94103 USA
415.832.7443
jdehaan@adobe.com

~~

End-User Software Engineering for System Administrators

Allen Cypher, Eben Haber, Eser Kandogan
User Experience Research Group
IBM Almaden Research Center
January 13, 2006

Our group at IBM Almaden has been studying system administration work practices and tools for the past three years. System administrators maintain the IT infrastructure on which our society depends. We conducted 14 ethnographic field studies at six different sites, both inside and outside IBM, through naturalistic observation, interviews, and surveys. In the course of these studies we found end-user programming to be pervasive throughout system administration work.

We observed end-user programming to be an important tool for system administrators for 1) monitoring systems and 2) automating important tasks. End-user monitoring tools are needed due to the complex and idiosyncratic nature of the systems being managed; systems typically comprise many components from different vendors, and off-the-shelf tools do not provide the scope or detail needed for a given installation. For example, at a database site we observed a locally-created set of Perl scripts that generate web pages which continually display a custom “dashboard” of all the aspects of database performance needed by the administrators. A more transient example was seen with a group of administrators debugging a problem involving the interactions of a web server with a web application server and a database. No tool existed for continually reporting the connections to the web server, so the group of administrators worked together for about 40 minutes to create one.

End-user programming is also used to automate important tasks. System administration tasks frequently involve complicated command-line commands and many steps, making them amenable to automation through scripting. The vast majority of administrators we observed used small scripts for executing common tasks. We also occasionally saw larger, shared tools for automating tasks, such as a database site where the “crontab” file contained a long list of common database maintenance commands, all commented out. When a command needed to be run, the administrator would uncomment the command, and it would then be run automatically.

End-User Software Engineering

Our studies have pointed to a critical need for Testability and Collaboration in end-user programming. These capabilities are currently well-supported in professional software engineering environments, but not for end-user programming.

Testability

System Administrators are responsible for the reliable operation of the computer systems on which businesses depend worldwide, and testability of the scripts they use is critical for ensuring reliability. One example of a deficiency in testability comes from some database administrators we observed preparing to perform a crucial operation during a limited time window. To get ready, they performed the same operation on a series of increasingly complex test systems. Part of the operation involved database scripts that needed to be customized for each test system. During our observations, an error crept in while the script was being edited. The database, however, had no way to verify a script’s syntax or semantics

without running the script. When the script was run and the error reported, it left the database in an unpredictable state.

Since sysadmins frequently write programs under time pressure, and need to produce working programs quickly, an integrated testing environment that immediately and interactively verified that a program was working would be of considerable value.

Collaboration

Although traditionally there has been some collaboration in end-user programming – such as the sharing of spreadsheets and VBScripts – we have seen how system administrators have needs for collaboration that far exceed current capabilities. Although sysadmins have considerable technical knowledge, they generally lack software engineering training. Access to collaborators could enhance their scripting abilities – and the robustness of their scripts – through scaffolding. For example, in a web hosting service installation we observed web administrators having to wait for several hours before they could execute their scripts for configuring a web application server. They had to wait for the database administrators, who were able to run the scripts that set up database tables.

We have also observed another need for collaboration: sysadmins sometimes share a programming task – with more than one person working on the program – and this leads to multiple versions of the program. In such a situation, a web-based environment could facilitate sharing, uniformity, and a common understanding. Web-based deployment also is valuable for system administration because it minimizes installation on critical systems.

Finally, beyond collaborating to develop tools, collaboration is also valuable during the use of the tools. In one case, we observed several sysadmins working together from different remote sites, trying to solve a system failure. We noted that miscommunication among the sysadmins kept them from resolving the problem. Shared monitoring tools can provide a consistent view of a system, helping to find problems more quickly.

Special Considerations for Software Engineering for System Administrators

In addressing the Testability and Collaboration needs for end-user programming by system administrators, there are some unique considerations that EUSE researchers should keep in mind.

First, a large amount of sysadmin scripting is done through a command line interface, since this environment is universally available on the computers being administered, and it is reliably present even when other parts of the software environment have failed. It is unlikely that any EUP solution which does not support command line interaction would be acceptable to this community.

Second, since sysadmins work with numerous heterogeneous software and hardware components, any EUP solution must be able to integrate the monitoring and control of these components. There is a large extant body of system management scripts, APIs, and frameworks, and sysadmins would appreciate the ability to incorporate them into EUP environments without much effort.

In conclusion, System Administrators are a large and important group of computer users who have a critical need for end-user programming. Advances in testability and collaboration could have a dramatic impact on their effectiveness. Our group at IBM has implemented a prototype end-user programming environment that addresses collaboration and information integration, but we have not worked on reliability and testability.

End-User Development in Small and Medium Enterprises: Research and development Issues

Position Paper for the CHI 2006 workshop on 'End-User Computing'

Matthias Betz, Jan Heß, Volkmar Pipek, Markus Rohde, Volker Wulf
{mathias.betz; jan.hess; volkmar.pipek; markus.rohde; volker.wulf}@uni-siegen.de
University of Siegen
Information Systems and New Media
Hoelderlinstrasse 3
D-57068 Siegen

Stefan Scheidl
stefan.scheidl@sap.com
SAP AG
Dietmar-Hopp-Allee 16
D-69190 Walldorf

Our research currently aims at the development of innovative strategies and techniques of end-user development for the business software market and focuses on small and medium enterprises (SMEs). The development of business software for this target group is a big challenge. Due to the fast changes in the market, flexibility and customisation are main requirements of such software. Most enterprises are not able to invest in individually programmed software but adjust the existing standard software to their own needs as long as possible. As only few options are adaptable, the level of modification is quite limited. Here, end user development can open up new perspectives. EUD strategies shall enable end-users (as non-professional developers) to manage their local IT-infrastructure within their organizational and process context.

Research Question and Approach

Our research questions in this context are: How can the necessary flexibility for business standard software be reached and how can one arrange these technologies in a way suitable to the users. What interface concepts and architectures can help to reach this goal? Where do we have to modify existing software engineering concepts? To answer these questions we will explore and evaluate EUD concepts for this context. (e.g. Concepts like Programming by Example, Incremental Programming or Model-based EUD, and software architectures for customizing like Service Oriented Architectures) .

Our EUD approach is based on two different and complementary perspectives: the development perspective and the appropriation perspective. On one side the development perspective focuses on the development of technologies, interfaces and methods to provide highly-tailorable, domain-oriented ERP software for small and medium enterprises. On the other side the appropriation perspective targets the activities that are being actively performed by end users in order to make sense of technology, and that usually go far beyond 'just' configuring technology. For the development of technology several points are interesting:

- What are 'good' decompositions of technology that make them flexible and manageable?

- What roles and competencies necessary to manage different levels of technological complexity? How can less competent users manage more complex technology?
- How can interface concepts be developed so that they can be easily specialised to serve users from different domains?

For the appropriation perspective, users are being perceived as a ‘Virtual Community of Technology Practice’, with support options in several directions:

- Articulation support: for the exchange of (online-/offline-) comments about the software
- Negotiation support: for the exchange of (online-/offline-) negotiation between end-users regarding software configuration
- Decision support: for collaborative decisions on software configuration solutions
- Observation support: with respect to practice of use (e.g., frequency and correlation of use patterns, configuration solutions etc.)
- Demonstration support: regarding the intended visualization of individual and collaborative use of software
- Recommendation support: establishment of a recommendation network regarding use patterns and configuration solutions
- Simulation support: use patterns and configuration solutions shall be simulated in a comprehensive way for end-users
- Exploration support: enhancement of the simulation by freely configurable, hypothetical use scenarios
- Version management support: storage and visualization of histories of use patterns and configuration solutions
- Delegation support: tasks of adaptation and configuration shall be delegated to specific users and roles in user communities

The research project

EUDISMES is a research project within the program of “Software Engineering 2006” promoted by the German “Federal Ministry of Education and Research” (BMBF). Research partners are SAP AG, Buhl Data GmbH and University of Siegen. SAP is an international key player in the area of ERP (Enterprise Resource Planning) software solutions. Buhl Data develops business software mainly for end-user. The chair of “Information Systems and New Media” at the University of Siegen has a long experience in the domain of End User Development (EUD). For more than five years we are organizing workshops regarding this issue. Soon the book “End User Development” will be released where Prof. Dr. Wulf functioned as co-editor. In 2005 the research group got the “IBM Eclipse Awards 2005” for a cooperative EUD concept with “Eclipse” (CHiC – Community Help in Context”). Furthermore we collaborate with two small (Natursteinwerk Schiffer GmbH and Dachdecker-Meisterbetrieb Vißer) and two medium (Alfred Sternjakob GmbH & Co. KG and Strähle+Hess GmbH & Co. KG) industry partners in order to gain practice experience. Via analysis of the existing business processes we want evaluate which techniques are best to be used. Later prototypes will be implemented. The prototypes (“Proof of Concepts”) will be reviewed and evaluated. From the gained experience we hope to create an integrated concept for EUD in SMEs. Additionally we plan to build up an EUD community to verify our concepts externally.

Gender in Domestic Programming: from *Bricolage* to *Séances d'Essayage*

Alan F. Blackwell

Computer Laboratory, University of Cambridge

Alan.Blackwell@cl.cam.ac.uk

ABSTRACT

Developments in ubiquitous computing mean that domestic appliances are increasingly programmable, providing new opportunities for end-user control and configuration. Unfortunately home programming, just as with end-user programming in professional contexts, is associated with stereotypically masculine learning styles. This is likely to result in future inequalities surrounding domestic technology. This paper summarises recent experimental evidence regarding the role of self-efficacy in learning through experimentation, demonstrates that similar gender-linked behaviour can be found in both domestic and professional contexts, and recommends a new approach to promoting such experimentation among women.

INTRODUCTION

In North America and the United Kingdom, computer programming has strong gender-specific connotations. Most professors of computer science are male, the computing “high culture” of hacking is overtly masculine [8], and universities (including my own) have great difficulty persuading female applicants to apply to study computer disciplines.

Do these patterns have any broader consequences, beyond a gender imbalance in the computing professions? In previous work, I have related the cognitive demands of computer programming, as practiced professionally, to the practice of programming on a smaller scale in order to control and configure domestic appliances [4]. Ubiquitous computing technologies increasingly introduce computers into our surroundings. In the domestic environment, these sometimes do little more than replacing device functions that would once have been achieved mechanically. However, an increasing number of domestic appliances also offer more powerful opportunities for configuration, no longer restricted to mechanical direct manipulation, but instead programming the appliance so that it will behave

differently in future. This paper investigates the possibility that gender imbalance in professional computing might extend to disempowerment of women in a domestic context where end-users program their home appliances.

A note on gender studies

The remainder of this paper describes a variety of behaviours that are presented as “stereotypically” male or female. It is important to note that these descriptions are not intended to be normative descriptions of men and women (either the way they *are*, or the way they *should be*). Indeed, many men act in ways that are stereotypically female, while many women act in ways that are stereotypically male. The motivation in describing and analyzing stereotypical behaviours is in order to identify resulting inequalities, and potentially act to correct them. In statistical terms, “stereotypically female” behaviours are more likely to be found in women, and experimental data is collected on this basis. The results should not, however, be applied indiscriminately to define the ability of individuals.

SOCIAL CONTEXT AND COGNITIVE STYLE

With Jennifer Rode and Eleanor Toye, I have investigated the social context of domestic end-user programming, finding that ordinary households own many programmable appliances, and that although specific appliances may fall into male or female domains of a household, both genders engage in programming behaviour [9,10]. If there is no gender-role obstacle to end user programming in the domestic context, it is reasonable to ask whether the gender imbalance in professional software engineering might result in a “trickle-down” of imbalance in everyday contexts of ubiquitous computing such as this domestic one. Evidence of this possibility can be seen in recent work with Beckwith, Kissinger et. al., which observed gender differences in end-user programming of spreadsheets [2]. These differences could not be directly attributed to social context (they were observed in an experimental context), but appear to be derived from cognitive styles associated with differing degrees of self-efficacy [1].

A previous proposal for gender-linked cognitive styles in learning to program was made by Turkle and Papert [11]. That work drew on Papert’s philosophy of “constructionism”, which emphasises learning by doing. Constructionism is derived from the cognitive development theories of Piaget, who reported that children first learn

through concrete, physical experience, and only later develop abstract and symbolic ways of learning. This natural progression from concrete to abstract understanding motivated Papert's educational programming language Logo, and also Kay's Smalltalk, designed as a component of a computer for children at Xerox PARC. Kay also believed that adults should learn this way, as in his constructionist motivation for the graphical user interface: "doing with icons makes symbols" [5].

What are the social implications of constructionism? The constructionist approach to learning is described by Papert as a kind of *bricolage*, a term used by anthropologist Claude Levi-Strauss to characterise the intellectual style of non-Western cultures. Levi-Strauss wished to emphasise the way that these cultures build social aggregates of experience, rather than the decontextualised theoretical structures typical of the West. In Turkle and Papert's work [11], *bricolage* is also a constructionist style of programming that creates "soft" and artistic arrangements of material rather than "hard" logical hierarchies of black boxes. They support this characterisation of adult learners from the personal experience of female students taking introductory programming classes at Harvard, who are reported to learn better when they are able to build by experimenting and adapting building-block materials.

BRICOLAGE IN THE HOME

This attitude to programming would appear highly appropriate to the domestic context. People programming home appliances do not wish to build theoretical constructs (although they certainly acquire theoretical understanding through successful performance). Indeed, home appliances do not support the design of sophisticated abstractions. Instead, appliances are used principally to achieve social and cultural ends, much as recommended for female students of programming by Turkle and Paper. Does *bricolage* provide an appropriate perspective for the introduction of end-user programmable ubiquitous computing into the home?

One problem with use of this term is the fact that it is already strongly associated with a particular kind of domestic activity. In informal French (outside of anthropology and cultural theory), "*bricolage*" is a synonym for the English "DIY", meaning the practice of amateurs, hobbyists or enthusiasts who maintain and modify their own houses. In France, this activity is certainly linked to gender. I asked a French student whether a French woman would ever engage in *bricolage*. She answered without hesitation: "No"!

I do not believe that this is an unfortunate linguistic accident. The kind of things that a male *bricoleur* or DIY-enthusiast might do around the house are often associated with hobbies rather than serious utility. Early experiments in ubiquitous computing for the home have a similar taint. It has been possible for over a decade to buy programmable home control systems that link appliances together,

controlling their behaviour from programs running on a central PC. The X10 standard for home automation is a popular tool for such hobbyists. If one were to identify opportunities for end-user software engineering in the home, this would seem to be an obvious target. Indeed, I was involved in a substantial research project aimed at end-user programming for home automation of this kind [6]. The many similar international research efforts aimed at developing future "smart homes" seem to be similarly masculine in their style and objectives. If home-owners are to be allowed to control and configure their homes via end-user programming, this will be a DIY/*bricoleur* heaven!

To summarise, Turkle and Papert recommend *bricolage* as an approach to programming that may be more appropriate to females. *Bricolage* seems likely to become a feature of end-user programming in the home, but might be framed in a way that is predominantly masculine.

TINKERING AND BRICOLAGE

The aspects of male DIY hobbyist behaviour that are least directed toward utilitarian outcomes are sometimes described as "tinkering". In the UK, this activity stereotypically takes place in a garden shed, where a man might take refuge from the social demands of the household to fiddle with pieces of wood or dismantled engines. Classic tropes of popular technology include the "backyard inventor", who, through such tinkering, achieves creative technical innovations.

One can certainly imagine that constant experimentation with tools, materials and components, whether woodwork, machinery or end-user software engineering, would lead over time to competence and even innovation. This is a positive, craft-oriented view of tinkering as a source of skill and expertise. It is related to Levi-Strauss' original adoption of the term *bricolage*, not to imply amateurism (as in the modern usage), but informal traditions of learning. In the domain of programming, Ben-Ari has in fact recommended that this style of engagement with computers is the best model for end-user programmers, whom he therefore describes as *bricoleurs* [3].

BRICOLAGE AND GENDER

Our recent study of tinkering in a conventional end-user software engineering domain, that of spreadsheets, found that males were indeed more likely to engage in tinkering [2]. Furthermore, those females who were more willing to tinker with the spreadsheet were more likely to learn. This willingness to tinker was associated with higher self-efficacy in females. However, increased tinkering in males was not always associated with improved performance in males. In fact, the opposite was true. It seems that an alternative connotation of the word tinkering, one associated with aimless time-wasting, was more typical of male behaviour in the end-user programming domain of spreadsheets.

Which of these interpretations of bricolage is likely to be true in the end-user programming domain of domestic appliance control? Will the smart homes of the future be of interest mainly to male hobbyists attracted to ubiquitous computing as the cyberspace equivalent of the garden shed? On the basis of popular literature such as technology magazines, one would have to conclude that the answer is yes. This is certainly the suspicion of female members of my own household. I believe it is true of many others.

However our study of end-user programming in existing home appliances [9] shows that women do already engage in programming at home, but for specific utilitarian purposes. It is worth asking whether the learning advantages experienced by females in our recent study of tinkering in spreadsheet programming, and recommended by Turkle and Papert for concrete experiences of object-oriented languages, might provide a basis by which females can be empowered to control and configure new pervasive computing technologies that enter their own homes.

AN EXPERIMENT IN DOMESTIC PROGRAMMING STYLES

In a recent (unpublished) study of domestic programming, Jennifer Rode, Eleanor Toye and I compared male and female approaches to the programming of a new DVD recorder. In a previous generation of domestic technology, "programming the VCR" was notorious as an activity that demonstrated lack of personal control over home technology. We wished to investigate this phenomenon in a controlled experimental context, in order to see whether there were any gender-linked effects of cognitive style that might influence home-owners' willingness to make the "attention investment" [4] involved in a transition from direct manipulation to appliance programming.

As in the work by Beckwith et. al [2], we saw a link between attention investment and self-efficacy. Low self-efficacy will result in an over-estimate of the costs involved in a novel abstraction strategy, and an under-estimate of the likelihood of success. Our experiment therefore compared participants' estimated likelihood of success in end-user programming of the appliance with their actual success in an experimental task. This task was designed to be as closely representative as possible of domestic experience of new technology. Participants were presented with a new DVD recorder and television, made by the same manufacturer, and purchased from the appliance department of a local department store. We had connected the recorder and television to power and aerial, but gave no further instructions on their use, simply giving the participant the appliance manuals, and asking them to program recording of a television show. Participants were interviewed before and after this task, in order to measure their self-efficacy.

Full results of this study will be published in due course. For the purpose of this workshop, it appears that the general trend with regard to self-efficacy for DVD programming is the same as that noted in the study of spreadsheet

programming by Beckwith et. al. Of the 24 participants in our study, the 12 women were less confident than the 12 men of their ability to complete the video programming task successfully. After the task, the confidence of the men increased, while the confidence of women decreased, as also observed in the Beckwith et. al. study. These effects were more pronounced when the task involved programming, rather than non-programming functions of the DVD recorder. Despite the drop in reported self-efficacy, the actual rates of success were equal for men and women (although more women were unsure afterwards whether they had correctly completed the task).

With regard to the consequences for attention investment decisions, women predicted that the task would take them longer than men predicted. This was true, in that average completion time was substantially longer for women. As in the experiment by Beckwith et. al., we might expect this to result from more periods of reflection by women. However the estimate by women of how long they had actually spent on the task was more than double the elapsed time (an estimate of 20 minutes, as opposed to average elapsed time of 9 minutes). Post-hoc estimates by men were that they had spent only 5 minutes on the task (actual average 4). In terms of attention investment, we would expect this biased estimate of actual attention required to perform a programming task to result in future avoidance of the task, because the attention investment would appear not to be justified. We therefore see that, in the home domain as in the spreadsheet domain, initial differences in self-efficacy lead to actual differences in programming competence.

EMPOWERMENT THROUGH ESSAYAGE

What skills do we wish to encourage, in order to establish competence in both genders to configure and control ubiquitous computing infrastructure in the domestic environment? In terms of the attention investment theory of abstraction use, we would like to assist all members of a household to make the transition from direct manipulation, to abstract specification of system behaviour. It is often the case that abstract specifications of appliance function are related to the functionality that can be controlled by direct manipulation, so the required competence is a matter of understanding direct manipulation behaviours sufficiently well to compose and modify them. This understanding of component behaviour is achieved informally, through a process of active experimentation, tinkering with the direct manipulation components, while the process of modifying and composing those components can be understood in terms of informal assembly or bricolage.

Based on our experimental findings, as well as the analysis of cultural connotations of tinkering and bricolage, it seems that these kinds of experimentation in the home are stereotypically masculine. Women are less likely to engage in either tinkering or bricolage with home appliances, and hence less likely to gain the expertise necessary to become competent end-user programmers in the home.

There are, however, other domains in which stereotypically female activity has characteristics that lead to competence in constructing abstractions. The conventional view of male dressing is that men select individual items of clothing according to immediate or functional requirements (a kind of direct manipulation) without proper consideration to the complete assemblage or “outfit”. Women, in contrast, are expected to be relatively expert in the coordination of items of clothing into an outfit or ensemble. This competence is not innate, but is developed through processes of deliberate experimentation, in which a woman experimentally tries on different items of clothing that she owns, in order to design ensemble outfits for use on later occasions. This form of experimentation, leading to expertise and the construction of abstract specifications from concrete elements, seems closely related to the kind of competences that are developed by men when they tinker with mechanical components.

We have noted that skill derived from tinkering is highly dependent on self-efficacy. Lack of confidence in one’s own ability does not encourage tinkering, and hence prevents sufficient familiarity for the move to abstract specification. In attention investment terms, low self-efficacy perpetuates reliance on direct manipulation. In the ubiquitous computing smart home, reliance on direct manipulation will be associated with lack of control, especially as home appliances incorporate increasing numbers of abstract specification functions [7]. Rather than submit to this perpetuation of gender-stereotyped competence in relation to technology, we might instead promote positive models of experimentation and abstract description within existing domains of female competence. Just as “bricolage” veers between social theory and mundane household gender roles in order to suggest a perhaps overly masculine model of technology use, we might recommend an alternative style of engagement based on the “séance d’essayage”. This phrase offers a relatively formalized recognition of the kind of female behaviour in which items of clothing are assembled into ensemble outfits. It encourages the kind of experimentation that leads to improved conceptual understanding in that domain, and it forms the basis for future competence.

The séance d’essayage is not currently associated with the kind of masculine competencies (tinkering and bricolage) that have been related to successful end-user programming. But this does not mean that such an association is impossible. Perhaps an alternative approach to software tools, one modeled on stereotypically female competence, would offer potential for greater balance in delivering the benefits of ubiquitous computing.

ACKNOWLEDGMENTS

This paper draws on experimental work that was carried out by Jennifer Rode and Eleanor Toye, and funded by the Engineering and Physical Sciences Research Council grant

GR/R87482 “Cognitive Ergonomics for Ubiquitous Computing.” It also draws on collaboration with Laura Beckwith during and subsequent to a visit to Cambridge that was funded by the NSF EUSES consortium. Laura’s work involved several other EUSES members, as in [2] We are grateful to participants in both these studies.

REFERENCES

1. Bandura, A. Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review* 8, 2 (1977), 191-215.
2. Beckwith, L., Kissinger, C., Burnett, B., Wiedenbeck, S., Lawrance, J., Blackwell, A. and Cook, C. Tinkering and gender in end-user programmers' debugging. To appear in *Proceedings of CHI 2006*.
3. Ben-Ari, M. Bricolage forever! In *Proceedings of the 11th Annual Workshop of the Psychology of Programming Interest Group*, (1999), 53-57.
4. Blackwell, A.F. First steps in programming: a rationale for attention investment models. In *Proc. IEEE Human-Centric Computing Languages and Environments* (2002), 2-10.
5. Blackwell, A.F. The reification of metaphor as a design tool. To appear in *ACM Transactions on CHI*.
6. Blackwell, A.F. and Hague, R. AutoHAN: An architecture for programming the home. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments* (2001), pp. 150-157.
7. Blackwell, A.F., Hewson, R.L. and Green, T.R.G. Product design to support user abstractions. In E. Hollnagel (Ed.) *Handbook of Cognitive Task Design*. Lawrence Erlbaum Associates, (2003) pp. 525-545.
8. Håpnes, T. and Sørensen, K.H. Competition and collaboration in male shaping of computing: A study of a Norwegian hacker culture. In K. Grint & R. Gill (Eds), *The Gender-Technology Relation: Contemporary theory and research*. London: Taylor & Francis (1995), pp. 174-191.
9. Rode, J.A., Toye, E.F. and Blackwell, A.F. The Fuzzy Felt Ethnography - understanding the programming patterns of domestic appliances. *Personal and Ubiquitous Computing* 8 (2004), 161-176.
10. Rode, J.A., Toye, E.F. and Blackwell, A.F. The domestic economy: A broader unit of analysis for end user programming. In *Proceedings CHI'05 (extended abstracts)*, (2005) pp. 1757-1760.
11. Turkle, S. and Papert, S. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior* 11, 1 (1992), 3-33. Available online at <http://www.papert.org/articles/EpistemologicalPluralism.html>

Games Programs Play: Obstacles to Data Reuse

Chris Scaffidi

Institute for Software Research Intl.
School of Computer Science
Carnegie Mellon University
cscaffid+isri@cs.cmu.edu

Mary Shaw

Sloan Software Industry Center &
School of Computer Science
Carnegie Mellon University
mary.shaw@cs.cmu.edu

Brad Myers

Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
bam@cs.cmu.edu

ABSTRACT

Information workers often reuse data by taking it from an existing representation, recombining it to create new data, and storing the new data in another representation. The sources and destinations include databases, spreadsheets, web sites, text documents, and emails. Recombination activities are similarly diverse and include copy/pasting, concatenating, visual reformatting, arithmetic/calculating, and so forth. Yet many obstacles impede such reuse. In this paper, we summarize the problems that users face as well as some strategies for overcoming these problems.

Author Keywords – data, reuse, software, interoperability

ACM Classification Keywords

H.3.5. Online Information Services: Data sharing.

OBSTACLES USERS HAVE ENCOUNTERED

We have recently conducted three studies that characterize numerous obstacles impeding effective data reuse by end users, professional programmers, and everyone in between.

First, preliminary analysis of our contextual inquiry of three administrative assistants and five managers at Carnegie Mellon University reveals that much of their work involves manually copying and pasting data among web pages, spreadsheets, and emails. Their work is highly repetitive and ripe for end-user programming—except that they lack suitable tools.

Second, our finished survey of 831 computer-savvy *Information Week* readers asks what software they use, followed by the open-response question, “In what ways has this software ‘gotten in the way’ of doing work in the past year?” [5] Of the 527 people who list problems in response, 25% mention obstacles related to data reuse, especially data incompatibility. (By comparison, only 15% mention bugs, glitches, or other software reliability problems.)

Third, preliminary analysis of telephone interviews with six people involved in creating Hurricane Katrina “person-locator” sites suggests that even technically capable people struggle to reuse data. As these sites redundantly proliferated in the weeks after Katrina, three of our respondents helped merge sites into a single whole. Though handcrafted scripts processed over 500,000 records, numerous problems forced volunteers to type in another 100,000 manually.

In general, users may perform the following six steps when reusing data, and obstacles abound at each step. (Below, “CI” refers to our contextual inquiry, “IW” refers to our *Information Week* survey, and “HK” refers to our interviews related to Hurricane Katrina person-locator sites.)

Step 1: Find data sources

Reusing data first requires finding it, which can prove tedious. One IW respondent has expressed unhappiness with his organization’s “very fragmented data management environment,” while another has complained, “Separate files in separate formats and folders causes [sic] confusion and need for good organizational skills.” In fact, our CI reveals that even if users only need a single piece of data to populate a spreadsheet or web form, they may struggle to find the datum using software and instead fall back on manual methods. For example, administrative assistants and managers fill out many expense reports that require a project code for each expense, but looking up codes is slow, usually involving scrolling through long lists onscreen, sending emails, or phoning peers. To overcome this obstacle, workers collaborate to maintain a “cheat sheet” (in Excel) which they each print and keep on a stand next to their monitors.

Step 2: Access data sources

Once workers locate data, accessing it may be hard. For instance, some HK site creators have refused to let aggregators access backend databases, so aggregators have resorted to using “screen scrapers.” As a second example, in order to analyze data in the accounting database, CI managers must first export the data to a file on their desktop computer; this export function is only accessible from browsers running on Windows XP. Our CI also reveals other access issues, some requiring intervention by technical staff.

Step 3: Vet and repair data quality

Ensuring data quality is a problem in any dataset, but even more so when humans generate the data. To deal with this,

HK aggregators have promulgated an XML standard for structuring data. This standard includes fields that help data users evaluate data's reliability so they know what data might need filtering or repair; for example, fields include the record's creation date and the contact information of the record's creator. However, data quality problems are not limited to hurricane-devastated areas but can be endemic to office environments. As one IW respondent has reported, poor data quality "leaves a lot of database cleaning to be done before the information can be used for intended purposes."

Step 4: Cope with incompatibility

After finding, accessing, and vetting data sources, users seek to combine data. Unfortunately, syntactic (meaning-free) incompatibility may interfere with combining data, often due to incompatibility in data layout or encoding. For example, HK data aggregation involves converting data from a rows-and-columns database representation into a hierarchical XML format, with its nested angle-bracket tags and rules for encoding many characters.

Other incompatibility occurs at a subtle, semantic level, where two apparently compatible data representations in fact have incompatible meanings. For example, end users of HK sites often have used the wrong web forms to enter data (e.g.: acting as if data about lost pets is semantically equivalent to data about lost humans, and then using the "missing persons" form to enter data about missing pets).

This problem's dual occurs when different systems interpret the same data in different ways. Formatting incompatibility is a particular case: Many IW respondents complain that different applications render data in different ways. For instance, Firefox and Internet Explorer render HTML differently, and WordPerfect and Microsoft Word render rich text differently. One IW user dislikes needing to "spend to [sic] much time making something look pretty," a sentiment shared by some CI spreadsheet users.

After coping with data incompatibility, users can combine the data by copy/pasting, concatenating, visual reformatting, arithmetic/calculating, and so forth.

Step 5: Store new data

Software limitations hamper storing new data due to performance, capacity, or access problems. For example, one HK interviewee notes the lack of scalability in Access for storing large data sets; similarly, several IW respondents have noted, "Excel can't handle much data."

Step 6: Publish new data

Users' ultimate goal is to publish new data, but helping others to find it can prove challenging. For many HK site creators, the main challenge has been getting the media to report sites' existence to the world. Data exposure is also a problem in offices; one IW reader has complained about the "limited ability for automated report distribution," while several CI users must print out documents and distribute them manually due to insufficient workflow automation.

TOOLS FOR FINDING / ACCESSING / REPAIRING DATA

End users often find data using commercial search tools whose main function is to draw together numerous scattered data sources into one index. Such tools are valuable because users still store and publish data via largely application-specific, decentralized, ad hoc mechanisms such as copying files to a web server or sending emails.

Researchers have recently focused on providing tools to help end users access and repair data. For example, tools exist that allow users to automate retrieval and manipulation of web page data [1]; Java-savvy users can even use such tools to populate spreadsheets [2]. Ensuring data quality remains difficult, but researchers have made progress in the web service [3] and spreadsheet [4] domains.

Integrating tools like these with search systems, and extending them to other domains such as databases and emails, may raise new usability and reliability challenges that deserve further exploration. However, our present research agenda centers on data incompatibility, which is the main subject of the following sections.

STRATEGIES FOR COPING WITH INCOMPATIBILITY

Shaw lists strategies to deal with *packaging* incompatibility between executable software components A and B [6]:

1. Replace A's representation with B's representation.
2. Publish an abstraction of A's representation.
3. Transform A on the fly to B's representation.
4. Negotiate to A and B's lowest common denominator.
5. Make B multilingual.
6. Provide B with import/export.
7. Transform A and B to intermediate representation C.
8. Attach a wrapper to A.
9. Maintain parallel consistent versions of A and B.

Some of these have natural analogues for coping with *data* incompatibility. For example, a user can combine data from spreadsheet A and web page B by running COM-based scripts on both documents (strategy 2), or by exporting the spreadsheet to HTML and referencing it in the web page with a <FRAME> tag (strategy 6).

Although existing tools lack support for some strategies, many strategies do prove useful in certain contexts. For example, database federation exemplifies several of these strategies [7]. In particular, federated systems must negotiate common protocols on the fly (strategy 4).

Whereas federation deals with database incompatibility, systems like Citrine deal with office application incompatibility [8]. Citrine transforms clipboard data from one representation to a standardized intermediate representation (strategy 7) so that users can copy/paste structured data among applications.

In terms of software architecture, many of these strategies can most easily be implemented by interposing a mediator component between A and B. For example, Microsoft COM DLLs act as mediators that expose an abstraction of web pages for scripting (strategy 2). Mediators are known

by various names: “converter” (if used in strategies 3 and 7), “broker” (if used in strategy 4), “translator” (if used in strategy 5), and “façade” (if used in strategy 8).

Unfortunately, there are inherent challenges to mediator-based implementation, as discussed below. Moreover, all nine strategies’ practical utility is limited, as no existing tool supports the full range of users’ data representations in database tables, groups of spreadsheet cells, web pages, documents, and emails.

TACTICS FOR SUCCESSFUL MEDIATION

Effective mediation ideally requires the mediator to recognize the *details* of the source and destination’s layout, encoding, and semantics. For example, Excel can export spreadsheets to a certain XML schema, but this serves no purpose if the user needs to import the data into a system that uses a slightly different XML schema than Excel does. This sensitivity to a representation’s details leads to two challenges for making mediator-based strategies successful.

First, in order to be cost-effective, any mediator implemented by a professional should ideally recognize *multiple* detailed representations. (Professionals are typically too expensive to have them create one mediator per detailed representation.) There are several tactics for achieving this:

1. Let the end user customize mediators’ behavior.
2. Let the end user (rather than a professional) create mediators in the first place.
3. Let the end user share customized / created mediators with other users (permitting further customization).
4. Let mediators automatically customize their own behavior when faced with new data representations.

Second, mediators are often not robust to evolution of representations, thus provoking manual reprogramming to prevent subtle semantic bugs from jeopardizing data quality. Researchers have worked toward automatic detection of evolution in web service semantics [3]; generalizing this tactic to other representations would be extremely valuable.

Tactics like these are essential to making mediator-based strategies successful, but some mediators are more amenable than others to these tactics.

FUTURE WORK: ENHANCEMENTS FOR CITRINE

In the future, we hope to apply several of the tactics and strategies listed above to produce an end user programming environment that supports a variety of data sources and a variety of ways to combine data from those sources. As a start, we will enhance Citrine, a mediator for copy/pasting structured data [8].

Currently, when end users paste data into a new web form that they have never before encountered, they each must train Citrine how to map the data into the form. Essentially, this equates to customizing the mediator’s behavior (tactic 1 in the list above). We will evaluate five enhancements that may reduce users’ effort:

1. We will enable users to save a capsule containing a form’s data so they can reload the capsule and skip the copy/paste step entirely when reusing data in that form.
2. We will automatically save a capsule each time a user completes a web form. Thus, the next time that the user completes similar forms, we may be able to use the user’s entries in some form fields to predict what values should go into other fields. This would eliminate manual reloading of capsules.
3. When a user maps data to a form, we will record the structure of this mapping in a central repository so that if other users face a similar situation, Citrine can offer a reasonable default mapping.
4. We will use machine learning to identify the most commonly occurring mappings so that Citrine can perform them automatically.
5. We will explore how visual cues on the page can help Citrine maintain high quality even if the data sources and destinations evolve in structure or semantics.

These enhancements should reduce the effort required to reuse data in web forms and reveal data patterns that may be of benefit as we tackle data reuse in other contexts.

ACKNOWLEDGMENTS

We thank Andrew Ko for his helpful questions and suggestions. This work was funded in part by the EUSES Consortium via NSF (ITR-0325273), by NSF under Grant CCF-0438929, by the Sloan Software Industry Center at Carnegie Mellon, and by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

1. Elbaum, S., et al. Helping End-Users “Engineer” Dependable Web Applications. *ISSRE’05*, 22-31.
2. Kandogan, E., et al. A1: End-User Programming for Web-based System Administration. *UIST’05*, 211-220.
3. Raz, O., Koopman, P., and Shaw, M. Semantic Anomaly Detection in Online Data Sources. *ICSE’02*, 302-312.
4. Rothermel, G., et al. A Methodology for Testing Spreadsheets. *TOSEM’01*, 110-147.
5. Scaffidi, C., Ko, A., Shaw, M., and Myers, B. Identifying Categories of End Users Based on the Abstractions That They Create, Tech Rpt CMU-ISRI-05-110/CMU-HCI-05-101, Carnegie Mellon University, Pittsburgh PA, 2005.
6. Shaw, M. Architectural Issues in Software Reuse: It’s not Just the Functionality, It’s the Packaging. *SSR’95*, 1-3.
7. Sheth, A., and Larson, J. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *CSUR* 22, 3 (1990), 183-236.
8. Stylos, J., Myers, B., and Faulring, A. Citrine: Providing Intelligent Copy-and-Paste. *UIST’04*, 185-188.

End User Software Engineering: Auditing the Invisible

Joshua B. Gross

School of Information Sciences and Technology
311B IST Building, Penn State University
University Park, PA 16802
+1814 865 9838
jgross@ist.psu.edu

ABSTRACT

In this paper, I will describe the need for new tools to engage end users in the software engineering process, and then describe an example of such a tool in a brief scenario.

INTRODUCTION

In his seminal article on the problems of software development, Brooks [2] cited the essential invisibility of software as one of the essential or natural problems that could never be resolved. His point is accurate, but limited in its perspective. Work in research and industry has shown that visibility can be lent to software, but that visibility is largely a veneer; an attempt to use physical or mechanical metaphor to explain the processes described in software.

Unfortunately, this approach is inevitably limited by the value of the metaphor. New approaches to visualization are necessary, ones that rely not on metaphor, but on new, artificial languages that bridge the gap between how computers operate and how the human mind functions. These languages must also account for the pragmatic applications of the software; this aspect is perhaps the most problematic, but the most critical to bridging the gap.

THE SOFTWARE ENGINEERING PROBLEM – REDUX

It seems almost superfluous to speak about problems related to software engineering. The norm for software engineering projects has been late delivery of overbudget, substandard, incomplete products. This is for the lucky projects that deliver at all; the United States has attempted to replace its air traffic control software three times in the past twenty years, but despite the millions of US dollars spent, no such replacement is available.

Much of the problem can be traced to software engineering (SE) as a discipline. Many software development processes begin (implicitly or explicitly) with the statement “assume fixed requirements.” Even if a process to capture such requirements were available, fixed requirements are a myth on the order of Sisyphus.

Numerous solutions to the problems of software engineering have been proposed, and inevitably they have offered some improvement. Some rely on tools (e.g. CASE tool, Business Rules), while others rely on processes (e.g. Extreme Programming and Rational

Unified Process), and others on visualizations that allow for design and explanation (e.g. the Unified Modeling Language).

All of these do address some aspect of what Brooks referred to as “accidents” of software development, but none solve the problem. Several researchers and practitioners have proposed that software needs either a “paradigm shift” or “sea change” to completely rewrite how software is built. Unfortunately, none has yet been successful.

It is not the aim of this paper to propose such a change; the hubris required to attempt such (especially in a three page workshop paper) is beyond this author. However, there are clues that show how existing tools, processes, and languages can be integrated and extended to improve software development, or, at the very least, lend it additional visibility.

THE END-USER SOFTWARE ENGINEER

When we consider a profession such as software engineering, we must initially ask whether end users can perform this function. As mentioned above, there is no reason to assume that they would be much worse than trained software engineers.

However, we cannot reasonably expect non-professionals to perform certain tasks. Designing taxonomies, creating flexible architectural components, and building the unexciting, exceptionally invisible interstitial software that manages the tiers of a business application are tasks with limited rewards for anyone other than a professional developer. Building a small application (e.g. in a spreadsheet) is within the grasp of many end users, but building an enterprise application is not.

So if end users cannot be software engineers, and developers cannot be domain experts, we must meet somewhere in the middle. Perhaps the best metaphor would be that of a library. A patron cannot be expected to build and organize the library, but similarly no librarian can fully understand the content and import of each volume. A library is only partly a building filled with books and periodicals; it is a meeting of minds, skills, and interests.

A SOFTWARE MEETING OF THE MINDS

Eric Evans has suggested that users, domain experts, and developers must jointly form a new “ubiquitous language” [3] that is shared and used by all people working on building a particular system. This language creates the possibility of an artificial space in which many abstract problems of the domain can be made concrete and “solved”, at least for the limited purpose of the application.

This idea is excellent, and shows a growing trend to incorporate the user more fully into the software development process. Another example can be found in Extreme Programming, in which an “on-site customer” is one of twelve required practices [1]. While these practices are growing in popularity, they often hit a roadblock due to disengaged and uninterested users.

As with Carroll & Rosson’s “active user”, the “engaged user” is something of a paradox, concerned with productivity, possibly at the expense of quality. The engineering gestalt, which emphasizes robust, reliable systems, cannot be expected to capture the hearts and minds of users everywhere.

THE NEED FOR CONVERGENCE

Despite potential limits of interest, we should not dismiss end-user software engineering. Unfortunately, we haven’t sufficiently mastered software production in order to allow us to completely automate the process. The ‘Big Red Button’ idea that magically translates requirements to code is not yet a reality.

The question arises, then, what role end users can take in the software engineering process? However, a slight modification of the question is more interesting: how can we modify the software engineering process to accommodate end users and improve the overall productivity and quality of the product?

This question allows us to find a convergence: a place where the needs of the various stakeholders in the process and outcome of large-scale software development can come together. In theory, any such convergence is a good thing, but as discussed above, the different interests and skills make a positive outcome seem unlikely.

ANSWERING THE CALL

Since we cannot yet solve software engineering problems *en masse*, our interim question must be how to take advantage of this convergence of need. This is not a question with a single answer, but this paper proposes that at least one answer can be offered and developed into a useful practice.

Two recent laws enacted in the United States have changed how businesses use and view information systems. HIPAA (the Health Insurance Portability and Accountability Act) regulates how all medical data is transferred, and the Sarbanes-Oxley Act has made corporate officers legally responsible for misreported corporate earnings and other financial statements.

In both cases, the new laws force organizations to produce a level of traceability that they have never had to deal with before. In addition, because both civil and criminal penalties can be imposed, these new business practices must be taken seriously. Interestingly, software developers are largely immune from penalties, but as others (end users) are not immune, they are greatly concerned with ensuring that the systems they use function properly.

Software engineering has an answer; software quality assurance (SQA), which is concerned with ensuring that software is validated (matched to requirements) and verified (technically correct). Unfortunately, SQA activities are seen as the least engaging, and while tools have improved (e.g. for requirements traceability and unit testing), we still have a problem that end users are probably unwilling to tackle.

I propose, instead, that we incorporate a new method of investigation, auditing, and create new tools to support auditing by end users.

To differentiate between auditing and traditional verification and validation, I will note several changes. First, auditing implies that someone external (in this case, to the development process) is performing the action; the end user is an ideal motivated auditor. Second, the distinction between verification and validation becomes moot; the end user does not care why software does or does not fail. Finally, the goal is different; the end user will not be concerned about the process that produced the artifact. The artifact itself is the only thing of interest. In other words, a piece of software may pass all validation and verification tests, but still fail an audit.

In order to properly audit software, however, we need new tools. These tools will be of use and interest to end users, but will probably enhance the development process. These tools must visualize how software is functioning.

A METAPHOR FOR MACHINES

We already have many visual languages in active use in software engineering. However, most (like UML) are designed to design systems, or, in other words, to explain how the system *will* work. At a much later point, a system is produced from the design, but the system may have little or no fidelity to the design. Also, even if the artifact is largely a product of the design, certain elements (often structural) never make it into the design.

So, what we need is not another design language, nor even an improved design language. Instead, we need a language and supporting tool that will allow an end user to trace aspects of the functioning system. This “auditing” tool might be seen as something like a debugger; it would allow the user to “open the hood” on a running process.

However, this is not a proposal for a visual debugger. The goal of a debugger is tracing, but an end user’s perspective on what should be traced will be quite different than the programmer’s perspective.

Additionally, the purpose of this tool is not to explain or explore the components (e.g. objects or functions) of the system, although those will be relevant. The purpose is to expose to the user those aspects that they believe are important. The scenario described below will explain one possible use.

A BRIEF SCENARIO: WHERE DID MY MONEY GO?

Jane is an end user involved in developing banking software. She has worked as a bank teller, personal banker, and business banker, and has been asked by the bank to participate in ensuring that the new banking software functions properly.

In order to perform this task, she has been given a new monitoring tool. The tool allows her to identify a variable of interest and follow it through the system. Jane has decided that she wants to see what happens to an amount of cash deposited into a checking account.

Jane begins by opening up the teller interface, and selecting the screen to enter a deposit. She identifies the deposit as cash, and selects the deposit amount using the monitoring tool. She then completes the transaction interaction.

At this point, the tool begins tracking the deposit amount. Because the new banking software is object-oriented, the amount is placed in a new instance of the Deposit class, and this object is presented to Jane in the center of the monitor tool screen. This object will remain at the center of the screen throughout Jane's interaction.

Jane uses the object as a launching point for her investigation. She follows a link from the Deposit object to the Account object, and verifies that the account information is correct. She then decides to watch the process continue.

The tool automatically stops whenever the members (instance variables) for the monitored object change. At one point, an instance of the Transaction class is created and placed in the object. When Jane sees this, she looks inside this object, and selects this as an additional object to monitor.

The tool later notes that the information from the Deposit class has been written to the database. At this point, Jane is concerned, because the transaction information has not been written. She again follows the link to the Account object, and verifies that the balance has been updated to reflect the deposit.

Now Jane knows something is wrong; banking regulations (and best business practices) dictate that a change to a balance cannot be recorded without first recording the transaction that caused it. Jane lets the tool complete, and notes that the transaction information is eventually written to the database, as well, but she still feels it should have been done first.

Jane immediately goes to talk to a developer to discuss this problem. The developer, Ludmilla, looks at the code,

and says to Jane, "Oh, that's OK, it's all happening in a transaction." Jane is confused; to her, a 'transaction' is a business process, not a technical process.

Jane explains her confusion, and Ludmilla realizes the mistake. Ludmilla explains the nature and purpose of isolated database transactions, in which all or none of a specified set of database writes are allowed to occur. Jane and Ludmilla use the point of confusion to propose some new terms.

As a result, the group explicitly uses the terms "database transaction" and "financial transaction", and the class Transaction has been renamed FinancialTransaction. Jane also uses this point to send an email to the developers of the monitoring tool to indicate that the tool should note the boundaries (beginnings and endings) of database transactions.

FINAL THOUGHTS

Looking at a traditional debugger, one might conclude that it could be used in the scenario described above. However, the amount of information on the screen, the monitoring and step points, and the programming knowledge needed to use a debugger make this unlikely.

Again, the goal is not to develop a new tool for its own sake. The idea is to develop a means to allow an end user to understand what is happening inside the world of a software application, in order to support a variety of tasks that can be categorized as auditing.

The advantage of using a visual language (and supporting tool) comes from using a new, potentially unbiased means of looking at the auditing problem that is necessarily limited in size.

We cannot immediately turn the reins of software engineering over to the end user, but we can use novel approaches to engage end users in the process at a deeper level. Traditionally, users have been kept at arm's length from the software artifact, but new interventions can bridge that gap.

REFERENCES

1. Beck, K. *Extreme Programming Explained*. Addison-Wesley Professional, 1999.
2. Brooks, F.P., Jr. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20 (4). 10-19.
3. Evans, E. *Domain-Driven Design: Tackling Complexity at the Heart of Software*. Addison-Wesley Professional, 2003.

End-User Programming Productivity Tools

Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Jeffrey Stylos

Human-Computer Interaction Institute

Carnegie Mellon University

Pittsburgh, PA 15213

ajko@cs.cmu.edu, bam@cs.cmu.edu, mcoblenz@andrew.cmu.edu, jsstylos@cs.cmu.edu

http://www.cs.cmu.edu/~marmalade

ABSTRACT

Our research focuses on developing interactive technologies for a broad range of end-user programming activities, including code construction, verification, debugging, and understanding. A common goal among all of these technologies is to identify core ideas that can be used across a variety of domains and programmer populations.

INTRODUCTION

Although end-user programmers' interests vary widely, spanning the web, animation, documents, databases, mail, and countless other types of information, all of these users use programming as a means to an end [10]. Therefore, to minimize the distractions from end users' primary goal, it is essential that end user programming tools are approachable, easy to learn, and immediately helpful [1].

We are designing several technologies that satisfy these criteria, including new interaction techniques for editing code, new languages that help end users identify mistakes, debugging tools that answer users' questions about their program's output, and workspaces that help them understand the answers. All of these technologies have been directly inspired by the empirical research of a variety of programmer populations and their difficulties [5, 6, 8, 11].

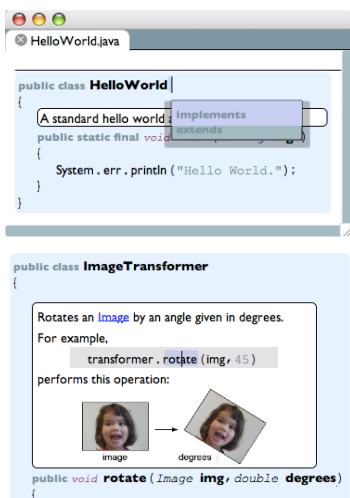


Figure 1. Barista [7], a Java editor that supports drag and drop, auto-complete menus, and text editing in a single editor, and embedded, in-context tools and visualizations.

CONSTRUCTING PROGRAMS

Syntax has long been a significant learning barrier in end-user programming systems, largely because of the difficulty of understanding and remembering the hidden and complex rules encoded in language grammars [5]. We have been working on a new class of code editors that try to help users construct code by choosing from different options rather than having to memorize the syntax. Barista [7], shown in Figure 1, is a Java editor that embodies this approach. It supports drag and drop interactions for creating and modifying code and syntactic and semantic auto-completion, as well as traditional text editing interaction techniques, all in a modeless editor. Barista also allows designers of end-user programming systems to embed tools and information in code, as illustrated by the method header on the bottom of Figure 1.

Although Barista is currently for Java, its underlying design and techniques could be an alternative to conventional text editors across the spectrum of programming languages.

DETECTING ERRORS

Some spreadsheet systems allow users to specify units (e.g. 5 lbs.) with their data in order to help detect unit errors in calculations. However, most data represented in spreadsheets is a measurement of a particular kind of object (e.g., 5 lbs of *apples*), and it is often inappropriate to perform calculations on data that represent different kinds of objects. Slate [2], shown in Figure 2, allows users to

	A	B	C
1	Fruit Prices		
2	\$0.45 / lb. (apples)	\$0.50 / lb. (oranges)	
3			
4	Fruit Sold	Revenue	
5	312 lb. (apples)	\$140.40 (apples)	
6	399 lb. (oranges)	\$179.55 (apples, oranges)	
7			
8			
9			
10			
11			
12			

Figure 2. Slate [2], a spreadsheet language that allows users to give data labels, in order to help identify incorrect input and formulas. For example, the label “(apples, oranges)” at the bottom right of the spreadsheet suggests an error, since nothing can be apples and oranges simultaneously.



Figure 5. The Mica web application. Mica includes a keyword sidebar on the left, which is generated from Google Web API search results shown on the right. Search result pages containing code are marked with an icon.

Mica finds related classes and methods in the standard Java APIs in the form of keywords (method, class and interface names on the left in Figure 5) and regular web search results (on the right in Figure 5). Mica determines API keywords by analyzing the content of the Google search result pages and comparing these to a list of all class and method names for the standard Java API. The keywords are ranked based on the frequency with which they appear in the search result pages for the query and the overall frequency with which they appear on all pages indexed by Google. The list of keywords dynamically updates as Mica loads and processes all of the search result pages.

We plan to expand Mica's to aid other aspects of API use, such as understanding high-level API concepts, finding example code, and integrating examples into programs.

CONCLUSIONS

Our research covers a broad spectrum of programming activities, and we anticipate that our techniques will generalize to a variety of domains and programmer populations. We hope that our broad focus will both inspire new ideas for commercial programming tools and drive innovations in end user software engineering research.

ACKNOWLEDGMENTS

We thank our collaborators, including Htet Htet Aung, Christopher Scaffidi, and David Weitzman. This work was supported by the National Science Foundation, under NSF grant IIS-0329090, and as part of the EUSES consortium (End Users Shaping Effective Software) under NSF grant ITR CCR-0324770. The first author was supported by an NDSEG fellowship.

REFERENCES

1. Blackwell, A., First Steps in Programming: A Rationale for Attention Investment Models, *IEEE Symposia on Human-Centric Computing Languages and Environments*, (2002), 2-10.
2. Coblenz, M. J., Ko, A. J., and Myers, B. A., Using Objects of Measurement to Detect Spreadsheet Errors, *IEEE Symposium on Visual Languages and Human-Centric Computing*, (2005), 314-316.
3. Furnas, G. W., Gomez, T. K. L. L. M., and Dumais, S. T., "The Vocabulary Problem in Human-System Communication," in *Communications of the ACM*, 30, 1987, 964-971.
4. Ko, A. J. and Myers, B. A., Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior, *Human Factors in Computing Systems*, (2004), 151-158.
5. Ko, A. J., Myers, B. A., and Aung, H., Six Learning Barriers in End-User Programming Systems, *IEEE Symposium on Visual Languages and Human-Centric Computing*, (2004), 199-206.
6. Ko, A. J., Aung, H., and Myers, B. A., Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks, *International Conference on Software Engineering*, (2005), 126-135.
7. Ko, A. J. and Myers, B. A., Barista: An Implementation Framework for Enabling New Interaction Techniques and Visualizations in Code Editors, *ACM Conference on Human Factors in Computing*, (2005), to appear.
8. Ko, A. J. and Myers, B. A., A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems, *Journal of Visual Languages and Computing*, 16, 1-2, (2005), 41-84.
9. Myers, B. A., Weitzman, D. A., Ko, A. J., and Chau, D. H., Answering Why and Why Not Questions in User Interfaces, *ACM Conference on Human Factors in Computing Systems*, (2005), to appear.
10. Nardi, B. A., *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: The MIT Press, 1993.
11. Panko, R., What We Know About Spreadsheet Errors, *Journal of End User Computing*, 2, (1998), 15-21.

Toward Sharing Reasoning to Improve Fault Localization in Spreadsheets

Joseph Lawrance, Margaret Burnett, Robin Abraham and Martin Erwig
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, Oregon 97331

{lawrance,burnett,abraharo,erwig}@eecs.oregonstate.edu

Abstract

Although researchers have developed several ways to reason about the location of faults in spreadsheets, no single form of reasoning is without limitations. Multiple types of errors can appear in spreadsheets, and various fault localization techniques differ in the kinds of errors that they are effective in locating. Because end users who debug spreadsheets consistently follow the advice of fault localization systems [9], it is important to ensure that fault localization feedback corresponds as closely as possible to where the faults *actually* appear.

In this paper, we describe an emerging system that attempts to improve fault localization for end-user programmers by sharing the results of the reasoning systems found in WYSIWYT [13, 14] and UCheck [1, 6]. By understanding the strengths and weaknesses of the reasoning found in each system, we expect to identify where different forms of reasoning complement one another, when different forms of reasoning contradict one another, and which heuristics can be used to select the best advice from each system. By using multiple forms of reasoning in conjunction with heuristics to choose among recommendations from each system, we expect to produce unified fault localization feedback whose combination is better than the sum of the parts.

1 Introduction

Spreadsheet systems like Excel are among the most widely used programming systems. Research estimates that the number of end-user programmers, which includes spreadsheet users, outnumbers professional programmers by an order of magnitude [15]. Both end-user programmers and professional programmers often make mistakes, but end-user programmers rarely possess the organized test suites and knowledge of software engineering methodologies that professional programmers have to mitigate problems. Unfortunately, up to 90% or more of spreadsheets contain faults [7, 10]. Because spreadsheets are often used for important tasks and decisions,

faults in them have been tied to costly errors.¹ The potential risks of spreadsheet faults extend beyond monetary costs, particularly in light of the Sarbanes-Oxley Act of 2002, a law which requires corporations to examine the validity of their spreadsheets [8].

Although spreadsheets are essentially a grid of cells, various information bases can be extracted out of spreadsheets, and each information base can highlight different categories of faults. For example, cells often contain explicit relationships to other cells, in the form of cell references, from which data flow graphs emerge; these data flow graphs can be used to identify reference faults² [5]. Furthermore, the juxtaposition of row and column headers against cells containing data within spreadsheets typically implies spatial relationships among cells, from which unit inference graphs emerge. Unit inference can be used to identify certain types of reference, range, and omission faults [2]. Other information bases supplied by end users can assist fault localization. For example, the value of cells is often expected to fall within certain intervals; by asserting intervals on cells, cells whose values fall outside their intervals can be located [4, 3, 5]. Adding assertions helped significantly with non-reference faults, suggesting that the addition of assertions into the environment fills a need not met effectively by the data flow testing methodology alone [5]. Furthermore, in several domains, particularly finance, it is often the case that two cells within a spreadsheet must add up to the same value; asserting relationships such as equality among groups of cells can be used to audit spreadsheets. Our work in progress to improve fault localization is based on the assumption that reasoning about faults in only one way is insufficient to locate several different categories of faults effectively.

Our emerging prototype relies on the results of the independent reasoning systems found in UCheck and in WYSIWYT. The two systems base their judgments on different information bases derived from spreadsheets: UCheck analyzes the spatial juxtaposition of row and column headers against data cells, whereas WYSIWYT uses data flow relationships in conjunction with users' judgments to locate faults. By leveraging the reasoning produced from two different information bases, we expect to produce better feedback. We believe that sharing the results of reasoning systems in a way sufficient to locate several categories of faults requires a shared reasoning database and heuristics to resolve competing and sometimes conflicting suggestions from different systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

¹<http://www.eusprig.org/stories.htm>

²One classification scheme we have found to be useful in our previous research involves two fault types: reference faults, which are faults of incorrect or missing references, and non-reference faults, which are all other faults.

2 Background

2.1 WYSIWYT (What You See is What You Test)

The fault localization system found in WYSIWYT relies on users checking off at least some of the cell values that are correct (with checkmarks) or incorrect (with X-marks) to locate cells containing faults. By allowing users to incrementally test spreadsheets as they develop them, the WYSIWYT fault localization and testing methodology maintains the interactive nature of spreadsheet systems [12, 11]. WYSIWYT provides automatic, immediate visual feedback about “testedness” for cell values through cell border colors, and users of WYSIWYT are able to improve their test effectiveness without training in testing theory [12]. From users’ judgments of cells, WYSIWYT determines fault likelihood for each cell based on the backwards slice of cells marked by users as wrong. WYSIWYT presents fault localization feedback to users by progressively shading cells darker the more likely they contain faults, as shown in Figure 1.

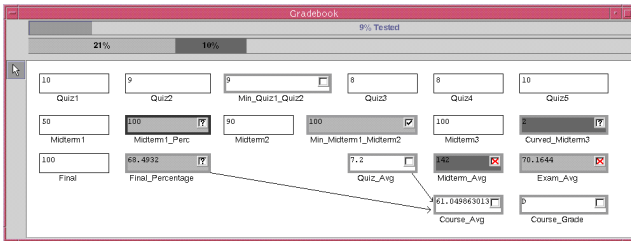


Figure 1. Users’ judgments and fault localization feedback

2.2 UCheck

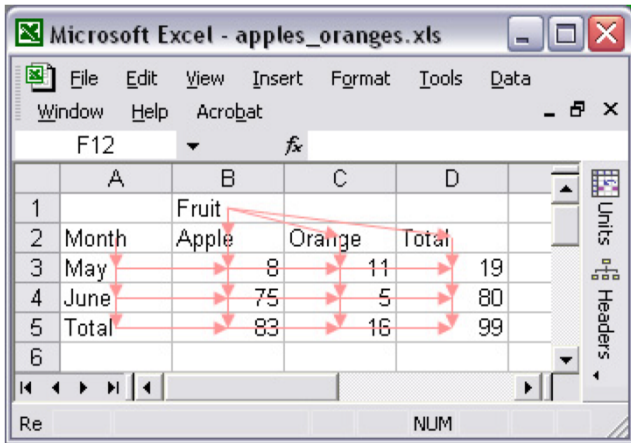


Figure 2. Headers inferred from spreadsheet layout

To locate faults, UCheck first analyses the spatial structure of the spreadsheet to then perform unit inference [1, 6]. UCheck examines the layout to determine the relationship between labels and data cells, as shown in Figure 2. From this information, UCheck can infer the units that apply to all non-blank cells in the spreadsheet. For example, UCheck understands that the unit of cell B3 in Figure 2 represents not just an apple, but also a kind of fruit. UCheck also understands that B3 also is associated with the month of May. From this understanding of units, UCheck can identify when cells inappropriately combine incompatible units, as shown in Figure 3.

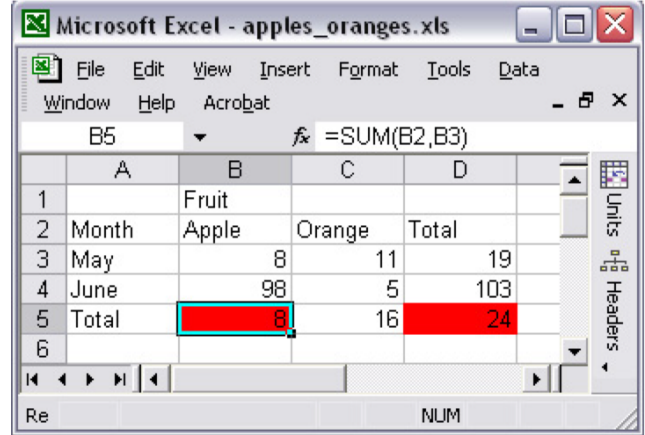


Figure 3. Range error identified from analysis

3 The evaluation testbed

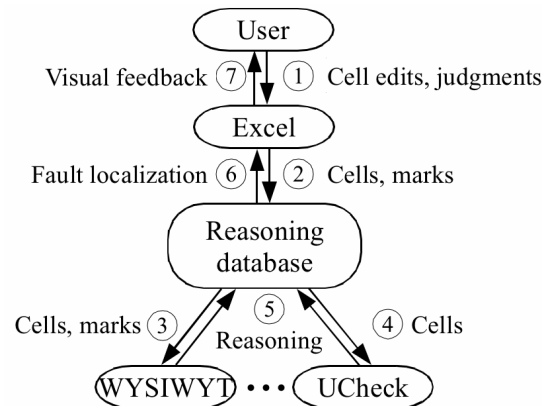


Figure 4. The design of the evaluation testbed

Figure 4 shows the design of the evaluation testbed and the sequential flow of information among the components in the proposed system. Between steps 2 and 6 in Figure 4, the reasoning database propagates cell edits to WYSIWYT and UCheck, then aggregates fault localization reasoning from each system and finally applies heuristics to select and combine fault localization feedback from the two systems to send back to Excel. Note that the design depicted in Figure 4 suggests the possibility of including additional reasoning systems in the future; for now, only the feedback from WYSIWYT and UCheck are used.

Evaluating the proposed system requires a comparison of the known faults in a spreadsheet with the feedback generated by WYSIWYT, UCheck, and the combined feedback from the two systems. Table 1 shows the four possible ways fault localization feedback corresponds to the actual faults for each cell.

Table 1. Fault localization feedback vs. actual faults

	Cell formula	
Fault localization feedback	Right formula	Faulty formula
Cell is Correct	CR	CF
Cell is Incorrect	IR	IF

We are in the process of implementing our prototype so as to empirically investigate the following questions:

- How well do these systems compare in correctly locating faults (IF)?
- When do these systems falsely identify correct cells as faults (IR)?
- When do these systems falsely identify faulty cells as correct (CF)?
- When do the systems disagree in their feedback?
- What heuristics are most effective in selecting and combining feedback?

4 Conclusion

We have presented our work in progress on experimenting with and empirically evaluating the effectiveness of sharing the results from multiple reasoning systems to improve spreadsheet fault localization. We hope that this approach will prove flexible and beneficial enough to allow a large portfolio of reasoning devices to be brought to bear on spreadsheet errors.

5 References

- [1] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.
- [2] R. Abraham and M. Erwig. How to communicate unit error messages in spreadsheets. In *WEUSE I: Proceedings of the first workshop on End-user software engineering*, pages 1–5, New York, NY, USA, 2005. ACM Press.
- [3] Y. Ayalew. *Spreadsheet Testing Using Interval Analysis*. PhD thesis, Universität Klagenfurt, 2001.
- [4] Y. Ayalew, M. Clermont, and R. Mittermeir. Detecting errors in spreadsheets. In *Proceedings of EuSpRIG 2000 Symposium: Spreadsheet Risks, Audit and Development Methods*, 2000.
- [5] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *International Conference on Software Engineering*, pages 93–103, 2003.
- [6] M. Erwig and M. Burnett. Adding apples and oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, pages 173–191, 2002.
- [7] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. In *Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.
- [8] R. R. Panko and N. Ordway. Sarbanes-Oxley: What about all the spreadsheets? In *European Spreadsheet Research Information Group*, 2005.
- [9] A. Phalgune, C. Kissinger, M. Burnett, C. Cook, L. Beckwith, and J. R. Ruthruff. Garbage in, garbage out? An empirical look at oracle mistakes by end-user programmers. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005.
- [10] K. Rajalingham, D. R. Chadwick, and B. Knight. Classification of spreadsheet errors. In *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2001.
- [11] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Trans. Software Engineering and Methodology*, 10(1):110–147, 2001.
- [12] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. In *ICSE '00: 22nd International Conf. Software Engineering*, pages 230–239, 2000.
- [13] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. F. II, and M. Main. End-user software visualizations for fault localization. In *Proceedings of ACM Symposium on Software Visualization*, pages 123–132, 2003.
- [14] J. R. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, and M. Burnett. Interactive, visual fault localization support for end-user programmers. *Journal of Visual Languages and Computing*, 16(1-2):3–40, 2005.
- [15] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In M. Erwig and A. Schürr, editors, *IEEE Symposium on Visual Languages and Human Centric Computing*, pages 207–214, 2005.

End-User Software Engineering in Natural Language

Henry Lieberman

MIT Media Lab
20 Ames St. 384A
Cambridge, MA 02139 USA
lieber@media.mit.edu

Hugo Liu

MIT Media Lab
20 Ames St.
Cambridge, MA 02139 USA
hugo@media.mit.edu

Ying Li

MIT Media Lab
20 Ames St.
Cambridge, MA 02139 USA
cyli@media.mit.edu

Abstract

In the search for easier-to-use environments for End-Users to do software development, everybody overlooks the obvious choice – using natural language to communicate between the user and the machine. Problems of ambiguity and imprecision are usually taken to be prohibitive, but we believe that modern natural language processing techniques and Common Sense reasoning can be used to create a workable environment for the creation and modification of programs. We present Metafor, a program outliner/editor that takes natural language input and allows a user to have a dialogue with the system about program construction.

Keywords

Natural language processing, Programming, Software Engineering, Dialogue management,

ACM Classification Keywords

D.1 PROGRAMMING TECHNIQUES (E), D.2 SOFTWARE ENGINEERING (K.6.3), D.3 PROGRAMMING LANGUAGES, I.2 ARTIFICIAL INTELLIGENCE

Natural Language Interaction for Software Engineering

We explore the idea of using descriptions in a natural language like English as a representation for programs. While we cannot yet convert arbitrary English descriptions to fully specified code, we can use a reasonably expressive subset of English as a conceptualization, visualization, editing and debugging tool. Simple descriptions of program objects and their behavior are converted to scaffolding (underspecified) code fragments, that can be used as feedback for the designer, and which can later be elaborated. Roughly speaking, noun phrases can be interpreted as program objects; verbs can be functions, adjectives can be properties. A surprising amount of information about program structure can be inferred by our parser from relations implicit in the linguistic structure. We refer to this phenomenon as *grammatical semantics*. We present a program editor, *Metafor*, that dynamically converts a user's stories into program code, and in a user study, participants found it useful as a brainstorming tool.

Metafor has some interesting capabilities for *refactoring* programs. Different ways of describing objects in natural language can give rise to different representation and implementation decisions as embodied in the details of the code. Conventional programming requires making up-front commitments to overspecified details, and saddles the user with having

to perform distributed, error-prone edits in order to change design decisions. Metafor uses the inherent "ambiguity" of natural language as an advantage, automatically performing refactoring as the system learns more about the user's intent.

The screenshot shows the Metafor programming environment with three panes. The top-left pane shows a conversation log:

```
[user] (01:28:32) Pacman is a character who loves to run through a maze and eat dots.
[agent] (01:28:32) Ok. I created a new agent Pacman that is a kind of character agent. I added the ability for Pacman to run, which can be through a maze. I added the ability for Pacman to eat. A dot is something which can be eaten.
[user] (01:28:53) Whenever Pacman eats a dot, it disappears and he wins a point.
[agent] (01:28:53) Ok. Now we are going to think about Pacman eating... When Pacman eats a dot, the dot disappears and Pacman wins a point. ...Now that we are done thinking about Pacman eating, we are going to resume thinking about __main__.
```

The top-right pane shows the DEICTIC STACK and DIR:

```
DEICTIC STACK: [('Pacman', ['he', 'him', 'his', 'her', 'she', 'hers', 'who']), ('character', ['it']), ('maze', ['it']), ('dots', ['they', 'them']), ('dot', ['it']), ('dot', ['it']), ('Pacman', ['he', 'him', 'his', 'her', 'she', 'hers', 'who']), ('dot', ['it']), ('point', ['it'])]
DIR: ['__main__.Pacman', '__main__.dot']
```

The bottom-right pane shows the CODETREE and the generated Python code:

```
CODETREE: [['__main__', 'FunctionT']]
def __main__():
    class Pacman(character):
        def run(maze):
            pass

        def eat(dot):
            dot.disappear()
            Pacman.win(point)

        def win(point):
            pass

    class dot:
        def disappear():
            pass
```

The bottom-left pane shows a note: "When Pacman is running through the maze, if a ghost catches him, then he loses and the game is over."

Figure 1. The Metafor programming environment. Natural language input at the lower left produces Python code at the lower right. The other two panes display system state and are not intended for the end-user.

References

Hugo Liu and Henry Lieberman (2005) Programmatic Semantics for Natural Language Interfaces. Proceedings of the ACM Conference on Human Factors in Computing Systems, CHI 2005, April 5-7, 2005, Portland, OR, USA. ACM Press.

Hugo Liu and Henry Lieberman (2005) Metafor: Visualizing Stories as Code. Proceedings of the ACM International Conference on Intelligent User Interfaces, IUI 2005, January 9-12, 2005, San Diego, CA, USA, to appear. ACM 2005.

Hugo Liu and Henry Lieberman (2004) Toward a Programmatic Semantics of Natural Language. Proceedings of VL/HCC'04: the 20th IEEE Symposium on Visual Languages and Human-Centric Computing. pp. 281-282. September 26-29, 2004, Rome. IEEE Computer Society Press.

Henry Lieberman and Hugo Liu. Feasibility Studies for Programming in Natural Language. H. Lieberman, F. Paterno, and V. Wulf (Eds.) Perspectives in End-User Development, to appear. Springer, 2006.

Abstractions for End-Users

Michael Toomim
Department of Computer Science
University of Washington
toomim@cs.washington.edu

ABSTRACT

Software Engineers use abstractions to make software scalable and avoid inconsistency errors. End-users, however, are abstraction-averse, preventing them from managing large, complex documents. We have been developing an environment-based technique, called Linked Editing, as a lightweight form of abstraction for end-users.

INTRODUCTION

Abstractions are fundamental tools in Software Engineering. They allow software to scale in size by encapsulating re-used concepts, and reduce errors by ensuring consistency across instantiations.

End-users, however, tend to work without abstraction. They use graphical direct manipulation environments—WYSIWYG word processors and web page editors, paint and illustration environments, spreadsheets, 3D CAD environments, music score editors—where they manipulate concrete objects of their interest with incremental actions and immediate visual feedback. Such environments have succeeded with end-users by providing *concrete* interaction models. However, concreteness forfeits the benefits of abstractions: when Direct Manipulation documents contain re-used or duplicated content (e.g. repeated styles) they can be difficult to scale and prone to inconsistencies.

As a result, interface designers have developed a variety of special-purpose abstraction facilities for these authoring environments. Powerpoint provides the concept of an abstract “master slide” that all concrete slides inherit from. Many music sequencers allow the user to specify bars that repeat for multiple measures. Microsoft Word and the W3C’s HTML introduce elaborate style sheet systems. Dreamweaver provides a “template” abstraction to maintain consistent headers, footers, and navigation bars across a website.

Unfortunately, the use of these abstraction features is often problematic. As put by Green & Blackwell: “Thinking in abstract terms is difficult: it comes late in children, it comes late to adults as they learn a new domain of knowledge, and it comes late within any given discipline.” [5] Abstraction features are difficult to learn, and each authoring environment has unique special-purpose abstraction mechanisms, in-

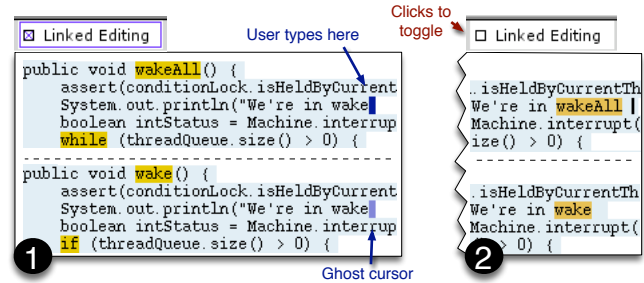


Figure 1: Prototype of Linked Editing for programmers in a text editor

hibiting knowledge transfer. Abstractions are difficult to design and implement: doing so requires much mental effort and planning, and unanticipated changes can require a re-architecture, giving users reason to put off abstraction for fear of premature commitment. Sometimes end-users avoid abstract or indirect interfaces because they find concrete operations easier to predict and safer to trust [2]. On the other hand, designers often constrain the power and applicability of abstractions in an effort to make them more concrete. The Powerpoint master slide, for instance: cannot be parameterized, is global and singular (users cannot create multiple slide styles per presentation), and has a fixed granularity (users cannot abstract content within a slide, nor abstract sets of multiple slides). Thus, the master slide’s applicability as an abstraction mechanism is limited. Content abstractions are far from panacea: they are difficult to learn, constrained in applicability, and place layers of indirection between the user and his or her objects of interest—defeating the original purposes and advantages of Direct Manipulation and concrete WYSIWYG interaction.

As a result, users often work without abstractions. Even expert programmers do so—studies show that the Linux kernel, Java JDK, FreeBSD, MySQL, PostgreSQL, and X Window System are all 20–30% duplicated, and some software is as much as 60% duplicated [6, 8, 9]. But end-users are dramatically more abstraction-averse. For instance, Blackwell found that, in a group of Microsoft Word users, end-users were less than a tenth as likely to create “text style” abstractions in their documents as programmers and scientists, even if they knew how to use that feature of Word [2, 3]. End-users were similarly less likely to create a range of other abstractions, such as nested directory structures, bookmark categories, and telephone quick-dial codes. In another domain, Bellotti reports that designers principally work in terms of concrete repre-

sentational artifacts, rather than abstract concepts, even if abstractions are available [1]. In software, novice programmers are known to create fewer abstractions than experts [4]. Thus, while programmers create far fewer abstractions than would be ideal, end-users and novices create very few at all.

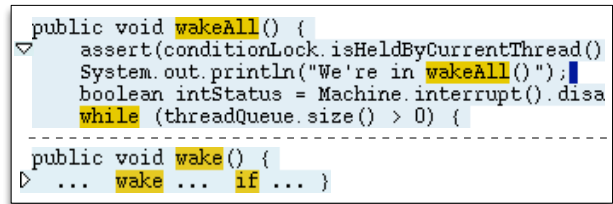
This is unfortunate. If end-users will not abstract away patterns of duplication, they will be unable to author, understand, and modify digital documents beyond a certain size and complexity. An abstractionless user could certainly not extend an abstractionless CNN.com, for example. Nor could such a user easily work in other domains; e.g. authoring a computer-graphic landscape with hundreds of trees, or editing an electronic music score with hundreds of voices. This would be a regretful scenario, since the ideas on CNN.com’s website are not difficult for an end-user to comprehend, edit or express—but rather the structural characteristics of their transcription in a website.

Linked Editing: Abstraction in concrete interfaces

We believe that an intelligent authoring environments can remedy this problem. Linked Editing [9] is a novel technique we are developing for visualizing and editing duplication without explicit abstraction or additional layers of indirection. Our hypotheses are that end-users will prefer its concrete interaction style over abstraction, and that it will let them edit larger, more complex documents than they would otherwise be able to, with fewer errors.

Figure 1 displays the current implementation of Linked Editing, which was developed for programmers, rather than end-users, to help them manage duplicated code. The system automatically finds duplicated code, and highlights common regions in blue and differences in yellow. Then the user can edit all instances at once by editing any single instance (a variant of Simultaneous Editing [7]). If the user wants to edit just a single instance, she toggles the “Linked Editing” mode checkbox on the toolbar before typing, and the system incrementally finds and highlights the new similarities and differences. Linked Editing allows duplication to be edited scalably with Simultaneous Editing. By highlighting similarities and differences, simultaneous edits are predictable (blue regions are guaranteed to be identical after arbitrary edits) and unintended inconsistencies are highlighted in yellow and thus can be avoided.

Linked Editing for web authoring We are now extending Linked Editing to a variety of end-user authoring environments. Here we will illustrate how we envision it assisting an end-user to modify CNN.com in a WYSIWYG web authoring environment. First, the system automatically analyzes the website and finds all patterns of duplication (using a custom algorithm we are developing). Then, as the user moves her cursor over a duplicated block of content, such as a navigation bar, the system provides a visualization of the block’s corresponding copies—miniature depictions of the navigation bars on other CNN web pages. The user can now change all navigation bars simultaneously by simply editing any one. The user can also make changes to any single instance or subset of instances. For example, the user may want to modify the navigation bars on all “science” pages to have additional entry for the “computer science” news category. First, she



```
public void wakeAll() {
    assert(conditionLock.isHeldByCurrentThread());
    System.out.println("We're in wakeAll()");
    boolean intStatus = Machine.interrupt().disa
    while (threadQueue.size() > 0) {
}

public void wake() {
    ... wake ... if ... }
```

Figure 2: An elided block of duplication looks similar to a function definition and use

would select one or two science pages from the miniaturized visualization. The system then infers by example that the user is selecting all pages with “science” in their header. It briefly highlights, in orange, the word “science” in each document’s header to indicate the pattern it inferred. The user now simultaneously edits the desired entry into all science navigation bars.

Note that introducing a new type of difference amongst duplicated instances, as was done here, would be much more difficult using a traditional template or function abstraction: the user would have had to add a new template definition, or parameter in the function definition, to represent the new type of difference (science or not science page) as well as modify each use of the function or template to provide the appropriate parameter or select the appropriate template. In general, abstraction systems become more complicated as additional differences are required. The system described here, however, adapts automatically to the concrete content created and infers an implied inheritance hierarchy behind the scenes.

Transitioning to traditional abstractions Figure 2 shows the result of clicking a button to elide the identical portions of block from view, leaving only the differences visible. This is similar to how a function call hides the function’s body and shows only parameters. In the future we also envision allowing the user to specify an optional name for a repeated block of content, and optionally transform the content into a traditional abstraction. By making names, elision, and simultaneous editing optional and independent, the system provides the user with a continuum of incremental abstraction, letting users work concretely or abstractly, at their discretion.

Other examples of duplication There are a variety of duplication situations in which Linked Editing could be useful beyond those already given. For instance, spreadsheet users often copy and paste complex formula between cells, to perform similar calculations. Accountants sometimes create duplicated versions of entire sheets, with minor changes, to analyze “what-if” scenarios. Secretaries periodically compose form letters and want them personalized for some recipients, which is difficult to accomplish with a “database merge” abstraction. Music composers repeat melodies and drum beats across an entire score, but modify them for some measures. Presenters copy graphical diagrams to multiple slides, modify them on certain slides to represent change, and then need to update an aspect of all diagrams at once. Researchers create multiple versions of a user study script for each condition of the experiment, and must be extremely careful to maintain differences (the independent variables) as they copy and

paste and evolve the scripts in parallel.

Preliminary results

We conducted a user study comparing Linked Editing with functional abstraction. Linked Editing took dramatically less time to implement and use, and resulted in code that programmers reported as being easier to understand and change [9]. These results are very encouraging, and we suspect they will be similar for end-users, in non-programming situations.

RELATED WORK

Lapis [7] introduced Simultaneous Editing, but supports one-off interactive edits rather than persistent abstractions, and differs from Linked Editing in other ways as described in [9]. Other projects have implemented demonstrational inference for specific subtypes of duplication (*e.g.* Tourmaline [10] infers styles in word processing documents) but are not as general as Linked Editing.

CONCLUSION

With or without abstractions, authoring and maintaining large documents is a major challenge in end-user software engineering. By providing abstraction-like scalability benefits without requiring layers of abstract indirection, Linked Editing may be a solution that end-users can benefit from.

REFERENCES

1. Victoria Bellotti, Simon Buckingham Shum, Allan MacLean, and Nick Hammond. Multidisciplinary modelling in hci design...in theory and in practice. In *Proceedings of the conference on Human Factors in Computing Systems*, pages 146–153, 1995.
2. Alan F. Blackwell. See what you need: Helping end-users to build abstractions. *Journal of Visual Languages and Computing*, 12:475–499, 2001.
3. Alan F. Blackwell. Personal Communication, 2004.
4. Francios Detienne. *Software Design – Cognitive Aspects*. Springer, 2002.
5. Thomas Green and Alan Blackwell. Cognitive dimensions of information artefacts: a tutorial. *BCS HCI Conference*, <http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/>, 1998.
6. Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th International Conference on Operating Systems Design and Implementation*, December 2004.
7. Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the USENIX Annual Technical Conference*, pages 161–174, 2001.
8. Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web portals. In *International Conference on Web Engineering, ICWE'05*, pages 252–262. Springer, July 2005.
9. Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *Proceedings of the IEEE Symposium on Human-Centered Computing and Visual Languages (to appear)*. IEEE, 2004.
10. Andrew J. Werth and Brad A. Myers. Tourmaline: Macrostyles by example. In *Proceedings INTER-CHI'93: Human Factors in Computing Systems*, page 532, April 1993.

End-User Programming at the University of Washington

Daniel S. Weld

Pedro Domingos

Raphael Hoffman

Sumit Sanghai

Department of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350 USA

Abstract

Over the past decade our research group at the University of Washington has investigated a number of techniques for improving end-user customization and programming. Much of this work has been reported in the AI literature, and we seek to participate in the Second Workshop on End-User Software Engineering in order to expand our understanding of existing work and alternative approaches.

1 Introduction

Starting with the Internet Softbots project [5], our research group at the University of Washington has been seeking new ways to facilitate end-user customization of their computational environment. Our work has included:

- Planning-based software agents, which synthesized and executed small programs from formal specifications [8; 7].
- The SMARTedit and SMARTpython programming-by-demonstration (PBD) systems, based on version-space algebra [10; 11].
- Relational Markov Models (RMMs), a learning method for predicting when a user may start executing a repetitive sequence of actions [1].
- Dynamic Markov Logic Networks, a statistical-relational learning engine, which improves on both version-space algebra and RMMs [17].
- The ASSIEME script recommendation engine.

End-user software engineering is especially important when programs are generated by demonstration with machine learning algorithms. Errors, debugging and visualization are important challenges for all programming environments, but are crucial when statistical or AI techniques are involved.

In the rest of this position paper we briefly describe some of our work and current directions.

2 Background

Mackay [13] studied the customization behavior of users of a Unix software environment and found that people do not take advantage of customization features, even if it made their

work more efficient. The main barrier was the difficulty in making modifications, and people only did customize when something broke or they had to learn a new environment. Carroll and Rosson [3] suggest that users are biased towards making concrete, short-term progress. As a result, they are more likely to stick with known procedures than invest time learning about system features. In contrast, a survey on the use of a word processor by Page et.al. [16] showed that 92% of the participants *did* perform some form of customization. However, the authors remark that most participants were heavy users and many of the considered customizations were simple to do.

Although many people seem to be reluctant to customize their software environment, Mackay [12] and Gantt and Nardi [6] discovered that members of an organization tend to share customizations. Typically, some people experiment with the system and inform other users about useful customizations.

From this work, we draw two conclusions, which motivate our work:

- Users will customize more if it is easier to do so. We hope PBD will simplify customization.
- Users are often spurred to customize, when inspired by other users who suggest about useful customizations. Possibly the interface, itself, could make these suggestions?

3 Research at the University of Washington

Due to space constraints, we limit our discussion to two PBD systems (one powered by version-space algebra and the other by dynamic Markov logic networks) and a system for recommending relevant Web browser customizations.

3.1 Programming by Demonstration

In 1998, we started working on machine learning approaches to programming by demonstration (PBD). Of course, PBD has been studied extensively [4], but most previous systems were domain-specific. We sought a domain-independent approach suitable for deep deployment that offered the expressiveness of a scripting language and the ease of macro recording, without its accompanying brittleness.

It is useful to think of a PBD-interface as having three components: 1) *segmentation* determines when the user is executing an automatable task, 2) *trace induction* predicts what the

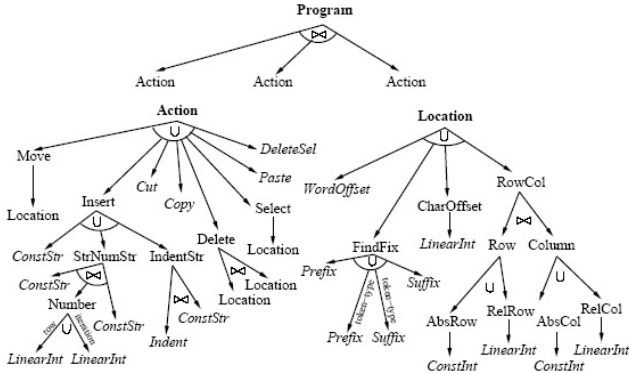


Figure 1: Composite version space for SmartEdit

user is doing from a prefix of her activity trace, and 3) *facilitation* manages user interaction to aid the user in completing her task. The next section treats segmentation in depth, but for our PBD work we assumed that the user would notify the interface when trace induction was desired, via “start” and “stop” buttons like those in a macro recorder. For the facilitation phase, we investigated decision-theoretic control [18], but many issues (e.g., saving learned procedures for future use, means for convenient invocation, etc.) remain. The initial focus of our work was on the trace induction phase.

We formalized PBD trace induction as a learning problem as follows. A repetitive task may be solved by a program with a loop, where each iteration solves one instance of the task. The PBD system must infer the correct program from a demonstration of the first few iterations. Each action (e.g., move, select, copy, paste, ...) the user performs during this demonstration causes a change in the state of the application (e.g., defines a mapping between editor states). Therefore, we modeled this problem as one of inferring the function that maps one state to the next, based on observations of the state prior to and following each user action.

3.2 Version-Space Algebra

PBD presents a particularly challenging machine learning problem, because users are extremely reluctant to provide more than a few training instances. Thus the learner must be able to generalize from a very small number of iterations. Yet in order to be useful, a wide range of programs must be learnable. Thus the problem combines a weak bias with the demand for low sample complexity. Our solution, called *version-space algebra*, lets the application designer combine multiple strong biases to achieve a weaker one that is tailored to the application, thus reducing the statistical bias for the least increase in variance. In addition, the learning system must be able to interact gracefully with the *user*: presenting comprehensible hypotheses, and taking *user* feedback into account. Version-space algebra addresses this issue as well.

Originally developed for concept learning, a *version space* is the subset of a hypothesis space which is consistent with a set of training instances [15]. If there is a partial order over candidate hypotheses, one may represent the version space implicitly (e.g., with boundary sets) and manage updates efficiently. Version-space algebra defines transformation operators (e.g., union, join, etc.) for combining simple version

spaces into more complex ones. We also developed a probabilistic framework for reasoning about the likelihood of each hypothesis in a composite version space. After constructing a library of reusable, domain-independent, component version spaces, we combined a set of primitive spaces to form a bias for learning text-editing programs (Figure 1), which was used in the SmartEdit implementation. Version-space algebra affords two benefits to a PBD system: 1) the ability to specify domain-specific details necessary to guide a learner with a simple algebraic expression (i.e., a formula equivalent to the structure of Figure 1), and 2) a fast learning method which uses this expression to guide consideration of possible programs.

3.3 PBD with Dynamic Markov Logic Networks

More recently, we have employed *dynamic Markov logic networks* (DMLNs) [17] to do PBD. DMLNs are a probabilistic extension of first-order and temporal logic which consist of weighted first-order formulas describing the temporal relationships between the objects in a system. DMLNs can be used to model and learn stochastic processes, i.e., the preconditions and effects of actions, the transitions between the actions and the relationships between the hidden and observed properties of objects in the domain. In most real-world domains, the effects of an action are uncertain and a DMLN represents this using weighted first-order rules where the higher the weight, the more likely the effect.

The major advantage of using DMLNs for PBD is that one can learn first-order rules that capture the preconditions and effects of an action or transitions between them. For example, an expert can demonstrate the task of saving emails to a newly created folder and would like the PBD system to complete it for them. Using a DMLN, one can learn that the user was trying to save only those emails that belonged to his thesis based on the contents and the sender and recipients. Such tasks cannot be easily (if at all) learned using propositional learners. Another advantage of DMLNs stems from its robustness to noisy training examples. It is capable of inducing a program even if the user makes a small error during demonstration (it can also identify these mistakes to verify that they were unintended).

DMLNs also allow us to combine the segmentation and trace induction phases of PBD. For example, we have modeled the desktop activity of a user simultaneously working on several tasks (i.e., switching between them). We use a DMLN to look for common transition patterns (both at the propositional and first-order level) between the actions to segregate the tasks and then learn models for each task. Our DMLN learning method is implemented and works on examples of the form described above, but has not yet been implemented into a full PBD system.

3.4 Sharing Browser Customizations

While a PBD system might become easier than manual programming, program reuse is the focus of the ASSIEME project. Motivated by Mackay’s observations [12], we seek ways for users to share browser customizations. In many ways our system is similar to alerting systems that advice novice users about system functionality that might be helpful, except that the likelihood of the user being unaware is even greater in our context.

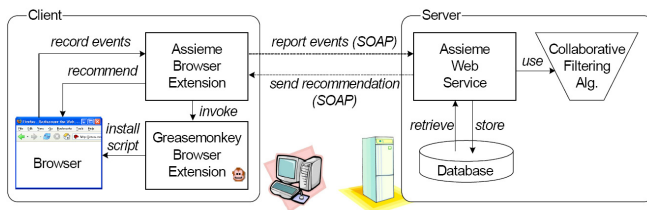


Figure 2: Architecture of ASSIEME.

Specifically, ASSIEME is a recommender system [2] for client-side Webpage customizations. ASSIEME—designed as an extension to the Firefox browser—records event traces of user browsing behavior. This recorded information is transmitted to a central server, and the server computes customization recommendations based on the similarity of multiple user models, which consist of event traces, installed customizations, and user responses to previous recommendations. Recommendations are transmitted back to the user who may accept or reject the installation of a new customization. We currently support client-side Webpage customizations written in JavaScript for the Greasemonkey Firefox extension.

Since the development of the client-side customization scripts requires programming skills, our system does at this point not yet offer the same flexibility as a PBD system. However, we believe that there are many customizations which have been developed and made publicly available. Our system facilitates sharing of these customizations, which often exhibit very complex behavior, because they are written by sophisticated programmers. Our main challenges lie in the design of an accurate recommendation algorithm and a secure communication protocol that respects every user’s privacy.

Our work is not the first to address sharing of customizations. Kahler [9] developed a system that allows users to explicitly share word processor customizations with colleagues. Unlike our approach, Kahler’s system does not automatically track customization usage nor provides personalized recommendations. Client-side customization for webpages has also been previously proposed. Miller and Myers [14] integrated a command shell into a web browser to enable simple forms of automation. Today, the Greasemonkey extension to the Firefox webbrowser enables simple installation of more than 3000 publicly available customization scripts.

4 Conclusions

We aspire to the CHI workshop on EUSE, because we stand to learn much from the community. In particular, our work has not yet paid sufficient attention to problems, such as informing the user the nature of the program induced by the PBD algorithm—this is a critical weakness and we believe that visual programming languages may be a key component of the solution. Furthermore, we hope that our background in AI and machine learning could contribute to the workshop discussions.

References

- [1] C. R. Anderson, P. Domingos, and D. S. Weld. Relational Markov models. *KDD-02*, August 2002.
- [2] John S. Breese, David Heckerman, and Carl Myers Kadie. Empirical analysis of predictive algorithms for collaborative filtering. *UAI*, p43–52, 1998.
- [3] John M. Carroll and Mary Beth Rosson. Paradox of the active user. *Interfacing thought: cognitive aspects of human-computer interaction*, pages 80–111, MIT Press, 1987.
- [4] Allen Cypher, editor. *Watch what I do: Programming by demonstration*. MIT Press, 1993.
- [5] O. Etzioni and D. Weld. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.
- [6] Michelle Gantt and Bonnie A. Nardi. Gardeners and gurus: patterns of cooperation among cad users. *CHI-92*, p107–117, 1992.
- [7] K. Golden and D. Weld. Representing sensing actions: The middle ground revisited. *KR-96*, p174–185, 1996.
- [8] Keith Golden, Oren Etzioni, and Dan Weld. Omnipotence without omniscience: Sensor management in planning. *AAAI-94*, p1048–1054, 1994.
- [9] Helge Kahler. More than words - collaborative tailoring of a word processor. *J. UCS*, 7(8):826–847, 2001.
- [10] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. *ICML-00*, p527–534, June 2000.
- [11] Tessa Lau, Pedro Domingos, and Daniel S. Weld. Learning programs from traces using version space algebra. *K-CAP-03*, p36–43, 2003.
- [12] W. E. Mackay. Patterns of sharing customizable software. *CSCW-90*, p209–221, 1990.
- [13] W. E. Mackay. Triggers and barriers to customizing software. *CHI-91*, p153 – 160, 1991.
- [14] Robert C. Miller and Brad A. Myers. Integrating a command shell into a web browser. *USENIX Annual Technical Conference*, p171–182, 2000.
- [15] T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.
- [16] Stanley R. Page, Todd J. Johnsgard, Uhl Albert, and C. Dennis Allen. User customization of a word processor. *CHI-96*, p340–346, 1996.
- [17] S. Sanghai, P. Domingos, and D. Weld. Learning models of relational stochastic processes. *ECML-05*, October 2005.
- [18] Steven A. Wolfman, Tessa Lau, Pedro Domingos, and Daniel S. Weld. Mixed initiative interfaces for learning tasks: Smartedit talks back. *IUI-01*, p167–174, January 2001.