07081 Abstracts Collection
End-User Software Engineering
— Dagstuhl Seminar —

Margaret M. Burnett[1], Gregor Engels[2], Brad A. Myers[3] and Gregg Rothermel[4]

[1] Oregon State University, US
`burnett@cs.orst.edu`
[2] University of Paderborn, DE
`engels@upb.de`
[3] Carnegie Mellon University - Pittsburgh, US
`bam@cs.cmu.edu`
[4] University of Nebraska - Lincoln, US
`grother@cse.unl.edu`

Abstract. From 18.01.07 to 23.02.07, the Dagstuhl Seminar 07081 "End-User Software Engineering" was held in the International Conference and Research Center (IBFI), Schloss Dagstuhl. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar as well as abstracts of seminar results and ideas are put together in this paper. The first section describes the seminar topics and goals in general. Links to extended abstracts or full papers are provided, if available.

Keywords. End user software engineering, end-user programming, human-computer interaction, programming language design

07081 Executive Summary – End-User Software Engineering

From 18.01.07 to 23.02.07, the Dagstuhl Seminar 07081, "End-User Software Engineering", was held in the International Conference and Research Center (IBFI), Schloss Dagstuhl. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. This document summarizes the event.

Keywords: End user software engineering, end-user programming, human-computer interaction, programming language design

Joint work of: Burnett, Margaret M.; Engels, Gregor; Myers, Brad A.; Rothermel, Gregg

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1098

Gender HCI Issues in End-User Software Engineering Environments

Laura Beckwith (Oregon State University, USA)

Although gender differences in a technological world are receiving significant research attention, much of the research and practice has aimed at how society and education can impact the successes and retention of female computer science professionals. The possibility of gender issues within software, however, has received almost no attention. We hypothesize that factors within software have a strong impact on how well female problem solvers can make use of the software. Evidence from other fields and investigations of our own have revealed evidence supporting this hypothesis.

Keywords: End-user software engineering, gender, self-efficacy

Joint work of: Beckwith, Laura; Burnett, Margaret; Wiedenbeck, Susan

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1076

End User Programming for Scientists: Modeling Complex Systems

Andrew Begel (Microsoft Research - Redmond, USA)

Towards the end of the 20th century, a paradigm shift took place in many scientific labs. Scientists embarked on a new form of scientific inquiry seeking to understand the behavior of complex adaptive systems that increasingly defied traditional reductive analysis. By combining experimental methodology with computer-based simulation tools, scientists gain greater understanding of the behavior of systems such as forest ecologies, global economies, climate modeling, and beach erosion. This improved understanding is already being used to influence policy in critical areas that will affect our nation's future, and the world's.

Keywords: StarLogo, Science, Modeling, Complex Systems

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1077

Empirical Foundations for EUSE and Interdisciplinary Design for EUSE

Alan Blackwell (Cambridge University, GB)

How does EUSE research build on empirical studies of programmers, and what kinds of empirical research might provide foundations for future EUSE research?

My own work on interdisciplinary design draws comparisons across academic and professional boundaries, applying the results to the design of new technologies, and the critical assessment of technology.

Keywords: Interdisciplinary design, Empirical Studies of Programmers, Psychology of Programming, Real World Research

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1078

Empirical Studies in End-User Software Engineering and Viewing Scientific Programmers as End-Users – POSITION STATEMENT –

Jeffrey Carver (Mississippi State Univ., USA)

My work has two relationships with End User Software Engineering. First, as an Empirical Software Engineer, I am interested in meeting with people who do research into techniques for improving end-user software engineering. All of these techniques need to have some type of empirical validation. In many cases this validation is performed by the researcher, but in other cases it is not. Regardless, an independent validation of a new approach is vital. Second, an area where I have done a fair amount of work is in software engineering for scientific software (typically written for a parallel supercomputer). These programmers are typically scientists who have little or no training in formal software engineering. Yet, to accomplish their work, they often write very complex simulation and computation software. I believe these programmers are a unique class of End-Users that must be addressed

Keywords: Empirical Studies

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1079

What is an end user software engineer?

Steven Clarke (Microsoft Research - Redmond, USA)

The group of people described as end user software engineers are a very large and diverse group. For example, research scientists building simulations of complex processes are described as end user software engineers as are school teachers who create spreadsheets to track the progress of their students. Given the difference in background and domains in which different end user software engineers work, I argue that it is important to distinguish between different categories of end user software engineers. Such distinctions will enable us to determine which tools and techniques are appropriate for which types of end user software engineers. Indeed, such distinctions will also make clear the differences and similarities between end user software engineers and so called professional software engineers.

Keywords: Personas, End user software engineer,Professional software engineer

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1080

Software environments for supporting End-User Development

Maria Francesca Costabile (University of Bari, I)

Our work on End-User Development primarily focuses on the needs of a specific community of users, namely professionals in diverse areas outside of computer science, such as engineers, physicians, geologists and physicist, who are not professional programmers. We refer to them as domain experts. We developed a participatory design methodology, called SSW (Software Shaping Workshop) methodology, aimed at designing software environments that support domain experts to become co-designers of their tools. The different stakeholders can contribute their own views on the problem to design, development and maintenance of an application, using their own languages and notations.We also proposed a model of the Interaction and Co-Evolution processes (ICE model) occurring between users and system. It extends a previous model of Human-Computer Interaction by considering an important phenomenon occurring during the use of interactive systems, called co-evolution of users and systems.

Keywords: Customized software environments, meta-design, participatory design, domain expert

Joint work of: Costabile, Maria Francesca; Piccinno, Antonio

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1081

Meta-UI for Ambient Spaces: Can MDE help?

Joelle Coutaz (Université de Grenoble, F)

This position paper introduces the concept of Meta-UI and outlines our technical approach. This approach draws upon the flexibility of MDE and SOA to allow users, by the way of a meta-UI, to control the ambient space in which they live.

Keywords: Meta-UI, MDE, SOA

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1082

Rethinking the Software Life Cycle: About the Interlace of Different Design and Development Activities

Yvonne Dittrich (IT University of Copenhagen, DK)

Software engineering research addresses professional ways of designing, developing and implementing software. So far, software engineering more or less takes for granted that software professionals have control over the material implementation of a piece of software. Though users might use the software innovatively or even customise it, neither end-user tailoring (EUT) nor end-user development (EUD) are treated systematically regarding the impact of deferring part of the design to the use context on software development technologies or processes. Especially the development, adaptation and configuration of software products, software that is used by more than one user in more than one organisation makes visible that different parallel ongoing development activities often distributed over more than two organisations have to be coordinated.

Keywords: End User Development, Software Processes

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1084

Requirements and Modeling for End-User Developers

Gregor Engels (Universität Paderborn, DE)

Eliciting the requirements and creating a model of a software system are standard activities in the development process of professional software development. The talk discusses whether these two development phases are also present in end-user software development and how they could look like. It is argued that one has to distinguish between at least two types of end-user software developers. Those, who are not professional software developers, but work in an engineering domain and follow stepwise development processes. They are used to have requirements specifications as well as models, too. But, non-professional, non-engineering end-users, e.g. spreadsheet developers, don't and would not like to distinguish between different steps in the development process. Therefore, we propose to hide the distinction between these different steps by closely interconnecting requirements specification, models and code, and by putting them into one development box. By offering appropriate interface functions like create, adapt, refine, etc. to the box, the end-user is supported in developing software without being aware that he is undergoing a stepwise refinement process from requirements specifications towards concrete code.

Keywords: End-User Modeling

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1085

Exploiting Domain-Specific Structures For End-User Programming Support Tools

Martin Erwig (Oregon State University, USA)

In previous work we have tried to transfer ideas that have been successful in general-purpose programming languages and mainstream software engineering into the realm of spreadsheets, which is one important example of an end-user programming environment.

More specifically, we have addressed the questions of how to employ the concepts of type checking, program generation and maintenance, and testing in spreadsheets. While the primary objective of our work has been to offer improvements for end-user productivity, we have tried to follow two particular principles to guide our research.

(1) Keep the number of new concepts to be learned by end users at a minimum.

(2) Exploit as much as possible information offered by the internal structure of spreadsheets.

In this short paper we will illustrate our research approach with several examples.

Keywords: Spreadsheet, program analysis

Joint work of: Abraham, Robin; Erwig, Martin

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1086

Meta-Design: A Conceptual Framework for End-User Software Engineering

Gerhard Fischer (University of Colorado, USA)

In a world that is not predictable, improvisation, evolution, and innovation are more than a luxury: they are a necessity. The challenge of design is not a matter of getting rid of the emergent, but rather of including it and making it an opportunity for more creative and more adequate solutions to problems.

Meta-design is an emerging conceptual framework aimed at defining and creating social and technical infrastructures in which new forms of collaborative design can take place. It extends the traditional notion of system design beyond the original development of a system. It is grounded in the basic assumption that future uses and problems cannot be completely anticipated at design time, when a system is developed. Users, at use time, will discover mismatches between their needs and the support that an existing system can provide for them. These mismatches will lead to breakdowns that serve as potential sources of new insights, new knowledge, and new understanding.

Keywords: Meta-design, consumers and designers, unself-conscious cultures of design

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1087

Dependability in Web Software

Marc Fisher (University of Nebraska, USA)

The web is an increasingly important platform used for a wide variety of tasks on a regular basis. And as the web becomes more important, the ways in which it is used grows increasingly sophisticated. End users build web pages and applications, use web applications in new and unexpected ways and use web macro tools to automate web-based tasks. All of these tasks are error-prone. In addition, they often depend on external components outside of the control of the developer or end user. Therefore we have been developing tools and methodologies to assist users with these

Keywords: Web Applications, Dependability

Joint work of: Elbaum, Sebastian; Fisher II, Marc; Rothermel, Gregg

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1089

A Methodology to Improve Dependability in Spreadsheets

Marc Fisher (University of Nebraska, USA)

Spreadsheets are one of the most commonly used end user programming environments. As such, there has been significant effort on the part of researchers and practitioners to develop methodologies and tools to improve the dependability of spreadsheets. Our work has focused on the development of the "What You See Is What You Test" (WYSIWYT) family of techniques. WYSIWYT is designed to be seamlessly integrated into a spreadsheet environment and the user's development processes. It uses visual devices that are integrated into the user's spreadsheet to guide the process of finding and fixing problems with the spreadsheet.

Keywords: Spreadsheets, Dependability, Testing

Joint work of: Burnett, Margaret; Fisher II, Marc; Rothermel, Gregg

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1088

Designers Need End-User Software Engineering

Mark Gross (Carnegie Mellon Univ. - Pittsburgh, USA)

This position paper for the End-User Software Engineering workshop outlines three systems that employ end user programming for designers: a constraint-based design environment; a sketch recognition interface for knowledge based systems, and a physical programming environment for building modular robots.

Keywords: Design, end-user, programming, physical, graphics, constraints

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1090

Barriers to Successful End-User Programming

Andrew J. Ko (Carnegie Mellon University, USA)

In my research and my personal life, I have come to know numerous people that our research community might call end-user programmers. Some of them are scientists, some are artists, others are educators and other types of professionals. One thing that all of these people have in common is that their goals are entirely unrelated to producing code. In some cases, programming may be a necessary part of accomplishing their goals, such as a physicist writing a simulation in C or an interaction designer creating an interactive prototype. In other cases, programming may simply be the more efficient alternative to manually solving a problem: one might find duplicate entries in an address book by visual search or by writing a short Perl script.

Keywords: End-user programming, learning, empirical studies

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1091

End-User Software Engineering Position Paper

Henry Lieberman (MIT - Cambridge, USA)

This position paper outlines work on making programming easier, and its relationship to end-user software engineering.

Keywords: End-user programming

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1092

End Users Creating More Effective Software

Brad Myers (Carnegie Mellon University, USA)

This position paper briefly summarizes paradigms used to create end-user software.

Keywords: End user software engineering

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1093

End-User Design

Alexander Repenning (University of Lugano, CH)

Are UML diagrams a good tool to teach middle school students how to make video games? Probably not, but what kinds end-user design aids such as mental models, scaffolding structures, examples or other kinds of objects to think we can we give to end-users in order to gradually introduce them to good programming practice?

Keywords: End-user programming, end-user development, computers in education, programming environment for kids

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1099

Position paper for EUSE 2007 at Dagstuhl

Mary Beth Rosson (Penn State University, USA)

This brief position paper summarizes several facets of research underway by the Informal Learning in Software Development group at Pennsylvania State University. The focus of the work reported is on end user web development, with discussion of user needs and tools that might help to meet these needs.

Keywords: End user web development

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1092

End-User Software Engineering and Professional End-User Developers

Judith Segal (The Open University, GB)

There is a great variety of end user developers and a great variety of contexts within which they develop. End user developers may have little or no experience of using computers or may be adept coders in general purpose programming languages.

They may develop their software on their own over a few minutes or in groups over years. The software produced may be for their own use only or for a large community of users. It may be inconsequential or the consequences of its failure may be great. In this paper, we identify and discuss the problems of one particular group of end user developers – professional end user developers – who have no fear of coding and who develop software which plays a vital part in furthering their professional goals.

Keywords: Professional end user developers, scientific computing

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1095

Helping Everday Users Establish Confidence for Everyday Applications

Mary Shaw (CMU - Pittsburgh, USA)

End users obtain their desired results by combining elements of information and computation from different applications. Software engineering provides little support for identifying, selecting, or combining these elements – that is, for helping end users to design computational support for their own tasks. Software engineering provides even less support to help end users to decide whether the resulting system is sufficiently dependable – whether it will meet their expectations. Many users, especially end users, base judgments about software on informal and undependable information, and they draw conclusions with informal rather than rational decision methods. We have been developing support for everyday dependability, with an emphasis on expressing expectations in abstractions familiar to the user and on obtaining software behavior that reasonably satisfies those expectations. In this Dagstuhl I would like to explore the differences between everyday informal reasoning and the rational processes of computer science in order to develop means for establishing credible indications of confidence for end users.

Keywords: Everyday users, everyday dependability, data feeds, task level of abstraction, topes

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1096

End-User Development Techniques for Enterprise Resource Planning Software Systems

Michael Spahn (SAP Research - Darmstadt, D)

The intent of this position paper is to present the focus of interest of our end-user development (EUD) related research at SAP Research CEC Darmstadt.

As we are in an early phase of research, research topics will be presented rather than detailed results. We focus on investigating and applying EUD techniques suitable for enterprise resource planning (ERP) software systems, especially for small and medium-sized enterprises (SMEs). Our current research addresses the sub-domains of workflow management and business intelligence.

Keywords: End-User Development (EUD), Enterprise Resource Planning (ERP), Workflow Management, Business Intelligence (BI)

Joint work of: Spahn, Michael; Scheidl, Stefan; Stoitsev, Todor

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1097

End-user (further) development: A case for negotiated semiotic engineering

Clarisse de Souza (PUC-Rio de Janeiro, BR)

Semiotic Engineering views human-computer interaction as a special case of computer-mediated communication. In it, designers communicate to users their design vision about computer artifacts: what they are meant to do and how, but also what benefits they bring to users' lives (as perceived by designers). The semiotic theories of meaning subscribed by Semiotic Engineering postulate that meanings always evolve - there are no fixed meanings. Thus, the meanings users assign to computer artifacts and the meaningful situations in which they expect such artifacts to be valuable evolve. End user development is thus a requirement for "useful artifacts". Users should be able to encode new meanings and meaningfulness in them. However, there are limitations for this encoding: helping designers and users negotiate new meanings at interaction time, through the mediation of systems' interfaces, is thus a key issue for Semiotic Engineering. In this scenario, explanations - especially those about what artifacts cannot do, have not done, and why - are cruacially important.

Keywords: Semiotic engineering, explanations, representation codes

Extended Abstract: http://drops.dagstuhl.de/opus/volltexte/2007/1083

End-user Programming of Ambient Narratives

Mark van Doorn (Philips Research Lab. - Eindhoven, NL)

Ambient Intelligence is a vision on the future of consumer electronics, telecommunications and computing in which devices move into the background while at the same time placing the user experience in the foreground. Producing Ambient Intelligent environments on a large scale is problematic however. First, it is technologically not possible in the foreseeable future to mass produce a product or service that generates Ambient Intelligence, given the current state-of-the-art in machine learning and artificial intelligence. Second, it is economically not feasible to manually design and produce Ambient Intelligence applications for each person individually. One of the main research questions in creating such environments is the design of a system capable of supporting mass customization of ambient experiences by means of end-user programming. A brief outline of the approach taken to address this question is described including future research.

Keywords: Ambient intelligence, storytelling, hypertext, end-user programming

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2007/1075

Dagstuhl Seminar 07081: End-User Software Engineering

Margaret M. Burnett, Gregor Engels, Brad Myers, Gregg Rothermel

1 March, 2007

The number of end users creating software is far larger than the number of professional programmers. These end users are using various languages and programming systems to create software in forms such as spreadsheets, dynamic web applications, and scientific simulations. This software needs to be sufficiently dependable, but substantial evidence suggests that it is not.

Solving these problems involves not just software engineering issues, but also several challenges related to the users that the end user software engineering intends to benefit. End users have very different training and background, and face different motivations and work constraints, than professional programmers. They are not likely to know about such things as quality control mechanisms, formal development processes, system models, language design characteristics, or test adequacy criteria, and are not likely to invest time learning about them.

It is important to find ways to help these users pursue their goals, while also alerting them to dependability problems, and assist them with their explorations into those problems. Further, it is important to work within the contexts with which these users are familiar, which can include programming environments that have not been directly considered by software engineering or programming languages researchers.

These challenges require collaborations by teams of researchers from various computer science subfields, including specialists in end-user-programming (EUP) and end-user development (EUD), researchers expert in software engineering methodologies and programming language design, human-computer interaction experts focusing on end-user programming, and empiricists who can evaluate emerging results and help understand fundamental issues in supporting end-user problem solving. Collaborations with industrial partners must also be established, to help ensure that the real needs of end-user programming environments in industry are met.

This Dagstuhl seminar was organized in order to bring together researchers from these various groups and with the various appropriate backgrounds, along with an appropriate selection of industrial participants. The seminar allowed the participants to work together on the challenges faced in helping end-user programmers create dependable software, and on the opportunities for research addressing these challenges. Our goals were to help these researchers better understand (1) the problems that exist for end-user programmers, (2) the environments, domains and languages in which those programmers create software, (3) the types of computing methodologies (especially in the areas of software engineering and programming language design) that can be brought to bear on these problems and in these domains, and (4) the issues that impact the success of research in this area. In addition, an overarching goal was to build awareness of the interdisciplinary connections and opportunities that exist for researchers working in the area.

The seminar included several tutorial-style presentations by experts on software engineering, programming languages, human-computer interaction, and empirical studies in relation to end-user software engineering. The program was complemented with brief presentations by some participants on topics of a more specialized nature, grouped into sessions on related topics. We also incorporated system demonstrations of prototypes and environments relevant to the topics. Ample time was allowed for interactive discussion sessions.

Most of the seminar participants provided white papers summarizing their primary interests in the area, including work that they are doing and open problems. These white papers are compiled into the seminar proceedings. Additional contributions to the seminar were provided as slides, and are available on the Dagstuhl website for the seminar.

A Methodology to Improve Dependability in Spreadsheets

Margaret Burnett
Oregon State University
burnett@eecs.oregonstate.edu

Marc Fisher II, Gregg Rothermel
University of Nebraska - Lincoln
{mfisher,grother}@cse.unl.edu

1 Introduction

Spreadsheets are one of the most commonly used end-user programming environments. As such, there has been significant effort on the part of researchers and practitioners to develop methodologies and tools to improve the dependability of spreadsheets.

Our work has focused on the development of the "What You See Is What You Test" (WYSIWYT) family of techniques. WYSIWYT is designed to be seamlessly integrated into a spreadsheet environment and the user's development processes. It uses visual devices that are integrated into the user's spreadsheet to guide the process of finding and fixing problems with the spreadsheet.

There are three major components to the WYSIWYT methodology: a testing and debugging methodology, an assertions mechanism, and the "Surprise-Explain-Reward" strategy.

2 Testing and Debugging Methodology

WYSIWYT provides a testing and debugging methodology [3]. As the user edits their spreadsheet they are provided with visual devices indicating the "testedness" (coverage relative to an underlying data-flow adequacy criterion) of the cells and the spreadsheet and the ability to mark the values in cells as correct or incorrect. If the user marks a cell's value as correct, the testedness of the contributing cells is updated as is the testedness of the spreadsheet. If, instead, the user marks a cell's value as incorrect, additional information is displayed to the user about the "fault likelihood" of cells based on the number of correct and incorrect values to which they contribute[4]. Figure 1 shows our visual devices in the Excel spreadsheet language.

In addition to tracking testedness and fault likelihood, WYSIWYT includes a "Help-Me-Test" feature that generates new test cases to cover unexercised portions of the spreadsheet [2] or replays test cases to re-validate changed portions of the spreadsheet.

3 Assertions Mechanism

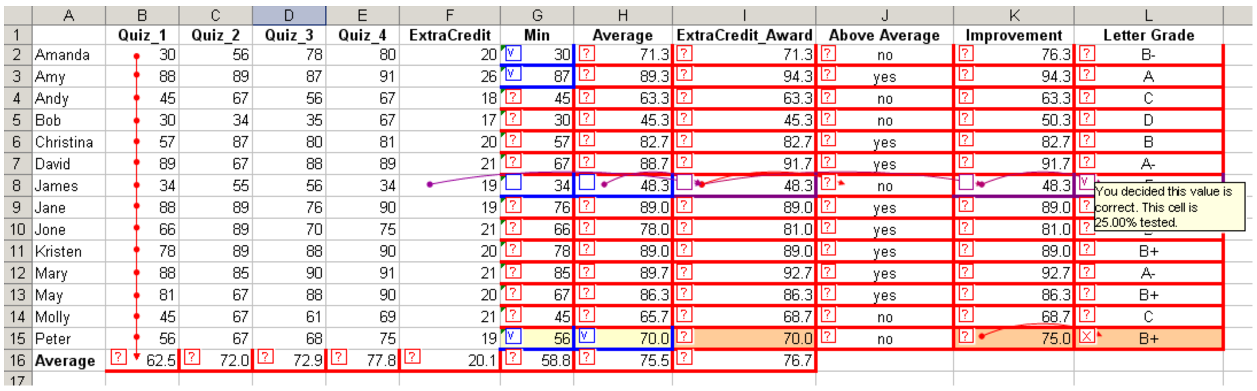
WYSIWYT also includes an assertions mechanism where users can supply a valid range of values for a cell [1]. These ranges are then propagated through dependent cells using interval arithmetic techniques, and conflicts between user-supplied ranges, propagated ranges and cell values are displayed to the user.

The assertions mechanism interacts with Help-Me-Test in two ways. User-supplied ranges on cells whose formulas are simple data values are used to limit the inputs used when generating new test cases. Help-Me-Test then attempts to generate test cases that violate the ranges on formula cells as they indicate that the user has an error either in a formula or a range.

4 Surprise-Explain-Reward

A key strategy in getting end users to effectively utilize the WYSIWYT methodology is Surprise-Explain-Reward [5]. Surprise-Explain-Reward relies on a user's curiosity about features in the environment. According to research about curiosity, if the user is surprised by something, such as the checkboxes in the spreadsheet, the surprise can arouse the user's curiosity, potentially causing the user to seek an explanation.

All features and feedback must therefore be able to explain themselves. These explanations must do two things: first, make the user aware of why the item is worthy of further attention (i.e., make the rewards clear), and second, help the user make an informed judgment as to whether the reward is worthwhile. Users explore a feature by viewing its explanation, on demand, via tool tips and other low-cost mechanisms.



	A	B	C	D	E	F	G	H	I	J	K	L
1		Quiz_1	Quiz_2	Quiz_3	Quiz_4	ExtraCredit	Min	Average	ExtraCredit_Award	Above Average	Improvement	Letter Grade
2	Amanda	30	56	78	80	20	30	71.3	71.3	no	76.3	B-
3	Amy	88	89	87	91	26	87	89.3	94.3	yes	94.3	A
4	Andy	45	67	56	67	18	45	63.3	63.3	no	63.3	C
5	Bob	30	34	35	67	17	30	45.3	45.3	no	50.3	D
6	Christina	57	87	80	81	20	57	82.7	82.7	yes	82.7	B
7	David	89	67	88	89	21	67	88.7	91.7	yes	91.7	A-
8	James	34	55	56	34	19	34	48.3	48.3	no	48.3	[illegible]
9	Jane	88	89	76	90	19	76	89.0	89.0	yes	89.0	[illegible]
10	Jone	66	89	70	75	21	66	78.0	81.0	yes	81.0	[illegible]
11	Kristen	78	89	88	90	20	78	89.0	89.0	yes	89.0	B+
12	Mary	88	85	90	91	21	85	89.7	92.7	yes	92.7	A-
13	May	81	67	88	90	20	67	86.3	86.3	yes	86.3	B+
14	Molly	45	67	61	69	21	45	65.7	68.7	no	68.7	C
15	Peter	56	67	68	75	19	56	70.0	70.0	no	75.0	B+
16	**Average**	62.5	72.0	72.9	77.8	20.1	58.8	75.5	76.7			
17												

You decided this value is correct. This cell is 25.00% tested.

Figure 1: WYSIWYT Devices in the Excel Spreadsheet Environment

The reward in the explanation informs the user when weighing costs, benefits, and risks in deciding how to complete a task. By providing users a projection of future benefits, they can better assess if the cost of using the feature is worth their time. If all goes well, if the user follows up as advised in the explanation, rewards will ensue, such as an increase in testing coverage or the discovery of an error.

5 Future Work

Our most recent research continues to investigate Surprise-Explain-Reward, focusing primarily on the explanations aspect. We are doing significant experimental prototyping and empirical work to understand what end-user programmers actually want to know when debugging, how explanations provided by the system can help them, and how they might be able to help each other. We are thus looking into strategies end-user debuggers follow, whether the system helps to support these strategies, and whether the explanations are helpful in improving their debugging strategies.

We are also interested in aspects of the end-user software engineering lifecycle beyond testing and debugging, and how to support them.

References

[1] M. Burnett, C. Cook, and G. Rothermel. End-user software engineering. *Communications of the ACM*, 47(9):53–58, September 2004.

[2] M. Fisher II, G. Rothermel, D. Brown, M. Cao, C. Cook, and M. Burnett. Integrating automated test generation into the wysiwyt spreadsheet testing methodology. *ACM Transactions on Software Engineering and Maintenance*, 15(2):150–194, April 2006.

[3] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Maintenance*, 27(1):110–147, January 2001.

[4] J. 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, February/April 2005.

[5] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. Harnessing curiosity to increase correctness in end-user programming. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 305–312, April 2003.

Barriers to Successful End-User Programming

Andrew Ko
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
ajko@cs.cmu.edu, http://www.cs.cmu.edu/~ajko

In my research and my personal life, I have come to know numerous people that our research community might call end-user programmers. Some of them are scientists, some are artists, others are educators and other types of professionals. One thing that all of these people have in common is that their goals are entirely unrelated to producing code. In some cases, programming may be a necessary part of accomplishing their goals, such as a physicist writing a simulation in C or an interaction designer creating an interactive prototype. In other cases, programming may simply be the more efficient alternative to manually solving a problem: one might find duplicate entries in an address book by visual search or by writing a short Perl script.

In either case, the fact that end-user programmers are motivated by their domain and not by the merits of producing high-quality, dependable code, means that most of the barriers that end users encounter in the process of writing a program are perceived as distractions. This is despite the fact that such barriers can represent fundamental problems in end-users' program's or their understanding of how to use a programming language effectively.

Much of my research has focused on understanding these barriers and how end users overcome them. When are they insurmountable and why? What happens when end users fail to overcome them? And how can tools help end-user programmers' improve their programs' dependability, while allowing them to remain focused on their goals, rather than their code?

Studies of Barriers

Some of my earlier investigations of these barriers involved observations of non-programmers using the Alice programming environment to create interactive 3D worlds (Ko and Myers 2005). Some of these observations were done in the field, in the context of teams of students, only one of which was programming, and other observations were performed in a lab, with an experimenter. There were several barriers that users encountered that seemed fundamental to programming and programming tools, and not just to Alice. For example, *premature commitment* was a major problem in numerous contexts: users were forced to make decisions before they had enough information to do so accurately. For example, they had to create an object before they could write code to manipulate it. Or, when a user was trying to diagnose their program's failure, they had to base their hypothesis of what caused the failure just on what they could see in the program's output, rather than on information about the program's execution. In many of these situations, users premature decisions led to errors.

These observations led to broader study, aimed at classifying major barriers (Ko, Myers and Aung 2004). I observed over thirty students learning to use Visual Basic.NET to create simple form-based applications and user interfaces. I attempted to document the barriers that students encountered by telling them that they could consult the teaching assistants with any problems they felt they could not overcome. When consulted, the teaching assistants recorded the problem that the student was stuck on and the strategies that the student had used to try to overcome it. After classifying all of the different barriers that students encountered, there were six major barriers that accounted for our data:

Design – Complex computational problems that users were not trained to solve, such as sorting and searching.

Selection – Finding code, usually part of an API, that produces a desired behavior, such as tracking time.

Use – Once some class, method, or data structure was found, learning how to properly use its programming interface, such as how to start and stop a timer.

Coordination – Learning rules about how entities can communicate, such as how to send data between forms.

Understanding – Forming hypotheses about the potential causes of a program's behavior.

Information – Gathering information to test hypotheses about the causes of a program's behavior.

These six barriers accounted for all of the situations we observed in our study, and we have continued to observe them in other languages and tools.

The Whyline

In addition to studying the barriers that end user programmers face, I have also attempted to lower them with tools. The Whyline (Ko and Myers 2005) is aimed at alleviating difficulties with the *understanding* and *information* barriers described above, specifically for the Alice programming environment. Essentially, it allows users to choose some aspect of the program's behavior, such as a change to the color of some object onscreen, and ask *why* and *why not* questions about it. The Whyline then gives answers in terms of a causal chain of events that caused or prevented the behavior to occur. In a user study it was highly effective, reducing debugging time by a factor of 8. The reasons for this improvement were simple. By allowing users to reason about the output of their program, it deferred the premature formation of hypotheses about the causes of the behavior until the Whyline provided information, helping lower the *understanding* barrier. By providing the information about the program's execution automatically, rather than having users gather it manually, it almost entirely eliminated the *information* barriers that we observed in our earlier study of Alice.

Future Directions

My studies of barriers in end-user programming revealed many important problems to address, and the Whyline demonstrates one example of addressing them. However, not only are there many other barriers that deserve attention, but the tools that we design to help with each of these are influenced by a number of factors for which we still have little knowledge.

For example, the generalizability of any end-user software engineering tool depends greatly on the similarity of the work contexts of the end users we intend to design for. The Whyline was designed for a single user; in a group context, where many people may be involved in diagnosing and fixing a bug, the tool suddenly has many shortcomings. Do end user programmers work in groups? If they do, how is the work divided? What information do they share?

Another issue that may vary across different work contexts is the set of languages and applications with which end users' programs must interact. We might be able to design tools for one language, but can we design general tools to support Excel scripters interacting with a proprietary internal company database? To what extent do such setups actually occur for end users?

Although end-user programming language design has received much attention in the past, there are still several important issues to understand. For example, to what extent must a language match the work that end-users do? How can we help end users bridge the expressive gap in the languages they use and the behaviors they want to express? Because end users often lack the training to create the abstractions necessary to bridge these gaps, this will continue to be an issue.

Another software engineering issue that end users may encounter are the long-term maintenance issues common to commercial software development. We frequently hear anecdotes about how a one-off excel spreadsheet meant to be temporary became the centerpiece of some accounting logic. How often do such organizational dependencies occur, and how important do such program's become? What can tools do to help the future owners of these programs learn about the program's history and design?

Finally, one challenge about end-user programming is that end-user programmers needs may vary so widely that we cannot design tools and languages general enough, yet specific in their aid to help everyone. Do we approach this problem by simplifying the creation of end-user programming environments and creating highly tailored languages on-demand, by helping end-users bridge expressive gaps in a smaller number of languages, or by some other means? What general research contributions can we make and what specifics do we have to leave to individuals and the market?

That we face so many complex issues is encouraging. Not only does this mean that we have lots of interesting work to do, but it also means that we are closer to addressing real concerns. Let us continue to tackle them with rigor and objectivity.

Acknowledgements

This work was funded in part by the National Science Foundation (NSF) under grant IIS-0329090, the EUSES consortium under NSF grant ITR CCR-0324770, and an NDSEG fellowship.

References

Ko, A. J. and Myers, B. A. (2005). A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems. *Journal of Visual Languages and Computing*, 16, 1-2, 41-84.

Ko, A. J. Myers, B. A., and Aung, H. (2004). Six Learning Barriers in End-User Programming Systems. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, September 26-29, 199-206.

Ko, A. J. and Myers, B. A. (2004). Designing the Whyline: A Debugging Interface for Asking Questions About Program Failures. *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, April 24-29, 151-158.

Dependability in Web Software

Sebastian Elbaum, Marc Fisher II, Gregg Rothermel
University of Nebraska - Lincoln
{elbaum,mfisher,grother}@cse.unl.edu

1 Introduction

The web is an increasingly important platform used for a wide variety of tasks on a regular basis. And as the web becomes more important, the ways in which it is used grows increasingly sophisticated. End users build web pages and applications, use web applications in new and unexpected ways and use web macro tools to automate web-based tasks. All of these tasks are error-prone. In addition, they often depend on external components outside of the control of the developer or end user. Therefore we have been developing tools and methodologies to assist users with these tasks.

One of our methodologies uses dynamic characterization of the web application interface to assist the application builder in finding anomalous behavior in their applications and to help users understand how they can access the application's features.

In other work, we have attempted to improve the maintainability and robustness of web macros. To do this we have developed a family of assertions that work with web macros to detect certain types of erroneous or changed behavior in the uses of web applications and indicate to the user when these assertions are violated so they can update the web macro accordingly.

2 Dynamic Characterization of Web Application Interfaces

Our early work in dynamic characterization of web applications was motivated by the use of existing web applications in mashups, web applications that combine data from multiple sources for some particular purpose [2]. This work included several static analysis methods to characterize certain properties of the interfaces to web applications, as well as one dynamic method for identifying mandatory and optional variables in the interface. The dynamic method operated by constructing and submitting multiple requests with different combinations of variables present in these requests, and characterized the result as either successful or unsuccessful. It was then able to use this collected information to identify which variables were required in a successful request.

Further work extended this basic method to detect a wider array of properties in the interfaces to a class of web applications we call specialized search engines [1, 3]. The new properties include dependencies between variables, ranges of allowable values for variables, and relationships between results for different values. We have applied our methodology to a variety of search applications and found anomalous behavior in the majority of these applications.

Dagstuhl Seminar Proceedings 07081
End-User Software Engineering
http://drops.dagstuhl.de/opus/volltexte/2007/1089

3 Assertions in Web Macros

Robofox is a Firefox plug-in that allows users to program web macros by demonstration [4]. Since web sites evolve over time (e.g., style, layout, flow) and these changes can cause faulty behavior in executing macros or prevent the macros from executing at all, we have developed a family of assertions within Robofox to detect and report anomalous run-time behavior to users.

As the user records a web macro, assertions are automatically created for each operation performed. These assertions can then be viewed and edited by the user. When the macro is run, these assertions are checked, and if any are violated, violations are reported to the user so that appropriate changes to the web macro can be made.

4 Future Work

We are continuing to work on both the dynamic characterization and web macro assertion methodologies. For dynamic characterization, we are extending our methodology to cover other classes of web applications beyond search. More specifically, we are currently looking at shopping carts and product configuration applications. Within the web macro dependability area, we will be evaluating the usability and effectiveness of the approach, and we are considering and investigating other dependability devices to assist in controlling the impact of a faulty macro.

References

[1] S. Elbaum, K.-R. Chilakamarri, M. Fisher II, and G. Rothermel. Web application characterization through directed requests. In *Proceedings of the* 4^{th} *International Workshop on Dynamic Analysis*, pages 46–56, May 2006.

[2] S. Elbaum, K.-R. Chilakamarri, B. Gopal, and G. Rothermel. Helping end-users "engineer" dependable web applications. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 31–40, November 2005.

[3] M. Fisher II, S. Elbaum, and G. Rothermel. Dynamic characterization of web application interfaces. In *Fundamental Approaches to Software Engineering*, March 2007 (to appear).

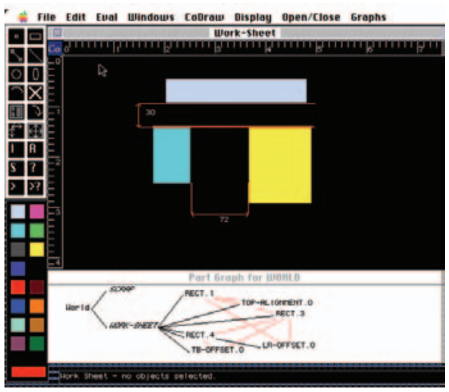
[4] A. Koesnander and S. Elbaum. Robofox: Web activities automation, integration, and customization. `http://esquared.unl.edu/wikka.php?wakka=AboutRobofox`, November 2006.

Designers Need End User Software Engineering

Mark D Gross, School of Architecture, Carnegie Mellon University

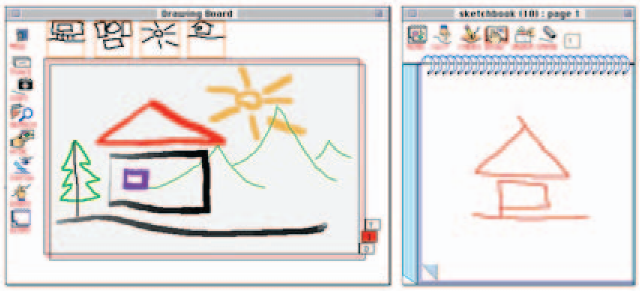
For many years I have been interested in how design works, and in how to support designers doing design. Inspired by tools like Macsyma (and later Mathematica, etc.), I became interested in building end-user languages and tools that are in a sense general-purpose systems and yet that end users can tailor or customize to correspond to their own needs. Whether there are specific structures and operations that are unique to design (across the disciplines, but distinct perhaps from other kinds of problem solving) I do not yet know, but it seems at least plausible that programming languages for design may have a special character. That question that has framed much of my work.

My earlier work was on design as exploring constraints, and computational support for end user designers to set up systems of constraints to describe, and then solve, problems couched in this framework. Inspired by Sketchpad, CoDraw [1] was a end-user extensible constraint based graphical editor (2D CAD system) in which every object was described as a collection of relationships among its parameters. Although CoDraw provided a few primitive objects to seed the system, end users could define new objects either by subclassing previously defined ones, or by identifying variables and their relationships. A CoDraw end user could extend the system on the fly, using a set of cards (each similar to a small spreadsheet) to define objects in terms of their variables, and relationships. Various graph display and editing features allowed end users to specify part-whole and sub/class relationships among objects in the system, to trace dependencies among constraints and inspect justification paths for derived values. As end users added new objects and relationships to the system they could also add items to tool palettes. In this system, end users could extend and build up the underlying language (a Lisp embedded constraint management system) as well as the graphical user interface that was used to build the CoDraw application.



Work Sheet - no objects selected.

Through experience with the CoDraw constraint based CAD system and its language, I learned that the designers I worked with were unsatisfied with describing their knowledge in terms of objects, variables, and constraints. They found even the menu and tool palette way of making drawings stilted and difficult to use. They wanted to draw. This led me to begin work on the Electronic Cocktail Napkin, a pen based freehand drawing system that is designed as an interface for knowledge-based systems [2]. The Napkin uses a symbol (glyph) recognizer to identify the freehand strokes that the designer draws. An end user trains the glyph recognizer on the fly, adding new symbols to the program's repertoire or showing the program new ways to draw previously defined symbols. The same is also true for more complex drawing configurations: an end user builds up a grammar (composed of previously defined glyphs and configurations) that corresponds to a specific diagrams in a specific domain. The end user builds the grammar by demonstrating examples of configurations; the Napkin program constructs a description of the relationships that



the user's configuration contains and the user then adjusts this description by making it more or less specific. These visual grammars may correspond to a specific knowledge domain (e.g., analog or digital electronics) or to a highly idiosyncratic way of using diagrams that is specific to the end user (as, for example, graphic or architectural designers may be wont to do).

I have recently become interested in what kinds of tools and environments might be useful for users who lack technical knowledge in programming or electronics to build working prototypes of embedded (tangible, pervasive, ubiquitous) computing systems. To build even a prototype of an embedded system may require expertise in programming, electronics (sensors and actuators), as well as mechanical, physical, and materials design. Each of these domains can alone be daunting and there are few designers who would be capable of managing the ensemble together. End user programming environments for microcontrollers include traditional coding (e.g. in C or Java) or visual languages (e.g., Scratch or Max/MSP). The electronics design similarly can require sophisticated knowledge of components, their interactions; and finally the physical, mechanical design demands that the end user master a 3D modeler, perhaps a kinematics simulator, and so on. We would like to build a "design fusion" environment where a novice programmer (a designer of embedded computing artifacts) can describe the set of desired behaviors and functions, and construct and debug the software, electronics, mechanical, and physical systems to implement these behaviors and functions. This design fusion environment should be powerful (not limiting the designer to a trivial subset of possibilities) yet simple so as to enable the designer to express the desired behaviors and functions without attending to irrelevant low-level language details.

One point in this space is roBlocks [3], a 'computationally enhanced construction kit' that is intended for young people to build simple robots out of blocks, without first demanding that they learn electronics, programming, and the associated manual skills. RoBlocks consists of small (40mm) cubes that snap together magnetically. Each block contains a microcontroller and provides either a sensor, an actuator, or some arithmetic or logic. The blocks snap together to make a robot, transmitting power and data from face to face. In this way a novice user can assemble (in one move) both the physical construction as well as the mechanics and programming necessary to make the robot behave. For example, snapping a photosensor block on a motor block would make a phototropic robot (that moves toward light). Adding an inverter block between the two would make a photophobic robot. We have built a small working set of roBlocks and are currently considering how to build a next-level screen-based language for users who have mastered the physical programming level and would like to change the behaviors of specific blocks.

1. Gross, M.D. Graphical Constraints in CoDraw. in Tanimoto, S. ed. IEEE Workshop on Visual Languages, IEEE Press, Seattle, 1992, 81-87.

2. Gross, M.D. and Do, E.Y.-L. Drawing on the Back of an Envelope: a framework for interacting with application programs by freehand drawing. Computers and Graphics, 24 (6). 835-849.

3. Schweikardt, E. and Gross, M.D., roBlocks: A Robotic Construction Kit for Mathematics and Science Education. in International Conference on Multimodal Interaction, (Banff, Alberta, 2006), ACM.

Empirical Studies in End-User Software Engineering and Viewing Scientific Programmers as End-Users -- POSITION STATEMENT --

Jeffrey Carver
Mississippi State University
carver@cse.msstate.edu

Abstract

My work has two relationships with End User Software Engineering. First, as an Empirical Software Engineer, I am interested in meeting with people who do research into techniques for improving end-user software engineering. All of these techniques need to have some type of empirical validation. In many cases this validation is performed by the researcher, but in other cases it is not. Regardless, an independent validation of a new approach is vital. Second, an area where I have done a fair amount of work is in software engineering for scientific software (typically written for a parallel supercomputer). These programmers are typically scientists who have little or no training in formal software engineering. Yet, to accomplish their work, they often write very complex simulation and computation software. I believe these programmers are a unique class of End-Users that must be addressed

1. Introduction

In this position paper, I will address work in two main areas related to End-User Software Engineering. The first area, discussed in Section 2, is related to the need for and use of empirical studies in End-User Software Engineering. This section provides the motivation for performing empirical studies, an overview of the types of studies that can be useful, and an example from my own experience.

The second area, discussed in Section 3, is related to a class of users who are not always considered in the discussion of End-User Software Engineering, the scientists and engineers. I argue that these users are not professional programmers, but rather they are a special class of End-Users that deserve unique attention and research.

2. Empirical Studies

The use of empirical studies is necessary in End-User Software Engineering for the same reasons that it is necessary in more traditional software engineering. An empirical study provides a researcher with the hard data necessary to make informed decisions, rather they relying only on hype or argumentation. Different types of empirical studies provide different types of evidence. Choosing the appropriate study and the appropriate evidence is important based on the goal of the research inquiry.

There are two main types of empirical studies that can be of use in this domain. Studies that are more exploratory and studies that are more confirmatory. In an exploratory study, the goal of the researcher is to understand the environment. This understanding could provide insight into identification of requirements for a new tool or interface or identification of necessary improvements in an existing interface. By gathering information about how the target users perform the task, the researcher can better understand the type of interface or tool that will best serve them. In addition, by observing users who are working with an existing interface or tool, researchers can understand how that tool or interface can be improved.

Software Engineering researchers have been doing these types of studies for a long time. Our studies focus on professional developers rather than end-users. And, our goals are typically to better understand or improve particular aspects of the software engineering process. But, the approaches used in study design and data analysis are similar to what is needed in the end-user domain [3, 6].

One important aspect of empirical studies that I believe I can offer to members of the EUSE community is independence and objectivity. One benefit of being independent, that is, not developing the end-user technologies myself, is that I have no vested interest in the outcome. One danger of a researcher performing empirical validation on his or her own tools or interfaces is that positive results are viewed with some skepticism. Studies conducted by an objective third party will lend additional validity to the results.

My experience in this domain comes from performing a series of experiments on the WYSIWYT prototype in Excel [7, 8]. In our study, we were interested in evaluating the use of WYSIWYT within the Excel environment to understand how the results from the Forms/3 environment translated. The goal of the study was to determine whether people would create a more correct, more tested spreadsheet when using WYSIWYT than they would when using the normal facilities provided by Excel [1].

In this study, the subjects were given the task of creating a spreadsheet based on a provided specification. They were instructed that their goal was to make the spreadsheet as correct as possible. The subjects were students in the Business Technology department at Mississippi State University who were taking a course on Spreadsheets. Therefore, they were representative of novice spreadsheet users, which is an interesting population for this study. The results of the study indicated that, while the WYSIWYT add-in did not improve overall correctness, it did decrease the amount of time required to reach the same level of correctness.

3. Scientists and Engineers as End-Users

High performance computing systems are used to develop software in a wide variety of domains including nuclear physics, crash simulation, satellite data processing, fluid dynamics, climate modeling, bioinformatics, and financial modeling. The TOP500 website (http://www.top500.org) lists the top 500 high performance computing systems. The diversity of government, scientific, and commercial organizations present on this list illustrates the growing prevalence and impact of HPC applications on modern society. These software systems are largely developed by experts in the scientific or engineering domain that is being modeled. Therefore, they have little or no training in formal software engineering.

This class of developers should be considered as a special type of end-users for the following reasons. First, they lack training in formal software engineering and often lack the interest in following correct software engineering principles. Second, for these developers, the production of software is a secondary goal. Their main interest is the science or engineering. To accomplish their goal, they must often write simulation code or computation code. While this code may often be shared and used by others, it is not the end goal of their work [2, 4, 5].

4. References

[1] Carver, J., Fisher II, M., and Rothermel, G. "An Empirical Evaluation of a Testing and Debugging Methodology for Excel". In *Proceedings of 2006 International Symposium on Empirical Software Engineering*. Rio de Janeiro. Sept. 21-22, 2006, 2006. p. 278-287

[2] Carver, J., Hochstein, L., Kendall, R.P., Nakamura, T., Zelkowitz, M.V., Basili, V.R., and Post, D., "Observations about Software Development for High End Computing." *CTWatch*, 2006. **November**: 33-37.

[3] Carver, J., Shull, F., and Basili, V.R., "Can Observational Techniques Help Novices Overcome the Software Inspection Learning Curve? An Empirical Investigation." *Empirical Software Engineering: An International Journal*, 2006. **11**(4): 523-539.

[4] Carver, J., Kendall, R.P., Squires, S., and Post, D. "Software Development Environments for Scientific and Engineering Software: A Series of Case Studies". In *Proceedings of 2007 International Conference on Software Engineering*. Minneapolis. 2007. p.

[5] Hochstein, L., Nakamura, T., Basili, V.R., Asgari, S., Zelkowitz, M.V., Hollingsworth, J.K., Shull, F., Carver, J., Voelp, M., Zazworka, N., and Johnson, P., "Experiments to Understand HPC Time to Development." *CTWatch*, 2006. **November**: 24-32.

[6] Maldonado, J., Carver, J., Shull, F., Fabbri, S., Doria, E., Martimiano, L., Mendonca, M., and Basili, V., "Perspective-Based Reading: A Replicated Experiment Focused on Individual Reviewer Effectiveness." *Empirical Software Engineering*, 2006. **11**(1): 119-142.

[7] Rothermel, G., Li, L., and Burnett, M. "Testing Strategies for Form-Based Visual Programs". In *Proceedings of 8th International Symposium on Software Reliability Engineering*. Albuquerque, NM USA: IEEE-CS. Nov., 1997. p. 96-107

[8] Rothermel, G., Burnett, M.M., Li, L., DuPuis, C., and Sheretov, A., "A Methodology for Testing Spreadsheets." *ACM Transactions on Software Engineering and Methodology*, 2001. **10**(1): 110-147.

End User Programming for Scientists: Modeling Complex Systems

Andrew Begel
Microsoft Research
andrew.begel@microsoft.com

Towards the end of the 20th century, a paradigm shift took place in many scientific labs. Scientists embarked on a new form of scientific inquiry seeking to understand the behavior of complex adaptive systems that increasingly defied traditional reductive analysis. By combining experimental methodology with computer-based simulation tools, scientists gain greater understanding of the behavior of systems such as forest ecologies, global economies, climate modeling, and beach erosion. This improved understanding is already being used to influence policy in critical areas that will affect our nation's future, and the world's.

Some computer tools enabled scientists to create models of phenomena from first principles, rather than from descriptive differential equations. These tools, which directly modeled complex adaptive systems, significantly lowered the mathematical burden required of scientists to understand and create models. Tools such as StarLogo (Klopfer & Begel, 2003), Swarm (Minar, Burkhart, Langton, & Askenazi, 1996), and Repast (North, Collier, & Vos, 2006) enable scientists to program a simulation of a system by describing the behaviors of the individual elements of the system (e.g. each animal eating another, each consumer purchasing a product, each molecule of air and particle of cloud, and each grain of sand and drop of water). These tools reduce the barrier to entry by providing a framework in which to develop models, but they require a degree of programming sophistication to accomplish even relatively simple tasks. Swarm and Repast require the scientist to program in Objective-C and Java, respectively. StarLogo reduces the barrier more than the others through its use of Logo, a more accessible language most often associated with children's programming projects. A more recent version of StarLogo, called TNG (Klopfer & Begel, In Press), improves accessibility to non-programmers further by using a graphical programming language.

Modeling follows the scientific method: hypothesis, experiment creation, observation, evaluation, only instead of studying a system in the real world, a model is created and studied instead. Rather than giving scientists black-box models in which they can only study what they have been given, and only tweak knobs that the author provided, the StarLogo programming environment enables scientists to be model designers and builders, by enabling them to program the behaviors of the entities they want to interact with using the Logo programming language. Programming is a means to an end, yet in order to enable scientists to model what they want to study, it is often the only means.

We have used StarLogo to teach the scientific method and modeling to high school students. Through a series of workshops, called Adventures in Modeling (Colella, Klopfer, & Resnick, 2001), high school students and teachers (and school district technology coordinators) have learned what complex systems are, how to program in StarLogo, how to model a complex system using StarLogo, and how to conduct scientific inquiries using the StarLogo modeling environment. Participants work through a series of participatory activities, games that involve the participant as one of the entities in a complex system. For example, in the majority-minority game, participants must discover what the majority of the group has decided, secretly, about which color chile they like the best, green or red. They can only move around while blindfolded, and whisper anything they like to whomever can hear them. At the end of each round, a vote is taken to determine which chile is the best; and each participant must vote with the choice they think the majority has chosen. The vote tallies initially begin quite divergent, but as the rounds progress, a kind of positive feedback loop forms, with the majority winner being whispered more often, and winning over more votes. Eventually, the majority dominates. The minority game is similar, but participants must pick the chile that the minority of people like. Vote tallies in this game often fluctuate from one extreme to the other; as participants hear more people saying one color chile, they pick the opposite, leading to an unstable dynamic that never converges. After playing the game, participants learn to program it in StarLogo. They develop variants, and run experiments to understand the behavior under different conditions, for example, greater or fewer people, no blindfolds, communication louder than a whisper, or communication only by touch.

The StarLogo workshops were successful at teaching non-experts to program, and we have heard many reports from scientists in many fields of study who have used StarLogo to model systems they were researchers. However, we have found that StarLogo programming can be difficult to pick up, especially when learned on a hobby basis, or without an instructor. Even worse, the longer a novice scientist goes between StarLogo programming sessions, the less they retain, and the more apprehensive they get about creating their own models. It is critical, however, for scientists to be able to design, build and conduct experiments in models that they build themselves. Model creation cannot be turned over to a programmer-for-hire without causing the model to become a black box. In order to ensure the validity of the model and stand behind its experimental results, the scientist must be intimately knowledgeable about its innards as well as its outward behaviors.

Thus, it is important to understand how non-programmers pick up programming languages when the task they want to complete cannot be accomplished in other ways. How does motivation drive learning in the absence of teachers, or a community of learners, which is the usual model of learning to program in school? Unless the fidelity of the finished model is quite high, even demonstrating the model to non-modeler audiences can prove difficult. How are search engines used to provide sample code, explanations, and project ideas, especially when the software modelers use is not widespread, or is new?

Learning a text-based programming language is difficult for novices who want to be programmers. In the first few weeks of learning a language, syntax rules are often the most difficult to comprehend, with semantics interleaved. Non-programmers face these problems, in addition to lacking an engineering mindset to help form mental models of how they want to make the computer do what they want. How does learning graphical programming languages like LabView, ProGraph or StarLogo TNG differ from learning text-based languages in this context? Is the floor lower? Is the ceiling lower? Are the walls more narrow? Graphical languages have not achieved popularity among computer scientists, but remain fashionable in educational settings. Is this making a difference? Does exposure to programming prior to college enable non-programmer scientists to understand and create models more easily?

How does one characterize an expert in a modeling language? When we, as computer scientists, see non-programmers' StarLogo programs, we might cringe at their inelegance. Yet, if the non-programmer is achieving their modeling goals, then their program is effective and just as valid as an elegant one. Is it important to turn expert non-programmers into proper engineers? Can experts teach other non-programmer novices properly? Does an expert's lack of formal instruction hinder their instruction or interfere with novice learning? Is it better or worse than no instructor at all? What can be done to ensure that a model's validity is not affected by poor programming? Can automated tools help a non-computer-scientist see coding flaws and help him to fix them?

Understanding how non-programmer scientists attain and disseminate expertise in programming will help us to design easier to use modeling environments that result in more understandable and maintainable programs. Our goal is to enable all scientists, even the ones who are apprehensive about computer programming, to create and study their own models of complex systems and use them in their research.

References

Colella, V., Klopfer, E., & Resnick, M. (2001). *Adventures in Modeling: Exploring Complex, Dynamic Systems with StarLogo.* Teachers College Press.

Klopfer, E., & Begel, A. (2003). StarLogo in the Classroom and Under the Hood. *Kybernetes , 32* (1/2), 15-37.

Klopfer, E., & Begel, A. (In Press). StarLogo TNG: An Introduction to Game Development. *Journal of E-Learning* .

Minar, N., Burkhart, R., Langton, C., & Askenazi, M. (1996). *The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulations.* Santa Fe: Santa Fe Institute.

North, M. J., Collier, N. T., & Vos, J. R. (2006). Experiences Creating Three Implementation of the Repast Agent Modeling Toolkit. *ACM Transactions on Modeling and Computer Simulation , 16* (1), 1-25.

End Users Creating More Effective Software

Brad A. Myers
Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
bam@cs.cmu.edu
http://www.cs.cmu.edu/~bam

Presented at:
Dagstuhl Seminar on End-User Software Engineering
http://www.dagstuhl.de/07081/
February 19, 2007

Abstract:

End-User Software is created using a variety of different techniques and paradigms. The "creating" part is defined as the process of representing the desired program in a computer-understandable form, and entering that representation into the computer. Programs can be represented using textual languages, visual (also called graphical) languages, spreadsheets (which are often included as a type of visual language), programming by example, or simple menu-based specifications. Textual languages range from general purpose languages such as Java, to special purpose languages such as StarLogo and MatLab for simulations and the HANDS language for kids, to "programming" using English as in Metafor. There is also research on how tools can make it easier to enter the text, such as the Alice syntax-directed editor. Visual Programming languages use 2D as part of the meaning of the language, and systems from members of the seminar include AgentSheets, Pursuit, Stagecast, StarLogo, and others. Programming by Example systems watch as the users perform tasks, and use AI (in particular, Machine Learning) to infer the program from the examples. Systems from members of the seminar include Eager, Robofox, Mondrian, and Gamut. Finally, there are some systems that allow limited customization of behaviors using menus and dialog boxes, such as simple parameter tweaking, specifying the behaviors of the characters in the Sims game, and drag-and-drop creation of dynamic web pages in CLICK. An orthogonal issue for software creation is the programming paradigm, such as imperative, event-based such as in Visual Basic and HANDS, functional such as in spreadsheets, and constraint programming as in CoDraw. One goal for EUP is to create what we call "gentle slope systems", where it is easy to get started ("low threshold"), sophisticated things can be done ("high ceilings"), and there are no barriers or walls where users must stop and learn many new things before making progress.

End-user (further) development: A case for negotiated semiotic engineering

Clarisse Sieckenius de Souza
Departamento de Informática, PUC-Rio
clarisse@inf.puc-rio.br

Semiotic engineering

Semiotics is a discipline devoted to studying signs and signification, which includes processes of representation, interpretation, sense making, and – for a number of semioticians – communication[1]. Its object of investigation is thus strongly connected with that of various sub-areas of Computer Science such as: Artificial Intelligence, Human-Computer Interaction, and even Theoretical Computer Science. In HCI, specifically, the most popular, although often superficial and restricted, use of Semiotics has been the famous classification of signs into *icons*, *indices* and *symbols* proposed by Peirce[2]. However, just as Cognitive Psychology has the power to provide the foundations of full-fledged theories of HCI, so does Semiotics. Semiotic Engineering[3] is the first proposed theoretical account of HCI in general based on semiotic theories (mainly Eco's and Peirce's).

The gist of Semiotic Engineering involves the following main concepts: metacommunication, semiosis, signification and communication. All of them are familiar to semioticians, and have not been originally proposed by Semiotic Engineering. What is new, however, is how they can be put together to characterize and explain HCI, to generate HCI research questions and methods, among which some related to end-user development. In fact, EUD holds an important position in this theory for the reasons briefly presented in the following paragraphs.

Metacommunication is classically *communication about/of communication*. Semiotic Engineering views HCI as a specific type of metacommunication, a process where systems' developers communicate to systems' users how they (users) can/should communicate with the system in order to achieve a particular range of intended effects. So, for example, if you are using a text editor to write a position paper, you are in fact getting (and reacting to) the developers' message about all the things you can do with their software in order to create great-looking documents. Of course this developer-to-user message is received progressively by users, as they interact with the communicative agent that represents the developers at interaction time: the system itself, or the designer's *deputy* as we say in Semiotic Engineering terms. Some important shifts of perspective follow from this. First, developers participate in interaction (the system speaks for *them*), which represents a radical change compared to the classical user-centered model of HCI[4], for instance. The change does not take users out of the scene. It includes designers/developers in it, and by so doing expands the topic of interactive exchanges from tasks to design intent, rationale and value. Second shift, problem-solving and cognition do not constitute the focus of investigation in this theory. Communication is the new focus. Thus, problem-solving and cognition are only covered by the theory inasmuch as they constitute the object or purpose of communication. Third shift, developers and users belong to the same ontological category – they are interlocutors in computer-mediated communication. To our knowledge, this is the only theory of HCI (and maybe one of the few, if not also the only, theoretical account that can be used to characterize Software Engineering, in a broad

Dagstuhl Seminar Proceedings 07081
End-User Software Engineering
http://drops.dagstuhl.de/opus/volltexte/2007/1083

sense) where software producers and software consumers, and their respective purposes and activities, can be described in terms of the same ontology.

Semiosis is process through which we generate (interpretive) signs in the presence of something that we take to stand for something else. For example, if you see this on a text page, you are likely to take it to mean *a hyperlink*. So, in your process of interpretation you generate other signs (*i.e.* things that, themselves, stand for various other things, to you). Among the signs you generate in your interpretation it is very probable that you will have a sign representing the expectation that when you put the mouse on the underlined blue text you see this. This sign will be part of your interpretation of this unless your expectation fails. All the signs generated in this interpretive process are part of what the original (base) sign means to you. Very importantly, they are subject to further revision, as the example shows. When expectations or inferences are contradicted by current factual evidence, you *change* your previous interpretation by generating other signs that accommodate the new information you just acquired. This "generate-revise" interpretive process, described and defined by Peirce as *abduction*[5], is continuous, and so we say that semiosis is *continuous*, or *unlimited* over time. Consequently, a semiotic theory of meaning, of Peircean breed, does not view meaning as a *static* entity associated to representations, but as an ongoing process that includes unpredicted and unpredictable signs. The fundamental role of semiosis in Semiotic Engineering is to characterize more precisely the developers-users interlocution at interaction time. Although both developers and users share the same interpretive capacities (that are species-specific for Peirce), computer mediation introduces a radical reduction in the developers' abilities to communicate productively with users during interaction. The system, unlike human beings, is not capable to carry on *unlimited semiosis*. Quite contrarily, it is in the nature of computer representations that, for all practical purposes, they need *grounding*. An examination of the semantics of computer programs can show the various pre-established meanings that developers have associated to the interactive signs that users will be exposed to and will be able to use in order to get the computer to exhibit various types of behavior. So, very briefly, although developers and users are communicating to each other (through the system), and thus are both interlocutors in the same conversation, computer mediation imposes an important limitation for both parties. Developers must realize that they won't have the usual unlimited human capacity to explain and revise what they mean by the kinds of interaction they invite users to have with the system they have designed. And users must realize that what they mean to communicate to the system will only be understood (and effective) if it is consistent with a pre-established range of meanings that have been encoded in it. Users can always explain and will constantly revise (and expand) *their* meanings, of course. And this is the fundamental link between Semiotic Engineering and End User Development.

Finally Semiotic Engineering uses two definitions from Eco's Semiotics[6]: signification and communication. **Signification** is the process by which certain *contents* are systematically assigned to certain *expressions* as a result of deep and strong cultural conventions. **Communication** is the process by which interlocutors *explore* the signification systems within their reach in order to produce signs meant to achieve an unlimited range of purposes and effects. They can not only pick up culturally established signs in the process, but they can also (and *extensively do so*) invent new expressions and/or use the signification system in innovative ways. The beauty of human communication is that just as sign producers are prepared (and actually inclined) to express themselves innovatively, sign consumers are equally well-equipped to interpret creative expressions, exactly because they are naturally born with the ability to think abductively. So, human communication is a negotiation of meanings, where abduction plays a central role, allowing interlocutors to revise constantly their assumptions and expectations about each other's understanding and intent. For the purpose of this discussion about EUD, Semiotic Engineering draws two important consequences from these notions. One is that it is *natural* for users to use any computer-encoded signification system in innovative ways. The other is that *usable* technologies must necessarily support revisions of the computer-encoded signification systems they support[7].

Designing at interaction time

Semiotic Engineering provides theoretical arguments to support what Liberman and co-authors express in the opening chapter of *End user development*: "We think that over the next few years, the goal of human-computer interaction (HCI) will evolve from just making system *easy to use* (even though that goal has not yet been completely achieved) to making systems that are *easy to develop*"[8]. Natural human communication is extensively innovative compared to the expressions that can be systematically derived from any given signification system. Innovation can focus on the expression side of the system (*e.g.* new expressions or expressive modes can be used to convey well known content), on the content side (*e.g.* a well known expression can be used to refer to a modified version of its previously known corresponding content), or on both (*e.g.* new expressions can be instantly produced to signify new content, or new expression/content correspondences can be created to achieve particular effects in communication). Using innovative forms of communication always requires additional interpretive efforts from interlocutors, who will typically engage in adbductive reasoning processes to interpret what they are being told. However, the efficacy and efficiency of communication can be very positively affected by precisely such innovations. At one end of the spectrum, it is clear that without them no evolution (of culture, society, science, and even personal lives) would be possible. At the other, it is also clear that communication constrained by perfectly ordinary situational factors, such as lack of time or space, wouldn't be possible otherwise. For instance, if this example worked for you, it's because you undertand innovative communication. If not, I need more time and space to explain it to you. But once you get the idea, you will be able to use this yourself to communicate things you mean.

To communicate is thus to *design* forms of expression that will effectively and efficiently cause your intentions to be fulfilled. Some of the EUD-related challenges for HCI within a Semiotic Engineering perspective are to: (i) let users communicate more naturally (hence, more effectively and efficiently) with systems; (ii) let developers communicate more effectively and efficiently to users *the limitations imposed by computer mediation* to their mutual understanding; (iii) help developers design various ways computer-encoded signification system manipulations that users can choose to explore interface languages in order to communicate innovation; and (iv) develop theoretical concepts and models to explain, characterize and expand the connections between HCI and EUD. Because the success of HCI for Semiotic Engineering is measured by the developers' ability to get their metacommunication message across to users[9], it is important that the gist of this message be preserved at least as a reference for further developments. Revisions of meanings encoded in an application's signification system must not destroy the original developers' message. Thus, the kind of EUD that Semiotic Engineering is prepared to deal with only involves negotiating meaning revisions with the designers' deputy at interaction time. This particular case of EUD might best be named end user *further* development.

SERG's related research publications

de SOUZA, C. S. ; BARBOSA, S. D. J. (2006) A semiotic framing for end-user development. In: Henry Lieberman; Fabio Paternò; Volker Wulf. (Org.). *End User Development: Empowering people to flexibly employ Advanced Information and Communication Technology*. New York: Springer, 2006, v. 9, p. 401-426

de SOUZA, C. S. (2005) Semiotic engineering: Bringing designers and users together at interaction time. *Interacting with Computers*. Vol. 17, n. 3, pp. 317-341.

BARBOSA, S. D. J., de SOUZA, C. S. (2001) Extending software through metaphors and metonymies. *Knowledge Based Systems*. Vol.14, n.1-2, pp.15-27.

de SOUZA, C. S., BARBOSA, S. D. J., SILVA, S. R. P. (2001) Semiotic Engineering Principles for Evaluating End-User Programming Environments. *Interacting With Computers*. Vol.13, n. 4, pp.467-495.

BARBOSA, S. D. J. (1999). *Programação via interfaces*. [Title in English: Programming via interface]. Ph.D.Thesis in Portuguese. Presentation: 23/12/1999. 109 p. Advisor: Clarisse Sieckenius de Souza.

[1] Eco, U. (1984) *Semiotics and the philosophy of language*. Indiana University Press.
[2] Houser, N. and Kloesel, C. (Eds.) (1992-1998) *The essential Peirce*. Vols. I, II. Indiana University Press.
[3] de Souza, C. S. (2005) *The semiotic engineering of human-computer interaction.* The MIT Press.
[4] Norman, D. A. (1986) Cognitive Engineering. In *User Centered System Design* (Norman & Draper, Eds.). Lawrence Erlbaum.
[5] See note 2.
[6] Eco, U. (1976) *A theory of semiotics.* Indiana University Press.
[7] For a discussion about usability and creative use see Adler, P. & Winograd, T. (1992) *Usability: Turning technologies into tools*. Oxford University Press.
[8] Lieberman, H.; Parternò, F.; Klann, M.; Wulf, V. (2006) End-user development: An emerging paradigm. In *End-User Development* (Lieberman, Paternò and Wulf, Eds.). Springer. p. 1.
[9] Prates, R. O., de Souza, C. S., and Barbosa, S. D. 2000. Methods and tools: a method for evaluating the communicability of user interfaces. interactions 7, 1 (Jan. 2000), 31-38.

End-User Design

Prof. Dr. Alexander Repenning
University of Lugano

Problem

Are UML diagrams a good tool to teach middle school students how to make video games? Probably not, but what kinds *end-user design* aids such as mental models, scaffolding structures, examples or other kinds of objects to think we can we give to end-users in order to gradually introduce them to good programming practice?

In the end-user programming arena the fundamental challenges have gradually shifted from basic syntactic challenges towards semantic challenges including the need to convey an understanding of design and engineering principles relevant to end-users. Visual programming has significantly lowered the threshold of programming [1] mostly by sharply reducing or even completely eliminating syntactic programming challenges. Visual programming languages using drag and drop mechanisms as programming approach make it virtually impossible to create syntactic errors. Even traditional text-based end-user programming approaches nowadays provide useful tools such as syntax coloring, symbol completion and wizards to reduce syntactic problems quite effectively. With the syntactic challenge being – more or less – out of the way we can focus on the semantic level of end-user programming.

At a semantic level the challenges ahead are substantially more complex. How can the user explore what a program does or design a new program in a systematic way? Programming languages including interactive debuggers such as Ruby, Lisp and Smalltalk allow users to comprehend code gradually by allowing them to decompose code into smaller code fragments such as individual methods that they could test in the context. The comprehension problem is a hard one but still relatively simple compared to the composition problem. If we have a specific problem in mind how do we think about the problem? How do we gradually map the problem, using some form of programming, to a solution? The “cursor is blinking in the upper left corner of an empty window” problem is hard because we know next to nothing about the users’ intensions. Are they trying to write a game or trying to solve a bookkeeping problem. How should they think about the problem in general? All of this leads to the challenge of end-user design.

Our Work

From systems perspective we have explored the notion of *tactile programming* [5] as means to make program comprehension and composition more concrete. Any piece of code can be tested and explained through an explanation generator interpreting a program and presenting in to the user. These explanations include specific representations of what function parameters means and even how their actual values should be interpreted. We have also explored knowledge-based approaches to computer supported program synthesis [4]. Programming by Analogous Examples has used small semantic annotation provided by the user to enable very high-level reuse.

From a pedagogical perspective we have tried to convey design thinking using a design scaffolding processes. Our Gamelet design [2] approach provides a couple of concrete design stepping stones and game design patterns to turn a game idea description into a running game. Using AgentSheets [3] and AgentCubes [1] we have employed variants of the approach in education. At the university level we have graduate and undergraduate computer science students using a version of the Gamelet design process including UML diagrams to build sophisticated games. At the middle school level we successfully experimented with more informal versions of the Gamelet design approach. This work has resulted in the world most compact game design course in which middle school children build for instance a simple Frogger game in very little time.

Challenges

Our initial efforts into end-user design suggest that it is possible to create semi-formal approaches to convey design knowledge to end-users. However, we feel that we are only at an early beginning and wonder if there is a more general *science of end-user design*.

- Can we create useful end-user design methods by scaling down existing design and engineering methods?
- Could – and should – there be something like an end-user UML diagram or and end-user pattern?
- Would we just be dumbing down real design issues and loose all value (e.g., The Idiots Guide to Software Design & Engineering)?
- If we make clever tools such as natural language problem statement parsers that would automatically create finished games would the users still be able to gain design knowledge?

References

[1] Repenning, A. and Ioannidou, A., AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D. in IEEE Symposium on Visual Languages and Human-Centric Computing 2006, (Brighton, United Kingdom, 2006), IEEE Press.

[2] Repenning, A. and Lewis, C., Workshop: Gamelet Design for Education. in Annual Games, Learning & Society Conference (GLS 2006), (Madison, Wisconsin, 2006).

[3] Repenning, A. and A. Ioannidou 2005. What makes End-User Development Tick? 13 design guidelines. End-User Development. F. Paterno and V. Wolf. Dordrecht, Kluwer.

[4] Ioannidou, A., Programmorphosis: a Knowledge-Based Approach to End-User Programming. in Interact 2003: Bringing the Bits together, Ninth IFIP TC13 International Conference on Human-Computer Interaction, (Zürich, Switzerland, 2003).

[5] Repenning, A., and J. Ambach, "Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing," Proceedings of the 1996 IEEE Symposium of Visual Languages, Boulder, CO, Computer Society, 1996, pp. 102-109.

End-User Development Techniques for Enterprise Resource Planning Software Systems

Michael Spahn, Stefan Scheidl, Todor Stoitsev

SAP AG, SAP Research CEC Darmstadt, Bleichstr. 8, D-64283 Darmstadt
{michael.spahn, stefan.scheidl, todor.stoitsev}@sap.com

Abstract. The intent of this position paper is to present the focus of interest of our end-user development (EUD) related research at SAP Research CEC Darmstadt, enabling other participants of the Dagstuhl seminar concerning end-user software engineering to prepare for fruitful and constructive discussions. As we are in an early phase of research, research topics will be presented rather than detailed results. We focus on investigating and applying EUD techniques suitable for enterprise resource planning (ERP) software systems, especially for small and medium-sized enterprises (SMEs). Our current research addresses the sub-domains of workflow management and business intelligence.

1 Customization of ERP systems

One dilemma of developing ERP software systems is to develop systems being on one hand generic enough to be used by a broad variety of companies and on the other hand offering solutions that match the concrete reality of a company as close as possible. As a consequence ERP systems are highly customizable, causing a long and costly implementation phase, involving external experts and consultants, which have to deal with the domain knowledge of users and adapt the software according to existing needs and processes. Since companies are not static and competitors, markets and customers are always on the move, influencing the strategies, products, services and processes of a company, a continuous need for adaptation exists which is not limited to the implementation phase of ERP software. End-users of ERP systems are domain experts but not necessarily IT professionals, limiting their ability to adapt the software by themselves to their own needs and forcing them to indirectly influence the adaptation processes by communicating their needs to IT professionals. Empowering the end-users to adapt the software by themselves is an important step in reducing customization costs and enabling high-quality tailoring of software and working environments to the needs of modern information and knowledge workers. EUD defined "as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artifact" [1] may deliver significant ideas of how to improve the evolutionary process of adapting ERP systems to changing company and user needs.

2 EUDISMES

Currently we are involved in the EUDISMES (End User Development in Small and Medium Enterprise Software Systems) project lead by Prof. Dr. Wulf of Siegen University and funded by the German Federal Ministry of Education and Research (BMBF) to further investigate EUD techniques. Focusing small and medium-sized enterprises (SMEs) aggravates the need for sophisticated EUD techniques. Large enterprises usually have a high management expertise and fine grained, standardized processes, which simplifies the adoption of predefined ERP processes and contents. In contrast to large enterprises SMEs have more human centric and less rigid processes. Having less financial resources and IT expertise, many SMEs fear the challenge of ERP implementation, operation and maintenance. As described in [2] we see good opportunities for applying EUD techniques in ERP systems for SMEs, especially in the sub-domains of managing agile workflows and business intelligence applications.

2.1 Creating and Managing Workflows

To achieve transparency and efficiency in the execution of business processes they are modeled and automated as workflows involving the routing of tasks and related documents and information. In addition to an initial implementation of processes, the dynamics of market requirements necessitates an ongoing adaptation of business models and hence workflow models. Although various visual tools provide an appealing user interface, the complexity of process (re-) engineering still exceeds the capabilities of most end users. At the same time, classical workflow systems are too rigid for many users, in particular knowledge workers, tackling complex processes with significant deviations from case to case.

As organizations seek to leverage their skilled people, reduce training time and support end-to-end workflow across all aspects of the business, application software now needs to be designed for the individual. Gartner has coined this people-process intersection the “process of me” [3]. End-user development tools can help to increase the adaptability of workflow systems to individual work practices. Techniques such as programming by example can help to overcome the separation between design- and runtime, giving the possibility of and confidence in successful system control back to end users.

2.2 Customizing Business Intelligence Applications

Business Intelligence (BI) applications are tools for analyzing data for the purpose of providing relevant information to enable better and faster business decisions. This purpose of BI applications turns every employee in a potential consumer of BI applications. Since the details of business decisions are specific to each company, their processes and the user context, there is no complete analytic application out of the box. Companies spend a huge amount of time and money on the customization and extension of commercial BI products to deliver business relevant information to the user in his work context in a business-user-oriented and easy-to-use way. Since the work of modern information and knowledge workers tends to consist of more and more non-routine, cognitive, analytic and interactive tasks, users have to be empowered to find, explore, process and analyze the data they need in the situation they want in an intuitive and user-friendly way. To meet their information needs, users create or adapt suitable informational artifacts, like reports or queries, confronting them with a huge information space of available data and technical details of data storage and querying. Ideally, users should be able to develop BI artifacts, like queries, reports or key performance indicators, by only using business concepts and terminology provided by an appropriate abstraction layer hiding technical details and reflecting only business relevant data entities and their relations. Related research addresses the question of how to build such abstraction layers on top of complex ERP systems, how to allow end-users to navigate huge information spaces, how to easily orchestrate queries in a descriptive way and how to enable end-users to interactively explore and analyze data to effectively improve speed and quality of business decisions.

3 Literature

[1] H. Lieberman, F. Paterno and V. Wulf: “End-User Development“, Springer, Dordrecht, 2006.

[2] A. Roth, S. Scheidl: “End-User Development for Enterprise Resource Planning Systems“ in Workshop “End User Development“ at “Informatik 2006“, Dresden, Germany, 2006.

[3] Gartner: “Business Application Vendors Face Challenge to Move to 'The Process of Me'”, 2006 (http://www.gartner.com/it/page.jsp?id=492897, accessed on Jan. 15, 2006)

End-user Programming of Ambient Narratives

Mark van Doorn
Media Interaction
Philips Research
mark.van.doorn@philips.com

1. INTRODUCTION

Ambient Intelligence is a vision on the future of consumer electronics, telecommunications and computing in which devices move into the background while at the same time placing the user experience in the foreground. Ambient intelligence is related to ubiquitous computing, pervasive computing but has a stronger connection to human computer interaction and design. Technically, Ambient Intelligence refers to the presence of a digital environment that is sensitive, adaptive, and responsive to the presence of people [1]. Producing Ambient Intelligent environments on a large scale is problematic however. First, it is technologically not possible in the foreseeable future to mass produce a product or service that generates Ambient Intelligence, given the current state-of-the-art in machine learning and artificial intelligence. Second, it is economically not feasible to manually design and produce Ambient Intelligence applications for each person individually. One of the main research questions in creating such environments is the design of a system capable of supporting mass customization of ambient experiences.

To address this research question an iterative top-down and bottom-up approach has been followed to gradually narrow down the solution space. The reason for adopting a top-down, analytical view was that it is easy to get lost in the wide variety of prototype systems, scenarios and examples that can be found in literature or gathered by doing empirical studies with end-users. The bottom-up, empirical view is necessary to ensure that any analytically derived concept is supported in practice, backed up by real world evidence. By repeatedly switching between these two perspectives, the concept is refined and eventually the design shaped.

2. BACKGROUND

The goal of Ambient Intelligence is to help people in performing their daily activities better, by making these activities more convenient and enjoyable: by introducing interactive media. Notice the word ‘performing’ in this description. In order to understand where and how Ambient Intelligence can be applied to support these performances, it is necessary to develop a better insight into what performances are and what it means to perform. Because performances vary so widely from medium to medium and culture to culture, it is hard to pin down an exact definition for performance. Schechner defines performance as “ritualized behavior conditioned/permeated by play” or “twice-behaved behavior” [9]. When people are performing, they show behavior that is at least practiced once before in a similar manner. In traditional performance arts this behavior can be detected easily: Actors in a theatre play, opera or movie rehearse their roles off-stage and repeat this behavior when they are on stage. But this twice-behaved behavior can also be seen in a priest conducting a wedding ceremony, a surgeon operating on a patient or a McDonald’s service employee behind the counter. Pine and Gillmore [8] argue how we live in an experience economy where work is theatre and every business a stage. In our own homes, we show signs of repeated behavior. This happens for example during everyday rituals, like brushing your teeth in front of a mirror in the morning, watching a soccer match with friends, or, coming home from work in the evening. Note that, here, the sending and receiving party in a ‘performance’ may be the same.

Viewing life as social theater is interesting for us for two reasons: First, if people behave according to social scripts, we can codify interactive media applications to support people in carrying out these scripts. Just as lighting and sound effects add to the overall drama of a theater play, Ambient Intelligence may thus be applied to enhance the performance described by these social scripts. Second, positioning Ambient Intelligence in performance theory gives us a well-studied and familiar frame of reference for the design of Ambient Intelligence environments and the underlying technology.

To model media-enhanced performances in the home and commercial service encounters in a machine understandable way, we choose to represent the structure and interrelationships of a set of related media-enhanced performances as an interactive or episodic narrative. Interactive narratives allow readers to affect, choose or otherwise change the plot of a story [7]. Most interactive narratives are situated either in the real world (e.g. live-action role playing games, improvisational theater) or in some virtual reality (e.g., hypertext novels, computer games). Another difference is that these media-enhanced performances are not really ‘read’ like a book or hypertext novel, but enacted like a theater play. We introduce the term *ambient narratives* to denote dramatic, interactive narratives that play in a mixed reality setting. We can look at ambient narratives from a consumer (reader) point of view or a producer (writer) perspective. From a reader point of view, interaction with the ambient narrative creates the perception of an intelligent surroundings. Interaction should be taken very broadly here as an ambient narrative can span both virtual and physical dimensions at the same time. Media-enhanced performances in different rooms may be linked to each other in one narrative structure, allowing people to influence the plot of the ambient narrative (the evolving Ambient Intelligence) by simply walking around for example. From a writer's perspective, the ambient narrative describes all possible media-enhanced performances and their interrelationships. Real-life environments are however highly complex and constantly changing so it is almost impossible for a producer to write ambient narratives for a given space in advance. Therefore end-users should be able to program their own ambient narratives.

The ambient narrative concept in itself is useful because it relates media, architecture and performance but in order to build a working system, we need to map the concept in a machine readable form. The underlying computer model and algorithms are inspired by interactive storytelling and hypertext systems such as [5, 10]. Essentially, implicit contextual information derived from sensors in the environment and explicit user feedback is fed into an interactive storytelling engine that will determine which media-enhanced performance scripts must be (de)activated given the database of scripts and the state of the ambient narrative engine. Each script consists of a preconditions part and an action part. If a script is selected and the preconditions are valid, the action is executed. Each action consists of three parts: an initialization, main and final part. The main part contains a description of a distributed hypermedia presentation in AmBX [2], a language and system used for enhancing game experiences with lighting and other ambient effects. The initialization section is used to set story values or triggers for other scripts before the amBX script is started. The final section gives the author the possibility to define story values or triggers that are executed right before the script is becoming inactive again. More information can be found in [4].

3. EVALUATION AND FUTURE RESEARCH

The ambient narrative concept and language model needs to be validated against the requirements placed by real world applications. To get a better notion of the type of language constructs needed, we performed a literature study where we analyzed existing scenarios, storyboards and prototype systems focused on the home and retail domain. The retail (and hospitality) domain is interesting in particular because no retail space is the same as each store wants to convey its own unique brand image. Furthermore, stores frequently change their collections. From this literature study we found that frequently appearing context paramaters are the location and identity of people, devices and objects, user roles and history.

To test the ambient narrative system we further made a prototype and refactored an intelligent shop window environment in ShopLab, the feasibility and usability laboratorium at the High Tech Campus in Eindhoven, to see if a typical ambient intelligent application would fit the model. The intelligent shop window reacts to the presence of users depending on their distance to the window and adapts its interaction style accordingly: When nobody is close, the transparent displays convey the style of the store by showing brand images. When a person stands directly in front, the transparent display switches to interaction mode. The person can then point or look at products in the shop and receive additional information about that product on the shop window, see Figure 1. We managed to map this application onto the ambient narrative model but found out we needed to filter out noise from the sensor data to make the system stable. Furthermore, some bypasses around the ambient narrative engine had to be made so that high frequency context data such as gaze input or pressure floor data could be fed directly to the devices if the device had gained focus by the narrative engine.

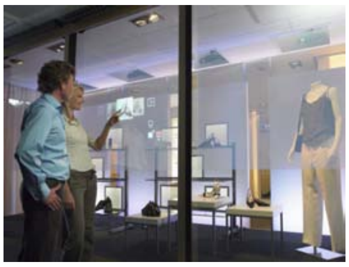


Figure 1: Touch interaction with the shop window display.

Currently, we are in the process of working out several different authoring strategies that allow end-users, in our case retail experience designers to quickly program environments such as the intelligent shop window. Several different programming strategies have been proposed in literature (e.g. desktop-based, in-situ, programming by example) e.g. [3, 11, 6] for ubiquitous computing environments. But in a series of workshops with designers of retail spaces we hope to collect qualitative feedback on which strategy they prefer when and see in how far they can think in the ambient narrative mental model. Finally, these end-user programming strategies need to be evaluated in ShopLab.

4. REFERENCES

[1] E. Aarts and S. Marzano, editors. *The New Everyday: Views on Ambient Intelligence.* 010 Publishers, 2003.

[2] amBX. http://www.ambx.com/.

[3] R. Hull, B. Clayton, and T. Melamed. Rapid Authoring of Mediascapes. In *UbiComp '04*, 2004.

[4] M. van Doorn and A. P. de Vries. Co-creation in Ambient Narratives. *Lecture Notes in Computer Science: Ambient Intelligence for Everyday Life*, (3964), 2006.

[5] M. Mateas and A. Stern. Facade: An Experiment in Building a Fully-Realized Interactive Drama. In *Game Developer's Conference: Game Design Track*, San Jose, California, 2003.

[6] T. McNerney. From turtles to Tangible Programming Bricks: explorations in physical language design. *Personal and Ubiquitous Computing*, 8(5):326–337, September 2004.

[7] J. Murray. *Hamlet on the Holodeck: The Future of Narrative in Cyberspace.* MIT Press, 1998.

[8] J. Pine and J. Gillmore. *The Experience Economy.* Harvard Business School Press, 1999.

[9] R. Schechner. *Performance Studies: An Introduction.* Routledge: New York, 2002.

[10] P. Stotts and R. Furuta. Petri-net-based hypertext: document structure with browsing semantics. *ACM Transactions on Information Systems*, 7(1):3–29, 1989.

[11] M. Weal, D. Michaelides, M. Tompson, and D. D. Roure. The Ambient Wood Journals - Replaying the Experience . In *ACM Hypertext*, Nottingham, UK, 2003.

End-User Software Engineering and Professional End-User Developers

Judith Segal

j.a.segal@open.ac.uk

Professional end-user developers

By the term 'professional end-user developers' is meant professionals working in a highly technical, knowledge-rich domain who develop their own software in order to further their professional work. I have conducted empirical studies of such developers working in the domains of financial mathematics ([1], [2]), earth and space sciences ([2], [3]), and, currently, structural biology. These developers are distinguished from other end-user developers in two ways. The first is that, consistent with their being familiar with formal notations and logical scientific reasoning, they tend to have few problems with coding per se. The second is that, as a class, they have a history of developing their own software which long predates the advent of the PC.

My empirical studies reveal that professional end-user developers work within a culture which, while recognising that software failure can have devastating consequences, nevertheless does not appear to value the knowledge, effort and skill required for software development. There is a widespread perception that software development is something that 'anyone can do' – it is regarded as just part of the armoury of skills that all such professionals are considered to have, or to be able easily to acquire (it was compared to glass-blowing by one earth scientist). The studies also demonstrate that professional end-users typically develop software in a highly iterative manner and requirements largely emerge, rather than being specified upfront.

In common with many other types of end-user development, professional end-user development is generally regarded as simply being a matter of coding. The consequence of this is that the products of professional end-user development can often be less than robust, and the underlying code is not written with a view to being comprehensible or easy to modify and maintain. This is especially problematic as software originally intended for individual, exploratory use might later be used by other scientists and take on a more formal role within the organisation.

Some problems of professional end-user development

The culture of professional end-user development poses some significant problems such as:

1. How does the professional end-user developer acquire the software development knowledge that it is assumed that 'everybody' has (or can easily acquire)? One exacerbating factor is the reluctance of some professional end-user organisations to expend resources on formal training. Another is that the community of practice of professional end-user developers can be small in any particular organisation and is inherently unstable. Those research scientists who develop software tend to be on short-term contracts and/or at the beginning of their careers. In the latter case, as they advance up their career ladder, they are likely to concentrate on their scientific endeavours, leaving others to develop their software for them. In either case, their knowledge of software development becomes lost to the community.

2. How can awareness that software development is more than merely coding, become embedded in the culture of professional end-user development? If this awareness were so embedded, then the quick construction of a piece of software that appears to address the domain problem at hand (rather than expending more resources on the construction of software which is robust and maintainable) would be the result of a conscious decision rather than the unthinking norm, as at present.

How might these problems be addressed?

There are various suggestions in the end-user literature as to how the product of end-user development might be improved. These suggestions include: end-user developers should adopt a software development methodology; software engineers might provide professional end-user developers with a library of customisable components, and the professional end-user developers themselves should develop some sort of library of reusable customisable components.

Each of these suggestions requires some change in work culture and practices. Given that such changes are very difficult to effect and especially so where software development is not the main focus of the organisation, I argue in [3] that we should 'cherry pick' methods, tools and practices which have proved effective in the context of professional software development and which support – or only slightly perturb – the way that professional end-user developers work naturally. Given the additional difficulty that professional end-user developers have in acquiring and sharing knowledge of software development, I argue that we should also look for methods, tools and practices which tend to strengthen their community of practice.

Agile methods appear to be a likely source of practices etc. which both support the way professional end-users currently develop software and strengthen the community of practice. However, the effect of the introduction of agile practices into a professional end-user developer community is still to be investigated.

References

[1] Segal J., 2001, 'Organisational Learning and Software Process Improvement: A Case Study', in *Advances in Learning Software Organizations,* K-D Althoff, R.L. Feldmann, W. Muller (Eds.), Lecture Notes in Computer Science, Vol. 2176, Springer, 68-82.

[2] Segal, J.,2005, 'Two principles of end-user software engineering research', Proceedings of the 1st Workshop on End User Software Engineering, WEUSE, International Conference of Software Engineering, St. Louis Missouri, May 2005.

[3] Segal J., 2005, 'When software engineers met research scientists: a case study', *Empirical Software Engineering,* 10, 517-536.

End-User Software Engineering Position Paper

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

PERSONAL WORK

My goal is to make the process of programming easier, especially for people who are not necessarily specialists in computer science. Why is it so much harder to program a computer than simply to use a computer application? I can't think of any good reason why this is so; we just happen to have a tradition of arcane programming languages and mystically complex software development techniques. We can do much better.

My background is in Human-Computer Interface and Artificial Intelligence, and my methodology is to use ideas from these fields to improve the situation. HCI has amassed an enormous body of knowledge about what makes interfaces easy to use, and this has been applied widely to many computer applications for end users. Oddly, little of this has been applied to making interfaces for programming easier to use. Non-experts tend to believe that programmers practice a kind of voodoo, perceived to be naturally arcane and mysterious. Since they can handle it so well, programmers aren't perceived as needing ease of use. But we all pay the price for this misconception.

Programming is the art of teaching new behavior to a computer. It's really the same problem as machine learning, which is where AI comes in. I believe the route to making programming easier is to make the computer smarter, make it capable of learning, and capable of accepting direction in the way that users feel most comfortable expressing it.

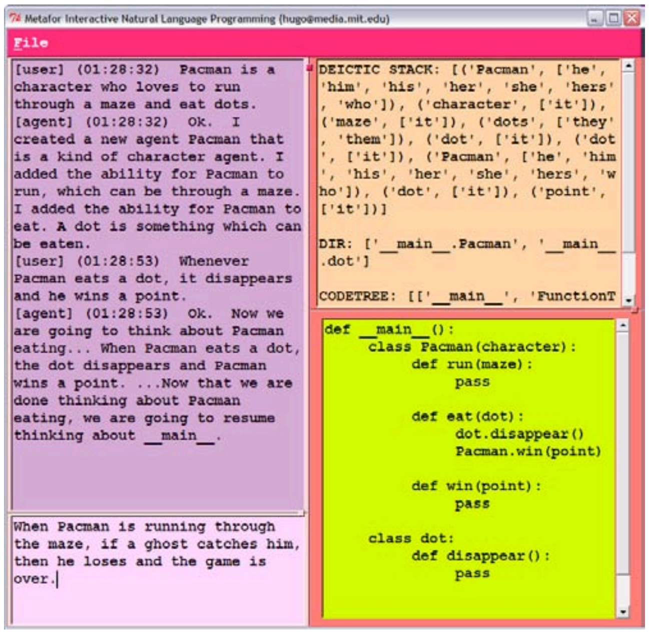
To that end, I've been exploring the following topics, among others:

Programming in Natural Language

Programming languages are a stumbling block for most beginning programmers. Why not just express what you want in English? Many believe this goal to be infeasible, but natural language understanding has made vast progress in recent years. We can use partial understanding, mixed initiative dialog models, and Commonsense reasoning to, at least partially, express procedural ideas in natural language [5]. I've also explored several ideas in Visual Programming, since some things are best expressed in pictures rather than words.

Programming by Example

People learn and teach best by example. But conventional programming languages require you to express procedures in the abstract, rather than through examples. I'd like to see the ability to demonstrate examples in concrete situations, have the system record them, and generalize them to yield a procedure capable of working on analogous examples. I've made several systems in this area, and edited a book [2].


Metafor Interactive Natural Language Programming (hugo@media.mit.edu)
File

```
[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__.
```

```
When Pacman is running through
the maze, if a ghost catches him,
then he loses and the game is
over.
```

```
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', 'w
ho']), ('dot', ['it']), ('point',
['it'])]

DIR: ['__main__.Pacman', '__main__
.dot']

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
```

Figure 1: Metafor Natural Language Programming system

Debugging

I think the most pressing need in software development is not programming per se, but *debugging*. Programmers spend roughly half their time debugging, but debugging tools have hardly improved since the earliest days of computing. I've worked on several innovative reversible, graphical debuggers, based on ideas from diagnostic reasoning in AI. [1]

I've been exploring the idea of *end-user debugging* [3], what one might call "debugging without programming". The idea is that even when ordinary application use fails to meet the expectations of users, they could fruitfully use debugging techniques to discover what went wrong.

END-USER DEVELOPMENT AND SOFTWARE ENGINEERING

My book [4], edited with Fabio Paterno and Volker Wulf, I think, provides a comprehensive, and up-to-date survey of the state of the art and future directions in this area. One of the big successes of the book, I believe, was to bring together the community of people who are working to make programming easier for beginners, such as children, with people from Software Engineering who are trying to move some of their techniques into the realm of less expert users. This workshop, of course, continues that interdisciplinary effort.

For the book, we chose the title *End-User Development*, rather than "Software Engineering", though the concerns were pretty much the same as this workshop. Software Engineering has a lot to contribute in terms of bringing design methodology, collaborative programming, testing, maintenance, and other larger concerns to programming for less expert users. But I think we still have a problem positioning this effort with respect to conventional Software Engineering. For the book, I was worried that a too-close association would scare off beginning and casual users.

I commend the bravery of the workshop organizers for reaching out to the SE community. Traditional Software Engineering is really largely about the management and organization of large software projects in industry. Sometimes they are not so sympathetic to efforts to make programming easier, because you don't want to make it too easy to modify a large software project. The risk of errors or miscoordination is too great. Reliability and efficiency sometimes trump ease of development, for applications like banks and airline reservations. The approaches advocated by SE are much too heavyweight for beginning users – bureaucratic design methodologies, abstract formal verification – at least, without some radical rethinking.

I think the ideal to shoot for, first, is to maximize ease of getting started in the beginning. Program development should be as informal, flexible, lightweight, agile, and dynamic as we can possibly make it. Formal methodologies be damned. If we can do this, we can make programming accessible to millions of people who are now scared to death of it.

But as programs (or their developers) mature, some of the legitimate concerns of Software Engineering do indeed come into play, even for nonprofessional users. Programs may then need to be maintained, tested, extended, shared with others, etc., in which case Software Engineering techniques could potentially yield benefits. But we shouldn't just, as conventional SE does, wag our fingers at the users, "Ya shoulda done it right in the first place".

I think our challenge is to figure out how to *smoothly* go from the initial conception of a project, vague and imprecise at it must necessarily be, to only gradually introduce more structured representations and abstract tools, all the while without placing undue burdens on the user. To do this, I think we have to give up the idea of a single representation for programs, be it a programming language or something else.

I also think that this process of solidifying a program should be *reversible*, so that any point, one can return to the more informal forms without needless loss of effort. This will encourage the user to learn new insights from the process of software development without feeling like they get stuck by their sunk investment in an initial approach.

Finally, as much as is possible, we should make this process as *automatic* as we can, though the use of program transformation, dependency maintenance, automated reasoning, mixed-initiative interfaces, visualization, and machine learning. Otherwise, I think it will be too much overhead for a non-expert user themselves to keep track of the myriad facets that software development entails. If we succeed in this, people will become End-User Software Engineers without their even realizing it.

REFERENCES

Most of these references can be found at:

http://www.media.mit.edu/~lieber/Publications/Publications.html

1. H. Lieberman, ed. Special Issue on *The Debugging Scandal*, Communications of the ACM, April 1997.
2. H. Lieberman, ed., Your Wish is My Command: Programming by Example, Morgan Kaufmann, 2001.
3. E. Wagner and H. Lieberman, End User Debugging for Electronic Commerce, ACM Conference on Intelligent User Interfaces, Miami Beach, January 2003.
4. H. Lieberman, F. Paterno and V. Wulf, eds. *End-User Development*, Springer Academic Publishers, 2006.
5. H. Lieberman and H. Liu, Metafor: Visualizing Stories as Code, ACM Conference on Intelligent User Interfaces (IUI-2005), San Diego, January 2005

Exploiting Domain-Specific Structures For End-User Programming Support Tools[*] — Position Paper —

Robin Abraham Martin Erwig
Oregon State University

1. PROGRAMMING LANGUAGE RESEARCH FOR SPREADSHEETS

In previous work we have tried to transfer ideas that have been successful in general-purpose programming languages and mainstream software engineering into the realm of spreadsheets, which is one important example of an end-user programming environment. More specifically, we have addressed the questions of how to employ the concepts of type checking, program generation and maintenance, and testing in spreadsheets. While the primary objective of our work has been to offer improvements for end-user productivity, we have tried to follow two particular principles to guide our research.

(1) Keep the number of new concepts to be learned by end users at a minimum.
(2) Exploit as much as possible information offered by the internal structure of spreadsheets.

In the following we will illustrate our research approach with several examples.

The idea behind the UCheck system [8] is to interpret the labels in a spreadsheet as annotations akin to type declarations in traditional programs. By identifying rules that express how labeled cells can be combined in formulas in a meaningful way, the information about cell labels can then be exploited to check the consistency of spreadsheet formulas [13]. To make this approach feasible we needed a way to automatically infer the information about which labels are to be used as type information and which cells are annotated by which labels [1], because a tool that required a spreadsheet user to annotate a spreadsheet with this information would probably not be very widely used due to the high additional cost involved. We have also begun to investigate ways to infer from the inconsistent use of labels in formulas suggestions for changes in formulas that can be reported to the end user [3].

We have also investigated a different approach to type checking that is based on the traditional notion of types, extended by a concept of formula shapes [6]. In addition to finding errors in spreadsheets, this approach can also be used to infer spreadsheet models, an aspect to be discussed below.

A strong point about the type checking approaches is that they operate fully automatically—all an end user has to do is to click a button, and sources of potential errors are found and highlighted instantly. At the same time, this advantage can also mean a drawback since end users might rely too much on the system, in particular, they might assume that their spreadsheet is correct when UCheck does not report an error, not knowing or ignoring the fact that automatic type checking cannot be complete in the sense of finding all errors in a program. Therefore, other methods to finding errors are needed to complement automatic type checking. One example is the WYSIWYT approach, invented by Rothermel, Burnett, and others [15], which supports end users with systematically testing their spreadsheets. An important objective of testing is to achieve sufficient coverage of the program being tested. Supporting the user in finding test cases attaining high coverage is the goal of automatic test case generators.

We have developed one such tool called "AutoTest" [4], which generates test suites that obtain 100% DU-coverage (for reachable code). This is an improvement over a previous approach, "Help Me Test", that was developed for the WYSIWYT framework [14]. AutoTest is also considerably more efficient than Help Me Test.

Once a user has identified through testing that a cell contains a wrong value, the next problem is to find out where the error is located in the spreadsheet and how to correct it. To this end, we have developed a method called "goal-directed debugging", or "GoalDebug", which asks the user for a correct value for that cell and then computes a ranked list of suggested changes for formulas, each of which would cause the specified target value to be computed [2]. These changes can be automatically applied, which eliminates a whole class errors introduced by end users during the editing of formulas. Using a systematic study based on mutation testing, we have found that GoalDebug consistently presents the correct changes among the most highly ranked suggestions [7].

While all the previously mentioned approaches try to detect errors, the goal of the Gencel system [11] is to prevent the introduction of errors into spreadsheets. The system is based on a concept of templates that capture the potential evolution of a spreadsheet over time. Changes to spreadsheets, such as insertion and deletion of (groups of) rows and columns are controlled by these templates that ensure the formulas will always be adjusted correctly. In fact, we can prove that spreadsheets maintained by Gencel based on these templates are always free from type, range, and reference errors [12]. Templates have a visual representation that is almost identical to the notation known to end users

[*]This work is partially supported by the National Science Foundation under the grant ITR-0325273 and by the EUSES Consortium (`http://EUSESconsortium.org`).

from spreadsheets [9] and can be created using a visual editor. We have also developed a method to infer templates from existing spreadsheets, which facilitates the use of Gencel for legacy spreadsheets [5]. The templates inferred by our system have been judged by experts to be better than those developed by novice and even expert users. We have extended the Gencel model to include more high-level modeling features while still retaining its visual attractiveness. The resulting ClassSheets model [10] also allows the integration of spreadsheet modeling into the UML modeling process.

All of our approaches to improve the quality of spreadsheets essentially exploit in some way or another the

- *simplicity* of the spreadsheet language, and
- *embedding* of computations in a spatial grid.

These two aspects allow the reasoning to be (a) simple enough because complicated language features, such as recursion and nested scope, need not be addressed and (b) supported by the spatial structure exhibited by the arrangement of cells.

2. FUTURE RESEARCH

Research for general-purpose languages has been quite successful and has produced important results from which most professional programmers benefit today. An example are the sound type systems now to be found in mainstream programming in languages, such as Java.

We have demonstrated with UCheck that it is indeed possible to bring the benefits of tools successfully employed in general-purpose languages to the realm of spreadsheets. Similarly, the Gencel/ClassSheets systems show that the idea of high-level modeling, as known from UML, can be employed successfully in the spreadsheet domain.

These experiences suggests as a successful strategy for future research:

> *Redesign methods known from general-purpose languages for end-user programming domains by exploiting application-specific structures and practices.*

In the spreadsheet domain, the spatial layout of cells entails the practice of end users to place closely related items in the same area, or in the same row or column. It is this combination of spatial structure and corresponding user practice that lets tools like UCheck or GoalDebug work so well. Therefore, identifying and exploiting such links might be a key step in designing successful end-user programming tools.

Example areas for new potential spreadsheet tools to be investigated are refactoring, version control, and use cases, to name just a few.

We believe that the successful transfer of concepts also requires a critical mass of researchers working in that area, which is currently hardly the case. Therefore, a second goal should be the following.

> *Persuade programming language and software engineering researchers to participate in the development of tools for end-user programming.*

The three most relevant papers are the following.

- UCheck, *JVLC 2007* [8]
- Gencel, *JFP 2006* [12]
- GoalDebug, *ICSE 2007* [7]

3. REFERENCES

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.

[2] R. Abraham and M. Erwig. Goal-Directed Debugging of Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 37–44, 2005.

[3] R. Abraham and M. Erwig. How to Communicate Unit Error Messages in Spreadsheets. In *1st Workshop on End-User Software Engineering*, pages 52–56, 2005.

[4] R. Abraham and M. Erwig. AutoTest: A Tool for Automatic Test Case Generation in Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 43–50, 2006.

[5] R. Abraham and M. Erwig. Inferring Templates from Spreadsheets. In *28th IEEE Int. Conf. on Software Engineering*, pages 182–191, 2006.

[6] R. Abraham and M. Erwig. Type Inference for Spreadsheets. In *ACM Int. Symp. on Principles and Practice of Declarative Programming*, pages 73–84, 2006.

[7] R. Abraham and M. Erwig. GoalDebug: A Spreadsheet Debugger for End Users. In *29th IEEE Int. Conf. on Software Engineering*, 2007. to appear.

[8] R. Abraham and M. Erwig. UCheck: A Spreadsheet Unit Checker for End Users. *Journal of Visual Languages and Computing*, 18(1):71–95, 2007.

[9] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual Specifications of Correct Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 189–196, 2005.

[10] G. Engels and M. Erwig. ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 124–133, 2005.

[11] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic Generation and Maintenance of Correct Spreadsheets. In *27th IEEE Int. Conf. on Software Engineering*, pages 136–145, 2005.

[12] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein. Gencel — A Program Generator for Correct Spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.

[13] M. Erwig and M. M. Burnett. Adding Apples and Oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 173–191, 2002.

[14] M. Fisher, G. Rothermel, D. Brown, M. Cao, C. Cook, and B. Burnett. Integrating Automated Test Generation into the WYSIWYT Spreadsheet Testing Methodology. *ACM Trans. on Software Engineering and Methodology*, 15:150–194, 2006.

[15] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, pages 110–147, 2001.

Gender HCI Issues in End-User Software Engineering Environments

Laura Beckwith, Margaret Burnett, and Susan Wiedenbeck[α]
Oregon State University
Corvallis, OR, USA
{beckwith, burnett}@eecs.oregonstate.edu

[α]Drexel University
Philadelphia, PA, USA
susan.wiedenbeck@cis.drexel.edu

INTRODUCTION

Until recently, research has not considered whether the design of problem-solving software, such as spreadsheets, multimedia authoring languages, and CAD systems, affect males and females differently. As a result, we began investigating how the two genders are impacted by problem-solving software and whether attention to gender differences is important in the design of software. Evidence from other domains, such as psychology and marketing (see [Beckwith and Burnett 2004]), strongly suggests that females process information and problem solve in very different ways than males do. This implies that without taking these differences into account in the design of problem-solving software, the needs of half the population for whom the software is intended are potentially being ignored. In fact, some research has shown that software is unintentionally designed for males.

To consider this issue, we are empirically investigating end users engaged in end-user software engineering activities, to inform the design of software to support end-user programmers of both genders.

METHOD

Our method for conducting this investigation consists of four steps: (1) draw from theory and previous gender difference empirical work from other domains—such as computer confidence, perceived risk, information processing, computing gaming, and technology adoption models—to hypothesize gender issues and their causes that could arise from gender-based differences in the use of problem-solving software [Beckwith and Burnett 2004], (2) use empirical methods to investigate whether these issues do actually arise in problem-solving software, (3) use the results of the first two steps along with qualitative empirical work involving low-cost prototyping to derive and refine approaches to address the issues, and (4) use quantitative empirical methods to evaluate the effectiveness of the approaches.

We have conducted four studies investigating gender differences relevant to end-user software engineering environments. Three of these are summarized here; more detail on the series of studies can be found in [Beckwith et al. 2006b].

EMPIRICAL EVIDENCE – SELF-EFFICACY

Guided by literature and early exploratory analyses, we performed a quantitative investigation of the impact of self-efficacy (a form of confidence) and gender on users' use of end-user testing and debugging features while debugging a spreadsheet [Beckwith et al. 2005]. The results of that study showed how these differences in self-efficacy negatively impacted acceptance of the features by females, and showed that the reduced feature acceptance can significantly reduce females' effectiveness at problem solving. More specifically:

- Females' self-efficacy was predictive of their effectiveness at using the debugging features, which was not the case for the males. See Figure 1. Thus, the (many) low self-efficacy females were unlikely to use the features, but the (few) low self-efficacy males were as likely to use the features as the high self-efficacy males were.
- Females were less likely than males were to accept the new debugging features (unfamiliar to all participants prior to the experiment). One reason females stated for this was that they thought the features would take them too long to learn—but there was no difference in the males' and females' learning of the new features.
- Although there was no gender difference in fixing the seeded bugs, females introduced more new bugs—which remained unfixed. This is probably explained by low acceptance of the debugging features: high effective usage of the features was a significant predictor of fixing bugs.

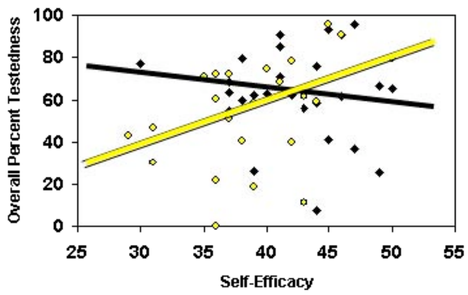


Figure 1. Females' (light) self-efficacy was a significant predictor of their effective use of the "check-off" feature, as the positively sloping line shows. For the males (dark), however, this was not the case.

QUALITATIVE EVIDENCE – FEATURES & MOTIVATION

A think-aloud study [Beckwith et al. 2006b] provided confirmatory evidence of females' beliefs and perceptions that seem tied to their avoidance of the debugging features. The experiment also revealed an interesting difference in the ways features were perceived by males and females. For example, female F2, in using the "guards" feature (akin to Excel's "data validation"), said:

> F2: "I don't think that you can get a -5 on the homework. No, it can't be. So 0 to 100 [is the guard I'm entering], ok. Ok, hmm… So, it doesn't like the -5 [...]. They can get a 0, which gets rid of the angry red circle."

In contrast to F2's focus on the guard feature as a way to get her spreadsheet to work correctly, the following male's initial focus was on the feature itself:

> M3: "The first thing I'm going to do is go through and check the guards for everything, just to make sure none of the entered values are above or below any of the ranges specified. So, homework 1—actually, I'm going to put guards on everything because I feel like it. I don't even know if this is really necessary, but it's fun."

Despite his initial interest in the feature for the fun of it, the male soon transitioned to its problem-solving advantages and was able to find and fix a bug with the aid of the feature. His use of guards because "it's fun" led us to the next study, investigating the role of exploratory investigation (or tinkering) as a manner of becoming comfortable with the features and environment.

EMPIRICAL EVIDENCE – TINKERING

In this quantitative study [Beckwith et al. 2006a], we originally anticipated that males' propensity to tinker (playfully experiment) would benefit their problem solving. However, we found that even small differences in the environments had big impacts on how gender and tinkering interacted and affected debugging effectiveness. More specifically:

- As in previous research, males tinkered more than females but, surprisingly, males' tinkering was often counterproductive to their effectiveness in debugging.
- One factor in the above result was the fact that the low-cost variant of the spreadsheet environment led some males to engage in unproductive, repeated tinkering, which was linked to poor understanding.
- Although they tinkered less, females' tinkering was effective: it was significantly tied to understanding and to successfully testing and debugging, regardless of environment. However, when tinkering in the more complex environment, females' tinkering was predictive of lower self-efficacy.

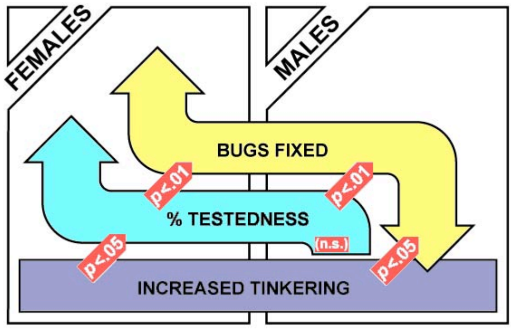


Figure 2. Males' and females' tinkering affected their debugging effectiveness, but in essentially opposite ways. Direction of stylized arrows depicts increase/decrease in a measure, and shaded arrows show significance of the regression relationships between measures.

- Tinkering with pauses allows for reflection and was helpful to everyone, but females were more likely than males to pause.

The essence of these results is depicted in Figure 2.

The implications are that designers should look for ways to encourage females' tinkering. Still, care must be taken to avoid at the same time encouraging males' tinkering further, since males' tinkering tended to be excessive and, when this was the case, was counterproductive.

SUMMARY

These results are being used to experiment with new ways software designs can counteract these effects. The outcomes of these experiments can provide the knowledge required to design future environments to better allow end-user programmers of *both* genders to succeed at end-user software engineering tasks.

REFERENCES

[Beckwith and Burnett 2004] Beckwith, L. and Burnett, M. Gender: An important factor in end-user programming environments? *IEEE Symp. Visual Languages and Human-Centric Computing Languages and Environments*, Sept. 2004, 107-114.

[Beckwith et al. 2005] Beckwith, L., Burnett, M., Wiedenbeck, S., Cook, C., Sorte, S., and Hastings, M. Effectiveness of end-user debugging features: Are there gender issues? *ACM Conf. Human Factors in Computing Systems*, April 2005, 869-878.

[Beckwith et al. 2006a] Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., and Cook, C. Tinkering and gender in end-user programmers' debugging. *ACM Conf. Human Factors in Computing Systems*, April 2006, 231-240.

[Beckwith et al. 2006b] Beckwith, L., Burnett, M., Wiedenbeck, S., and Grigoreanu, V. Gender HCI: What about the software? *IEEE Computer*, Nov. 2006, 83-87.

Helping Everyday Users Establish Confidence for Everyday Applications

Mary Shaw
Institute for Software Research, School of Computer Science
Carnegie Mellon University, Pittsburgh PA
mary.shaw@cs.cmu.edu

January 2007

Abstract

End users obtain their desired results by combining elements of information and computation from different applications. Software engineering provides little support for identifying, selecting, or combining these elements – that is, for helping end users to design computational support for their own tasks. Software engineering provides even less support to help end users to decide whether the resulting system is sufficiently dependable –whether it will meet their expectations. Many users, especially end users, base judgments about software on informal and undependable information, and they draw conclusions with informal rather than rational decision methods. We have been developing support for everyday dependability, with an emphasis on expressing expectations in abstractions familiar to the user and on obtaining software behavior that reasonably satisfies those expectations. In this Dagstuhl I would like to explore the differences between everyday informal reasoning and the rational processes of computer science in order to develop means for establishing credible indications of confidence for end users.

Everyday Dependability for Everyday Users

"Dependability" is an overarching property of software systems that includes, to various viewers and to various extents, elements of correctness, reliability, fault-tolerance, performance, security, usability (without surprises), robustness, accuracy, and numerous other properties. Everyday dependability provides enough assurance to carry out ordinary activities. Everyday software systems may be undependable, but the consequences of that undependability are not catastrophic, a human will probably notice and intervene before the effects spread, and the number of people affected is modest. For everyday software, it is generally not difficult to recover from failures.

Everyday users are not computing professionals; they create small software systems from available information and computing elements, they use abstractions drawn from their problem domains, theybare ill-equipped to evaluate the elements they use, and they are often mystified by the behavior and requirements of their software. We focus on their design and use of suites of these information and computing elements elements, applications, data sources, web pages, and other distribugted content rather than on correct programming within an application.

Within my group,

- Orna Raz explored dependability of online data feeds. The semantics of these data feeds sometimes goes awrt\y, but different users are sensitive to erroneous values to different extents. Raz developed a technique for end users to describe their individual expectations about a data feed and to translate those expectations to predicates that could monitor the data feed dynamically and adapt themselves to certain kinds of gradual systematic change in the data feeds.
- Vahe Poladian is exploring ways to provide users of mobile devices with the most satisfactory service given the limited resources of the device. Using a "task level" abstraction, he selects a

Dagstuhl Seminar Proceedings 07081
End-User Software Engineering
http://drops.dagstuhl.de/opus/volltexte/2007/1096

sequence of suites of applications that can perform the sequence of tasks planned by the user. Using models of resource availability and the resource consumption of candidate applications, he chooses the configurations that will lead to the best expected value of the user's utility.

- Chris Scaffidi is exploring abstractions that support the activities of specific identified groups of end user programmers. He refined Boehm's estimate of the number of end user programmers and collaborated with Information Week on a survey that suggested three distinct types of end users based on the kinds of abstractions they use (interestingly, they clustered by type of abstraction, not by type of application). He is developing a technique for helping users create data abstractions that correspond to the users' problems, to reformat the information as required by applications, and to express the degree of confidence a user should justifiably have that a value satisfies the expected abstraction.

Everyday Decisions

Everyday users do not have rich and robust mental models of their computing systems: they fail to do backups, misunderstand storage models (especially local and network storage), execute malware, and innocently engage in other risky behavior. The responses of computer science to the mismatch between computing systems and users' models has been to attempt to simplify the systems and to seek ways to help the users act "rationally". I hope to explore two aspects of this discrepancy at Dagstuhl.

First, we rely on "high ceremony" techniques (formal verification, systematic testing, empirical studies) for reasoning about correctness and dependability of systems. Everyday users have available a great deal of "low ceremony" evidence (reviews, reputation, informal experience, …) that is of variable accuracy and credibility. Nevertheless, many decisions are based on this sort of evidence. Can we find ways to track and manage the justifiable confidence in this evidence and to draw conclusions that are adequate for decisions about everyday software?

Second, psychologists have established that human decision-making does not adhere to rules of rational deduction, statistical analysis, and abstract reasoning about general cases. Rather, human decisions are very strongly shaped by examples, by cases that come easily to mind, and by similarity with experience. Simple linear models of a few inputs often out-perform even human experts, and there is some evidence that visual displays help with reasoning about probabilities. Can we find ways to accommodate typical human decision strategies in software design decisions, using some of the established techniques for improving informal reasoning and providing explicit indications of confidence in the results?

Acknowledgements

The work reported here is currently supported in part by the National Science Foundation (ITR-0325273) via the EUSES Consortium, by NSF Grants NSF-CCF-0438929, NSF-CNS-0613823, and NSF-CCF-0613822, and by the A.J. Perlis Chair of Computer Science at Carnegie Mellon University. The work was done in collaboration with Brad Myers, Vahe Poladian, Orna Raz, and Chris Scaffidi, and it has had the benefit of feedback from members of the EUSES Consotrium.

References

Vahe Poladian, Joao Pedro Sousa, David Garlan, and Mary Shaw. Dynamic Configuration of Resource-Aware Services. Position paper for *ICSE-2004, 26th Int"l Conf on Software Engineering*, Edinburgh, Scotland, May 2004, pp.604-613

Orna Raz, Rebecca Buchheit, Mary Shaw, Philip Koopman, and Christos Faloutsos. Automated Assistance for Eliciting User Expectations*. International Conference on Software Engineering and Knowledge Engineering (SEKE"04)*, Banff, Canada, June 2004, pp. 80-85.

Chris Scaffidi, Andrew Ko, Brad Myers, and Mary Shaw. Dimensions Characterizing Programming Feature Usage by Information Workers. *VL/HCC'06: Proceedings of the 2006 IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 59-62, 2006.

Chris Scaffidi, Mary Shaw, and Brad Myers. Games Programs Play: Obstacles to Data Reuse. *Position paper for 2nd Workshop on End User Software Engineering (WEUSE)*, at the Conference on Human Factors in Computing Systems (CHI), 2006, unpaginated.

Chris Scaffidi, Mary Shaw, and Brad Myers. Estimating the Numbers of End Users and End User Programmers. *VL/HCC'05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 207-214.

Joao Pedro Sousa, Vahe Poladian, David Garlan, Bradley Schmerl, and Mary Shaw. Task-based Adaptation for Ubiquitous Computing. in *IEEE Transactions on Systems, Man and Cybernetics - Part C: Applications and Reviews*, Vol. 36, No. 3, May 2006, pp. 328-339.

Interdisciplinary Design Research for End-User Software Engineering

Alan Blackwell

Dagstuhl seminar on End User Software Engineering, February 2007

My research style involves constantly drawing comparisons from one field to another – across academic disciplines, and also across application domains. In these terms, End-User Software Engineering is neither an application domain, nor an academic discipline, but a technological attitude or strategy, applicable in many domains, while also profiting from many research methods and theory bases. In this respect, it is an ideal opportunity for the multi-disciplinary enquiry and analogical comparisons on which I habitually base my own research [1].

The phrase "end-user software engineering" itself relies on an analogy, in the sense that software engineering is a professional discipline, whereas the end-users whom we hope to assist are defined precisely by the fact that they are not professionals (at least, not software professionals). Our aim in this research is to identify those techniques within software engineering that might offer most benefit to end-users, potentially including tools for specification, debugging, revision management and so on.

As a teacher of professional software engineering, I often draw on the experience of other professional fields, especially design disciplines such as architecture, typography and performance composition [2]. There are certain recurring themes across these design disciplines that I have found to offer substantial insights to professional software engineering. I believe that these same themes can also be productive sources of innovation, by making new analogies to end-user software engineering. In the remainder of this statement, I reflect on some of these analogies.

Design takes place in a social context, and is a social process. Our studies of end-user configuration and automation of domestic technologies demonstrate the extent to which family relations and gender roles spill over into practices of end-user programming [3].

Design processes involve modeling – simplifying or abstracting some aspects of the problem domain in order to plan and evaluate design decisions. The use of representations to reason about future consequences is fundamental to end-user software engineering. The constraints that representations place on design activities are described by the cognitive dimensions of notations framework [4], and in turn by a great variety of research into visual representations.

Abstract reasoning about the future can be described in terms of the attention investment model [5]. A productive approach to end-user software engineering is to modify users' perception of this investment, whether by Burnett's Surprise-Explain Reward strategy, or by the use of machine learning techniques to infer possible abstractions that might be suggested to the user [6].

Finally, I am interested in the extent to which all designers experience their work as creative. This experience should be available to end-users too, not only creative professionals. In studies of choreographers and musicians, my students and I research and develop new notations and programming languages that offer artistic experiences to their users [7]

Many of these activities extend well beyond the bounds of software engineering, empowering users to control and enhance their computer tools in new ways. This was the same motivation that led to the innovations of the modern graphical user interface [8], and I believe that EUSE research might well transform the general purpose user interfaces of the future.

References and Further Reading

1. Blackwell, A.F. and Good, D.A. (in press). Languages of innovation. To appear in H. Crawford & L. Fellman (Eds.). Artistic Bedfellows: Collaborative History and Discourse. University Press of America.
2. Blackwell, A., Bucciarelli, L, Clarkson, P.J., Earl, C.F., Eckert, C., Knight, T., Macmillan, S., Stacey, M. and Whitney, D. (2005). Comparative study of design - application to engineering design. Presented at International Conference on Engineering Design.
3. Rode, J.A., Toye, E.F. and Blackwell, A.F. (2005). The domestic economy: A broader unit of analysis for end user programming. In proceedings CHI'05 (extended abstracts), pp. 1757-1760
4. Blackwell, A.F. and Green, T.R.G. (2003). Notational systems - the Cognitive Dimensions of Notations framework. In J.M. Carroll (Ed.) HCI Models, Theories and Frameworks: Toward a multidisciplinary science. San Francisco: Morgan Kaufmann, 103-134.
5. Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 2-10.
6. Blackwell, A.F. (2001). SWYN: A Visual Representation for Regular Expressions. In H. Lieberman (Ed.), Your wish is my command: Giving users the power to instruct their software. Morgan Kauffman , pp. 245-270.
7. Blackwell, A. and Collins, N. (2005). The programming language as a musical instrument. In Proceedings of PPIG 2005, pp. 120-130.
8. Blackwell, A.F. (2006). The reification of metaphor as a design tool. ACM Transactions on Computer-Human Interaction (TOCHI), 13(4), 490-530.

Position Paper for the Dagstuhl Seminar "End-User Software Engineering", February 2007

Meta-Design: A Conceptual Framework for End-User Software Engineering

Gerhard Fischer
University of Colorado, Center for LifeLong Learning and Design (L3D)
Department of Computer Science, 430 UCB
Boulder, CO 80309-0430 – USA
gerhard@colorado.edu

1. Summary of own most relevant work for EUSE

Meta-Design

In a world that is not predictable, improvisation, evolution, and innovation are more than a luxury: they are a necessity. The challenge of design is not a matter of getting rid of the emergent, but rather of including it and making it an opportunity for more creative and more adequate solutions to problems.

Meta-design is an emerging conceptual framework aimed at defining and creating social and technical infrastructures in which new forms of collaborative design can take place. It extends the traditional notion of system design beyond the original development of a system. It is grounded in the basic assumption that future uses and problems cannot be completely anticipated at design time, when a system is developed. Users, at use time, will discover mismatches between their needs and the support that an existing system can provide for them. These mismatches will lead to breakdowns that serve as potential sources of new insights, new knowledge, and new understanding.

Consumers and Designers

Cultures are substantially defined by their media and their tools for thinking, working, learning, and collaborating. A great amount of new media is designed to see humans only as consumers. The importance of meta-design rests on the fundamental belief that humans (not all of them, not at all times, not in all contexts) want to be and act as designers in personally meaningful activities. Meta-design encourages users to be actively engaged in generating creative extensions to the artifacts given to them and has the potential to break down the strict counterproductive barriers between consumers and designers.

Many computer users and designers today are domain professionals, competent practitioners, and discretionary users, and should not be considered as naïve users or "dummies." They worry about tasks, they are motivated to contribute and to create good products, they care about personal growth, and they want to have convivial tools that make them independent of "*high-tech scribes*" (whose role is defined by the fact that the world of computing is still too much separated into a population of elite scribes who can act as designers and a much larger population of intellectually disenfranchised computer phobes who are *forced* into consumer roles). The experience of having participated in the framing and solving of a problem or in the creation of an artifact makes a difference to those who are affected by the solution and therefore consider it personally meaningful and important.

A fundamental challenge for the next generation of computational media and new technologies is not to deliver predigested information to individuals, but to provide the opportunity and resources for social debate, discussion, and collaborative design. In many design activities, learning cannot be restricted to finding knowledge that is "out there." For most design problems (ranging from urban design to graphics design and software design, which we have studied over many years), the knowledge to understand, frame, and solve problems does not exist; rather, it is constructed and evolved during the process of solving these problems, exploiting the power of *"breakdowns"*. From this perspective, *access* to existing information and knowledge (often seen as the major advance of new media) is a very limiting concept.

Dagstuhl Seminar Proceedings 07081
End-User Software Engineering
http://drops.dagstuhl.de/opus/volltexte/2007/1087

Unself-conscious and Self-conscious Design Cultures

The theory of *unself-conscious* and *self-conscious* design cultures (C. Alexander) provides an initial analytical framework for gaining a systematic understanding of the fundamental difference between domain experts and software professionals.

Self-conscious design culture. Dictated by a self-conscious design culture, the major focuses of software engineering research are understanding, representing correctly, and satisfying what the users want; creating software systems that have high production values; and providing the development process that achieves the highest economic efficiency and that is repeatable. The distinct separation of users and developers is one of the most important tacit assumptions underlying software engineering research and many research problems framed under this assumption.

Unself-conscious design cultures. Domain experts who engage in software development activities are not interested in the system per se, but rather in the domain-specific tasks that have to be performed with the help of the system. For them, because they are not professional software developers, software systems are tools, and the introduction of new tools changes the tasks and practices, which in turn begets new needs for tools. This *co-evolution* of tools and tasks determines that a large class of software systems can never be completely delegated to external professional software developers, and can be developed only by those domain experts who own the problem and have both the inside knowledge of the application domain and software development skills.

2. Future Questions for EUSE

Understanding the Impacts of Meta-Design on Software Development

EUSE research should explore the following hypotheses / claims:

- **Hypothesis$_1$: Requirements are generated differently.** Because developers are users themselves, there is no need for an elaborate requirement analysis phase as a major activity preceding the construction of the software system. Rapid changes of requirements need not be avoided; quite to the contrary, they are desired because the computer in such contexts is used to explore the new possibilities and to find the "undreamed-of requirements"
- **Hypothesis$_2$: Software testing is conducted differently**. Because domain expert developers themselves are the primary users, complete testing is not as important as in the case when the developers are not the users.
- **Hypothesis$_3$: Collaboration takes place along different dimensions**. In self-conscious software development, a team of developers is often organized before the project starts — in unself-conscious software development, a predefined project team does not exist. Collaboration is spontaneous and opportunistic rather than planned.
- **Hypothesis$_4$: The path to the acquisition of knowledge and skill for software development is different**. Due to the lack of interest in software per se and the lack of professional training, domain experts are more likely to acquire software knowledge in a piecemeal fashion and demand-driven manner. Their knowledge is more fragmental than systematic.
- **Hypothesis$_5$: Software will evolve in a different style**. The system is evolved gradually by a large number of people who make small contributions each time. Evolution is more spontaneous and situational due to the co-adaptivity of tools and their users.

Trade-off between Standardization and Improvisation

Meta-design creates an inherent tension between standardization and improvisation. The SAP Info (July 2003, page 33) argues to reduce the number of customer modifications for the following reasons: *"every customer modification implies costs because it has to be maintained by the customer. Each time a support package is imported there is a risk that the customer modification my have to be adjusted or re-implemented. To reduce the costs of such on-going maintenance of customer-specific changes, one of the key targets during an upgrade should be to return to the SAP standard wherever this is possible"*. Finding the right balance between standardization (which can suppress innovation and creativity) and improvisation (which can lead to a Babel of different and incompatible versions) has been noted as a challenge in open source environments in which forking has often led developers in different directions.

From "Ease-of-Use" to "Low Threshold and High Ceiling"

"Ease-of-use" along with the "burden of learning something" are often used as arguments for why people will not engage in design. Building systems that support users to act as designers and not just as consumers is often less successful than the meta-designers have hoped for.

The end-user modifiability and end-user programming features themselves add often considerably more functionality to already very complex environments (such as high functionality applications and large software reuse libraries) — and our empirical analyses clearly show that not too many users of such systems are willing to engage in this additional learning effort.

Based on our work with user communities, it is obvious that serious working and learning do not have to be unpleasant — they can be empowering, engaging, and fun. Many times the problem is not that *programming is difficult, but that it is boring* (as we were told by an artist). Highly creative owners of problems struggle and learn tools that are useful to them, rather than believing in the alternative of "ease-of-use," which limits them to preprogrammed features.

Motivation and Rewards

What makes people, over time, become active contributors and designers and share their knowledge requires a new "design culture", involving a mindset change and principles of social capital accumulation. But before new social mindsets and expectations emerge, users' active participation comes as a function of simple motivational mechanisms and activities considered personally meaningful.

One focus of meta-design is the design of socio-technical environments in which interactive systems are embedded, and in which users are recognized and rewarded for their contributions and can accumulate social capital. *Social capital* is based on specific benefits that flow from the trust, reciprocity, information, and cooperation associated with social networks

Additional Topics (only enumerated here)

1. relationship between: end-user development, end-user software engineering, meta-design, web 2.0 approaches;
2. the relevance of EUSE as a contribution to a "science of design";
3. support for the "seeding/location/comprehension/modification/sharing (sLCMS)" model;
4. putting owners of problems in charge by redefining the roles of high-tech scribes;
5. relationship between EUSE and different design methodologies (e.g., professionally dominated design, user-centered design, participatory design, learner-centered design);
6. EUSE does not only require reflective practitioners but *reflective communities*.

3. References Relevant to EUSE

Fischer, G., & Giaccardi, E. (2006) "Meta-Design: A Framework for the Future of End User Development." In H. Lieberman, F. Paternò, & V. Wulf (Eds.), *End User Development — Empowering people to flexibly employ advanced information and communication technology,* Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 427-457.
http://l3d.cs.colorado.edu/~gerhard/papers/EUD-meta-design-online.pdf

Fischer, G., Giaccardi, E., Eden, H., Sugimoto, M., & Ye, Y. (2005) "Beyond Binary Choices: Integrating Individual and Social Creativity," International Journal of Human-Computer Studies (IJHCS) Special Issue on Computer Support for Creativity (E.A. Edmonds & L. Candy, Eds.), 63(4-5), pp. 482-512.
http://l3d.cs.colorado.edu/~gerhard/papers/ind-social-creativity-05.pdf

Fischer, G. (2005) "From Reflective Practitioners to Reflective Communities." In: Proceedings of the HCI International Conference (HCII), Las Vegas, July 2005, (published on CD).
http://l3d.cs.colorado.edu/~gerhard/papers/reflective-communities-hcii-2005.pdf

Fischer, G. (2005) "Computational Literacy and Fluency: Being Independent of High-Tech Scribes." In J. Engel, R. Vogel, & S. Wessolowski (Eds.), *Strukturieren - Modellieren - Kommunizieren. Leitbild mathematischer und informatischer Aktivitäten*, Franzbecker, Hildesheim, pp 217-230;
http://l3d.cs.colorado.edu/~gerhard/papers/hightechscribes-05.pdf

Ye, Y., & Fischer, G. (2005) "Reuse-Conducive Development Environments," *International Journal Automated Software Engineering, Kluwer Academic Publishers, Dordrecht, Netherlands*, 12(2), pp. 199-235
http://l3d.cs.colorado.edu/~gerhard/papers/J-ASE-final.pdf

Fischer, G. (2002): "Beyond 'Couch Potatoes': From Consumers to Designers and Active Contributors", in FirstMonday (Peer-Reviewed Journal on the Internet),
http://firstmonday.org/issues/issue7_12/fischer/

Position paper for the End-User Software Engineering Dagstuhl Workshop (Feb. 2007)

Meta-User Interfaces for Ambient Spaces: Can Model-Driven-Engineering Help?

Joëlle Coutaz
Université Joseph Fourier, Lab.Informatique de Grenoble (LIG)
385 rue de la Bibliothèque, BP 53, 38041 Grenoble Cedex 9, France
joelle.coutaz@imag.fr

PERSONAL WORK RELEVANT TO THE WORKSHOP

My goal is to develop concepts and techniques that allow users to control and understand the ambient interactive spaces in which they live. With ambient computing, we are shifting from the control (and understanding) of systems and applications confined to a single computer to that of a dynamic computational aura where the boundaries between the physical and the digital worlds are progressively disappearing, where everything is highly dynamic and adaptive.

As a result, the pre-packaged well-understood solutions provided by shells and desktops that allow end-users to control their computing environments are inadequate for a continuous moving universe. To address this problem, I propose the concept of *meta-UI*. In addition, user interfaces that used to be defined once for ever for a well-identified context of use, must evolve dynamically. In my research group, we are addressing this problem under the umbrella of *UI plasticity*. Our approach to UI plasticity brings together MDE (Model Driven Engineering) and SOA (Service Oriented Architecture) within a unified framework that covers both the development stage and the runtime phase of interactive systems.

META-UI

A meta-UI is a special kind of end-user development environment whose set of functions is necessary and sufficient to control and evaluate the state of an interactive ambient space. This set is *meta*- because it serves as an umbrella *beyond* the domain-dependent services that support human activities in this space. It is *UI*-oriented because its role is to allow users to control and evaluate the state of the ambient interactive space. By analogy, a meta-UI is to ambient computing what desktops and shells are to conventional workstations.

As shown in Fig. 1, a meta-UI is characterized by its *functional coverage* in terms of *services* such as object discovery and coupling, and *object types*. *Objects discovery* allows users (and the system) to be aware of the objects that can be coupled. By coupling objects, users (and the system) build new constructs whose components play a set of roles (or functions). In conventional computing, roles are generally predefined. In ambient computing, where serendipity is paramount, *assigning roles to objects* becomes crucial. For example, Bob and Jane meeting in a café use spoons and lumps of sugar to denote the streets and buildings of the city they are talking about. Bob couples a spoon with the table by laying it down on the table while uttering "this is Champs-Elysées". The system can then discover the presence of the spoon and assign it the role of interaction resource (phicon). By doing so, Bob has dynamically defined a *mixed-by-contruction object*.

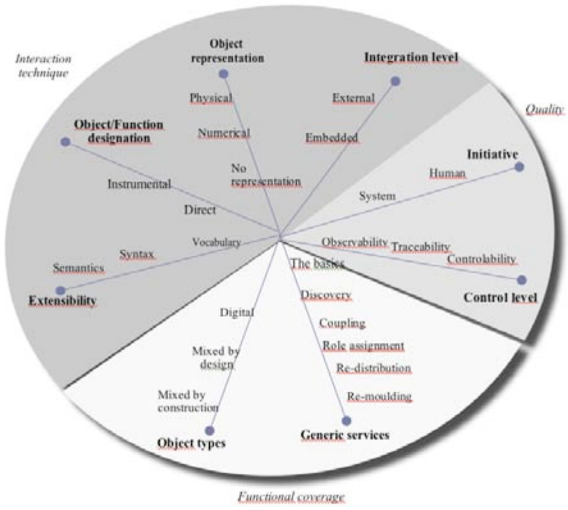


Fig. 1. A dimension space for meta-UI's.

UI re-distribution is another important generic service to be provided in ambient spaces. It denotes the re-allocation of UI elements of the interactive space to different interaction resources. For example, the GUI of a web site may dynamically switch from a centralized rendering on a PC screen to a distributed UI between a PDA and a wall-mounted display. In turn, UI re-distribution may require UI *re-moulding*, that is the capacity of the UI to reconfigure itself or to be reconfigured (under end-user's control) by suppressing, adding, and/or re-organizing UI elements.

Services and objects are invoked and referenced by the way of an *interaction technique* (i.e. a UI) that provides users with some *level of control* (observability only, traceability over time, and controllability or programmability). An interaction technique is a language (possibly *extensible*) characterized by the *representation* (vocabulary) used to denote objects and functions as well as by the way users construct sentences and assemble them into programs (including how they select/*designate* objects and functions).

Given the role of a meta-UI, the elements of the interaction technique of the meta-UI cohabit with the UI's of the domain-dependent services that it governs. The *integration level* expresses this relationship: all or parts of the UI elements of the meta-UI are *embedded* with (or weaved into) the UI components of the domain-dependent services. For example, Collapse-to-zoom uses the weaving approach. Alternatively, UI elements of the meta-UI services may be *external*, i.e. not mixed with the UI components of the domain-dependent services.

MDE and SOA

MDE aims at integrating different technological spaces using models, models transformations and mappings as key mechanisms. SOA defines the appropriate meta-model for a particular class of models: the runtime components. The flexibility offered by SOA fits our requirements for dynamic UI re-distribution and UI re-molding.

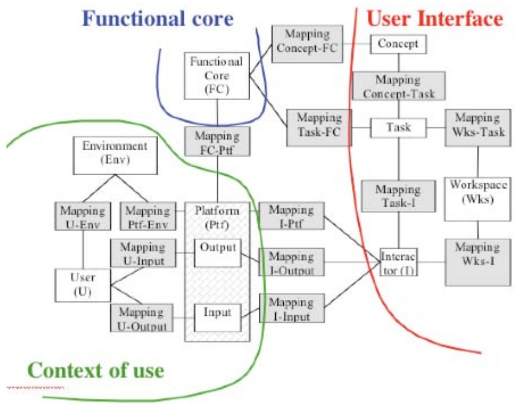


Fig. 2 An interactive system is a graph of models related by mappings and transformations.

As shown in Fig. 2, an interactive system is a graph of models that expresses and maintains multiple perspectives on the system. As opposed to previous work, an interactive system is not limited to a set of linked pieces of code. Models developed at design-time, which convey high-level design decision, are still available at runtime. A UI may include a task model, a concept model, an Abstract UI model (expressed in terms of workspaces), and a Concrete UI model (expressed in terms of interactors) all of them linked by mappings. Tasks and Concepts are mapped to entities of the Functional Core of the interactive system, whereas the Concrete UI interactors are mapped to I/O devices (interaction resources) of the platform. Mappings between interactors and I/O devices support the explicit expression of centralized versus distributed UIs.

Transformations and Mappings are models as well expressed in ATL (QVT could be an option as well). In the conventional model-driven approach to UI generation, transformation rules are diluted within the tool. Model transformers are encapsulated as services within a middleware infrastructure that includes services to support context awareness, UI re-moulding and UI re-distribution: The *situation synthesizer* computes the current situation from the information provided by observers. An *evolution engine* elaborates a reaction in response to the new situation. For example, "if a new PDA arrives, move the control panel to the PDA". The evolution engine identifies the components of the UI that must be replaced and/or suppressed and provides the configurator with a plan of actions. The *Configurator* executes the plan. If new components are needed, these are retrieved from the *storage space* by the *component manager*. Components of the storage space are described with conceptual graphs and retrieved with requests expressed with conceptual graphs. By exploiting component reflexivity, the configurator stops the execution of the "defectuous" components specified in the plan, gets their state, then suppresses or replaces them with the retrieved components and launches these components based on the saved state of the previous components. The components referred to in the action plan do not necessarily exist as executable code. They may instead be high-level descriptions such as task models. If so, the configurator relies on *models transformers* to produce executable code.

We are currently experimenting the flexibility provided by the interplay between modeling an interactive system as a graph of models, the existence of a meta-UI and of UI transformers encapsulated as OSGi services. In our example of a Home Control Heating System (HHCS), the user's task is to set the temperature of the rooms of the home. The meta-UI provides the end-user with access to the task and the platform models. For example, the platform model indicates that a PC HTML and a PC XUL are currently available in the home. By selecting a task of the task model then selecting the platform(s) on which the user would appreciate to perform the selected task, the UI is re-computed and redistributed on the fly.

ISSUES TO BE DISCUSSED

Programming (and debugging) ambient spaces is yet another challenge. Embracing this challenge as a whole may be too complex. Shall we study it based on a classification of ambient spaces (e.g., domestic, public, mobile settings, a day of "my" life, etc.). By extension, what is the problem space of EUSE? How does current approaches cover the problem space? And then, what is the solution space?

REFERENCES

In addition to the classics (A. Cypher, B. Myers, H. Lieberman, etc.), I would like to suggest the following ref. related to ambient spaces as well as to our own work on UI plasticity and meta-UI.

1. L. Balme, A. Demeure, N. Barralon, J. Coutaz, G. Calvary. CAMELEON-RT: a Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces. In Proc. *second European Symposium on Ambient Intelligence*, EUSAI 2004, LNCS 3295, Markopoulos et al. pp. 291-302

2. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, J. Vanderdonckt. A Unifying Reference Framework for Multi-Target user interfaces. *Interacting with Computers*, Special Issue on Computer-Aided Design of User Interface, 15(3), Elsevier Publ., June 2003, pp. 289-308.

3. Coutaz J. Meta-User Interface for Ambient Spaces, Invited talk. In proceedingd *TAMODIA'06*, Hasselt, Belgium, October 2006, Springer LNCS publ., p. 1-15.

4. Dey, A., Hamid, R., Beckmann, C., Li, Y., Hsu, D., a CAPpella: Programming by Demonstration of Context-Aware Applications, In *Proceedings of the ACM SIGCHI'04*, Vienne, 33-40.

5. Sohn, T.Y., Dey, A.K., iCAP: An Informal Tool for Interactive Prototyping of Context-Aware Applications. In *Proceedings of the International Conference on Pervasive Computing 2006*. Dublin, Ireland, May 2006, 974–975.

Model-Driven Development for End Users, too!?

Gregor Engels, University of Paderborn
engels@upb.de

Own work

Software Engineering is the discipline which develops and evaluates concepts, languages, methods and tools for professional software development in order to yield high-quality software systems. During the last decade, the paradigm of a model-driven development (MDD) has become prominent and is nowadays the accepted method for industrial software development. The main idea is to work with intermediary models in order to bridge the semantic gap between high-level, abstract user requirements and low-level, concrete programs and to support a stepwise refinement process.

This development is supported by agreeing on a standardized unified modelling language (UML). In order to cope with different application domains, the UML provides also standardized means (termed stereotypes) to customize the language, ending up with domain-specific (modelling) languages (DSL).

The UML is strong in modelling the internal functionality of a software system, but weak in modelling user interface aspects of a system. Thus, with respect to the well-known MVC (Model-View-Controller) pattern, the UML supports the Model aspect, while neglecting the View and Controller aspect.

While the model and functionality aspect is of high interest for a software developer, the view and model aspects are of particular interest for end users. Thus, any support for end users in customizing or even changing a software system should first concentrate on means to support the adaption of user interface aspects.

Within in our research on multimedia software systems, we extended and customized the standard UML towards a domain-specific modelling language for multimedia systems ([4], [5]). In particular, we introduced language concepts to define concrete layout aspects as well as complex interactive behaviour. In [1], we developed sophisticated tool support for end-users in order to customize multimedia user interfaces.

In a research cooperation with M. Erwig at Oregon State University, we investigated an approach to introduce an object-oriented model level for the development of spreadsheets [3]. We showed how such a model level prevents end users from hidden, hard to detect errors within spreadsheet applications.

Besides facilitating the development due to abstract, domain-related modelling concepts a model-based development provides the additional advantage of analysing model properties. For instance, in [2], we developed concepts based on patterns to prove the conformity of models with constraints as they might be expressed in given (ISO) standards.

Currently, several PhD students are working on the topic of “Model Quality”, in order to develop concepts and tools to understand, define and check the quality of any developed model.

Future Research Topics

While the model-driven development paradigm forms a well-accepted approach towards software development for professional software development, such an approach is in its infancy in end-user software development. It has to be investigated and evaluated, whether end-users at all are willing to develop high-level, abstract models instead of directly dealing with low-level programs. In addition, it has to be investigated what kind of modelling languages are appropriate for end-users. This has to be accompanied by setting up and evaluating case studies within real scenarios.

Furthermore, it has to be investigated whether and how standard model analysis techniques can be transferred to end user software development. In particular, appropriate explanation and help systems have to be designed which translate analysis results into a representation which is understandable by end users.

References

[1] S. Sauer, G. Engels: *Easy Model-Driven Development of Multimedia User Interfaces with GUIBuilder*. In Proc. 4th International Conference on Universal Access in Human-Computer Interaction (UAHCI 2007), July 2007,Bejing, LNCS, Springer, 2007 (to appear).
.
[2] A. Förster, T. Schattkowsky, G. Engels, R. Van Der Straeten: *A Pattern-driven Development Process for Quality Standard-conforming Business Process Models.* In Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Brighton 2006.

[3] G. Engels, M. Erwig: ClassSheets: *Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications*. In Proc. 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA, November 7-11, 2005, pp. 124-133.

[4] G. Engels, S. Sauer: *Object-oriented Modeling of Multimedia Applications*. In S.K. Chang (ed.), Handbook of Software Engineering and Knowledge Engineering, vol. 2, pp. 21-53, World Scientific, Singapore, 2002.

[5] S. Sauer, G. Engels: *UML-based Behavior Specification of Interactive Multimedia Applications.* In Proc. IEEE Symposia on Human-Centric Computing Languages and Environments (HCC'01), September 2001, Stresa, Italy, pp. 248-255. IEEE Computer Society Press.

Position Paper for Dagstuhl 2007 EUSE Workshop

Mary Beth Rosson
Computer-Supported Collaboration and Learning Lab
Center for Human-Computer Interaction
The Pennsylvania State University
mrosson@psu.edu

In this brief position paper, I summarize four strands of work underway in our Informal Learning in Software Development research group at Pennsylvania State University. The work contributes to end user software engineering by considering the needs and characteristics of end user web developers.

1 - Analysis of end users' needs with respect to web development

We have conducted survey and interview studies of nonprogrammers who have some experience with web development to assess current practices, problems, needs, and attitudes. Our results indicate that the informal web development population is quite diverse, with participants ranging in age from under 20 to over 55. In fact in an opportunity sample web survey, the largest age segment represented was over 55.

Self-reported web development expertise is correspondingly diverse, ranging from self-taught expert programmers to users who rely entirely on high-level application builders. However even the least experienced users report a need for relatively sophisticated web technologies, for example database interaction and user authentication. Attitudes and practices related to software engineering (e.g., attention to design and testing) are correlated with self-reported expertise, but vary in complex ways with other personal characteristics like curiosity and carefulness.

2 - Analysis of personal variables related to end users' development of web applications

Again by combining across survey results and lab studies, we are starting to see some patterns in personal variables associated with web development expertise and practices. Of particular interest has been gender as a factor. Quantitative analyses have been difficult because person variables tend to be highly inter-correlated and opportunity samples tend to have fewer women than men. However, using multiple regression techniques, we have found that gender is associated with some measures of web development expertise; other factors associated with expertise include the context in which the development is done, age, and the "carefulness" of a person's general working style. In a lab study, we also saw a small role of gender in predicting success with an experimental tool for end user web development, but cognitive abilities like visualization and logical reasoning were more influential as factors.

3 - Analysis of critical obstacles to end users' development of web applications

These findings also resulted from a combination of web surveys and interviews. In interviews with community webmasters, we found that many obstacles to providing working applications were socio-organizational in nature, for example the requirement to use a tool mandated by an organization but that causes problems for the developer. In general across both the interviews and surveys, we found that one of the most common problems in end users' web development activities was collecting, merging, and formatting content from other colleagues. At a technical level, some of the most irritating and frequent problems were also those that are most basic and amenable to tools – making sure that all the links are always working and getting layouts and format to look right and to work correctly on different browsers.

4 - Development and evaluation of a tool for end user web development

We have found that a significant proportion of end users' concepts for web applications can be satisfied with a tool that supports simple data-oriented applications (e.g., member directories, personal information or inventory management). CLICK (Click, a Lightweight Internet Construction Kit) is a prototype tool for supporting such development. It uses an interactive drag-and-drop user interface, with scaffolding provided by built-in wizards for common tasks (e.g., setting up a data table), dialog boxes that guide behavioral specification (e.g., prompting with currently available actions that can be connected to widgets), and a to-do list that monitors tasks that are still incomplete (e.g., creating a web page referenced in another part of the system). Usability evaluations have confirmed that sophisticated end users can learn to use CLICK enough to build a simple application in about one hour.

Future issues related to EUSE research

A significant issue for EUSE is related to end users' motivation to learn and use software engineering practices or techniques. People are active users; they do not want to stop what they are doing so as to evaluate their progress, make corrections, and do a 'better job'. The learning that they accomplish must come through *informal* means, for example goal-oriented help information, interactions with colleagues, or intelligent systems with just the right amount of initiative. As researchers we need to bear this in mind as we invent new techniques and methods: if we build it, will they come?

A related issue concerns the use of intelligent systems techniques. Intelligent systems can address users' minimal motivation by monitoring or correcting work. However, these systems are difficult to build with just the right amount of initiative—knowing when to jump in and with what level of assistance is critical, as too much help may be annoying or patronizing (as well as wrong) and may also decrease what the user is able to learn on his or her own.

One approach to helping users raise the quality of their problem analyses, designs and coding techniques is to support collaboration within a community of end users (e.g., within programming domains like web development). Although any individual user may not be willing to take the time to discover a solution or a useful tool, a community may be able to provide this support. However we still know very little about how end users may or may not wish to collaborate in their development activities, and more generally about how to build effective online community systems.

There is enormous diversity among end users who build software, particularly web software. As we build tools and training for EUSE, we must be careful to analyze the differential needs of varying user groups and create systems and tools that support a broad range of learning styles, motivation, and work contexts. An important societal concern lurking behind this programmatic suggestion is the digital divide—as end user development tools become more useful and available, the gaps and consequences of varying levels of computer literacy may become even more pronounced, with the consequence that some population segments become even more marginalized.

References of interest

Rode, J., Rosson, M.B., & Pérez-Quiñones, M.A. 2004. End-users' mental models of web engineering concepts. *Proceedings of Visual Languages and Human-Centric Computing 2004* (pp. 215-222). New York: IEEE.

Rode, J., Rosson, M.B., & Pérez-Quiñones, M.A. 2006. End user development of web applications. In Lieberman, H., Paterno, F., & Wulf, V. (Eds.), *End-User Development*. Kluwer/Springer.

Rosson, M.B., Ballin, J., & Nash, H. 2004. Everyday programming: Challenges and opportunities for informal web development. *Proceedings of Visual Languages and Human-Centric Computing 2004* (pp. 123-130). New York: IEEE.

Rosson, M.B., Ballin, J., & Rode, J. 2005. Who, what and why? A survey of informal and professional web developers. *Proceedings of Visual Languages and Human-Centric Computing 2005* (pp. 199-206). New York: IEEE.

Rosson, M.B., Ballin, J., Rode, J., & Toward, B. 2005. 'Designing for the Web' revisited: A survey of informal and experienced web developers. In *Proceedings of the International Conference on Web Engineering* (pp. 522-532). Kluwer/Springer.

Rosson, M.B. & Seals, C. 2001. Teachers as simulation programmers: Minimalist learning and reuse. *Proceedings of CHI 2001* (pp. 237-244). New York: ACM.

Rethinking the Software Life Cycle:
About the Interlace of Different Design and Development Activities

Position Paper for the Dagstuhl Seminar 07081
End User Software Engineering.

Yvonne Dittrich
IT University of Copenhagen
Software Development Group
Rued Langaardsvej 7
DK 2300 Copenhagen, Denmark
+45 7218 5177
ydi@itu.dk

Software engineering research addresses professional ways of designing, developing and implementing software. So far, software engineering more or less takes for granted that software professionals have control over the material implementation of a piece of software. Though users might use the software innovatively or even customise it, neither end-user tailoring (EUT) nor end-user development (EUD) are treated systematically regarding the impact of deferring part of the design to the use context on software development technologies or processes. Especially the development, adaptation and configuration of software products, software that is used by more than one user in more than one organisation – makes visible that different parallel ongoing development activities often distributed over more than two organisations have to be coordinated. To illustrate and develop related issues, let me first present three different projects I was and am involved in:

PD in the Wild

As part of a project focusing on 'shop floor IT management'[1], we addressed the interlacing of the user side integration and adaptation of software and flexible software processes as part of developing an adequate infrastructure for one-stop service provision. [1] The possibility to tailor system interfaces to allow for exchanging data between different programs and the adjustment the program a developing practice were central for the ongoing design-in-use.

The most appreciated software provider was a small company developing and maintaining a booking system mainly meant for sports facilities. Besides being configured to the specific facilities a municipality provided, the program contained a module which could be tailored to generate ASCII files transferring data e.g. for different economy systems or number-code based access control. The company implemented an agile development process in order to react quickly on bug reports and change proposals. For example the need for an extra field in the customer data to add a mobile number alongside with the landline could be implemented without delay. [2]

Designing for Change

The second project addresses the design and development of tailorable systems for changing business practices at a telecommunication provider.[2] The main results with respect to this article are that the design of flexible and tailorable systems does not only depend on the requirements from the use context but also the technical environment and the organisation of software development respectively the interaction of these domains. [3] Functional requirements for a tailoring interface might be traded against requirements for maintainability depending on whether the software is developed in-house and small maintenance tasks therefore are less problematic.

Tailoring and end-user development that expands beyond adaptation of applications to individual preferences and - like in this case - affects the model of the common work object implemented by the software requires additional features for designing, testing and debugging have to be provided for the end-user. [4] But even if that can be done, when the limits of the variability provided for the users is reached, software engineers have to take over in order to evolve the program. [5] And here again the connection between use and development plays an important role.

Design of Evolvable Software Productions[3]

The software that provides the cases for this recently started project are ERP systems and simulation software for hydrodynamic system. [6]

Before the software can be applied in a specific organisation or to simulate a specific river system, the software has to be configured, partly by providing data – about the organisation respectively the river system – and partly by configuring the software. In many cases even adaptation to individual preferences is possible. Most of the existing ERP systems can

[1] *DitA* Design of IT in Use – supportive technologies for public service provision, funded by VINNOVA 2000–2002 at Blekinge Institute of Technology.

[2] The research described here was first funded as a single project *Design in Use of Database Applications* (2000 – 2002). The cooperation then continued as part of *BESQ – Blekinge Engineering Software Qualities*, a profile project (2002-2008) funded by the same agency: The Knowledge Foundation (KKS) at Blekinge Institute of Technology.

[3] *ESP – Designing Evolvable Software Products*, funded the Danish government through NABIIT, a research council for nano-, bio- and information technologies from 2006-2008.

also be adapted through a programming interface. Adaptation are often done by independent software houses but can also be implemented by super-user in the end-user organisation.

The different activities of configuration, tailoring and adaptation often implemented using different technical solutions are all contributing to finalising design of the software. And – as could be expected – evolution takes place on all different levels, posing a number of technical and social coordination problems.

Based on the experience from the above-described projects a number of aspects of end-user development become visible as topics for research:

Software Engineering for EU developers

End-user tailoring and development is not only about adjusting personal performance support to individual preferences, but often needed to adjust the existing software to developing work and business processes. This requires to design, implement and test changes in coordination with other users or even with a whole organisation. There is some research on the organisational end-user development, but regarding tools and methods for user-developers more research is needed.

Cooperating across different development sites

There is neither *the* user nor *the* developer. Different members of the use organisation take on different responsibilities regarding the continuing development and adaptation of the software infrastructure. And on the software provider side, development of the base system, configuration and adaptation are often distributed between different organisations. How can the development in the different contexts be organised so that the parallel development processes in the other contexts can be taken into account?

This issue becomes visible when addressing the upgrade of software products. In different projects, we have observed periods of a few weeks, three to four times peer year, once a year and several years. What different ways to organise software development correspond to these different upgrade periods. What does that imply for the tailoring and EUD practices of the different products? How can experiences of tailors and EU developers inform the evolution of the base product?

Technical coordination of layered of development

Evolution takes place in parallel on the different layers of a software product. For the different layers, different technologies are used: part of the configuration might result in selection between precompiled alternatives. Other configuration might be saved as meta-data to be interpreted at run-time. The application itself can be changed through a programming language interface. The challenges of this situation become visible when the basic software is updated: base-data, production data, and configuration can be transformed semi-automatically but might have to be complemented depending on new functionality; adaptations often have to be reworked from scratch or perhaps can now be replaced through tailoring. Every software product provides examples of how to combine different technologies to provide different layers of adaptation and end-user development.

Product line architecture and variability management addresses some of the design issues but mainly in contexts where the deferred design is resolved within the development organisation. There is no systematic categorisation of different possibilities and their combination for multi-layered and end-user development.

Representations of Variability

How to represent complex configuration and adaptation possibilities respectively their constraints so that consultants and end-users can develop a qualified co-design of business organisation and software?

ACKNOWLEDGEMENTS

Thanks to my colleagues Sara Eriksén, Jeanette Eriksson, Christina Hansson and Olle Lindeberg from Blekinge Institute of Technology and Hataichanok Unphon, Peter Sestoft and Sebastien Vaucouleur from the IT University of Copenhagen who have been or are participating in the research projects, this position paper is based on.

REFERENCES

[1] Y. Dittrich, S. Eriksén, C. Hansson PD in the Wild: Evolving Practices of Design in Use. In: T. Binder, J. Gregory, I. Wagner *Proceedings of the PDC 2002*, Malmö, Sweden.

[2] C. Hansson, Y. Dittrich, D. Randall How to Include Users in the Development of Off-the-Shelf Software: A Case for Complementing Participatory Design with Agile Development. In: Proceedings of the HICSS 2006.

[3] Y. Dittrich, O. Lindeberg Designing for Changing Work and Business Practices. To be published in: N. Patel (ed.) *Evolutionary and Adaptive Information Systems,* Idea Group Publishing: 2002.

[4] J. Eriksson, Y. Dittrich Combining Tailoring and Evolutionary Software Development for Rapidly Changing Business Systems. Accepted for the Journal of Organizational and End-User Computing.

[5] Y. Dittrich, O. Lindeberg, L. Lundberg End- User Development as Adaptive Maintenance. Two Industrial Cases. In: V. Wulf, H. Lieberman, F. Paternó (eds.) *End User Development: Empowering people to flexibly employ advanced information and communication technology*, Springer 2006.

[6] ESP web address

Software environments for supporting End-User Development

Maria Francesca Costabile, Antonio Piccinno
Dipartimento di Informatica, Università degli Studi di Bari, via Orabona, 4,
Bari, 70125, Italy
costabile@di.uniba.it, piccinno@di.uniba.it

In the Information Society, end users keep increasing very fast in number, as well as in their demand with respect to the activities they would like to perform with computer environments, without being obliged to become computer specialists. There is a great request to provide end users with powerful and flexible environments, tailorable to the culture, skills and needs of a very diverse end user population. Moreover, the evolution in the culture of computing and the evolution of the roles of designers, programmers, and end users in the life cycle of software products lead to a new perspective in the development of software systems. Current work organizations require end users to tailor their software environments for better adapting them to their needs, and even to create or modify software artefacts. These are defined activities of End-User Development (EUD), to which a lot of attention is currently devoted by various researchers in Europe and all over in the world.

Our work in EUD has been carried out in the last few years together with Piero Mussio, of the University of Milan, also participating in EUD-Net (thematic network on EUD, funded by the UE). The work has provided various contributions to EUD, as summarized in the following.

We highlighted the needs of a community of users that is the most specific target audience for EUD, namely professionals in diverse areas outside of computer science, such as engineers, physicians, geologists and physicist, who are not professional programmers. This is described in papers that report experiences we collected by developing interactive systems used by such professional people, also called *domain experts* [1][2][3][4]. We identified two classes types of end user activities, as reported in [2][5], also mentioned by H. Lieberman, F. Paternò, M. Klann and V. Wulf in the first chapter of the Springer book “End-User Development”.

We developed a participatory design methodology, called SSW (Software Shaping Workshop) methodology, aimed at designing software environments that support end users to become co-designers of their tools. The SSW methodology emerges from the experiences gained in different application domains, and stresses the need for collaboration of different stakeholders, namely software engineers, HCI experts and domain experts.

The importance of considering user diversity is also considered. End users are very diverse because of their culture, education, skill, age, and training. In many domains, there are different communities of end users that need to collaborate to reach a common goal. Members of each community should need an appropriate software environment, suitable to them to manage their own view of the activity to be performed. This environment is called Software Shaping Workshop (SSW), since it is developed by exploiting the metaphor of the artisan workshop, where an artisan finds all and only the tools necessary to carry out her/his activities and properly shapes various materials (wood, iron, etc.) into usable products. In analogy, people should find in the SSWs all and only the tools to shape software artefacts. Such tools must be perceived and must behave so as to be usable in the current situation. To this aim, in the SSW methodology, representative of end users are required to participate in the design and implementation process as co-authors. Such representatives are involved in the design of the final workshops to be used by all the end users belonging to the specific sub-community to which they belong too. In this manner users have a twofold rule: users and designers of their own software environment.

This methodology, first presented in [1], has been refined in the following years [2][3][4][7]. It is a participatory, meta-design approach [Fischer G., Giaccardi E., Ye Y., Sutcliffe A. G., Mehandjiev N., Meta-design: a manifesto for end-user development, *Communications of the ACM*, Vol. 47(9), Sept. 2004, 33-37.] in which the different stakeholders can contribute their own views on the problem to design, development and maintenance of an application, using their own languages and notations.

Many types of hurdles are studied, which induce users to make errors and mistakes, and to break the continuity of their reasoning while carrying out a working task with the computer. As a consequence, negative emotional states, such as frustration, dissatisfaction, anxiety, may arise. The Software Shaping Workshop (SSW) methodology drives the development of interactive systems that are correctly perceived and interpreted by end users, thus becoming more acceptable and favouring positive emotional states [8].

We proposed a model of the Interaction and Co-Evolution processes (ICE model) occurring between users and system [6]. It extends the previous model of Human-Computer Interaction by considering an important phenomenon occurring during the use of interactive systems, called *co-evolution of users and systems* and based on the following two observations: 1) once people gain proficiency in system usage, they would like to use the system in different ways and need different interfaces than those they required when they were novice users (*user evolution*); 2) designers are then forced to evolve the system to meet the new users' needs (*system evolution*). The ICE model leads to re-examine the way interactive system are designed and forces a perspective of meta-design.

We consider a primary challenge for EUSE the creation of methodologies and tools which permit the creation of systems localized to end user culture and situation, so that end users may access knowledge sources, comprehend their content and perform their tasks without hurdles deriving from different cultures and traditions. These novel systems should also allow end users to tailor them at use time according to their needs and preferences. We believe that a meta-design approach must be stressed, which distinguishes two-phases: the first phase being designing the design environment that will be used by various experts (stakeholders) in the design team in order to design the specific applications; the second one being designing the applications using that design environment. The different stakeholders should be enabled to collaborate, also respecting their different viewpoints, both at design and use time.

In our view, meta-design is a process in which humans are able to act as designers of the system they use and to contribute to the co-evolution of such system. Meta-design must support humans in shaping their socio-technical environments and in adapting their tools to their needs.

[1] Costabile, M.F., Fogli, D., Fresta, G., Mussio, P., Piccinno, A., "Computer Environments for Improving End-User Accessibility", *Proc. of 7th ERCIM Workshop "User Interfaces For All"*, Paris, October 23-25, 2002, pp. 187-198.

[2] Costabile, M. F., Fogli, D., Fresta, G, Mussio, P., Piccinno, A., Building Environments for End-User Development and Tailoring, *Proc. 2003 IEEE HCC' 03*, Auckland, New Zealand, October 2003, pp. 31-38.

[3] Costabile, M. F., Fogli, D., Mussio, P., Piccinno, A., "End-User Development: the Software Shaping Workshop Approach". In (Lieberman, H.; Paternò, F.; Wulf, V. Eds.): End user development. Springer, Dordrecht, The Netherlands, 2006, pp. 183-205.

[4] Costabile, M. F., Fogli, D., Lanzilotti, R., Fresta, G, Mussio, P., Piccinno, A., "End-User Development Supporting Work Practice", *Journal of Organizational and End User Computing*, Volume 18, Number 4, 2006, pp. 43-65.

[5] Costabile, M.F., Fogli, D., Letondal, C., Mussio, P., Piccinno, A., "Domain-Expert Users and their Needs of Software Development", Special Session on EUD, UAHCI Conference, Crete, June 2003, pp. 532-536.

[6] Costabile, M.F., Fogli, D., Marcante, A., Piccinno, A., Supporting Interaction and Co-evolution of Users and Systems. Proc. AVI 2006, ACM Press, pp. 143-150.

[7] Costabile, M. F., Fogli, D., Mussio, P., Piccinno, A., "Visual Interactive Systems for End-User Development: a Model-based Design Methodology", *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, to appear.

[8] Fogli, D., Piccinno, A., "Environments to support context and emotion aware visual interaction", *Journal of Visual Languages and Computing*. Volume 16, pp. 386–405, 2005.

WHAT IS AN END-USER SOFTWARE ENGINEER?

STEVEN CLARKE, MICROSOFT CORPORATION

INTRODUCTION

I work in a multi-disciplinary team at Microsoft that is responsible for designing and building the user experience for users using the Visual Studio .Net suite of products. Visual Studio .Net is a large product suite, comprising a variety of software tools such as code profilers, debuggers, bug tracking tools, testing tools, code editors and language compilers. Multiple programming languages are supported, such as Visual Basic .Net, C# and C++.

Given the large variety of tools and languages that are supported by Visual Studio .Net, we are responsible for designing user experiences for a large variety of different users working in a large variety of different scenarios. For example, on one project we might be designing the user experience for building small web based applications while on another project we might be designing the user experience for a team of developers building a large distributed enterprise application. In both scenarios, the users that participate in the scenarios might differ in their work styles and characteristics just as much as the scenarios differ from each other.

To address the challenge of developing a shared understanding of the users that participate in each scenario we have developed a set of personas that describe the work styles, characteristics and motivations that are common to particular groups of people using our products. The personas help us communicate these characteristics by humanizing them, increasing the empathy that team members have for these fictional users.

There are a couple of things that are of particular interest about these personas that I would like to expand upon:

- We need more than one persona to adequately describe the different work styles, motivations and characteristics that we have observed of people using our products.
- We do not differentiate personas on expertise, experience or educational background.

MULTIPLE PERSONAS

We developed the personas by observing people using our products and noting the work styles, characteristics and motivations of each person. Over a period of approximately 12 months we observed people working in our usability labs and in their own workplaces, working in multiple scenarios. After this time, we were able to identify work styles, characteristics and motivations that were common across many of the observations that we had made. These formed the basis for the three personas that we defined.

We developed three different personas which describe the three sets of work styles, characteristics and motivations that we had observed. These personas are briefly described below:

THE SYSTEMATIC DEVELOPER

- Writes code defensively. Does everything they can to protect their code from unstable and untrustworthy processes running in parallel with their code.
- Develops a deep understanding of a technology before using it.
- Prides themselves on building elegant solutions.

THE PRAGMATIC DEVELOPER

- Writes code methodically.
- Develops a sufficient understanding of a technology to enable them to use it.
- Prides themselves on building robust applications.

THE OPPORTUNISTIC DEVELOPER

- Writes code in an exploratory fashion.
- Develops a sufficient understanding of a technology to understand how it can solve a business problem.
- Prides themselves on solving business problems.

We have been using these personas for four or five years now and have found them to be an invaluable resource in developing a shared understanding of who the user is when designing user experiences.

DIFFERENTIATE ON WORK STYLES, NOT EXPERTISE

One of the big challenges we've faced in spreading the word about these personas throughout Microsoft (and amongst our own customers) is correcting the assumption that the three personas describe developers with different levels of skill and educational backgrounds. We chose to represent work styles, motivations and characteristics as these are less liable to change over time as opposed to levels of expertise, educational background etc. Our observations have shown us that the work styles we described in the personas are shared by people with varying levels of expertise and educational background. It is not the case that someone starts out as an opportunistic developer then becomes a pragmatic developer after gaining a certain level of experience and expertise.

TRANSFER OF LEARNING

When developing and describing the personas we did not make a distinction between the job roles or titles of the people that we observed. Instead, we simply made observations of people who said that they used our products or other tools to develop software while at work. Many of these people did not describe themselves as software engineers. The variety of job titles that people used included 'Rocket Scientist', 'Surveyor', 'Customer support' as well as 'Software engineer', 'Software developer' etc. In addition we did not observe any relationship between job titles and work styles.

Given this, it is possible that one or more of the personas we developed would apply equally as well in discussions of end user software engineers. Identifying the commonalities between end user software engineers and so called professional software engineers would help enormously in identifying opportunities for transfer of learning between research focused on either community.

For example, Beckwith et al (2005) describe an investigation into the effect of gender on the effectiveness of end user debugging features and report that females were less willing to use new debugging features than males. In addition, females spent their time editing spreadsheet formulas as opposed to learning how to use the new debugging features. These results are similar to observations we make of opportunistic developers who focus on solving the business problem rather than learning how a particular feature works. The challenges are the same for both groups – how to encourage the use of tools that will help solve the business problem.

REFERENCES

Beckwith, L., Burnett, M., Wiedenbeck, S., Cook, C., Sorte, S., and Hastings, M.: Effectiveness of end-user debugging software features: are there gender issues? *ACM Conference on Human Factors in Computing Systems,* April 2005