

The State of the Art in End-User Software Engineering

ANDREW J. KO

The Information School, DUB Institute, University of Washington

ROBIN ABRAHAM

Microsoft Corporation

LAURA BECKWITH

<http://hciaresearcher.com>

ALAN BLACKWELL

The Computer Laboratory, University of Cambridge

MARGARET BURNETT, MARTIN ERWIG, AND CHRIS SCAFFIDI

School of Electrical and Engineering and Computer Science, Oregon State University

JOSEPH LAWRANCE

MIT CSAIL

HENRY LIEBERMAN

MIT Media Laboratory

BRAD MYERS

Human-Computer Interaction Institute, Carnegie Mellon University

MARY BETH ROSSON

Information Sciences and Technology, Penn State University

GREGG ROTHERMEL

Department of Computer Science and Engineering, University of Nebraska at Lincoln

MARY SHAW

Institute for Software Research, Carnegie Mellon University

SUSAN WIEDENBECK

College of Information Science and Technology, Drexel University

Most programs today are written not by professional software developers, but by people with expertise in other domains working towards goals for which they need computational support. For example, a teacher might write a grading spreadsheet to save time grading, or an interaction designer might use an interface builder to test some user interface design ideas. Although these end-user programmers may not have the same goals as professional developers, they do face many of the same software engineering challenges, including understanding their requirements, as well as making decisions about design, reuse, integration, testing, and debugging. This article summarizes and classifies research on these activities, defining the area of End-User Software Engineering (EUSE) and related terminology. The article then discusses empirical research about end-user software engineering activities and the technologies designed to support them. The article also addresses several crosscutting issues in the design of EUSE tools, including the roles of risk, reward, and domain complexity, and self-efficacy in the design of EUSE tools and the potential of educating users about software engineering principles.

Categories and Subject Descriptors: D.2 [**Software Engineering**], D.3 [**Programming Languages**], H.5 [**Information Interfaces and Presentation**], K.4 [**Computers and Society**], J.4 [**Social and Behavioral Sciences**]

General Terms: Reliability, Human Factors, Languages, Experimentation, Design

Additional Key Words and Phrases: end-user software engineering, end-user programming, end-user development, visual programming, human-computer interaction.

1. INTRODUCTION

From the first digital computer programs in the 1940's to today's rapidly growing software industry, computer programming has become a technical skill of millions. As this profession has grown, however, a second, perhaps more powerful trend has begun to take shape. According to statistics from the U.S. Bureau of Labor and Statistics, by 2012 in the United States there will be fewer than 3 million professional programmers, but more than 55 million people using spreadsheets and databases at work, many writing formulas and queries to support their job [Scaffidi et al. 2005]. There are also millions designing websites with Javascript, writing simulations in MATLAB [Gulley 2006], prototyping user interfaces in Flash [Myers et al. 2008], and using countless other platforms to support their work and hobbies. Computer programming, almost as much as computer use, is becoming a widespread, pervasive practice.

What makes these “end-user programmers” different from their professional counterparts is their *goals*: professionals are paid to ship and maintain software over time; end users, in contrast, write programs to support some goal in their own domains of expertise. End-user programmers might be secretaries, accountants, children [Petre and Blackwell 2007], teachers [Wiedenbeck 2005], interaction designers [Myers et al. 2008], scientists [Segal 2007] or anyone else who finds themselves writing programs to support their work or hobbies. Programming *experience* is an independent concern. For example, despite their considerable programming skills, many system administrators view programming as only a means to keeping a network and other services online [Barrett et al. 2004]. The same is true of many research scientists [Carver et al. 2007, Segal 2007].

Despite their differences in priorities from professional developers, end-user programmers face many of the same software engineering challenges. For example, they must choose which APIs, libraries, and functions to use [Ko et al. 2004]. Because their programs contain errors [Panko 1998], they test, verify and debug their programs. They also face critical consequences to failure. For example, a Texas oil firm lost millions of dollars in an acquisition deal through an error in a spreadsheet formula [Panko 1995]. The consequences are not just financial. Web applications created by small-business owners to promote their businesses do just the opposite if they contain bad links or pages that display incorrectly, resulting in loss of revenue and credibility [Rosson et al. 2005].

Software resources configured by end users to monitor non-critical medical conditions can cause unnecessary pain or discomfort for users who rely on them [Orrick 2006].

Because of these quality issues, researchers have begun to study end-user programming practices and invent new kinds of technologies that collaborate with end users to improve software quality. This research area is called *end-user software engineering* (EUSE). This topic is distinct from related topics in end-user development in its focus on software quality. For example, there have been prior surveys of *novice* programming environments [Kelleher and Pausch 2005], discussing systems that either help students acquire computing skills or enable the creation of computational artifacts; while quality is a concern in these contexts, this work focuses largely on learning goals. There have also been surveys on end-user *programming* [Sutcliffe and Mehandjiev 2004, Lieberman et al. 2006][Wulf et al. 2006], but these focus on the *construction* of programs to support other goals, but not on engineering activities peripheral to construction, such as requirements, specifications, reuse, testing, and debugging.

In this article, these software engineering activities are our primary focus. We start by proposing definitions of *programming*, *end-user programming*, and *end-user software engineering*, focusing on differences in intents and priorities between end-user programming and professional software development. We follow with a lifecycle-oriented treatment of end-user software engineering research, organizing more than a decade of research on incorporating requirements, design, testing, verification, and debugging into end users' existing work practices. We then discuss a variety of crosscutting issues in end-user software engineering research, including the role of risk, reward, and domain of practice on end users' decision-making, as well as strategies for persuading users to engage in more rigorous software engineering activities as part of their normal work. We also discuss individual factors, such as self-efficacy and gender, and their influence on how effectively people use EUSE tools.

What we found in our review of these research efforts were two different histories. First, *studies* of end-user software engineering concerns have had a consistently broad scope. Researchers have studied children [Petre and Blackwell 2007], middle-school students [Baker 2007, Kelleher et al. 2007], system administrators [Barrett et al. 2004], people at home [Blackwell 2004], knowledge workers in large companies [Bogart et al. 2008, Scaffidi et al. 2006], interaction designers [Ko et al. 2004, Brandt et al. 2008, Myers et al. 2008], natural scientists [Carver et al. 2007, Segal 2007], software architects [Lakshminarayanan et al. 2006], bioinformatics professionals [Letondal 2006], web de-

signers [Rode and Rosson 2003], and even volunteers helping with disaster relief [Scaffidi et al. 2007].

In contrast to studies of EUSE, contributions to EUSE tools have historically had a narrow scope. Early work focused largely on spreadsheets and event-based computing paradigms and on perfective aspects of end-user software engineering, such as testing, verification and debugging. Part of this historical bias is due to the fact that doing research on a particular paradigm has required mature and flexible programming languages, platforms, and IDEs on which to build more helpful software engineering tools, and most end-user programming platforms have not exhibited these properties. More recently however, this bias has been eliminated, with recent work focusing on a much broader set of domains and paradigms, including the web, mobile devices, personal information management, business processes, and programming in the home. Researchers have also extended their focus from perfective activities to design, including work on requirements, specifications, and reuse. Part of this shift is due to the advent of interactive web applications, as sharing code is becoming much more common.

These trends, coupled with the fact that computing is rapidly being incorporated into an incredible array of human activities, suggest that EUSE research will become similarly diverse. This will pose many challenges for the field, since the various domains studied and supported by research may have little in common. This is also an opportunity, however, for researchers to identify what *is* common across these diverse areas of practice. This article represents an effort at identifying some of these fundamental challenges, grounded in lessons from prior work.

2. DEFINITIONS

One contribution of this article is to identify existing terms in EUSE research and fill in terminology gaps, creating a well-defined vocabulary upon which to build future research. In this section, we start with a basic definition of programming and end with a definition of end-user software engineering.

2.1. Programming and Programs

We define *programming* similarly to modern English dictionaries, as *the process of planning or writing a program*. This leads to the need for a definition of the term *program*. Some definitions of “program” are in terms of the language, in which the program is written, requiring, for example, that the notation be Turing complete, and able to specify sequence, conditional logic and iteration. However, definitions such as these are heavily

influenced by the type of activity being automated. To remove these somewhat arbitrary constraints from the definition, for the purposes of this paper we define a program as *a collection of specifications that may take variable inputs, and that can be executed (or interpreted) by a device with computational capabilities*. Note that the variability of input values requires that the program has the ability to execute on future values, which is one way it is different from simply doing a computation once manually. This definition captures general purpose languages in wide use, such as Java and C, but also notations as simple as VCR programs, written to record a particular show when the time of day (input) satisfies the specified constraint, and combinations of HTML and CSS, which are interpreted to produce a specific visual rendering of shapes and text. It also captures the use of report generators, which take some abstract specification of the desired report and automatically create the finished report.

2.2. End-User Programming

We now turn to *end-user programming*, a phrase popularized by Nardi [1993] in her investigations into spreadsheet use in office workplaces. An *end user*¹ is simply any computer user. We then define *end-user programming* as *programming to achieve the result of a program primarily for personal, rather public use*. The important distinction here is that program itself is not primarily intended for use by a large number of users with varying needs. For example, a teacher may write a grades spreadsheet to track students' test scores, a photographer might write a Photoshop script to apply the same filters to a hundred photos, or a caretaker might write a script to help a person with cognitive disabilities be more independent [Carmien and Fischer 2008]. In these end-user programming situations, the program is a means to an end and only one of potentially many tools that could be used to accomplish a goal. This definition also includes a skilled software developer writing "helper" code to support some primary task. For example, a developer is engaging in end-user programming when writing code to visualize a data structure to help diagnose a bug. Here, the tool and its output are intended to support the developers' particular task, but not a broader group of users or use cases.

In contrast to end-user programming, *professional programming* has the goal of producing code for *others* to use. The intent might be to make money, or to write it for fun, or perhaps as a public service (as is the case for many free and open source projects).

¹ The "end" in "end user" comes from economics and business, where the person who purchases a software product may be different from the "end user" who uses it. Our use of the phrase in this article is more for historical consistency than because we need to make this distinction.

Therefore, the moment novice web designers move from designing a web page for themselves to designing a web page for someone else, the nature of their activity has changed. The same is true if the developer mentioned above decides to share the data structure visualization tool with the rest of his team. The moment this shift in intent occurs, the developer must plan and design for a broader range of possible uses, increasing the importance of design and testing, and the prevalence of potential bugs.

It is also important to clarify two aspects of this “intent”-based definition. First, our definition is not intended to be dichotomous, but continuous. After all, there is no clear distinction between a program intended for use by five people and a program intended for fifty. Instead, the key distinction is that as the number of intended uses of the program increases, a programmer will have to increasingly consider software engineering concerns in order to satisfy increasingly complex and diverse constraints. Second, even if a programmer does not *intend* for a program to be used by others, circumstances may change: the program may have broader value, and the code which was originally untested, hacked together, and full of unexercised bugs may suddenly require more rigorous software engineering attention.

While our definition of end-user programming is a departure from previously published definitions, we do so both to bring clarity to field and to discuss some of the underlying dimensions of historical use. For example, a number of connotations of the phrase have emerged in research, many using it to refer to “novice” programming or “non-professional” programming, or system design that involves the participation of end users. Many have also used it to describe an individual’s *identity* [Nardi 1993]. We believe these connotations conflate a number of related, but non-equivalent concepts.

Class of people	Activities of programming and tools and languages used
System administrators	Write scripts to glue systems together, using text editors and scripting languages
Interaction designers	Prototype user interfaces with tools like Visual Basic and Flash
Artists	Create interactive art with languages like Processing (http://processing.org)
Teachers	Teach science and math with spreadsheets [Niess et al. 2007]
Accountants	Tabulate and summarize financial data with spreadsheets
Actuaries	Calculate and assess risks using financial simulation tools like MATLAB
Architects	Model and design structures using FormZ and other 3D modelers
Children	Create animations and games with Alice [Dann et al. 2006] and Scratch
Middle school girls	Use Alice to tell stories [Kelleher and Pausch 2006, Kelleher and Pausch 2007]
Webmasters	Manage databases and websites using Access, FrontPage, HTML, Javascript
Health care workers	Write specifications to generate medical report forms
Scientists/engineers	Use MATLAB and Prograph [Cox et al. 1989] to perform tests and simulations
E-mail users	Write e-mail rules to manage, sort and filter e-mail
Video game players	Author “mods” for first person shooters, online multiplayer games, and The Sims
Musicians	Create digital music with synthesizers and musical dataflow languages
VCR and TiVo users	Record television programs in advance by specifying parameters and schedules
Home owners	Write control schedules for heating and lighting systems with X10
Apple OS X users	Automate workflow using AppleScript and Automator
Calculator users	Process and graph mathematical data with calculator scripting languages
Managers	Author and generate data-base backed reports with Crystal Reports

Table 1. A *partial* list of class of people who write programs and the kinds of programs they write.

For example, consider its use as an *identity*. A wide variety of people may engage in end-user programming; Table 1 gives just a glimpse of the diversity of people’s computational creations. While it is natural to use the phrase “end-user programmers” to describe these groups, it is not always accurate, because a persons’ *intent* in programming can depend on their task. For example, an accountant may use spreadsheets at home to keep track of a family loan, but use and share a spreadsheet at work to manage annual tax preparation activities with other accountants. It would be inaccurate to call this accountant an end-user programmer in all situations, when the spreadsheet at home is intended for personal, short-term use, but the one at work is heavily maintained and repeatedly used by multiple people. Thus, when we use the phrase “end-user programmer” in this paper, we are indicating the *intent* behind a programming task, not a fundamental aspect of the programmer’s identity.

It is also important to not conflate end-user programming with *inexperience*. Professional developers with decades of experience can engage in end-user programming by writing code for personal use, with no intent to share their program with others. They may complete their end-user programming task more quickly and with fewer bugs, but they will not approach the work with the same quality goals that they would for production code. To illustrate this distinction, consider Figure 1, which portrays programming *experience* and *intent* as two separate dimensions. Computer science students and professional programmers code with the intent of creating software for people to use (or grade), but vary in their experience. Similarly, end-user programming involves programming for personal use, but can include a wide range of programming expertise (of course, there are many more *inexperienced* programmers than experienced ones; we are not arguing that the distribution of experience is uniform).

Similarly, it is important to not conflate end-user programming with the use of “simple” languages. Experienced developers may use general purpose languages like C++ to achieve end-user programming goals, and in fact, many scientists do use such languages to do exploratory scientific analyses, without the intent of sharing the code or polishing it for future use [Segal 2007]. Similarly, experienced developers may use simple markup languages such as HTML and CSS to design commercial web sites. In general, end-user programming can involve a wide range of languages, from macro recording, to domain-specific languages, to conventional, general-purpose languages. The key distinction in the choice of language is whether it helps a person achieve their personal goal (e.g., “choosing the right tool for the job”).

Related to the choice of language, end-user programming should not be conflated

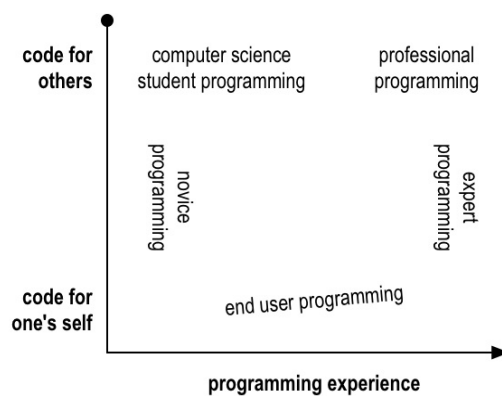


Figure 1. Programming activities along dimensions of *experience* and *intent*. The diagram is not intended to suggest the *distribution* of programmers (as there are many more without experience than with), but simply the underlying dimensions that characterize programming activity. The upward slant in *end-user programming* indicates that people with more experience tend to plan for other uses of their code.

with the use of particular *interaction technique* for constructing code. Java programs can be written with a text editor or a visual editor [Ko and Myers 2006] and languages that are traditionally written with visual editors, such as dataflow languages like Yahoo Pipes (<http://pipes.yahoo.com>) and Prograph [Matwin & Petrzykowski 1985], can also be expressed in textual syntax. The use of visual code editors has more to do with the difficulty of learning and manipulating textual syntax than the intent of the programmer.

There are a number of phrases related to “end-user programming” that are worth discussing relative to our definition. *End-user development* has been 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” [Lieberman et al. 2006]. This notion of end-user development also focuses on the use and adaptations of software over time, and focuses on elements of the software lifecycle beyond the stage of creating a new program. More specifically, *mutual-development*, *co-development*, and *participatory design* refer to activities in which end users are involved in a system design, but may or may not be involved in its actual coding [Henderson and Kyng 1991, Costabile et al. 2009, Mackay 1990]. These accounts of organizational, social, and collaborative perspectives on end-user development offer several valuable perspectives on how people appropriate and customize software, most of which are beyond the scope of this survey.

Terms such as *customization*, *configuring* [Eagan and Stasko 2008], and *tailoring* [Trigg and Bødker 1994, Kahler 2001] include parameterization of existing programs, but not direct modification of a program’s source code. *Visual programming* refers to a set of interaction techniques and visual notations for expressing programs. The phrase often implies use by end-user programmers, but visual notations are not always targeted at a particular type of programming practice. *Domain-specific languages* are programming languages designed for writing programs for a particular kind of context or practice. End-user programming may or may not involve such languages, since what defines end-user programming is the *intent*, not the choice of languages or tools. Finally, *scripts* and *scripting languages* are often distinguished from programs and programming languages by the use of machine interpretation rather than compilation and their “high-level” use in coordinating the functions of multiple programs or services. The phrase end-user programming, because it is often conflated with inexperience, often connotes the use of scripting languages since these languages have the reputation of being easier to learn.

Software Engineering Activity	Professional SE	End-user SE
Requirements	<i>explicit</i>	<i>implicit</i>
Specifications	<i>explicit</i>	<i>implicit</i>
Reuse	<i>planned</i>	<i>unplanned</i>
Testing and Verification	<i>cautious</i>	<i>overconfident</i>
Debugging	<i>systematic</i>	<i>opportunistic</i>

Table 2. Qualitative differences between professional and end-user software engineering.

2.3. End-User Software Engineering

With definitions of programming and end-user programming, we now turn to the central topic of this article, *end-user software engineering*. As we discussed in the previous section, the *intent* behind programming is what distinguishes end-user programming from other activities. This is because programmers’ intents determine to what extent they consider concerns such as reliability, reuse, and maintainability and the extent to which they engage in activities that reinforce these qualities, such as testing, verification, and debugging. Therefore, if one defines *software engineering* as systematic and disciplined activities that address software quality issues², the key difference between *professional software engineering* and *end-user software engineering* is the amount *attention* given to software quality concerns.

In professional software engineering, the amount of attention is much greater: if a program is intended for use by millions of users, all with varying concerns and unique contexts of use, a programmer must consider quality regularly and rigorously in order to succeed. This is perhaps why definitions of software engineering often imply *rigor*. For example, IEEE Standard 610.12 defines software engineering as “*the application of systematic, disciplined, quantifiable approaches to the development, operation, and maintenance of software.*” Systematicity, discipline, and quantification all require significant time and attention, so much so that professional software developers spend more time testing and maintaining code than developing it [Tassey 2002] and they often structure their teams, communication, and tools around performing these activities [Ko et al. 2007].

In contrast, *end-user software engineering* still involves *systematic and disciplined activities that address software quality issues*, but these activities are secondary to the goal that the program is helping to achieve. Because of this difference in priorities and

² Because the meaning of the phrase *software engineering* is still under much debate, we use the definition from current IEEE standards.

because of the opportunistic nature end-user programming [Brandt et al. 2008], people who are engaging in end-user programming rarely have the time or interest in systematic and disciplined software engineering activities. For example, Segal [2007] reported on several teams of scientists engaging in end-user programming, finding that software itself is not valued, that the process of creating software was highly iterative and unpredictable, and that testing was not considered important relative other domain-specific risks [Segal 2007]. These differences are coarsely summarized in Table 2, showing that end-user software engineering can be characterized by its unplanned, implicit, opportunistic nature, due primarily to the priorities and intents of the programmer (but perhaps also to inexperience).

Given these differences, the challenge of end-user software engineering research is to find ways to incorporate software engineering activities into users' *existing* workflow, without requiring people to substantially change the nature of their work or their priorities. For example, rather than expecting spreadsheet users to incorporate a testing phase into their programming efforts, tools can simplify the tracking of successful and failing inputs incrementally, providing feedback about software quality as the user edits the spreadsheet program. Approaches like these, and the ones reported throughout the rest of this article, allow users to stay focused on their primary goals (teaching children, recording a television, making scientific discoveries, etc.), while still achieving software quality.

It is important to note that we do not discuss the issue of *educating* users about software engineering practices in this article. Many of the techniques discussed in our review may have the side effect of teaching users about the importance of testing, for example, but this is not the primary goal of these techniques. There is a case to be made that anyone creating software with some potential for costly failure *ought* to engage in more rigorous and disciplined software engineering activities. This viewpoint and any research associated with it, is outside the scope of this article.

3. END-USER SOFTWARE ENGINEERING RESEARCH

End-user software engineering research is interdisciplinary, drawing from computer science, software engineering, human-computer interaction, education, psychology and other disciplines. Therefore, there are several dimensions along which we could discuss the literature in this area, including tools, language paradigm, research approach, and so on. However, because we aim to contrast EUSE with professional software engineering,

we chose to organize the literature by software engineering activities commonly listed in software engineering textbooks (e.g., [Ghezzi et al. 2002]). For each of these, however, we frame the discussion from the perspective of an end user:

1. *Requirements*. How the software should behave in the world.
2. *Design and specifications*. How the software behaves *internally* to achieve the requirements.
3. *Reuse*. Using preexisting code to save time and avoid errors (including integration, extension, and other perfective maintenance).
4. *Testing and verification*. Gaining confidence about correctness and identifying failures.
5. *Debugging*. Repairing known failures by locating and correcting errors.

In our discussion of each of these, we will review research in understanding and supporting these activities, and characterize the historical emphasis on particular paradigms or end-user programming domains. In doing so, however, we do not imply that these activities take place in sequence; indeed, the waterfall model [Ghezzi et al. 2002] is even less appropriate in end-user programming than it is in professional development.

It is important to note that we do *not* explicitly discuss the *implementation* and *use* of end-user programs, even though these activities are a central part of end-user programming activity. Surveys of *implementation* issues in end-user programming have been discussed extensively in previous literature [Sutcliffe and Mehandjiev 2004, Kelleher and Pausch 2005, Lieberman et al. 2006][Wulf et al. 2006]. The *use* of end-user programs depends largely on for what purpose they were created and most end-user software engineering research has attempted to be independent of purpose. We do, however, discuss the *maintenance* of end-user programs in our discussion of *sharing* in Section 3.3.

3.1. What Should My Program Do? — Requirements

The term “requirements” refers to statements of how a program should behave in the world (as opposed to the *internal* behavior of a program, which is how it achieves these external concerns). For example, a requirement for a tax program might be “Create a properly formatted 1040 tax form based on my financial data.” This is a statement of a desired result, but not of how the result is achieved.

In considering the history of work on this activity, the contributions have largely focused on understanding the sources and types of requirements of different domains of end-user programming and contrasting these with the role of requirements in professional software engineering.

For example, in professional software engineering, projects usually involve a requirements gathering phase that results in requirements specifications. These specifications can be helpful in anticipating project resource needs and for negotiating with clients. For end-user software engineering, however, the notion of requirements has to be reinvented. Because their motivations are not related to the software, but to some other goal, people engaging in end-user programming rarely have an interest in explicitly stating their requirements. This means they may be less likely to learn formal languages in which to express requirements or to follow structured development methodologies. Furthermore, in many cases, end users may not know the requirements at the outset of a project; the requirements may only become clear in the process of implementation [Costabile et al. 2006, Fischer and Giaccardi 2006, Mørch and Mehandjiev 2000, Segal 2007]. While this is also true in Agile development [Coplien and Harrison 2004], Agile developers explicitly recognize that requirements will evolve and have tools and processes that plan for emergent requirements. In contrast, people engaging in end-user programming are unlikely to plan in this way.

Another difference between requirements in professional and end-user software engineering is the *source* of the requirements. In professional settings, the customers and users are usually different people from the developers themselves. In these situations, requirements analysts use formal interviews and other methods [Beyer and Holtzblatt 1998] to arrive at the requirements. End-user programmers, on the other hand, are usually programming for themselves or for a friend or colleague. Therefore, end-user software engineering is unlike other forms of software engineering, where the challenge of requirements definition is to understand the context, needs and priorities of other people and organizations. For end users, requirements are both more easily understood (because the requirements are their own) and more likely to change (because end users may need to negotiate such changes only with themselves). Furthermore, end users' requirements are able to be implicit, and perhaps not even consciously recognized.

The situation in which an end user programs also affects the *type* of requirements. For example, at an office, the requirements are often to automate repetitive operations (such as transferring or transforming pieces of information such as customer names, products,

accounts, or documents). In this context, a decision to write a program at all corresponds directly to real investment since time is money. End users who become successful at automating their own work often find that their programs are passed on to others, whether by simple sharing of tools between peers [MacLean et al. 1990], or as a means for managers to define office procedures. These social contexts start to resemble the concerns of professional software developers, for whom requirements analysis extends to the definition and negotiation of work practices [Ko et al. 2007].

At home, end-user software engineering is seldom about efficiency (except in the case of office-like work that is done at home, such as taxes). Instead, typical tasks include automation of future actions, such as starting a cooker or recording television. It is often the case that one member of a household becomes expert in operating a particular appliance, and assumes responsibility for programming it [Rode et al. 2005, Blackwell 2004]. In this context, requirements are negotiated within the social relations of the household, in a manner that might have some resemblance to professional software experiences. Sometimes there are no requirements to start with; for example, there is a long tradition of “tinkering,” in which hobbyists explore ways to reconfigure and personalize technology with no definite end in mind [Blackwell 2006]. Even though these hobbyists might have envisioned some scenario of use when they made the purchase [Okada 2005], those motivations may be abandoned later. Instead, requirements evolve through experimentation, seeing what one can do, and perhaps motivated by the possibility of exhibiting the final product to others as a demonstration of skill and technical mastery.

In online contexts, end users must often consider the *users* of the web site or service they are creating [Rode et al. 2006], demonstrating that the distinction between the intent behind end-user programming and professional programming is a continuum rather than a mutually exclusive categorization. Further, in some situations, requirements are shared and negotiated, as happens with professional software developers. For example, Scaffidi et al. interviewed six Hurricane Katrina site developers and found that three relied on teammates for evaluating what features should be present and whether the site was viable at all [Scaffidi et al. 2006]. In this same study, requirements were derived from beliefs about the users of the program. One writer on the aggregators' email distribution list recognized that this “loosey goosey data entry strategy” would provide end users with maximal flexibility. Unfortunately, the lack of validation led to semantic errors that software propagated into the new database.

In educational contexts, programming is often used as a tool to educate students about mathematics and science. What makes these classroom situations unique is how requirements are delivered to and adapted by students. For example, Rosson et al. [2002] describe a participatory design workshop in which pairs of students and senior citizens created simulation projects to promote discussion about community issues. In this situation, requirements emerged from interpersonal communication in conversation and then were later constrained by the capabilities of the simulation tool. This contrasts with a classroom study of AgentSheets [Ioannidou et al. 2006], in which small groups of elementary school students followed a carefully designed curriculum to design biological simulations. In this situation, the instructions set the scope of the programming and students chose the detailed requirements within this scope. In other contexts [Niess 2007], the teachers and the students are end-user programmers. The degree to which the teachers understood the abilities and limitations of spreadsheets affected not only the requirements they developed in lab activities, but also the degree to which the students understood the abilities and limitations of spreadsheets.

In general, research has not attempted to explicitly support requirements capture, and the studies we have discussed should help reveal why. There are some techniques, however, that can be viewed as a form of requirements elicitation. For example, the Whyline [Ko and Myers 2004], which allows users to ask “why” questions about their program’s output, is an implicit way of learning about what behavior the user intended and did not intend. The same is true of the goal debugging work in the spreadsheet paradigm [Abraham and Erwig 2007b], which allows users to inquire about incorrect values in spreadsheet output. Both of these systems are a form of requirements elicitation, in which the requirements are used to support debugging.

3.2. How Should My Program Work? — Design and Design Specifications

In software engineering, *design specifications* specify the *internal* behavior of a system, whereas the requirements are *external* (in the world). In professional software engineering, software designers translate the ideas in the requirements into design specifications. These specifications can be helpful in coordinating implementation strategies and ensuring the right prioritization of software qualities such as performance and reliability. Design *processes* can ensure that all of the requirements have been accounted for.

In end-user programming, the challenge of translating one’s requirements into a working program can be daunting. For example, interview studies of people who wanted

to develop web applications revealed that people are capable of envisioning simple interactive applications, but cannot imagine how to translate their requirements into working applications [Rosson et al. 2005]. Further, in end-user software engineering, the benefits of explicit design processes and specifications may be unclear to users. Most of the benefits of being explicit come in the long term and at a large scale, whereas end users may not expect long-term usage of their programs, even though this is not particularly accurate. For example, studies of spreadsheets have shown that end users are creating more and more complex spreadsheets [Shaw 2004], with typical corporate spreadsheets doubling in size and formula content every three years [Whittaker 1999].

In general, research on incorporating specifications into end-user programming has been quite pragmatic. If systems have supported any form of specifications, they have been used (1) to support a particular kind of design process, such as prototyping or exploratory activities, (2) as the primary programming language, (3) or as an intermediate language that either makes it easier to generate correct programs or helps with program validation. Most of these technologies have focused on improving the creation and validation of spreadsheets, the prototyping of web sites, and the expression of preferences in the privacy domain. There is considerably less work on model- and specification-based approaches for interactive and web-based applications, though this is beginning to change. In the rest of this section, we review these approaches in light of the various imbalances and biases.

3.2.1. Design Processes. Software design *processes* constrain how requirements are translated into design specifications and then implementations. They also involve a careful consideration of tradeoffs among conflicting goals such as reliability, maintainability, performance and other software qualities. These processes are usually learned by professionals through experience or training. Many end-user programmers, however, are “silent designers” [Gorb and Dumas 1987], with no training in design and often seeing no benefit to ensuring such qualities.

Some have proposed dealing with this lack of design experience by enforcing particular design processes. For example, Ronen et al. propose a design process that focuses on ensuring that spreadsheets are reliable, auditable, and safe to update (without introducing errors) [Ronen et al. 1989]. Powell and Baker define strategies and best practices for spreadsheet design to improve the quality of created spreadsheets [Powell and Baker 2004]. Outside of the spreadsheet domain, Rosson et al. tested a design process with end-user web programmers based on scenarios and concept maps, finding that the process was

useful for orienting participants towards particular design solutions [Rosson et al. 2007]. One problem with dictating proper design practices is that end-user programmers often design alone, making it difficult to enforce such processes.

An alternative to enforcing good behavior is to let end users work in the way they are used to working, but inject good design decisions into their existing practices. One crucial difference between trained software engineers' and end users' approaches to problem solving is the extent to which they can anticipate design constraints on a solution. Software engineers can use their experience and knowledge of design patterns to predict conflicts and dependencies in their design decisions [Lakshminarayanan et al. 2006]. End-user programmers, however, often come to understand the constraints on their programs' implementations only in the process of writing their program [Fischer and Giaccardi 2006].

Because end-user programmers' designs tend to be emergent, like their requirements, requirements and design in end-user programming are rarely separate activities. This is reflected in most design approaches that have been targeted at end-user programmers, which largely aim to support evolutionary and exploratory prototyping, rather than up-front design. For example, DENIM, a sketching system for designing web sites, allows users to leave parts of the interface in a rough and ambiguous state [Newman et al. 2003]. This characteristic is called *provisionality* [Green et al. 2006], where elements of a design can be partially, and perhaps imprecisely stated.

Letondal 2006, Stevens et al. 2006, Won et al. 2006, Wulf et al. 2008]. As Pipek and Kahler point out, tailorability is a rich area, including not only issues of how to support low-level tailoring, but also numerous collaborative and social aspects [Pipek & Kahler 2006].

3.2.2. Writing Specifications. In professional software engineering, one way to ensure that requirements have been satisfied is to write explicit design specifications and then have tools check the program for inconsistencies with these specifications. In general, tools and languages for expressing specifications tend to be declarative in nature, allowing users to express what they want to happen, but not necessarily how.

In applying this idea to end-user software engineering, one approach is for a tool to require up-front design. For example, ViTSL separates the modeling and data-entry aspects of spreadsheet development [Erwig et al. 2005]. The spreadsheet model is captured as a template [Abraham et al. 2005] like the one in Figure 3. The ellipsis under row 3 indicates that the row can be repeated downwards; each row stores the scores of a student enrolled in the course. These templates can then be imported into a system called Gencil [Erwig et al. 2005, Erwig et al. 2006], which can be used to generate spreadsheets that are guaranteed to conform to the model represented by the template. For example, an instance of the template in Figure 3 is shown in Figure 4. The menu bar on the right allows the user to perform insertion and deletion, protecting the user against unintended changes. One limitation of this approach is that once a spreadsheet is generated from a template, edits to the generated spreadsheet cannot be propagated back to the template. (This same problem occurs in code generation systems in software engineering, where changes to the code are not reflected back to the specifications.)

Some systems are intended to support the exploration of specifications by supporting modeling for a particular type of application. For example, Berti et al. [2004] describe CTTE, a system that helps users convert natural language descriptions of tasks and scenarios into a hierarchy of subtasks. This is essentially a modeling language that helps users to express the design and underlying workflow of a user interface. In a similar system, Lin and Landay [2008] describe Damask, a system that allows designers to prototype ubiquitous computing applications and test them with users. In both of these systems, the modeling languages were carefully designed with a particular domain and class of applications in mind.

Most other systems that support specification writing are used for later verification and checking, rather than generating programs. For example, Topes [Scaffidi et al. 2008] allowing users to define string-based data types that can be used to check the validity of data and operations in any programming language that stores information as strings. Other researchers have developed end-user specification languages for privacy and security. For example, Dougherty et al. [2006] describe a framework for expressing access-control policies in terms of domain concepts. These specifications are stated as “differences” rather than as absolutes. For example, rather than stating who gets privileges in a declarative form, the system supports statements such as “after this change, students should not gain any new privileges.” Cranor et al. [2006] describe Privacy Bird, a related approach, which includes a specification language for users to express their privacy preferences in terms of the personal information being made accessible. Privacy Bird then uses these specifications to warn users about web sites’ violations of these preferences.

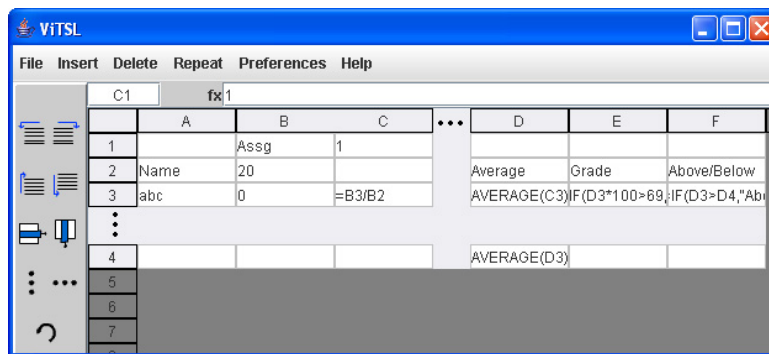


Figure 3. A ViTSL template, specifying the underlying structure of a grading spreadsheet. The names appear in rows and the assignments appear in columns, with the ellipses indicating repetition. Reproduced from [Abraham and Erwig 2006c] with permission from authors.

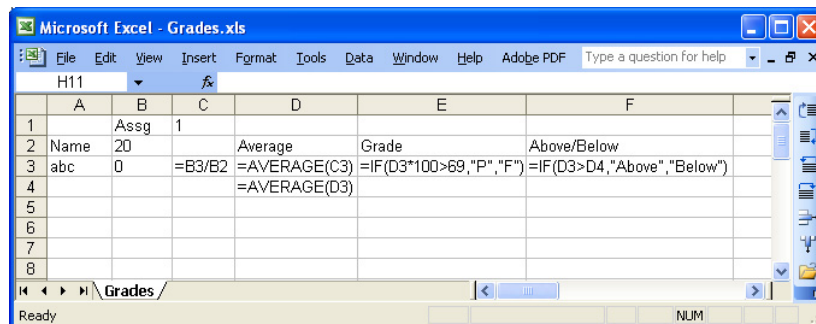


Figure 4. An instance of the grade sheet template from Figure 3 loaded into Excel. The operations in the toolbar on the right utilize the spreadsheet’s underlying structure to help users avoid introducing errors into the structure. Reproduced from [Abraham and Erwig 2006c] with permission from authors.

Finally, some approaches for writing specifications take a direct manipulation, what you see is what you get (WYSIWYG) approach, moving the description of behavior and appearance to the user's language, rather than a machine language. For example, many WYSIWYG web site tools allow users to directly manipulate the design of a web site and then let the tool generate the HTML and CSS for use in a web browser (most notably at the time of this writing is Adobe's Dreamweaver). To enable direct manipulation, such tools often limit the range of design possibilities to facilitate code generation, requiring users to learn the underlying language to express more complicated designs.

3.2.3. Inferring specifications. One approach to the problem of how to support a design process is to eliminate it, replacing it with technologies that can determine requirements automatically through various forms of inference.

Several systems have used a *programming by example* approach to this problem (such systems are described in detail in [Lieberman 2000]). These systems allow users to provide multiple examples of the program's intended behavior and the tool observes the behavior and attempts to generalize from it. For example, Abraham and Erwig developed an approach for automatically inferring the templates discussed in the previous section from an example spreadsheet [Abraham and Erwig 2006a], allowing users more flexibility in redefining the spreadsheet template as requirements change. In the domain of event-based simulations, the AgentSheets environment [Repenning and Perrone 2000] lets the programmer specify that a new type of "part" is just like an existing part, except for its icon; the tool will then generate all of the instructions necessary for the new part. McDaniel and Myers [1999] describe an approach to inferring interaction specifications, allowing users to click and drag objects from one part of the screen to another to demonstrate a desired movement at runtime.

Recent work has begun to apply programming by example to web sites. For example, Toomim et al. [2009] allow users to select example data from web sites and automatically generate a range of user interface enhancements. Nichols and Lau [2008] describe a similar system, which allows users to create a mobile version of a web site through a combination of navigating through the desired portion of the site and explicitly selecting content. Macias and Paterno [2008] take a similar approach, in which users directly modify the web page source code. These modifications are used as a specification of preferences, which are then generalized and applied to other pages on the same site. Yet another approach allows users to identify the same content with multiple markings, increasing the robustness of the inference [Lingham and Elbaum 2007].

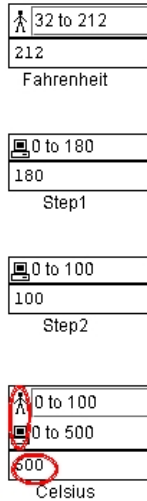


Figure 5. An assertion conflict in Forms/3. The user wrote the assertion on the Celsius cell (0 to 100), which conflicts with the computer generated assertion (0 to 500). This prompts the user to check for errors in the cells' formulas [Burnett et al. 2003]. Original figure obtained from authors.

One problem with programming by example approaches is that the specifications inferred are difficult to reuse in future programs, since most systems do not package the resulting program as a reusable component or a function. There are some exceptions to this, however. For example, Scaffidi et al. [2007] describe an approach to inferring data type specifications from unlabeled textual examples and then allowing users to review and customize the specification. The specification can then be easily packaged and reused for use in other applications. Another counter example is [Smith et al. 2000].

An alternative to programming by example is to elicit aspects of the specification directly from end users. Burnett et al. [2003] describe an approach for spreadsheets, attaching *assertions* to each cell to specify intended numerical values. In this approach, seen in Figure 5, users can specify an intended range of a cell's value at any time. Then, the system propagates these ranges through cell formulas, allowing the system to further reason about the correctness of the spreadsheet. If a conflict is found between a user-generated assertion and a system-generated assertion, the system circles the two assertions to indicate the conflict. This assertions-based approach has been shown to increase people's effectiveness at testing and debugging [Wilson et al. 2003, Burnett et al. 2003]. Scaffidi describes a similar approach for validating textual input [Scaffidi et al. 2008]; we describe this approach in Section 3.3.4.

Other approaches take natural language descriptions of requirements and attempt to translate them into code. For example, Liu and Lieberman [2005] describe a system that

takes descriptions of the intended behavior of a system and generates Python declarations of the objects and behaviors described in the descriptions. Little and Miller [2006] developed a similar approach for Chickenfoot [Bolin et al. 2005] (a web scripting language) and Microsoft Word’s Visual Basic for Applications. Their approach exploits the user’s familiarity with the vocabulary of the application domain to express commands in that domain. Users can state their goals in terms of the domain keywords that they are familiar with and the system generates the code. Other systems have attempted to teach commands to users when their effect may not be visible, as in the case of scriptable groupware applications [Wulf 2000].

3.3. What Can I Use to Write My Program? — Reuse

Reuse generally refers to either a form of *composition*, such as “gluing” together components APIs, or libraries, or *modification*, such as changing some existing code to suit a new context or problem. In professional programming, most of the code that developers write involves reuse of some sort, whether copying code snippets, adapting example code, or using libraries and frameworks by invoking their APIs [Bellon et al. 2007]. Traditionally, the motivations for these various types of reuse are usually to save time, to avoid the risk of writing erroneous new code, and to support maintainability [Ye and Fischer 2005, Ravichandran and Rothenberger 2003].

While these practices are also true in end-user programming, in many ways they are made more difficult by end users’ shift in priorities. In particular, finding, reusing, and even sharing code becomes more opportunistic, as the goals of reuse are more to save time and less to achieve other software qualities. Furthermore, in end-user programming, reuse is often what makes a project possible, since it may be easier for an end user to perform a task manually or not at all than to have to write it from scratch without other code to reuse [Blackwell 2002a].

Prior work on reuse has largely focused on studies of reuse in more conventional programming languages with large APIs or object hierarchies. Consequently, many of the challenges that professionals face, end users face as well. Where these populations differ is in how APIs, libraries, and frameworks must be designed to support a certain population. While APIs designed for professionals use often focus on optimizing flexibility, end users often need much more focused support for achieving their domain-specific goals. In this section, we characterize prior work on these different reuse activities and compare and contrast the role of reuse in end-user programming and professional development.

3.3.1. Finding Code to Reuse. A fundamental challenge to reuse is finding code and abstractions to reuse or knowing that they exist at all [Ye and Fischer 2005]. For example, Ko found that students using Visual Basic.NET to implement user interfaces struggled when trying to use search tools to find relevant APIs, and instead relied on their more experienced peers for finding example code or APIs [Ko et al. 2004]. This is similar to Nardi's finding that people often seek out slightly more experienced coworkers for programming help [Nardi 1993]. Example code and example programs are one of the greatest sources of help for discovering, understanding, and coordinating reusable abstractions, both in professional programming [Rosson and Carroll 1996, Stylos and Myers 2006] and end-user programming [Wiedenbeck 2005, Rosson et al. 2005]. In many cases the examples are fully functional, so the programmer can try out the examples and better understand how they work [Rosson and Carroll 1996, Walpole and Burnett 1997].

Researchers have also invented a number of tools to help search for both example code and APIs. For example, the CodeBroker system watches the programmer type code and guesses what API functions the programmer might benefit from knowing about [Ye and Fischer 2005]. Other systems also attempt to predict which abstractions will benefit a professional programmer [Mandelin et al. 2005, Bellettini et al. 1999]. Mica [Stylos and Myers 2006] lets users search using domain-specific keywords. While all of these approaches are targeted at experienced programmers, many of the same ideas are beginning to be applied to languages intended for end-user programming. For example, CoScripter's support for sharing and finding others' scripts not only helps users search for examples, but utilizes other meta data such as a users' social network to help users find relevant programs [Bogart et al. 2008].

3.3.2. Reusing Code. Even if end users are able to find reusable abstractions, in some cases, they may have difficulty using abstractions that were packaged, documented, and provided by an API. One study of students using Visual Basic.NET for user interface prototype showed that most difficulties relate to determining how to use abstractions correctly, coordinating the use of multiple abstractions, and understanding why abstractions produced certain output [Ko et al. 2004]. In fact, most of the errors that the students made had more to do with the programming environment and API, and not the programming language. For example, many students had difficulty knowing how to pass data from one window to another programmatically, and they encountered null pointer exceptions and other inappropriate program behavior. These errors were due primarily to choosing the

wrong API construct or violating usage rules in the coordination of multiple API constructs. Studies of end-user programming in other domains, such as web programming [Rode et al. 2003, Rosson et al. 2004], and numerical programming [Nkwocha and Elbaum 2005], have documented similar types of API usage errors.

There are several ways of addressing mismatch between code and the desired functionality. In the case of code, one way is to simply modify the code itself, customizing it for a particular purpose. A special case of adapting such examples is the concept of a *template*. For example, Lin and Landay [2008], in their tool for prototyping user interfaces across multiple devices, provide a collection of design pattern examples [Beck 2007] that users can adapt, parameterize, and combine. Some end-user development platforms, such as Adobe Flash, implement user interface components as modifiable templates. Templates have also been used to facilitate the creation of scripts for mobile devices to support the independencies of people with cognitive disabilities [Carmien and Fischer 2008].

The mismatch between code and desired functionality can sometimes be addressed through *tailoring* by the end user. In this case, the user supplies parameters, rules, or even more complex information to the component or application, thereby changing its behavior. Tailoring enables component developers to offload some adaptive maintenance activities onto end users [Dittrich et al. 2006], who essentially become asynchronous collaborators with the component developers [Mørch and Mehandjiev 2000]. In order for a component to be tailorable, the component designer must engage in significant upfront planning; in particular, the designer must consider how users in the target population will differ from one another, then determine which aspects of the component should accordingly be tailorable [Dittrich et al. 2006]. There are several ways to uncover user needs, such as including users in the design and construction of the component [Letondal 2006], performing ethnographies [Stevens et al. 2006], or interviewing users about their likely requirements [Eagan and Stasko 2008]. Such approaches assume that the users involved in component design can represent the diverse needs of the target population. Since users often vary not only in their requirements, but also in their level of tailoring skill, some component designers provide multiple mechanisms for tailoring, so that users with more skill can take advantage of more complex mechanisms in order to effect more specialized tailoring [Eagan and Stasko 2008, Mørch and Mehandjiev 2000, Wulf 1999, Wulf et al. 2008]. Tailorability can be greatly facilitated by integrated support for collaborated tailoring [Pipek and Kahler 2006], integrity checks for detecting tailoring mistakes [Won et al.

2006, Wulf et al. 2008], and evaluation features for helping users to understand the impact of tailoring [Won et al. 2006, Wulf et al. 2008].

Actually modifying the source code of APIs, libraries and other kinds of abstractions is generally not possible, since the code for the abstraction itself is not usually available. The programmer can sometimes write additional code to adapt an API [DeLine 1999], but there are certain characteristics of APIs and libraries, such as performance, that are difficult to adapt. Worse yet, when an end-user programmer is trying to decide whether some API or library would be suitable for a task, it is difficult to know in advance whether one will encounter such difficulties (this is also true for professionals [Garlan et al. 1995]).

Another issue for API and abstraction use is whether future versions of the abstraction will introduce mismatch because of changes to the implementation of the API behavior. For example, ActionScript [DeHaan 2006] (the programming language for Adobe Flash) and spreadsheet engine upgrades often change the semantics of existing programs' behavior. In the world of professional programming, one popular approach to detecting such changes is to create regression tests [Onoma et al. 1988]. Another possibility is to proxy all interactions with the API and log the API's responses; then, if future versions of the API respond differently, the system can show an alert [Rakic and Medvidovic 2001]. Regression testing has been used in relation to spreadsheets [Fisher et al. 2002b]; beyond this, these approaches have not been pursued in end-user development environments.

3.3.3. Creating and Sharing Reusable Code. Thus far, we have discussed reusing existing code, but most end-user programming environments also provide ways for users to create new abstractions. Table 3 lists several examples of reusable abstractions, distinguishing between behavioral abstractions and data abstractions. Studies of certain classes of end users suggest that data abstractions are the most commonly created type [Rosson et al. 2005, Scaffidi et al. 2006].

Environment	Domain	Behavioral abstractions	Data abstractions
AutoHAN [Blackwell and Hague 2001]	Home automation	Channel Cubes can map to scripts that call functions on appliances.	Aggregate Cubes can represent a collection of other Media Cubes.
BOOMS [Balaban et al. 2002]	Music editing	Functions record series of music edits.	Structures contain notes and phrases.
Forms/3 [Burnett et al. 2001]	Spread-sheets	Forms simultaneously represent a function and an activation record.	Types are structured collections of cells and graphical objects.
Gamut [McDaniel and Myers 1999]	Game design	Behaviors are learned from positive and negative examples.	Decks of cards serve as graphical containers with properties.
Janus [Fischer and Girgensohn 1990]	Floor plan design	Critic rules encode algorithms for deciding if a floor plan is “good.”	Instances of classes may possess attributes and sub-objects.
KidSim [Smith et al. 1994]	Simulation design	Graphical rewrite rules describe agent behavior.	Agents possess properties and are cloned for new instances.
Lapis [Miller and Myers 2002]	Text editing	Scripts automate a series of edits.	Text patterns can contain sub-structure.
Pursuit [Modugno and Myers 1994]	File management	Scripts automate a series of manipulations.	Filter sets contain files and folders.
QUICK [Douglas et al. 1990]	UI design	Actions may be associated with objects (that are then cloned).	Objects may have attributes and be cloned and/or aggregated.
TEXAO [Textier and Guittet 1999]	CAD	Formulas may drive values of attributes on cloneable objects.	Instances of classes may possess attributes and sub-objects.

Table 3. Behavioral and data abstractions in some end-user programming environments.

Although end users have the option of creating such reusable abstractions, examples are the more common form of reusable code. Unfortunately, it is extremely time-consuming to maintain a well-vetted repository of code. For example, Gulley [2006] describes the challenges in maintaining a repository of user-contributed Matlab examples, with a rating system and other social networking features. For this reason, many organizations do not explicitly invest in creating such repositories. In such cases, programmers cannot rely on search tools but must instead share code informally [Segal 2007, Wiedenbeck 2005]. This spreads repository management across many individuals, who share the work of vetting and explaining code.

Although it is common for end-user programmers to view the code they create as “throw away,” in many cases such code becomes quite long-lived [Mackay 1990]. Someone might write a simple script to streamline some business process and then later, someone might reuse the script for some other purpose. This form of “accidental” sharing is one way that end-user programmers face the same kinds of maintainability concerns as professional programmers. In field studies of CoScripter [Leshed et al. 2008, Bogart et al. 2008], an end-user development tool for automating and sharing “how-to” knowledge, scripts in the CoScripter repository were regularly copied and duplicated as starting points for new scripts, even when the original author never intended such use [Bogart et

al. 2008]. This unplanned sharing also means that improvements to the originally copied or shared code do not propagate to the copies that need it. This distinction between planned and unplanned reuse can demand different reuse technologies. For example, many professional development tools that support copying, such as the “linked editing” technology described by Toomim et al. [2004], require users to plan their copying activities.

3.3.4. Designing Reusable Code for End Users. One way to facilitate reuse by end users is to choose the right abstractions for their problem domains. This means choosing the right concepts and choosing the right level of abstraction for such concepts. For example, the designers of the Alice 3D programming system [Dann et al. 2006] consciously designed their APIs to provide abstractions that more closely matched peoples’ expectations about cameras, perspectives, and object movement. The designers of the Visual Basic.NET APIs based their API designs on a thorough study of the common programming tasks of a variety of programmer populations [Green et al. 2006]. The Google Maps API is related in that the key to its success has been the relative ease with which users can annotate geographical images with custom data types.

In other cases, choosing the right abstractions for a problem domain involves understanding the *data* used in the domain, rather than the behavior. For example, *Topes* [Scaffidi et al. 2008] is a framework for describing string data types unique to an organization, such as room numbers, purchase order IDs, and phone number extensions (see Figure 6). By supporting the design of these custom data types, end-user programmers can more easily process and validate information, as well as transform information between different formats. This is a fundamental problem in many new domains of end-user programming, such as “mashup” design tools [Wong and Hong 2007] and RSS feed processors (e.g., <http://pipes.yahoo.com>).

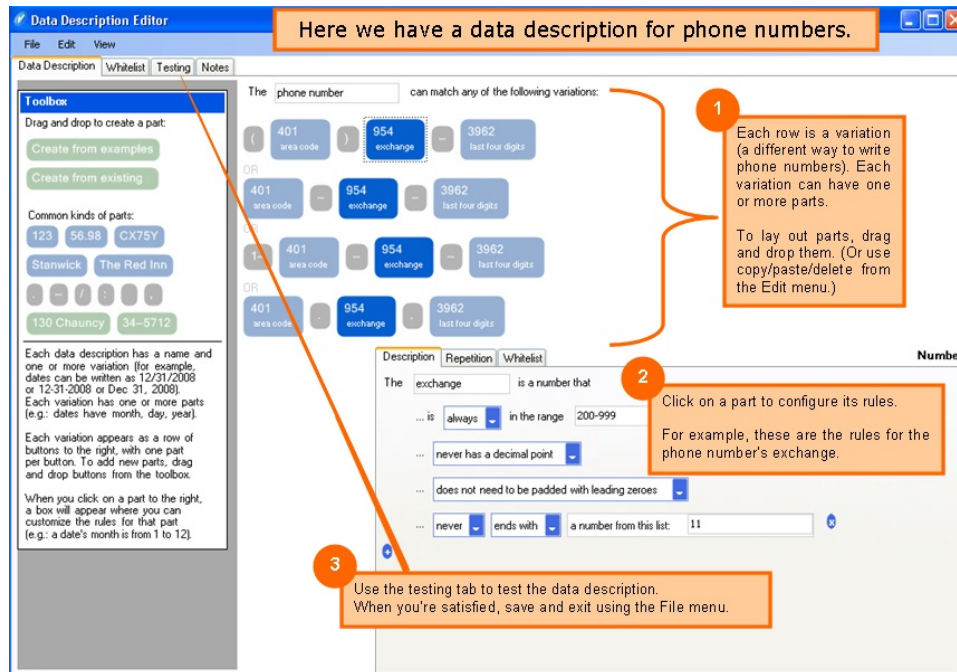


Figure 6. The Toped++ pattern editor [Scaffidi et al. 2008], allowing the creation of string data types that support recognition of matching strings and transformation between formats. Original figure obtained from authors.

Of course, as with any design, designing the right abstractions has tradeoffs. Specializing abstractions can result in a mismatch between the functionality of a reusable abstraction and the functionality needed by a programmer [Ye and Fischer 2005, Wiedenbeck 2005]. For example, many functional mismatches occur because specialized abstractions often have non-local effects on the state of a program [Biggerstaff and Richter 1989]. In addition to functional mismatch, non-functional issues can cause abstractions not to mesh well with the new program [Ravichandran and Rothenberger 2003, Shaw 1995]. End-user software engineering research is only beginning to consider this space of API and library design issues.

3.4. Is My Program Working Correctly? — Verification and Testing

There is a large range of ways to gain confidence about the correctness of a program, including through verification, testing, or a number of other approaches. The goals of testing and verification techniques are universal: they enable people to have a more objective and accurate level of confidence than they would if they were left unassisted. Where EUSE and professional SE differ is that end-user programmers' priorities often lead to overconfidence in the correctness of their programs.

Research on testing and verification in end-user programming has primarily focused on helping end users manage their overconfidence, and primarily for the spreadsheet paradigm³. More recent work has broadened support for testing and verification to the web and researchers are also beginning to generalize the spreadsheet-focused technologies to other paradigms and domains. In this section, we discuss these various contributions and organize research by the different approaches to helping end-user programmers overcome overconfidence.

3.4.1. Oracles and Overconfidence. A central issue for any type of verification is the decision about whether a particular program behavior or output is correct. The source of such knowledge is usually referred to as an *oracle*. Oracles might be people, making more or less formal decisions about the correctness of program behavior, or oracles can be explicitly documented definitions of the correct and intended behavior.

People are typically imperfect oracles. Professional programmers are known to be overconfident [Leventhal et al. 1994, Teasley and Leventhal 1994, Lawrance et al. 2005], but such overconfidence subsides as they gain experience [Ko et al. 2007]. Some end-user programmers, in comparison, are notoriously overconfident: many studies about spreadsheets report that despite the high error rates in spreadsheets, spreadsheet developers are heedlessly confident about correctness [Panko 1998, Panko 2000, Hendry and Green 1994]. In one study, overconfidence about the correctness of spreadsheet cell values was associated with a high degree of overconfidence about the spreadsheets' *overall* correctness [Wilcox 1997]. In fact, for spreadsheets, studies report that between 5% and 23% of the value judgments made by end-user programmers are incorrect [Ruthruff et al. 2005a, Ruthruff et al. 2005b, Phalgune et al. 2005]. In all of these studies, people were much more likely to judge an incorrect value to be right than a correct value to be wrong.

These findings have implications for creators of error detection tools. The first is that immediate feedback about the values a program computes, *without* feedback about correctness, leads to significantly higher overconfidence [Rothermel et al. 2000, Krishna et al. 2001]. Second, because end users' negative judgments are more likely to be correct than positive judgments, a tool should “trust” negative judgments more. One possible strategy for doing so is to implement a “robustness” feature that guards against a large

³ One possible explanation for this bias is that Microsoft Excel, the most widely used spreadsheet language, tends to produce output even in the presence of errors. This then leads to user overconfidence in program correctness, which researchers have tried to remedy through better testing tools. Furthermore, much of the original research on end-user software engineering was inspired by Nardi's investigation of spreadsheet use in business contexts [1992].

number of positive judgments swamping a small number of negative judgments, e.g., as in [Ruthruff et al. 2005b]. This approach was empirically tested in [Phalgune et al. 2005] and was found to significantly improve the tool’s feedback.

3.4.2. Detecting Errors with Testing. One approach to helping end-user programmers detect errors is supporting testing. Testing is judging the correctness of programs based on the correctness of program outputs. Systematic testing—testing according to a plan that defines exactly what tests are needed and when enough testing has been done—is crucial for success. Without it, the likelihood of missing important errors increases [Rothermel et al. 2001]. Furthermore, stronger (and more expensive) systematic testing techniques have a demonstrated tendency to outperform weaker ones [Frankl and Weiss 1993, Hutchins et al. 1994]. Unfortunately, being systematic is often in conflict with end-user programmers’ goals, because it requires time on activities that they usually perceive as irrelevant to success. Therefore, research on testing tools for end-user programmers has focused on testing approaches that are integrated with users’ work and are incremental in their feedback.

The most notable of these approaches is the “What You See Is What You Test” (WYSIWYT) methodology for doing “white box” testing of spreadsheets [Rothermel et al. 1998, Rothermel et al. 2001, Burnett et al. 2002]. With white box testing, the code is available to the tester [Beizer 1990]; in the case of spreadsheets, the formulas are the source code. Since testing most programs would require an infinite number of test cases in order to actually prove correctness, most white box approaches include a *test adequacy criterion*, which measures when “enough” testing has been done according to some code-based measure. Some criteria include *branch coverage* (test cases that exercise every branch), and *statement coverage* (exercising every statement in an imperative program) [White 1987]. With WYSIWYT, the criterion used is *definition-use coverage*, which (in the spreadsheet context) involves exercising every data dependency that could feasibly execute [Rothermel et al. 1998, Rothermel et al. 2001].

With WYSIWYT, as users develop a spreadsheet, they can also test that spreadsheet incrementally and systematically. At any point in the process of developing the spreadsheet, the user can validate any value that he or she believes is correct (the issues of oracles and overconfidence aside). Behind the scenes, these validations are used to measure the quality of testing in terms of a test adequacy criterion based on data dependencies. These measurements are then projected to the user using several different visual devices, to help them direct their testing activities.

Student Grades							
	NAME	ID	HWAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4	? B ?
2	Farnes, Joan	7,649	92	94	92	92.6	? A ?
3	Green, Matt	2,314	78	80	75	77.4	? C ?
4	Smith, Scott	2,316	84	90	86	86.6	? B ?
5	Thomas, Sue	9,857	89	89	89	93.45	? A ?
6							
7	AVERAGE		86.4	88.8	85.6	87.69	? ?

Figure 7. The WYSIWYT testing approach. Checkmarks represent decisions about correct values. Empty boxes indicate that a value has not been validated under the current inputs. Question marks indicate that validating the cell would increase testedness [Burnett et al. 2002]. Original figure obtained from authors.

For example, suppose that a teacher is creating a student grades spreadsheet and has reached the point shown in Figure 7. During this process, whenever the teacher notices that a value in a cell is correct, she can check it off in the decision box in the upper right corner of that cell. A checkmark appears in the decision box, indicating that the cell’s value has been validated under current inputs. The validated cell’s border also becomes more blue, indicating that dependencies between the validated cell and cells it references have been “exercised” in producing the validated values. Red borders mean untested, blue borders mean tested, and any color in between means partially tested. From the border colors, the user is kept informed of which areas of the spreadsheet are tested and to what extent. The tool also supports more fine-grained access to testing the data dependencies in the spreadsheet, as well as a “percent tested” bar at the top of the spreadsheet, providing the user with an overview of her testing progress.

To help users think of values to test, users can invoke the “Help Me Test” utility to automatically generate suitable test values [Fisher et al. 2002a, Fisher et al. 2006b]. This approach finds values that follow unexplored paths in the spreadsheet’s dataflow, as well as reuse prior test case values for regression testing after a spreadsheet has changed. Abraham and Erwig describe an alternative approach to generating test values by back-propagating constraints on cell values, showing that it can be more effective in terms of efficiency and predictability [Abraham and Erwig 2006b].

WYSIWYT is the most mature error-detecting approach for end-user programmers. It includes support for reasoning about regions of cells with shared formulas [Fisher et al. 2006b, Burnett et al. 2002] and also interacts with assertions (covered in Section 3.2),

fault localization, debugging (covered in Section 3.5), reuse of prior test cases [Fisher et al. 2002b], and the “Help Me Test” functionality mentioned earlier. There has also been research into how WYSIWYT can be applied to visual dataflow languages [Karam and Smedley 2002] and to the kind of “screen transition” programming being developed for web page design [Brown et al. 2003].

3.4.3. Checking Against Specifications. Another approach to detecting errors in programs is by checking values computed by the program against some form of specification; these specifications then serve as the oracle for correctness. For example, as discussed in Section 3.2, one form of specifications that can be entered is assertions about the values that a spreadsheet cell can have. Such assertions can be propagated to infer new assertions, using interval arithmetic [Ayalew and Mittermeir 2003, Burnett et al. 2003]. Assertions that conflict with one another are also highlighted, showing errors in the assertions or the formulas through which they propagated. Other approaches validate string input against flexible data type definitions [Scaffidi et al. 2008]. In all of these approaches, values that do not conform to the assertions are highlighted.

Elbaum et al. [2005] describe an approach for capturing *user session data* from users who utilize web applications, and using this data to distill relevant testing information. The approach can be abstractly thought of as identifying specification information about a web application in the form of an *operational abstraction* of usage of that application. By focusing on usage, the approach allows verification relative to an (often shifting) operational profile; this can detect errors not foreseen by developers of the application, who often have unrealistic expectations about application usage.

3.4.4. *Consistency Checking*. Instead of using a human oracle or external specifications, some systems define correctness heuristics about the *internal* consistency of a program's code. One approach for spreadsheets is a form of type inference called "unit inference and checking systems" [Chambers and Erwig 2009, Abraham and Erwig 2004, Abraham and Erwig 2007b, Ahmad et al. 2003, Antoniu et al. 2004, Coblenz et al. 2005]. These approaches are based on the idea that users' layout of data, especially the labeled row and column headers, offer a form of user defined type called a *unit* [Erwig and Burnett 2002]. For example, the label (column head) "apples" would represent entries of type apple. The "apples" label gets propagated to other formulas that use this value, and the labels are combined in different ways depending on the operator. The program can then be checked against these units for consistency. To illustrate, consider the spreadsheet in Figure 8, using the UCheck system [Abraham and Erwig 2004, Abraham and Erwig 2007a]. Because a column is labeled "Apples," the entries in that column can be considered of unit Apples. The approach begins with an analysis of spatial layout, also taking into account referencing relationships in formulas, to determine the relationships among header labels for rows and columns, their relationship to data entries, and how far in the spreadsheet these labels apply. Because the row labeled "Total" contains sums of the "Apples" entries, the system decides that "Total" marks the end of the "Apples" column.

The system can also reason about transformations that happen through formula operators/function calls, such as inferring that the sum of two Apples entries is also of type

	A	B	C	D	E	F
1		Fruit				
2	Month	Apple	Orange	Plum	Total	
3	May		4	5	6	15
4	June		7	7	8	22
5	July		5	5	8	10
6	Total		16	17	14	47
7						

Figure 8. The UCheck system for inferring units from headers. The arrows represent unit inferences based on the column and row labels [Abraham and Erwig 2007b].

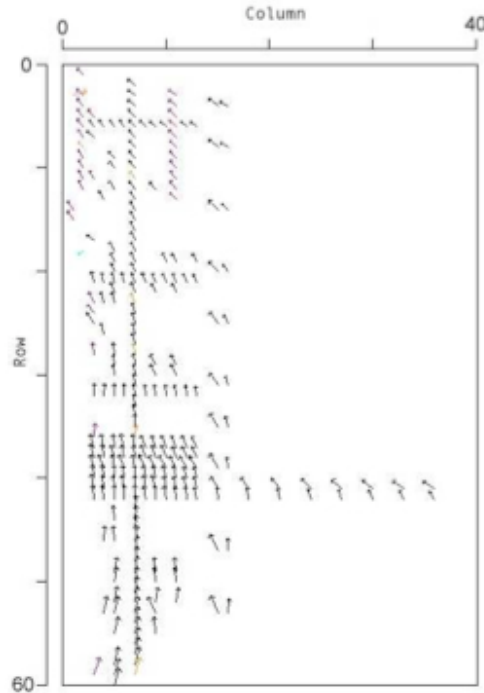


Figure 9. A “data dependency flow” of a spreadsheet’s dependencies. Reproduced from [Ballinger et al. 2003] with permission from authors.

Apples, even if it is not in the Apples column. These inferences can be crosschecked for contradictions, and, just as in type inference, these contradictions are strong indications of logic errors. For example, if the sum of two Apples entries occurs in the middle of the Oranges column, the system could consider this to be a case of conflicting type information and generate a unit error. Empirical studies suggest that end users are successful at using these features to detect errors [Abraham and Erwig 2007b].

Another form of internal consistency checking is *statistical outlier finding*, which involves identifying invalid data values that are mixed among a set of valid values. Miller and Myers [2001] used this approach to help detect errors in text editing macros. Scaffidi [2007] developed a similar algorithm that infers a format from an unlabeled collection of examples that may contain invalid values. The generated format is presented in human-readable notation, so end-user programmers can review and customize the format before using it to find outliers that do not match the format. Raz et al. [2002] used anomaly detection to monitor on-line data feeds incorporated in web-based applications for possible erroneous inputs. All of these approaches use statistical analysis and interactive techniques to direct end-user programmers’ attention to potentially problematic values.

3.4.5. *Visualizations*. Another way to check the correctness of a program is to *visualize* its behavior. Visualization tools enable end-user programmers to apply their knowledge of correctness to certain features of their program’s behavior. For example, Igarashi et al. present comprehension devices that can aid spreadsheet users in dataflow visualization and editing tasks, and finding faults [1998]. More recent spreadsheet visualization approaches include detecting semantic regions and classes [Clermont 2003, Clermont and Mittermeir 2003, Fisher et al. 2006b], ways to visualize trends and “big picture” relationships in spreadsheets that elide a number of low-level details [Ballinger et al. 2003] (see *Figure 9*); and visual auditing features in which similar groups of cells are recognized and shaded based on formula similarity [Sajaniemi 2000]. This latter technique builds on earlier work on the Arrow Tool, a dataflow visualization device proposed by Davis [1996].

While most of these visualization tools have been created for the spreadsheet paradigm, there is a growing interest in allowing users to test and verify the behavior of machine learning algorithms. For example, Talbot et al. [2009] describe a system that allows users to partition data, view weights on different classifiers, and use a confusion matrix visualization to assess the behavior of the resulting classifier. Like the spreadsheet visualizations, this system focuses on portraying more global trends in the program behavior.

3.5. Why is My Program Not Working? —Debugging

Whereas verification and testing detect the *presence* of errors, debugging is the process of *finding* and *removing* errors. Debugging continues to be one of the most time-consuming aspects of both professional and end-user programming [LaToza et al. 2007, Ko et al. 2005, Ko et al. 2007]. Although the process of debugging can involve a variety of strategies, studies have shown across a range of populations that debugging is fundamentally a hypothesis-driven diagnostic activity [Brooks 1977, Littman et al. 1986, Katz and Anderson 1988, Robillard et al. 2004, Gugerty and Olson 1986, Wiedenbeck 2004, Ko and Myers 2004b]. What makes debugging difficult in general is that programmers typically begin the process with a “why” question about their program’s behavior, but must translate this question into a series of actions and queries using low-level tools such as breakpoints and print statements [Ko and Myers 2008b].

A number of issues make debugging even more problematic for end-user programmers. Many lack accurate knowledge about how their programs execute and, as a result, they often have difficulty conceiving of possible explanations for a program's failure [Ko

and Myers 2004b]. Furthermore, because end users often prioritize their external goals over software reliability, debugging strategies often involve “quick and dirty” solutions, such as modifying their code until it appears to work. In the process of remedying existing errors, such strategies often lead to additional errors [Ko and Myers 2003, Beckwith et al. 2005a].

Although prior studies of debugging have focused on a broad set of domains and language paradigms, the technologies to support debugging have generally focused on spreadsheets and event-based imperative languages. In this section, we organize approaches to supporting debugging in light of this historical bias and discuss their potential to generalize beyond these paradigms.

3.5.1. Analyzing dependencies. Dependencies in a program’s execution can involve control dependencies (such as a statement only executing if a particular condition is true) and data dependencies (such as a variable’s depending on the sum of two other variables) [Tip 1995]. Such dependencies are the basis of a number of end-user debugging tools.

One approach in the spreadsheet domain is an extension to the WYSIWYT testing framework, which was discussed in Section 3.4 [Ruthruff et al. 2005b]. To illustrate, see Figure 10 and recall the grades spreadsheet example in Figure 7. Suppose in the process of testing, the teacher notices that row 5’s Letter grade (“A”) is incorrect. The teacher indicates that row 5’s letter grade is erroneous by “X’ing it out” instead of checking it off. Row 5’s Course average is also wrong, so she X’s that one, too. As Figure 10 shows, both cells now contain pink (gray in this paper), but Course is darker than Letter because Course contributed to two incorrect values (its own and Letter’s) whereas Letter contrib-

	NAME	ID	HMAVG	MIDTERM	FINAL	COURSE	LETTER
1	Abbott, Mike	1,035	89	91	86	88.4	B
2	Farnes, Joan	7,649	92	94	92	92.6	A
3	Green, Matt	2,314	78	80	75	77.4	C
4	Smith, Scott	2,316	84	90	86	86.6	B
5	Thomas, Sue	9,857	89	89	89	93.45	A
6							
7	AVERAGE		86.4	88.8	85.6	87.69	

Figure 10. After the teacher marks a few successful and unsuccessful test values, the system helps her narrow down the most likely location of the erroneous formula [Ruthruff et al. 2005b]. Original figure obtained from authors.

uted to only its own. These colors reflect the *likelihood* that the cell formulas contain faults, with darker shades reflecting greater likelihood. Although this example is too small for the shadings to contribute a great deal, in one study, users who used the technique on larger examples did tend to follow the darkest cells and were better at finding bugs than those without the tool [Ruthruff et al. 2005a].

To determine the colors from the X marks, three different algorithms have been used to calculate the WYSIWYT-based fault likelihood colorings [Ruthruff et al. 2006]. All three are based on variants of program slicing [Tip 1995]. The most effective algorithm is based on the sheer number of successful and failed test cases that have contributed to a cell's outcomes [Ruthruff et al. 2006]. Ayalew and Mittermeir [2003] devised a similar method of fault tracing in spreadsheets based on “interval testing” and slicing. This strategy reduces the search domain after it detects a failure, and selects a single cell as the “most influential faulty”. It has some similarities to the assertions work presented in Section 3.2.3 [Burnett et al. 2003], but it not only detects the presence of possible errors, but also what cells are most likely to contain faulty formulas.

A new class of tools based on *question asking* rather than on test outcomes has re-

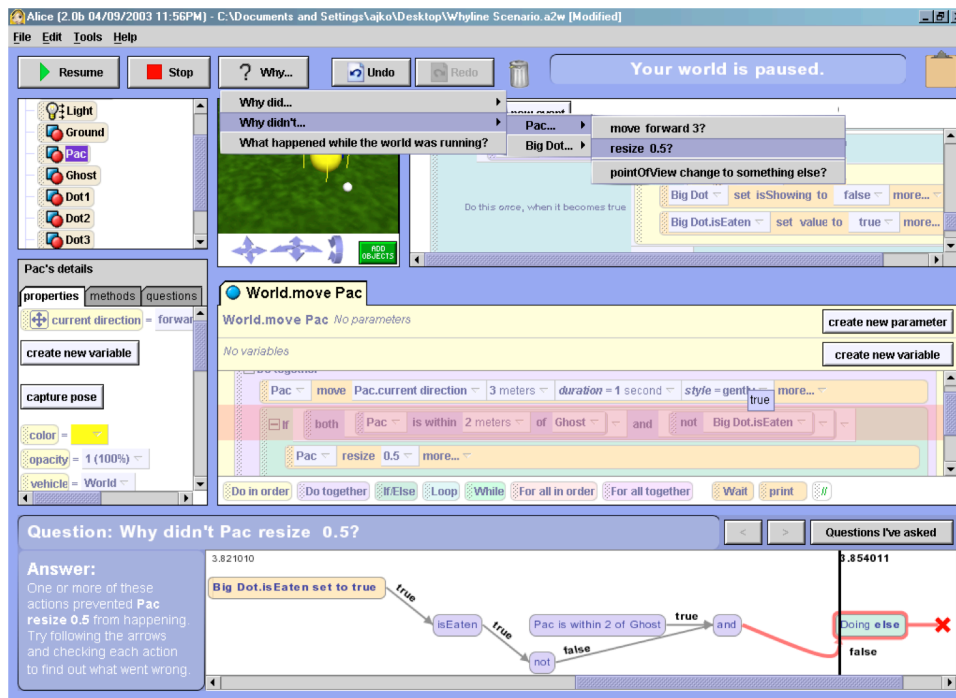


Figure 11. The Whyline for Alice. The user has asked why Pac Man failed to resize and the answer shows a visualization of the events that prevented the resize statement from executing [Ko and Myers 2004a]. Original figure obtained from authors.

cently emerged and has proven effective. The first tool to take this approach was Ko and Myers' Whyline [2004a], which was prototyped for the Alice programming environment [Dann et al. 2006] and is shown in Figure 11. Users execute their program, and when they see a behavior they have a question about, they press a "Why" button. This brings up a menu of "why did" and "why didn't" questions, organized according to the structure of the visible 3D objects manipulated by the program. Once the user selects a question, the system analyzes the program's execution history and generates an answer in terms of the events that occurred during execution. In a user study, the Whyline reduced debugging time by a factor of 8 and helped users get through 40% more tasks, when compared to users without the Whyline [Ko and Myers 2004a]. In a similar approach, Myers et al. [2006] describe a word processor that supports questions about the document and the application state (such as preferences about auto-correction and styles). This system enabled the user to ask the system questions such as "why was teh replaced with the?" The answers were given in terms of the user-modifiable document and application state that ultimately influenced the undesirable behavior. Recent work has shown that these question-asking tools, particularly those that answer "why" questions, can help users understand software behavior more accurately and more in-depth than systems that support "what if" and "how to" types of questions [Lim et al. 2009].

3.5.2. Change Suggestions. An entirely different approach to debugging goes a step further in automation. GoalDebug is a semi-automatic debugger for spreadsheets [Abraham and Erwig 2007a] that allows the user to select an erroneous value, give an expected value, and get a list of changes to the spreadsheet's formulas that would result in the cell having the desired value. Users can interactively explore, apply, refine, or reject these change suggestions. The computation of change suggestions is based on a formal inference system that propagates expected values backwards across formulas. Empirical results so far showed that the correct formula change was the first suggestion in 59% of the cases, and among the first five in 80% of the cases [Abraham and Erwig 2007b]. Of course, there will certainly be situations with such tools where the necessary change is far too complex for the system to infer. This approach also suffers from the oracle problem (Section 3.4.1), because it assumes that users can specify correct values.

3.5.3. Sharing Reasoning. Given the variety of debugging tools that both detect and locate errors in spreadsheets, recent work has developed ways to combine the results of multiple techniques. For example, Lawrance et al. [2006] developed a system to combine the reasoning from UCheck [Abraham and Erwig 2004] and WYSIWYT [Ruthruff et al.

2005b]. The combined reasoning demonstrated both the importance of the information base used to locate faults and the mapping of this information into visual fault localization feedback for end-user programmers, replicating the findings of [Ruthruff et al. 2005a]. They found that UCheck's static analysis of the spreadsheet effectively detected a narrow class of faults, while WYSIWYT (which was driven by a probabilistic model of users derived from previous work [Phalgune et al. 2005]) detected a broader range of faults with moderate effectiveness, and that certain combinations of the two were more effective than either alone. Additionally, by manipulating the mapping, they were able to improve the effectiveness of the feedback.

3.5.4. Social and Cognitive Support. Aside from using tools to help end users debug, there are other approaches that take advantage of human and social factors of debugging. For example, a study of end-user debugging found that when end users worked in pairs rather than alone, they were more systematic and objective in their hypothesis testing [Chintakovid et al. 2006]. This approach was inspired by similar research on the benefits of pair programming for professional programmers.

Kissinger et al. [2006] categorized people's comments during a debugging task in the lab, finding a number of questions that people ask of themselves, including "Am I smart enough?" "Is this the right value?" and "What should I do next?" These questions demonstrated the importance of supporting the individual's questions about planning and their meta-cognitive strategies, not just their questions about the debugging problem itself. These findings led to a video-based approach to teaching debugging strategies, in which a user could ask for help from videotaped human assistants [Subrahmaniyan et al. 2007, Grigoreanu 2008]. In the study of this approach, participants chose better debugging strategies as a result of viewing the videos in the context of their problems, and had correspondingly more success at debugging.

4. CROSS-CUTTING ISSUES IN END-USER SOFTWARE ENGINEERING

As illustrated in the previous sections, there has been significant progress in understanding and relieving the software engineering challenges that arise when people engage in end-user programming. However, in addition to this collection of studies and technologies, researchers have also explored several crosscutting issues that affect the degree to which people engage in software engineering activities or use software engineering tools. For example, in addition to intent, there are many other factors that distinguish end-user programming from professional software development. There are also many issues in

how well tools can generalize across paradigms and user groups. Additionally, simply creating EUSE tools is not enough: users must see the value in using them and they must be able to use them effectively, despite individual differences in strategy. In this section, we organize prior work on these issues and discuss the potential for future work.

4.1. Risk, reward, and the role of domain

While our definitions of end-user programming and end-user software engineering draw a clear line at the *intent* behind programming, this definition almost certainly does not capture the diversity of programming activities in the world. Some researchers have begun to document programming activities in different domains [Scaffidi et al. 2006, Scaffidi et al. 2007a, Rosson and Kase 2006, Wiedenbeck 2005, Carver et al. 2007, Segal 2007, Myers et al. 2008, Petre and Blackwell 2007], finding that a wide range of people are engaging in programming to support their work or hobbies and not all of them are novice programmers. Some are skilled professionals, such as scientists and analysts, who also happen to have programming skills that they can apply to their work. It may not be meaningful to group these professionals together with *novice* programmers with similar intents, but different perceptions around learning new tools and a different willingness to redefine their work practices.

Part of what underlies this distinction is that people vary in their *tolerance* and *perceptions* of risk and reward. For example, some teachers may not be willing to learn a new testing tool because they may not see the eventual payoff or may be skeptical about their own success. A financial analyst faced with performing thousands of manual transactions may see the situation differently. The costs involved in learning how to automate a task may be so high that it may seem more economical to find a manual alternative (or to persuade someone else to write the program). Blackwell's Attention Investment model [Blackwell and Green 1999, Blackwell 2002] provides a cognitive model of these insights, describing individuals' allocations of attention as cognitive "investments." According to the model, a user weighs four factors (not necessarily explicitly) before taking an action: (1) perceived benefits, (2) expected pay-off, (3) perceived cost, and (4) perceived risks. For example, imagine that an administrator in a small art museum might be considering adopting one of the spreadsheet verification tools in Section 3.4.2 to detect errors, because of recent problems in inventory tracking. The administrator might see a *benefit* in that the enhancement would allow her to find and fix errors more quickly. The *expected pay-off* is that inventory tracking will be dependable thus relieving her from the

additional effort of supplementary audits. The *perceived cost* is that she will have to spend time learning to use the new features, while the *perceived risk* is that the features do not aid her enough to make it worth her effort. According to the Attention Investment model, her decision is based on a calculus of these factors. Some end-user programming techniques have attempted to support this decision making by measuring a trade-off point (for example, how many items must be processed manually before the time saved is more than that needed to author the automation [Stylos et al. 2004, Miller and Myers 2001]).

The irony of attention investment is that even this careful thought involves the investment of attentional effort. It might even be the case that truly rigorous analysis of requirements can be more costly than writing another program (a phenomenon that plagues the advocates of formal specification languages). Users who have a choice between programming or manual procedures are likely to avoid such careful analysis of requirements not because they are lazy or careless, but simply because it would be a poor investment of attention to do so much thinking in advance, rather than making iterative adjustments or simply reverting to manual procedures. A further risk in the attention investment equation is that the program may malfunction, failing to bring the anticipated benefits of automation, or perhaps even resulting in damage. The effort involved in testing or debugging to avoid this eventuality is yet another investment of attention.

In addition to differences in perceptions of risk and reward, the domain of practice can also have a significant influence on how willing users are to engage in software engineering activities. For example, the *domain complexity*, or the types of concepts modeled by software, can vary in nature. Weather simulations, for instance, are likely more complex than a teacher's grading system and are likely to involve different types of computational patterns and different software architectures. A related factor is an end-user programmer's *domain familiarity*. This is the difference between a banker writing banking software and a professional programmer writing banking software. The banker would have to learn to program, whereas the professional would have to learn banking concepts. The domain for which a program is written and its underlying characteristics, are a fundamental part of Sutcliffe's exploration of reuse [Sutcliffe 2002], in which he describes the role of *granularity* (how "large" an abstraction is) and abstraction (how it is partitioned). Both of these factors can influence how easily code can be appropriated for a particular task.

Finally, people in different domains of practice may also *collaborate* differently. Professional developers work in teams [Ko et al. 2007], which can change the constraints on

programming decisions, but this is often not the case in end-user programming. Teachers may work alone [Wiedenbeck 2005]; designers may work with other developers [Myers 2008]; web developers may work with users [Scaffidi et al. 2007]. The *cultural values* around software development itself can also vary, influencing tool adoption and motivations to invest in learning software engineering concepts [Segal 2005]. Further, the end user's organizational context imposes constraints and values of its own [Mehandjiev et al. 2006].

4.2. Persuading people to use EUSE tools

Given the discussion in the previous section, and the lower priority of software engineering concerns in end-user programming, getting users to use EUSE tools at all is a significant challenge. End users may be reluctant to use new, unknown features of a system, because they may perceive the features as risky or unhelpful [Blackwell 2002]. Furthermore, because many people who engage in end-user programming lack training in software engineering principles, they may not see immediate, or even long-term value in using software engineering tools.

One approach to this problem is to *train* end-user programmers about software engineering and computer science principles rather than (or in addition to) trying to design tools around end users' existing habits. Some of the tools discussed in this survey have the side effect of training users, but do not explicitly intend to train. For example, a central issue in many of the tools described in this survey is the tension between *formality* and *accessibility*. With more explicit requirements, tests, and verifications, come more precise analysis, but more difficulty in expression (a related concept is *provisionality* [Green et al. 1996], which is the ability in a tool or notation to express things tentatively or imprecisely). Although this paper has demonstrated notations that are both accessible and precise, there are only a few such examples [Beckwith et al. 2005b, Gross and Do 1996] and no work has assessed to what extent end users learn more general principles by using these technologies.

Some work has attempted to explicitly train end users in software engineering and computer science principles. Umarji et al. [2008] explored one approach, focusing on teaching quality assurance, reuse, and documentation best practices, but the training's influence on software quality is not yet known. The authors do suggest, however, that training may help end users make more informed decisions about when and when not to engage in software engineering activities, relative to their goals and priorities. Perhaps

professional software developers develop similar instincts through experience and these instincts could be distilled into advice that can be taught.

An alternative to explicitly training users in principles is to find “teachable moments” to provide a concrete, contextual illustration of the benefits of software engineering ideas. *Surprise-Explain-Reward* [Wilson et al. 2003, Robertson et al. 2004, Ruthruff et al. 2004] is one such approach, aimed at changing end-user programmers’ perceptions of risk and reward. The strategy consists of three basic steps:

1. *Surprise* the user in order to raise curiosity about a feature,
2. Provide *explanations* to satisfy the user’s curiosity and encourage trying out the feature, and
3. Give a *reward* for trying the feature, encouraging future use of the feature.

A simple example of Surprise-Explain-Reward is enticing users to try out the WYSIWYT testing features [Ruthruff et al. 2004], described in Section 3.4.2. One of the best ways to surprise users and get their attention is to *violate* their assumptions. For example, the red border in cell Exam_Avg in Figure 12 (grey in this paper) may be surprising if the coloring is unexpected. If the user hovers over the surprising red cell border, a tool tip pops up with an explanation that “0% of this cell has been tested,” a passive form of feedback that allows, but does not require, the user to attend to it [Robertson et al. 2004]. The user may respond by examining the cell value, deciding that it is correct, and placing a checkmark (✓) in the decision box at the upper right corner of the cell. As described in Section 3.4.2, this decision results in an increase of the cell’s testedness, changing its color, and more importantly, an increase in the progress bar (at the top of Figure 12). Some of these rewards are functional (e.g., carrying out a successful test), and others are perceivable rewards that do not affect the outcome of the task (e.g., the progress bar that informs the user how close he or she is to completing the testing). Research has shown that such perceivable rewards can significantly improve users’ understanding and performance [Ruthruff et al. 2004].

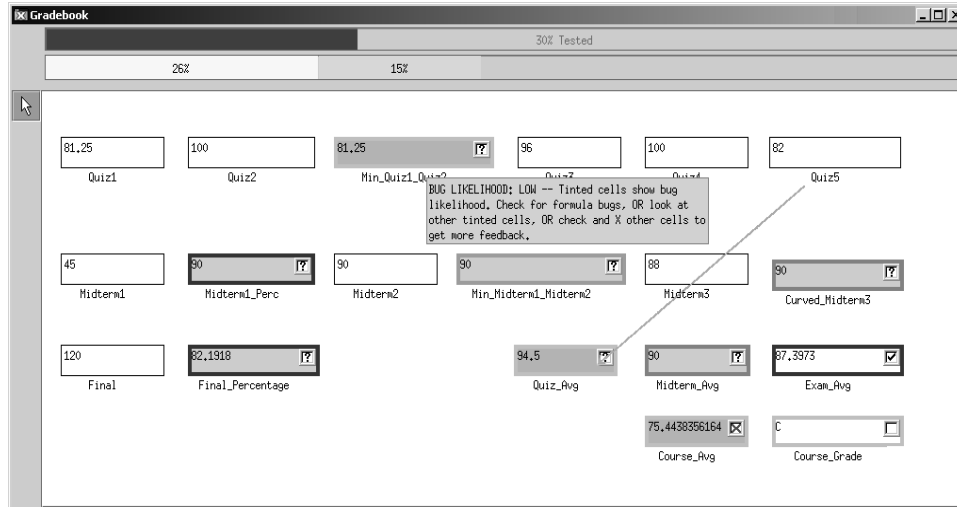


Figure 12. The Surprise-Explain-Reward strategy in Forms/3. The changing colors surprise users, the tooltips explain the potential rewards, and the further changes in colors and the percent testedness bar at the top are the rewards [Wilson et al. 2003]. Original figure obtained from authors.

The same Surprise-Explain-Reward strategy was used in designing the assertions described in Section 3.2.2. An empirical study of the feature [Wilson et al. 2003] found that although users had no prior knowledge of assertions, they entered a high number of assertions and viewed many explanations about assertions. Use of assertions was rewarded by more correct spreadsheets, as well as users' perceptions that assertions helped them to be accurate.

One danger with such an approach may be that users may "game the system," using the system and its features in order to achieve goals, such as coloredness, other than the intended one, namely correctness. This has been observed in studies of computer-based learning environments [Baker 2007], where the primary goal is learning, but some students learn how to manipulate the system to avoid learning. In the case of WYSIWYT, this might mean checking off all of the cells as correct without actually assessing the correctness of the cells' values, just to attain 100% testedness. Users might do this because they do not understand the meaning of the system's feedback, or possibly because the system makes it difficult to avoid using a feature. Because of this possibility, the aphorism of "garbage-in garbage-out" comes into play. In lab studies, this behavior was not enough to outweigh the effectiveness of WYSIWYT in helping users find errors [Burnett et al. 2004].

4.3. Self-efficacy, gender, and strategy in EUSE tool use

Even if one accounts for perceptions of risk and reward and makes a significant effort to train users about the benefits of a software engineering mindset, there is some evidence that personal factors such as self-efficacy and problem solving strategies can significantly influence how effective EUSE tools are at ensuring software quality. Researchers have only recently begun to consider the role of these individual differences [Beckwith and Burnett 2004, Grigoreanu et al. 2006, Beckwith et al. 2007, Subrahmaniyam et al. 2008, Grigoreanu et al. 2008].

Some of these investigations have been done in the context of *self-efficacy*, a psychology construct that represents an individual's belief in their ability to accomplish a specific task [Bandura 1977] (not to be confused with *self-confidence*, which refers to one's more general sense of self-worth). Research has linked it closely with performance accomplishments, level of effort, and the persistence a person is willing to expend on a task [Bandura 1977]. Because software development is a challenging task, a person with low self-efficacy may be less likely to persist when a task becomes challenging or may calculate attention investment tradeoffs differently [Blackwell et al. 2009].

One study considered the self-efficacy of males and females in a spreadsheet debugging task and how it interacted with participants' use of the WYSIWYT testing/debugging features present in the environment [Beckwith et al. 2005a]. The result in this study and others that followed [Beckwith et al. 2006, Beckwith et al. 2007] was that self-efficacy was predictive of the females' ability to use the debugging features effectively, but it was not predictive for males. The females, who had significantly lower self-efficacy, also were less likely than males to engage with the features they had been unfamiliar with prior to the study (regardless of whether the feature had been taught in the tutorial). Females expressed that they were afraid it would take them too long to learn about one of these features, but they actually understood the features as well as the males did. Because the females chose to rely on features they were familiar with already, they used formula editing rather than the debugging features to debug and, as a result, inserted more formula errors than the males.

Another study considered gender differences in "tinkering," a form of playful experimentation encouraged in educational settings because of its documented learning benefits [Rowe 1978]. Research suggests that tinkering is more common among males [Jones et al. 2000, Martinson 2005, Van Den Heuvel-Panhuizen 1999], especially in computing [Rode 2008]. Findings such as these prompted an experiment investigating the effects of

tinkering and gender on end-user debugging [Beckwith et al. 2006]. The results found that females' tinkering was *positively* related to success, whereas the males' tinkering was *negatively* related to success. This was because females were more likely to *pause* between their actions than the males were, leaving more time for analysis and interpretation of the changes that occurred due to their action. Also, males tinkered more and were less likely to pause.

These gender difference results led to the design of a new variant of these features, which adds explicitly “tentative” versions of the WYSIWYT features, aimed primarily at benefiting low-confidence females [Beckwith et al. 2005b]. These changes also slightly raise the cost of tinkering, aimed at reducing males' tendency to tinker excessively. Follow-on monitoring of feature usage showed encouraging trends toward closing of the gender gap in feature usage [Beckwith et al. 2007], and a lab study combining that feature enhancement with strategy explanation support showed significant reduction in the gender gap in feature usage and tinkering by improving females' usage without negative impact to males [Grigoreanu et al. 2008].

In a separate line of research, Kelleher investigated issues of motivation in the domain of animations and storytelling [Kelleher and Pausch 2006]. The goal was not to identify gender differences in performance, but to identify design considerations that would motivate middle school girls to tell stories using interactive animations. To do this, girls were asked to create detailed storyboards of stories they wanted to tell and annotate them with textual descriptions. Analyses of the storyboards revealed a small number of animations necessary to support storytelling, including speech bubbles for talking and thinking, walking, changing body positions, and touching other objects. These features resulted in most of the participants of a study sneaking in extra time during class breaks to work on their storytelling projects [Kelleher and Pausch 2007]. Kafai has also studied gender differences in programming, but for a different audience in a different domain: ten-year-old children programming video games. Her work reported significant gender differences in game character development and in the kind of game feedback that the children programmed [Kafai 1997].

The implications of these findings on the design of end-user software engineering tools reach more broadly than just gender: it suggests that there are barriers to success at end-user software engineering activities for males and females. The body of work also suggests that the designs for features to support end-users can be done in a way that helps to remove these barriers, regardless of whether the person encountering them is male or

female. Future work should better understand not only these barriers, but also ways of detecting when such barriers are encountered.

5. CONCLUSIONS

Most programs today are written not by professional software developers, but by people with expertise in other domains working towards goals supported by computation. This article has offered definitions that distinguish this practice from professional software development and it has organized decades of research on incorporating requirements, design, testing, verification, and debugging concerns into users' existing work practices.

What we have found in our review is an early and in-depth focus on testing and debugging in spreadsheets and imperative event-based languages. Recent work, however, is moving in several directions at once: (1) to more platforms and paradigms, including the web and mobile devices, (2) to explore a broader array of software engineering concerns, including specification and reuse, and (3) to a focus more broadly on application domains, rather than language paradigm alone. The web in particular is becoming a dominant platform on which to study and support end-user programming, with much of the work occurring in the past few years [Little et al. 2007][Macias and Paterno 2008][Nichols et al. 2008][Toomim et al. 2009]. This is probably due to the rapid increase in the use of the web (as opposed to offline desktop applications) to support computational work.

In general, this recent surge in the diversity of end-user software engineering research is both a blessing and a curse. The sheer diversity of domains that researchers are studying may lead researchers to find that the truly difficult problems in end-user software engineering arise from the domain itself, and not from the software engineering challenges. If this is the case, research in end-user software engineering will likely shift to better understanding particular domains of practice, rather than on particular paradigms or technologies. However, this diversity is also an opportunity: if among these widely diverse domains of practice there are fundamental software engineering challenges, the field of software engineering research has the opportunity to dramatically improve the broader use of computational tools in human endeavors.

Part of this challenge is to consider the generalizability of software engineering tools. For example, do these groups need different fundamental software engineering support, or do they just need software engineering support tailored to their domain of practice? For example, the Whyline [Ko and Myers 2004a], which began as a debugging tool for

end-user programming of animations in the Alice environment, was successfully adapted for professional Java programmers [Ko and Myers 2008a]. The differences between these two approaches are not in the larger concept, but in how the concept was tailored to the differing information needs of the two target user populations. Other tools, such as the testing and verification approaches described in Section 3.4, transform “batch” testing techniques to incremental approaches. Others still are traditional concepts that exploit properties of a particular language or the types of programs created with a language (for example, exploiting the layout of spreadsheets in a grid). Therefore, it is possible that the primary challenges in tool design are not fundamental conceptual differences in software engineering, but the adaptation of these concepts to particular domains of practice and differing priorities. This makes the previously mentioned work on motivation even more important, in that the adaptations may be primarily in reframing the presentation and interaction of more general software engineering tool concepts.

Another type of tool generalizability is the extent to which software engineering concepts can transfer between paradigms. For example, the notion of interrogating program output [Ko and Myers 2004a] was adapted to the spreadsheet paradigm [Abraham and Erwig 2007] successfully, but the two prototypes have a number of differences. For example, the Whyline focused on *finding* the code that caused or prevented a particular output. In contrast, in spreadsheets, finding the formula that caused a problem is generally trivial. Instead, Abraham and Erwig transformed the concept from one of asking questions to one of stating an expected value. The implementations of these systems were non-trivial and have little in common. Therefore, it is possible that while the general concepts involved in bringing software engineering concerns to end-user programming may generalize, very little else may transfer. Future work will reveal to what extent ideas that support other aspects of software engineering, such as specification and reuse, can generalize between paradigms and domains.

Finally, in all of this research, it is important to remember that the programs that end-user programmers create are just small parts of the much larger contexts of their lives at work and at home. Understanding how programming fits into end users’ everyday lives is central to not only the design of the EUSE tools, but our understanding of why people program at all. The research on motivations and perceptions presented in Section 4 is just a glimpse of the contextual factors that influence programming activity. We expect that future work will discover and explain an even greater number of these factors and better inform the design of end-user software engineering tools.

ACKNOWLEDGEMENTS

This work was supported in part by the EUSES Consortium via NSF grants ITR-0325273/0324861/0324770/0324844/0405612 and also by NSF grants ITWF-0420533 and IIS-0329090. The first author was also supported by a National Defense Science and Engineering Grant and an NSF Graduate Research Fellowship. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

- ABRAHAM R. AND ERWIG M. 2004. Header and unit inference for spreadsheets through spatial analyses. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, September, 165–172.
- ABRAHAM R., ERWIG M., KOLLMANSBERGER S., AND SEIFERT E. 2005. Visual specifications of correct spreadsheets. *IEEE International Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, September, 189-196.
- ABRAHAM R. AND ERWIG M. 2006. Inferring templates from spreadsheets. *International Conference on Software Engineering*, Shanghai, China, May, 182-191.
- ABRAHAM R. AND ERWIG M. 2006. AutoTest: A tool for automatic test case generation in spreadsheets. *Visual Languages and Human-Centric Computing*, Brighton, UK, September, 43-50.
- ABRAHAM R. AND ERWIG M. 2006. Type Inference for Spreadsheets. *ACM International Symposium on Principles and Practice of Declarative Programming*, 73-84.
- ABRAHAM R. AND ERWIG M. 2007. GoalDebug: A spreadsheet debugger for end users. *International Conference on Software Engineering*, Minneapolis, Minnesota, May, 251-260.
- ABRAHAM R. AND ERWIG M. 2007. UCheck: A spreadsheet unit checker for end users. *Journal of Visual Languages and Computing*, 18(1), 71-95.
- ABRAHAM R., ERWIG M. AND ANDREW S. 2007. A type system based on end-user vocabulary. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 215-222.
- AHMAD Y., ANTONIU T., GOLDWATER S., AND KRISHNAMURTHI S. 2003. A type system for statically detecting spreadsheet errors. *International Conference on Automated Software Engineering*, Montreal, Quebec, Canada, October, 174-183.
- ANTONIU T., STECKLER P.A., KRISHNAMURTHI S., NEUWIRTH E., AND FELLEISEN M. 2004. Validating the unit correctness of spreadsheet programs. *International Conference on Software Engineering*, Edinburgh, Scotland, May, 439-448.
- AYALEW Y. AND MITTERMEIR R. 2003. Spreadsheet debugging. *European Spreadsheet Risks Interest Group*, Dublin, Ireland, July 24–25.
- BAKER S.J. 2007. Modeling and understanding students’ off-task behavior in intelligent tutoring systems. *ACM Conference on Human Factors in Computer Systems*, San Jose, California, April, 1059-1068.
- BALABAN M., BARZILAY E., AND ELHADAD M. 2002. Abstraction as a means for end user computing in creative applications. *IEEE Transactions on Systems*, 32(6), November, 640-653.
- BALLINGER D., BIDDLE R., AND NOBLE J. 2003. Spreadsheet visualisation to improve end-user understanding. *Asia-Pacific Symposium on Information Visualisation*, 24, 99-109.
- BANDINI S. AND SIMONE C. 2006. EUD as integration of components off-the-shelf. In *End-User Development* H. Lieberman, F. Paterno, and V. Wulf (eds).. Springer, 183-205.
- BANDURA A. 1977. Self-efficacy: Toward a unifying theory of behavioral change. *Psychological Review* 8(2), 191-215.

- BARRETT R., KANDOGAN E., MAGLIO P.P., HABER E.M., TAKAYAMA L.A., PRABAKER M. 2004. Field studies of computer system administrators: analysis of system management tools and practices. *ACM Conference on Computer Supported Cooperative Work*, Chicago, Illinois, USA, 388-395.
- BECK, K. 2007. *Implementation Patterns*. Addison-Wesley.
- BECKWITH L. AND BURNETT, M. 2004. Gender: An important factor in problem-solving software? *IEEE Symposium on Visual Languages and Human-Centric Computing Languages and Environments*, September, Rome, Italy, September, 107-114.
- BECKWITH L., BURNETT M., WIEDENBECK S., COOK C., SORTE S., AND HASTINGS M. 2005. Effectiveness of end-user debugging features: Are there gender issues? *ACM Conference on Human Factors in Computing Systems*, Portland, Oregon, USA, April, 869-878.
- BECKWITH L., SORTE S., BURNETT M., WIEDENBECK S., CHINTAKOVID T., AND COOK C. 2005. Designing features for both genders in end-user programming environments, *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, TX, Sept., 153-160.
- BECKWITH L., KISSINGER C., BURNETT M., WIEDENBECK S., LAWRENCE J., BLACKWELL A. AND COOK C. 2006. Tinkering and gender in end-user programmers' debugging. *ACM Conference on Human Factors in Computing Systems*, Montreal, Quebec, Canada, April, 231-240.
- BECKWITH, L., INMAN D., RECTOR K. AND BURNETT M. 2007. On to the real world: Gender and self-efficacy in Excel. *IEEE Symposium on Visual Languages and Human-Centric Computing*, September 23-27, 119-126.
- BECKWITH, L. 2007. Gender HCI issues in end-user programming, Ph.D. Thesis, Oregon State University.
- BEIZER B. 1990. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY.
- BELLETTINI C., DAMIANI E., AND FUGINI M. 1999. User opinions and rewards in reuse-based development system. *Symposium on Software Reusability*, Los Angeles, California, USA, May, 151-158.
- BELLON S., KOSCHKE R., ANTONIOL G., KRINKE J. AND MERLO E. 2007. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9), September, 577-591.
- BERTI, S., PATERNÒ, F., SANTORO, C. 2006. Natural development of nomadic interfaces based on conceptual descriptions. *End User Development*, 143-160.
- BEYER H. & HOLTZBLATT, K. 1998. *Contextual Design: Defining Customer-Centered Systems*. San Francisco: Morgan Kaufmann.
- BEYER S., RYNES K., PERRAULT J., HAY K. AND HALLER S. 2003. Gender differences in computer science students. *Special Interest Group on Computer Science Education*, Reno, Nevada, USA, February, 49-53.
- BIGGERSTAFF T. AND RICHTER C. 1989. Reusability Framework, Assessment, and Directions. *Software Reusability: Vol. 1, Concepts and Models*, 1-17.
- BLACKWELL A. AND GREEN T. R. G. 1999. Investment of attention as an analytic approach to cognitive dimensions. In Green, T.R.G, Abdullah T., and Brna P. (Eds.), *11th Workshop of the Psychology of Programming Interest Group*, 24-35.
- BLACKWELL A., AND HAGUE, R. 2001. AutoHAN: An architecture for programming the home. *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, Virginia, USA, September, 150-157.
- BLACKWELL, A. F. 2002. First steps in programming: A rationale for attention investment models. *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, Virginia, USA, September, 2-10.
- BLACKWELL A. AND BURNETT M. 2002. Applying attention investment to end-user programming. *IEEE Symposia on Human-Centric Computing Languages and Environments*, Arlington, Virginia, USA, September, 1-4.
- BLACKWELL, A.F. 2004. End user developers at home. *Communications of the ACM* 47(9), 65-66.
- BLACKWELL, A.F. 2006. Gender in domestic programming: From bricolage to séances d'essayage. Presentation at *CHI Workshop on End User Software Engineering*.

- BLACKWELL, A.F., RODE, J.A. AND TOYE, E.F. 2009. How do we program the home? Gender, attention investment, and the psychology of programming at home. *International Journal of Human Computer Studies* 67, 324-341.
- BOEHM, B.W. 1988. A spiral model of software development and enhancement. *IEEE Computer*, 21(5), May, 61-72.
- BOGART C., BURNETT M.M., CYPHER A. AND SCAFFIDI C. 2008. End-user programming in the wild: A field study of CoScripter scripts. *IEEE Symposium on Visual Languages and Human-Centric Computing*, to appear.
- BOLIN M. AND WEBBER M. AND RHA P. AND WILSON, T. AND MILLER, R. 2005. Automation and customization of rendered web pages. *ACM Symposium on User Interface Software and Technology*, Seattle, Washington, October, 163-172.
- BRANDT J., GUO P., LEWENSTEIN J., AND KLEMMER S.R. 2008. Opportunistic programming: How rapid ideation and prototyping occur in practice. *Workshop on End-User Software Engineering (WEUSE)*, Leipzig, Germany.
- BROOKS R. 1977. Towards a theory of the cognitive processes in computer programming. *International Journal of Human-Computer Studies*, 51, 197-211.
- BROWN D., BURNETT M., ROTHERMEL G., FUJITA, H. AND NEGORO F. 2003. Generalizing WYSIWYT visual testing to screen transition languages. *IEEE Symposium on Human-Centric Computing Languages and Environments*, Auckland, New Zealand, October, 203-210.
- BURNETT M. 2001. Software engineering for visual programming languages, in *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, volume 2.
- BURNETT M., CHEKKA, S. K. AND PANDEY, R. 2001. FAR: An end-user language to support cottage e-services. *IEEE Symposium on Human-Centric Computing*, Stresa, Italy, September, 195-202.
- BURNETT M., ATWOOD J., DJANG R.W., GOTTFRIED H., REICHWEIN J., YANG S. 2001. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming* 11(2), March, 155-206.
- BURNETT M., SHERETOV A., REN B., ROTHERMEL G. 2002. Testing homogeneous spreadsheet grids with the 'What You See Is What You Test' methodology. *IEEE Transactions on Software Engineering*, June, 576-594.
- BURNETT M., COOK C., PENDSE O., ROTHERMEL G., SUMMET J., AND WALLACE C. 2003. End-user software engineering with assertions in the spreadsheet paradigm. *International Conference on Software Engineering*, Portland, Oregon, May, 93-103.
- BURNETT M., COOK C., AND ROTHERMEL G. 2004. End-user software engineering. *Communications of the ACM*, September, 53-58.
- BUSCH T. 1995. Gender differences in self-efficacy and attitudes toward computers. *Journal of Educational Computing Research*, 12: 147-158.
- BUXTON, B. 2007. *Sketching user experiences: Getting the design right and the right design*. Morgan Kaufmann.
- CARMEN, S.P., FISCHER, G. 2008. Design, adoption, and assessment of a socio-technical environment supporting independence for persons with cognitive disabilities. *ACM Conference on Human Factors in Computing Systems*, Florence, Italy, 597 – 606.
- CARVER J., KENDALL R., SQUIRES S., AND POST D. 2007. Software engineering environments for scientific and engineering software: a series of case studies. *International Conference on Software Engineering*, Minneapolis, MN, May, 550-559.
- CHAMBERS, C. AND ERWIG, M.: Automatic Detection of Dimension Errors in Spreadsheets. *Journal of Visual Languages and Computing*, 20, 2009.
- CHINTAKOVID T., WIEDENBECK S., BURNETT M., GRIGOREANU V. 2006. Pair collaboration in end-user debugging. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, UK, September, 3-10.
- CLERMONT M., HANIN C., AND MITTERMEIR R. 2002. A spreadsheet auditing tool evaluated in an industrial context. *Spreadsheet Risks, Audit, and Development Methods*, 3, 35-46.

- CLERMONT M. 2003. Analyzing large spreadsheet programs. *Working Conference on Reverse Engineering*, November, 306–315.
- CLERMONT M. AND MITTERMEIR R. 2003. Auditing large spreadsheet programs. *International Conference on Information Systems Implementation and Modeling*, April, Brno, Czech Republic, February, 87-97.
- COBLENZ M.J., KO A.J., AND MYERS B.A. 2005. Using objects of measurement to detect spreadsheet errors. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, September 23-26, 314-316.
- COPLIEN J.O., AND HARRISON N.B. 2004. *Organizational patterns of Agile Software Development*. Prentice-Hall, Inc., Upper Saddle River, NJ.
- COOPER A. AND REIMANN R. 2003. *About Face 2.0: The Essentials of Interaction Design*. Indianapolis, IN, Wiley.
- COSTABILE M.F., FOGLI D., MUSSIO P., AND PICCINNO A. 2006. End-user development: The software shaping workshop approach. In *End-User Development*, H. Lieberman, F. Paterno, and V. Wulf (eds.) Springer, 183-205.
- COSTABILE M.F., MUSSIO P., PROVENZA L.P., AND PICCINNO A. 2009. Supporting end users to be co-designers of their tools. *2nd International Symposium on End-User Development (LNCS 5435)*, Siegen, Germany, Springer-Verlag, Berlin Heidelberg, March 2-4, 70-85.
- COX P.T., GILES F.R. AND PIETRZYKOWSKI T. 1989. Prograph: a step towards liberating programming from textual conditioning. *IEEE Workshop on Visual Languages*, 150-156.
- CRANOR, L. F., GUDURU, P., AND ARJULA, M. 2006. User interfaces for privacy agents. *ACM Transactions on Computer-Human Interaction*. 13(2), June, 135-178.
- DANN W., COOPER S., PAUSCH R. 2006. *Learning to program with Alice*. Prentice Hall.
- DAVIS J.S. 1996. Tools for spreadsheet auditing. *International Journal on Human-Computer Studies*, 45, 429-442.
- DELINE R. 1999. A Catalog of Techniques for Resolving Packaging Mismatch. *Symposium on Software Reusability*, Los Angeles, California, USA, 44-53.
- DITTRICH Y., LINDBERG O., AND LUNDBERG L. 2006. End-user development as adaptive maintenance. In *End-User Development*, H. Lieberman, F. Paterno, and V. Wulf (eds.) Springer, 295-313.
- DOUGHERTY D.J., FISLER K., KRISHNAMURTHI S. 2006. Specifying and reasoning about dynamic access-control policies. *International Joint Conference on Automated Reasoning*, Seattle, Washington, USA, 632-646.
- DOUGLAS S., DOERRY E., AND NOVICK D. 1990. Quick: A user-interface design kit for non-programmers. *ACM Symposium on User Interface Software and Technology*, 1990, Snowbird, Utah, USA, 47-56.
- EAGAN, J. R. AND STASKO, J.T. 2008. The buzz: supporting user tailorability in awareness applications. *ACM Conference on Human Factors in Computing Systems*, Florence, Italy, April 05 - 10, 1729-1738.
- ELBAUM S., ROTHERMEL G., KARRE S. AND FISHER II M. 2005. Leveraging user session data to support web application testing. *IEEE Transactions on Software Engineering*, 31(3), March, 187-202.
- ERWIG M., AND BURNETT, M. 2002. Adding apples and oranges. *4th International Symposium on Practical Aspects of Declarative Languages*, LNCS 2257, Portland, Oregon, January, 173-191.
- ERWIG M., ABRAHAM R., COOPERSTEIN I., AND KOLLMANSBERGER S. 2005. Automatic generation and maintenance of correct spreadsheets. *International Conference on Software Engineering*, St. Louis, Missouri, May, 136-145.
- ERWIG M., ABRAHAM R., KOLLMANSBERGER S., AND COOPERSTEIN I. 2006. Gencel—A program generator for correct spreadsheets. *Journal of Functional Programming*, 16(3): 293-325.
- EZRAN M., MORISIO M., TULLY C. 2002. *Practical Software Reuse*, Springer.
- FISCHER G. AND GIRGENSOHN A. 1990. End user modifiability in design environments. *ACM Conference on Human Factors in Computing Systems*, Seattle, Washington, USA, April, 183-192.
- FISCHER, G. AND 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 em-*

ploy advanced information and communication technology, Kluwer Academic Publishers, Dordrecht, The Netherlands, 427-457.

- FISHER II M., CAO M., ROTHERMEL G., COOK C.R., AND BURNETT M.M. 2002. Automated test case generation for spreadsheets. *International Conference on Software Engineering*, Orlando, Florida, May 19-25, 141-151.
- FISHER II M., JIN D., ROTHERMEL G., AND BURNETT M. 2002. Test reuse in the spreadsheet paradigm. *IEEE International Symposium on Software Reliability Engineering*, Raleigh, North Carolina, USA, November, 257-264.
- FISHER II M., ROTHERMEL G., CREELAN T. AND BURNETT M. 2006. Scaling a dataflow testing methodology to the multiparadigm world of commercial spreadsheets. *IEEE International Symposium on Software Reliability Engineering*, Raleigh, North Carolina, USA, November, 13-22.
- FISHER II M., CAO M., ROTHERMEL G., BROWN D., COOK C.R., AND BURNETT M.M. 2006. Integrating automated test generation into the WYSIWYT spreadsheet testing methodology. *ACM Transactions on Software Engineering and Methodology* 15(2), 150-194, April.
- FRANKL P.G. AND WEISS S.N. 1993. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19(8), August, 774-787.
- GARLAN D., ALLEN R., AND OCKERBLOOM J. 1995. Architectural mismatch or why it's hard to build systems out of existing parts. *International Conference on Software Engineering*, Seattle, Washington, April, 179-185.
- GHEZZI, C., JAZAYERI, M., MANDRIOLI D. 2002. *Fundamentals of software engineering*. Prentice Hall: NJ, USA.
- GORB, P. AND DUMAS, A. 1987. 'Silent Design', *Design Studies*, 8, 150-156.
- GREEN T.R.G., BLANDFORD A., CHURCH L. ROAST C., CLARKE S. 2006. Cognitive Dimensions: achievements, new directions, and open questions. *Journal of Visual Languages and Computing*, 17(4), 328-365
- GRIGOREANU, V., BECKWITH, L., FERN, X., YANG, S., KOMIREDDY, C., NARAYANAN, V., COOK, C., BURNETT, M.M. 2006. Gender differences in end-user debugging, revisited: What the miners found. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 19-26.
- GRIGOREANU, V., CAO, J., KULESZA, T., BOGART, C., RECTOR, K., BURNETT, M., WIEDENBECK, S. 2008. Can feature design reduce the gender gap in end-user software development environments? *IEEE Symposium on Visual Languages and Human-Centric Computing*, Herrsching am Ammersee, Germany, September.
- GROSS, M. D. AND DO, E. Y. 1996. Ambiguous intentions: a paper-like interface for creative design. *ACM Symposium on User Interface Software and Technology*, Seattle, Washington, November 6-8, 183-192.
- GUGERTY, L. AND OLSON, G. M. 1986. Comprehension differences in debugging by skilled and novice programmers. *Empirical Studies of Programmers*, Soloway, E. and Iyengar, S., Washington, DC: Ablex Publishing Corporation, 13-27.
- GULLY, N. 2006. Improving the Quality of Contributed Software on the MATLAB File Exchange. *Second Workshop on End-User Software Engineering*, in conjunction with the *ACM Conference on Human Factors in Computing*, Montreal, Quebec.
- HENDERSON, A., AND KYNG, M. 1991. There's no place like home: Continuing design in use. In J. Greenbaum, & M. Kyng (Eds.), *Design At Work* (pp. 219-240). Hillsdale, New Jersey: Lawrence Erlbaum Associates.
- HENDRY, D. G. AND GREEN, T. R. G. 1994. Creating, comprehending, and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model. *International Journal of Human-Computer Studies*, 40(6), 1033-1065.
- HUTCHINS M., FOSTER H., GORADIA T. AND OSTRAND T. 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. *International Conference on Software Engineering*, May, 191-200
- IGARASHI T., MACKINLAY J.D., CHANG B.-W., AND ZELLWEGER P.T. 1998. Fluid visualization of spreadsheet structures. *IEEE Symposium on Visual Languages*, Halifax, Nova Scotia, September 1-4, 118-125.
- IOANNIDOU, A., RADER, C., REPENNING, A., LEWIS, C., & CHERRY, G. 2003. Making constructionism work in the classroom. *International Journal of Computers for Mathematical Learning*, 8(1), 63-108.

- JONES, M. G., BRADER-ARAJE, L., CARBONI, L. W., CARTER, G., RUA, M. J., BANILOWER, E. AND HATCH, H. 2000. Tool time: Gender and students' use of tools, control, and authority. *Journal of Research in Science Teaching* 37(8), 760-783.
- KAFAI, Y. 1996. Gender differences in children's constructions of video games. In *Interacting with Video*, P. Greenfield and R. Cocking (Eds.), Greenwood Publishing Group, 39-66.
- KAHLER, H. 2001. More Than WORDs - Collaborative tailoring of a word processor. *Journal of Universal Computer Science*, 7(9), 826-847.
- KARAM M. AND SMEDLEY T. 2002. A Testing methodology for a dataflow based visual programming language. *IEEE Symposia on Human-Centric Computing*, Arlington, Virginia, September 3-6, 86-89.
- KATZ I.R. AND ANDERSON J.R. 1988. Debugging: An analysis of bug-location strategies, *Human Computer Interaction*, 3, 351-399.
- KELLEHER C. AND PAUSCH R. 2005. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys* 37(2), 83-137.
- KELLEHER, C.; PAUSCH, R. 2006. Lessons learned from designing a programming system to support middle school girls creating animated stories. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 165-172.
- KELLEHER, C., PAUSCH, R., AND KIESLER, S. 2007. Storytelling Alice motivates middle school girls to learn computer programming. *ACM Conference on Human Factors in Computing Systems*, San Jose, California, 1455-1464.
- KISSINGER C., BURNETT M., STUMPF S., SUBRAHMANYAN N., BECKWITH L., YANG S., AND ROSSON M.B. 2006. Supporting end-user debugging: What do users want to know? *Advanced Visual Interfaces*, Venezia, Italy, 135-142.
- KO A.J. AND MYERS B.A. 2003. Development and evaluation of a model of programming errors. *IEEE Symposium Human-Centric Computing Languages and Environments*, Auckland, New Zealand, October,7-14.
- 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, 151-158.
- KO A. J., MYERS B. A., AND AUNG, H.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. 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. and Myers, B. A. 2006. Barista: An implementation framework for enabling new tools, interaction techniques and views for code editors. *ACM Conference on Human Factors in Computing Systems*, Montreal, Canada, April, 387-396.
- KO, A. J. DELINE, R., VENOLIA, G. 2007. Information needs in collocated software development teams. *International Conference on Software Engineering*, Minneapolis, MN, May 20-26, 344-353.
- KO, A.J. AND MYERS, B.A. 2008. Debugging reinvented: Asking and answering why and why not questions about program behavior. *International Conference on Software Engineering (ICSE)*, Leipzig, Germany, May 10-18, 301-310.
- KO, A.J. 2008. Asking and answering questions about the causes of software behaviors, Ph.D. thesis, *Human-Computer Interaction Institute Technical Report CMU-CS-08-122*, May.
- KRISHNA V., COOK C., KELLER D., CANTRELL J., WALLACE C., BURNETT M., AND ROTHERMEL G. 2001. Incorporating incremental validation and impact analysis into spreadsheet maintenance: An empirical study. *IEEE International Conference on Software Maintenance*, Washington, DC, November, 72-81.
- KRISHNAMURTHI S., FINDLER R.B., GRAUNKE P. AND FELLEISEN M. 2006. Modeling web interactions and errors. In Goldin, D., Smolka S. and Wegner P., (Eds.) *Interactive Computation: The New Paradigm*, Springer Lecture Notes in Computer Science. Springer-Verlag.
- LAKSHMINARAYANAN V., LIU W., CHEN C.L., M. EASTERBROOK S. AND PERRY D. E. 2006. Software architects in practice: Handling requirements. *16th International Conference of the IBM Centers for Advanced Studies*, Toronto, Canada, 16-19.

- LATOZA T., VENOLIA G. AND DELINE R. 2006. Maintaining mental models: A study of developer work habits, *International Conference on Software Engineering*, Shanghai, China, 492-501.
- LAWRANCE J., CLARKE S., BURNETT M., AND ROTHERMEL G. 2005. How well do professional developers test with code coverage visualizations? An empirical study. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, September, 53-60.
- LAWRANCE J., ABRAHAM R., BURNETT M., AND ERWIG M. 2006. Sharing reasoning about faults in spreadsheets: An empirical study. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, UK, September, 35-42.
- LESHED, G., HABER, E. M., MATTHEWS, T., AND LAU, T. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. *ACM Conference on Human Factors in Computing Systems*, Florence, Italy, April 05 – 10, 1719-1728.
- LETONDAL, C. 2006. Participatory programming: Developing programmable bioinformatics tools for end users. In *End-User Development*, H. Lieberman, F. Paterno, and V. Wulf (eds.) Springer, 207-242.
- LEVENTHAL L.M., TEASLEY B.E. AND ROHLMAN D.S. 1994. Analyses of factors related to positive test bias in software testing. *International Journal of Human-Computer Studies*, 41, 717-749.
- LIEBERMAN H. AND FRY C. 1995. Bridging the gulf between code and behavior in programming. *ACM Conference on Human Factors in Computing*, Denver, Colorado, 480-486.
- LIEBERMAN H. AND FRY C. 1997. ZStep 95: A reversible, animated, source code stepper, in *Software Visualization: Programming as a Multimedia Experience*, Stasko, J., Domingue, J. Brown, M. and Price, B. eds., MIT Press, Cambridge, MA.
- LIEBERMAN H. (ed.) 2000. *Your Wish is My Command: Giving Users the Power to Instruct their Software*. San Francisco: Morgan Kaufmann.
- LIEBERMAN H., PATERNO F. AND WULF V. (eds) 2006. *End-User Development*. Kluwer/ Springer.
- LINGAM, S. AND ELBAUM, S. 2007. Supporting end-users in the creation of dependable web clips. *International Conference on World Wide Web*, Banff, Alberta, Canada, May 953-962.
- LIM B., DEY A., AVRAHAMI D. 2009. Why and why not explanations improve the intelligibility of context-aware intelligent systems. *ACM Conference on Human Factors in Computing Systems*, Boston, MA, Apr. 4-9, 2119-2128.
- LIN J. AND LANDAY J.A. 2008. Employing Patterns and Layers for Early-Stage Design and Prototyping of Cross-Device User Interfaces. *ACM Conference on Human Factors in Computing Systems*, Florence, Italy, 1313–1322.
- LITTLE, G., LAU, T. A., CYPHER, A., LIN, J., HABER, E. M., AND KANDOGAN, E. (2007). Koala: capture, share, automate, personalize business processes on the web. *ACM Conference on Human Factors in Computing Systems*, San Jose, California, USA, April 943-946.
- LITTLE G. AND MILLER R.C. 2006. Translating keyword commands into executable code. *ACM Symposium on User Interface Software and Technology*, Montreux, Switzerland, October, 135-144.
- LITTMAN D.C., PINTO J., LETOVSKY S. AND SOLOWAY, E. 1986. Mental models and software maintenance, *Empirical Studies of Programmers, 1st Workshop*, Washington, DC, 80-98.
- LIU H. AND LIEBERMAN H. 2005. Programmatic semantics for natural language interfaces. *ACM Conference on Human Factors in Computing*, Portland, Oregon, USA, April, 1597-1600.
- MACÍAS J.A. AND PATERNÒ F. 2008. Customization of web applications through an intelligent environment exploiting logical interface descriptions. *Interacting with Computers*, 20(1): 29-47.
- MACKAY, W. E. 1990. Patterns of sharing customizable software. *ACM Conference on Computer-Supported Cooperative Work*, Los Angeles, CA, USA, October, 209-221.
- MACLEAN, A., CARTER, K., LÖVSTRAND, L. AND MORAN, T. 1990. User-tailorable systems: Pressing the issue with buttons. *ACM Conference on Human Factors in Computing Systems*, April, 175-182.
- MANDELIN D., XU L., AND BODIK R., AND KIMELMAN D. 2005. Jungloid mining: Helping to navigate the API jungle. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 12-15, 48-61.
- MARGOLIS J. AND FISHER A. 2003. *Unlocking the Clubhouse*, MIT Press, Cambridge, MA.

- MARTINSON A.M. 2005. Playing with technology: Designing gender sensitive games to close the gender gap. *Working Paper SLISWP-03-05, School of Library and Information Science, Indiana University.*
- MATWIN, S. AND PIETRZYKOWSKI, T. 1985. Prograph: a preliminary report. *Computer Languages* 10(2): 91-126.
- MCDANIEL R., AND MYERS B. 1999. Getting more out of programming-by-demonstration. *ACM Conference on Human Factors in Computing Systems*, Pittsburgh, Pennsylvania, USA, May, 442-449.
- MEHANDJIEV N., SUTCLIFFE A. AND LEE, D. 2006. Organizational view of end-user development. In *End-User Development*, H. Lieberman, F. Paterno, and V. Wulf (eds.) Springer, 371-399.
- MILLER R. AND MYERS B.A. 2001. Outlier finding: Focusing user attention on possible errors. *ACM Symposium on User Interface Software and Technology*, Orlando, Florida, USA, November, 81-90.
- MILLER R. AND MYERS B.A. 2001. Interactive simultaneous editing of multiple text regions. *USENIX Annual Technical Conference*, Boston, MA, June, pp. 161-174.
- MILLER R., AND MYERS B. 2002. LAPIS: Smart editing with text structure. *ACM Conference on Human Factor in Computing Systems*, Minneapolis, Minnesota, USA, April, 496-497.
- MITTERMEIR R. AND CLERMONT M. 2002. Finding high-level structures in spreadsheet programs. *Working Conference on Reverse Engineering*, Richmond, Virginia, USA, October, 221-232.
- MODUGNO F., AND MYERS B. 1994. Pursuit: Graphically representing programs in a demonstrational visual shell. *ACM Conference on Human Factors in Computing Systems*, Boston, Massachusetts, USA, April, 455-456.
- MØRCH A. AND MEHANDJIEV N.D. 2000. Tailoring as collaboration: The mediating role of multiple representations and application units, *Computer Supported Cooperative Work* 9(1), 75-100.
- MYERS B., PARK S., NAKANO Y., MUELLER G., AND KO. A.J. 2008. How designers design and program interactive behaviors. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Sept 15-18, 2008, Hersching am Ammersee, Germany, to appear.
- NARDI B.A. 1993. *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: The MIT Press.
- NEWMAN M.W., LIN J., HONG J.I., AND LANDAY J.A. 2003. DENIM: An informal web site design tool inspired by observations of practice. *Human-Computer Interaction*, 18(3), 259-324.
- NICHOLS J., AND LAU T. 2008. Mobilization by demonstration: using traces to re-author existing web sites. *Intelligent User Interfaces*, Maspalomas, Gran Canaria, Spain., January, 149-158.
- NISS, M., SADRI, P., AND LEE, K. 2007. Dynamic Spreadsheets as Learning Technology Tools: Developing Teachers' Technology Pedagogical Content Knowledge (TPCK). *American Educational Research Association*.
- NKWOCHA F. AND ELBAUM F. 2005. Fault patterns in Matlab. *International Conference on Software Engineering, 1st Workshop on End-user Software Engineering*, St. Louis, MI, May, 1-4.
- OKADA, E.M. 2005. Justification effects on consumer choice of hedonic and utilitarian goods. *Journal of Marketing Research*, 62, 43-53.
- ONOMA K., TSAI W-T, POONAWALA M. AND SUGANUMA H. 1988. Regression testing in an industrial environment. *Communications of the ACM*, 41(5), May, 81-86.
- ORRICK E. 2006. Position Paper, *Second Workshop on End-User Software Engineering*, in conjunction with the *ACM Conference on Human Factors in Computing*, Montreal, Quebec.
- PANKO R. 1995. Finding spreadsheet errors: most spreadsheet models have design flaws that may lead to long-term miscalculation. *Information Week*, May, 100.
- PANKO R. 1998. What we know about spreadsheet errors. *Journal of End User Computing*, 2, 15-21.
- PANKO R. 2000. Spreadsheet errors: what we know. what we think we can do. *Spreadsheet Risk Symposium*, July 2000.
- PETRE, M. AND BLACKWELL, A.F. 2007. Children as unwitting end-user programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 239-242.

- PHALGUNE A., KISSINGER C., BURNETT M., COOK C., BECKWITH L., AND RUTHRUFF J.R. 2005. Garbage in, garbage out? An empirical look at oracle mistakes by end-user programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, TX, September, 45-52.
- PIPEK V. AND KAHLER H. 2006. Supporting collaborative tailoring. In *End-User Development*, H. Lieberman, F. Paterno, and V. Wulf (eds.) Springer, 315-345.
- POWELL S. G. AND BAKER K.R. 2004. *The Art of Modeling with Spreadsheets: Management Science, Spreadsheet Engineering, and Modeling Craft*, Wiley.
- PRABHAKARARAO S., COOK C., RUTHRUFF J., CRESWICK E., MAIN M., DURHAM, M., AND BURNETT M. 2003. Strategies and behaviors of end-user programmers with interactive fault localization. *IEEE Symposium on Human-Centric Computing Languages and Environments*, Auckland, New Zealand, September, 15-22.
- RAKIC M. AND MEDVIDOVIC N. 2001. Increasing the confidence in off-the-shelf components: A software connector-based approach. *ACM SIGSOFT Software Engineering Notes*, 26(3), 11-18.
- RAVICHANDRAN T. AND ROTHENBERGER M. 2003. Software reuse strategies and component markets. *Communications of the ACM*, 46(8), 109-114.
- RAZ O., KOOPMAN P. AND SHAW M. 2002. Semantic anomaly detection in online data sources. *International Conference on Software Engineering*, Orlando, Florida, May, 302-312.
- REPENNING, A. AND SULLIVAN, J. 2003. The pragmatic web: Agent-based multimodal web interaction with no browser in sight. *International Conference on Human-Computer Interaction*, Zurich, Switzerland.
- REPENNING A. AND IOANNIDOU A. 1997. Behavior processors: Layers between end users and Java virtual machine. *IEEE Symposium on Visual Languages*, Isle of Capri, Italy, September, 23-26.
- REPENNING A. AND PERRONE C. 2000. Programming by analogous examples. *Communications of the ACM*, 43(3), 90-97.
- ROBERTSON T. J., PRABHAKARARAO S., BURNETT M., COOK C., RUTHRUFF J. R., BECKWITH L., AND PHALGUNE A. 2004. Impact of interruption style on end-user debugging. *ACM Conference on Human Factors in Computing systems*, Vienna, Austria, April, 287-294.
- ROBILLARD M.P., COELHO W., AND MURPHY G.C. 2004. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30, 12, 889-903, 2004.
- RODE J. AND ROSSON M.B. 2003. Programming at runtime: Requirements and paradigms for nonprogrammer web application development. *IEEE Symposium on Human-Centric Computing Languages and Environments*, Auckland, New Zealand, September, 23-30.
- RODE J. AND BHARDWAJ Y. AND PEREZ-QUINONES M.A. AND ROSSON M.B. AND HOWARTH J. 2005. As easy as "Click": End-user web engineering. *International Conference on Web Engineering*, Sydney, Australia, July, 478-488.
- RODE J., ROSSON M.B. AND QUINONES M.A.P. 2006. End user development of web applications, In Lieberman H., Paterno F., and Wulf V. (Eds.), *End-User Development*. Springer-Verlag.
- RODE J.A., TOYE E.F., AND BLACKWELL A.F. 2004. The fuzzy felt ethnography—understanding the programming patterns of domestic appliances. *Personal Ubiquitous Computing*, 8, 3-4, 161-176.
- RODE, J.A., TOYE, E.F. AND BLACKWELL, A.F. 2005. The domestic economy: A broader unit of analysis for end-user programming. *ACM Conference on Human Factors in Computing Systems*, April, 1757-1760.
- RODE, J.A. 2008. An ethnographic examination of the relationship of gender & end-user programming, Ph.D. Thesis, University of California Irvine.
- RONEN B. AND PALLEY M.A. AND LUCAS JR. H.C. 1989. Spreadsheet analysis and design, *Communications of the ACM*, 32(1):84-93.
- ROSSON M. AND CARROLL J. 1996. The reuse of uses in Smalltalk programming. *ACM Transactions on Computer-Human Interaction*, 3(3), 219-253.
- ROSSON, M.B., CARROLL, J.M., SEALS, C., & LEWIS, T. 2002. Community design of community simulations. *Proceedings of Designing Interactive Systems*, 74-83.
- ROSSON M.B. AND CARROLL J.M. 2003. Scenario-based design. In Jacko J.A. and Sears A. (Eds.), *The Human-Computer Interaction Handbook*. Mahwah, NJ, Lawrence Erlbaum, 1032-1050.

- ROSSON M.B., BALLIN J., AND NASH H. 2004. Everyday programming: Challenges and opportunities for informal web development. *IEEE Symposium on Visual Languages and Human-Centric Computing Languages and Environments*, Rome, Italy, September, 123-130.
- ROSSON M.B., BALLIN J., AND RODE J. 2005. Who, what, and how: A survey of informal and professional web developers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, September, 199-206.
- ROSSON, M.B., KASE, S. 2006. Work, play, and in-between: Exploring the role of work context for informal web developers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 151-156.
- ROSSON, M.B., SINHA H., BHATTACHARYA, M., AND ZHAO, D. 2007. Design planning in end-user web development. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 189-196.
- ROTHERMEL G., LI L., DUPUIS C., AND BURNETT M. 1998. What you see is what you test: A methodology for testing form-based visual programs. *International Conference on Software Engineering*, Kyoto, Japan, April, 198-207.
- ROTHERMEL G., BURNETT M., LI L., DUPUIS C. AND SHERETOV A. 2001. A Methodology for testing spreadsheets. *ACM Transactions on Software Engineering Methodologies*, 10(1), 110-147.
- ROTHERMEL G., HARROLD M.J., VON RONNE J., AND HONG C. 2002. Empirical studies of test suite reduction. *Journal of Software Testing, Verification, and Reliability*, 4(2), December.
- ROTHERMEL K., COOK C., BURNETT M., SCHONFELD J., GREEN T.R.G., AND ROTHERMEL G. 2000. WYSIWYT testing in the spreadsheet paradigm: An empirical evaluation. *International Conference on Software Engineering*, Limerick, Ireland, June, 230-239.
- ROWE M.D. 1978. *Teaching Science as Continuous Inquiry: A Basic* (2nd ed.). McGraw-Hill, New York, NY.
- RUTHRUFF J., CRESWICK E., BURNETT M., COOK C., PRABHAKARARAO S., FISHER II M., AND MAIN M. 2003. End-user software visualizations for fault localization. *ACM Symposium on Software Visualization*, San Diego, California, USA, June, 123-132.
- RUTHRUFF J.R., PHALGUNE A., BECKWITH L., BURNETT M., AND COOK C. 2004. Rewarding “good” behavior: End-user debugging and rewards. *IEEE Symposium on Visual Languages and Human-Centered Computing*, Rome, Italy, September, 115-122.
- RUTHRUFF J., BURNETT M., AND ROTHERMEL G. 2005. An empirical study of fault localization for end-user programmers. *International Conference on Software Engineering*, St. Louis, Missouri, May, 352-361.
- RUTHRUFF J., PRABHAKARARAO S. REICHWEIN J., COOK C., CRESWICK E. AND BURNETT M. 2005. Interactive, visual fault localization support for end-user programmers. *Journal of Visual Languages and Computing* 16(1-2), 3-40, February/April.
- RUTHRUFF J.R., BURNETT M., AND ROTHERMEL G. 2006. Interactive fault localization techniques in an end-user programming environment. *IEEE Transactions on Software Engineering*, 32(4), 213-239, April.
- SAJANIEMI J. 2000. Modeling spreadsheet audit: A rigorous approach to automatic visualization. *Journal of Visual Languages and Computing*, 11(1), 49-82.
- SCAFFIDI C., SHAW M., AND MYERS B.A. 2005. Estimating the numbers of end users and end user programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, UK, September, 207-214.
- SCAFFIDI C., KO A.J., MYERS B., SHAW M. 2006. Dimensions characterizing programming feature usage by information workers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, UK, September, 59-62.
- SCAFFIDI C. 2007. Unsupervised inference of data formats in human-readable notation. *Proceedings of 9th International Conference on Enterprise Integration Systems, HCI Volume*, 236-241.
- SCAFFIDI C., MYERS B., AND SHAW M. 2007. Trial by water: creating Hurricane Katrina “person locator” web sites. In Weisband S., *Leadership at a Distance: Research in Technologically-Supported Work* (ed), Lawrence Erlbaum.
- SCAFFIDI C., MYERS B.A., AND SHAW M. 2008. Topes: Reusable abstractions for validating data. *International Conference on Software Engineering*, Leipzig, Germany, May 2008, 1-10.

- SEGAL J. 2005. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10, 517-536, 2005.
- SEGAL, J. 2007. Some problems of professional end user developers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, 111-118.
- SHAW M. 1995. Architectural issues in software reuse: It's not just the functionality, it's the packaging. *Symposium on Software Reusability*, Seattle, Washington, USA, April, 3-6.
- SHAW, M. 2004. Avoiding costly errors in your spreadsheets. *Contractor's Management Report* 11, 2-4.
- SMITH D., CYPHER A., AND SPOHRER J. 1994. KidSim: Programming agents without a programming language. *Communications of ACM*, 37(7), July, 54-67.
- SMITH D., CYPHER A., AND TESLER L. 2000. Programming by example: Novice programming comes of age. *Communications of the ACM*, 43(3), 75-81.
- STEVENS G., QUAISSE G., AND KLANN M. 2006. Breaking it up: An industrial case study of component-based tailorable software design. In *End-User Development*, H. Lieberman, F. Paterno, and V. Wulf (eds.). Springer, 269-294.
- STYLOS J. AND MYERS B.A. 2006. Mica: A web-search tool for finding API components and examples. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Brighton, UK, September, 195-202.
- STYLOS J., MYERS B.A., AND FAULRING A. 2004. Citrine: Providing intelligent copy-and-paste. *ACM Symposium on User Interface Software and Technology*, Santa Fe, NM, October 24-27, pp. 185-188.
- SUBRAHMANYAN N., KISSINGER C., RECTOR K., INMAN D., KAPLAN J., BECKWITH L., AND BURNETT M.M. 2007. Explaining debugging strategies to end-user programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Couer d'Alene, Idaho, Sept. 23-27, 127-134.
- SUBRAHMANYAN, N., BECKWITH, L., GRIGOREANU, V., BURNETT, M., WIEDENBECK, S., NARAYANAN, V., BUCHT, K., DRUMMOND, R., AND FERN, X. 2008. Testing vs. code inspection vs. what else?: male and female end users' debugging strategies. *ACM Conference on Human Factors in Computing Systems*, Florence, Italy, April 05 - 10, 617-626.
- SUTCLIFFE A. AND MEHANDJIEV, N. 2004. End-user development. *Communications of the ACM*, 47(9), 31-32.
- SUTCLIFFE, A. G. 2002. *The domain theory: patterns for knowledge and software reuse*. Mahwah NJ: Lawrence Erlbaum Associates.
- TALBOT, J., LEE, B., KAPOOR, A., AND TAN, D. S. 2009. EnsembleMatrix: interactive visualization to support machine learning with multiple classifiers. *ACM Conference on Human Factors in Computing Systems*, Boston, MA, April, 1283-1292.
- TASSEY, G. 2002. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, RTI Project Number 7007.011.
- TEASLEY B. AND LEVENTHAL L. 1994. Why software testing is sometimes ineffective: Two applied studies of positive test strategy. *Journal of Applied Psychology* 79(1), 142-155.
- TEXIER G., AND GUITTET L. 1999. User defined objects are first class citizens. *International Conference on Computer-Aided Design of User Interfaces*, Louvain-la-Neuve, Belgium, October, 231-244.
- TIP F. 1995. A survey of program slicing techniques. *Journal of Programming Languages* 3(3), 121-189.
- TOOMIM, M., DRUCKER, S. M., DONTCHEVA, M., RAHIMI, A., THOMSON, B., AND LANDAY, J. A. 2009. Attaching UI enhancements to websites with end users. *ACM Conference on Human Factors in Computing Systems* Boston, MA, April 1859-1868.
- TOOMIM, M., BEGEL, A., GRAHAM, S.L. 2004. Managing duplicated code with linked editing. *IEEE Symposium on Visual Languages and Human Centric Computing*, 173-180, September.
- TRIGG, R. H. AND BÖDKER, S. 1994. From implementation to design: tailoring and the emergence of systematization in CSCW. *ACM Conference on Computer Supported Cooperative Work*, Chapel Hill, North Carolina, October 45-54.
- UMARJI, M., POHL, M., SEAMAN, C., KORU, A. G., AND LIU, H. 2008. Teaching software engineering to end-users. *International Workshop on End-User Software Engineering*, Leipzig, Germany, May 40-42.

- VAN DEN HEUVEL-PANHEIZEN, M. 1999. Girls' and boys' problems: Gender differences in solving problems in primary school mathematics in the Netherlands. In Nunes T. and Bryant P. (Eds.), *Learning and Teaching Mathematics: An International Perspective*, 223-253. Psychology Press, UK.
- WALPOLE R. AND BURNETT M. 1997. Supporting reuse of evolving visual code. *IEEE Symposium on Visual Languages*, Isle of Capri, Italy, September, 68-75.
- WHITE L.J. 1987. Software testing and verification. In *Advances in Computers*. Academic Press, Boston, MA, 335-390.
- WHITTAKER, D. 1999. Spreadsheet errors and techniques for finding them. *Management Accounting* 77(9), 50-51.
- WIEDENBECK S. AND ENGBRETSON A. 2004. Comprehension strategies of end-user programmers in an event-driven application. *IEEE Symposium on Visual Languages and Human Centric Computing*, Rome, Italy, September, 207-214.
- WIEDENBECK S. 2005. Facilitators and inhibitors of end-user development by teachers in a school environment. *IEEE Symposium on Visual Languages and Human-Centric Computing*, Dallas, Texas, September, 215-222.
- WILCOX E., ATWOOD J., BURNETT M., CADIZ J., AND COOK, C. 1997. Does continuous visual feedback aid debugging in direct-manipulation programming systems? *ACM Conference on Human Factors in Computing Systems*, Atlanta, Georgia, USA, March 258-265.
- WILSON A., BURNETT M., BECKWITH L., GRANATIR O., CASBURN L., COOK C., DURHAM M., AND ROTHERMEL G. 2003. Harnessing curiosity to increase correctness in end-user programming. *ACM Conference on Human Factors in Computing Systems*, Fort Lauderdale, Florida, USA, April, 305-312.
- WOLBER D. AND SU Y. AND CHIANG Y.T. 2002. Designing dynamic web pages and persistence in the WYSIWYG interface. *International Conference on Intelligent User Interfaces*, San Francisco, California, USA, January, 228-229.
- WON M., STIEMERLING O., AND WULF V. 2006. Component-based approaches to tailorable systems. In *End-User Development*, H. Lieberman, F. Paterno, and V. Wulf (eds.) Springer, 115-141.
- WONG, J. AND HONG J.I. 2007. Making mashups with Marmite: Re-purposing web content through end-user programming. *Proceedings of ACM Conference on Human Factors in Computing Systems*.
- WULF, V. 1999. "Let's see your search-tool!"—collaborative use of tailored artifacts in groupware. *ACM SIGGROUP Conference on Supporting Group Work*, Phoenix, AZ, United States, November, 50-59.
- WULF, V. 2000. Exploration environments: Supporting users to learn groupware functions. *Interacting with Computers*, 13, pp. 265 – 299.
- WULF V. PATERNO F. AND LIEBERMAN H. 2006, Eds. *End User Development*. Kluwer Academic Publishers.
- WULF V., PIPEK V., AND WON, M. 2008. Component-based tailorability: Enabling highly flexible software applications. *International Journal of Human-Computer Studies* 66, 1-22.
- YE Y. AND FISCHER G. 2005. Reuse-conducive development environments. *International Journal of Automated Software Engineering*, 12(2), 199-235.