The State of the Art in End-User Software Engineering

ANDREW J. KO
The Information School, University of Washington
ROBIN ABRAHAM AND LAURA BECKWITH
Microsoft Corporation
ALAN BLACKWELL
The Computer Laboratory, University of Cambridge
MARGARET BURNETT, MARTIN ERWIG, AND JOSEPH LAWRANCE
School of Electrical and Engineering and Computer Science, Oregon State University
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
CHRIS SCAFFIDI AND 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 supported by computation. 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 requirements gathering, de-sign, specification, reuse, 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 challenges in de-signing EUSE tools, including the power of surprise in affecting tool use and the influence of gender.

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 US Bureau of Labor and Statistics, by 2012 there will be more than 55 million people using spreadsheets and databases at work in the USA, 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 computers 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 domain of expertise. End-user programmers might be secretaries, accountants, children [Petre and Blackwell 2007], teachers [Wiedenbeck 2005], interaction designers [Myers et al. 2008], and anyone else who finds themselves writing programs to support their work or hobbies. Programming experience is an independent concern, as shown in Figure 1. For example, despite their considerable skill, many system administrators view programming as 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 this difference in priorities from professionals, end-user programmers *do* face many of the same software engineering challenges. 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

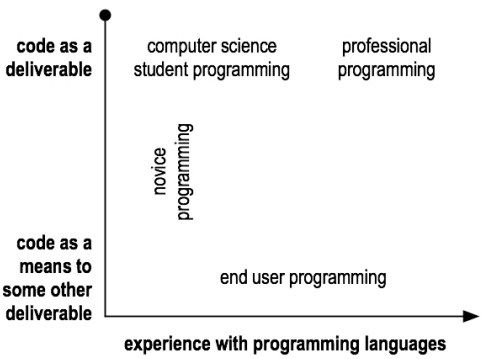

Figure 1. Programming activities along dimensions of experience and goals.

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 linked 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 software engineering technologies that collaborate with end users to improve software quality. This research area is called *end-user software engineering* (EUSE). There have been some surveys on end-user *programming* [Kelleher and Pausch 2005, Sutcliffe and Mehandjiev 2004, Lieberman et al. 2006], but to date, none has focused specifically on the topic of end-user *software engineering*.

In this article, we aim to define and organize this research area. We start by proposing definitions of programming, end-user programming, and end-user software engineering. We follow with a lifecycle-oriented treatment of end-user software engineering research, organizing more than a decade of research on requirements, design, testing, verification, and debugging. We then discuss issues surrounding the design of EUSE tools, including the power of surprise in affecting tool use and the influence of gender. We conclude with a discussion of open questions in EUSE research.

2. DEFINITIONS

A contribution of this paper 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, ending with a definition of end-user software engineering.

2.1. Programming and Programs

We define *programming* 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 can be exe-*

cuted by a computational device at some future time, on input values that can vary. 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 on a schedule, and markup languages like HTML, which are interpreted to produce a specific visual rendering of shapes and text. Our definition also captures related terms such as *customization* and *tailoring* [Eagan and Stasko 2008], which imply parameterization of existing programs.

2.2. End User Programming

We now turn to *end-user programming*, a phrase perhaps popularized by Nardi [1993] in her investigations into spreadsheet use in office workplaces. An *end user* is simply a computer user. We define *end-user programming* as *programming to achieve the result of a program, rather than the program itself.* For example, a teacher may write a grades spreadsheet to track students' test scores and a photographer might write a Photoshop script to apply the same filters to a hundred photos. In these situations, the program is a means to an end and only one of many tools used to accomplish a goal. In contrast, in professional software engineering, the goal is to produce the program itself for sale, for contract, for fun, or as a service.

An *end-user programmer* is *someone who uses a program to achieve some goal other*

Class of people	**Kinds of programs 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 BASIC
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 formal 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	Control 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

Table 1. Classes of people who write programs and the kinds of programs they write.

than the program itself. This definition captures a variety of people and their work; Table 1 gives just a glimpse of the diversity of people's computational creations. The phrase "end-user programmer" should be used with some caution, however: because the *goal* of programming is the key difference between end-user programming and other kinds of programming, it is not appropriate to say a particular *individual* is an end-user programmer. Instead, end-user programming is a role and a state of mind (and when we use the phrase "end-user programmer" in this paper, we mean it in this sense). This is a departure from previous literature's definitions of "end-user programmer" as an identity [Nardi 1993].

Because end-user programming is a role, one person can be both an end-user programmer and a professional programmer in different situations. For example, a professional programmer may create a spreadsheet to manage a start-up company's business plan, viewing the spreadsheet as a means to an end, and not applying the usual standard practices that they use at work to program, even though the quality of the spreadsheet formulas is critical to the business planning. Similarly, an end-user programmer can shift into the role of a professional programmer simply by making the *program* the primary goal, for example, by deciding to sell a program and thus maintain it and debug it for customers. In fact, whole communities have been formed around such peoples' work (e.g. http://applescriptcentral.com).

Given the somewhat inconsistent use of phrases in this research area, it is worth mentioning other phrases related to end-user programming. *End user development* [Lieberman et al. 2006] has the same basic meaning as end-user software engineering, but also implies user participation in the software development process. *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 these 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 work 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.

2.3. End User Software Engineering

With definitions of programming and end-user programming, we now turn to *end-user software engineering*. According to IEEE Standard 610.12, software engineering is

(1) the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, that is, the application of engineering to software, and (2) the study of approaches as in (1).

Although there is still much debate about the meaning of this phrase and its implications, we define *end-user software engineering* as *end-user programming involving systematic and disciplined activities that address software quality issues* (such as reliability, efficiency, usability, etc.). In essence, end-user programming focuses mainly on how to allow end users to create their own programs, and end-user software engineering considers how to support the entire software lifecycle and its attendant issues. Professional and end-user software engineering differ in how they are structured. In *professional* software engineering, the *engineering concerns* and customer needs structure the work, resulting in requirements, design, testing, issue trackers, version control systems, and other formal tools and processes. Software developers can be informal within these phases [Ko et al. 2007], but this informality is usually confined within a systematic process. In *end-user* software engineering, the end-user programmers' *goals* structure the work. Engineering concerns and their formalities occur, if at all, in the process of accomplishing other goals.

3. END-USER SOFTWARE ENGINEERING

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. We chose to organize the literature by major software engineering activities, framing the discussion from the perspective of an end user:

1. *Requirements*. What a program accomplishes, as opposed to how.
2. *Design and specifications*. How a program meets 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 errors.
5. *Debugging*. Repairing known failures.

We discuss each of these, examining both empirical and technical research across a variety of application domains, language paradigms, and research approaches. We do *not* discuss implementation issues surrounding language design or code editors (which would go between 2 and 3), since these topics have already received attention in end-user programming literature [Sutcliffe and Mehandjiev 2004, Kelleher and Pausch 2005, Lieberman et al. 2006].

3.1. What Should My Program Do? — Requirements

The term "requirements*"* refers to statements of *what* users would like a program to do, as opposed to *how* the program should do it. For example, a requirement for a tax program might be "Fill in my 1040 tax form automatically with salary numbers from my bank account." This is a statement of a desired result, but not of how the result is achieved.

In professional software engineering, projects usually involve a requirements gathering phase that results in requirements specifications. These specifications can 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 of their motivations, end users may not tolerate overhead that is not rewarded in the short term. 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 become clear in the process of implementation [Costabile et al. 2006] [Fischer and Giaccardi 2006][Mørch and Mehandjiev 2000].

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 captured (because the requirements are often their own) and more likely to change (because end users may need to negotiate such changes only with themselves).

The situation in which an end user programs also affects the *type* of requirements. For example, at the office the goal is often to automate repetitive operations, such as transferring or transforming pieces of information such as customer names, products, accounts, or documents. A decision to write a program at all corresponds directly to real investment, since time is money. Therefore, end users must estimate the costs of programming and compare those to the cost of continuing repeated manual operations [Nardi 1993]. In these contexts, 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], these 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]. 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 [Neiss 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.

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.

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].

3.2.1. Design Processes. Software design processes constrain how requirements are translated into design specifications and then implementations. Such processes are often used to ensure quality goals such as reliability and maintainability and are learned through experience or training in software engineering practices. 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, requirements and design in end-user software development are rarely separate activities. This is reflected in most design approaches that have been targeted at end-user programmers, which are largely designed 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

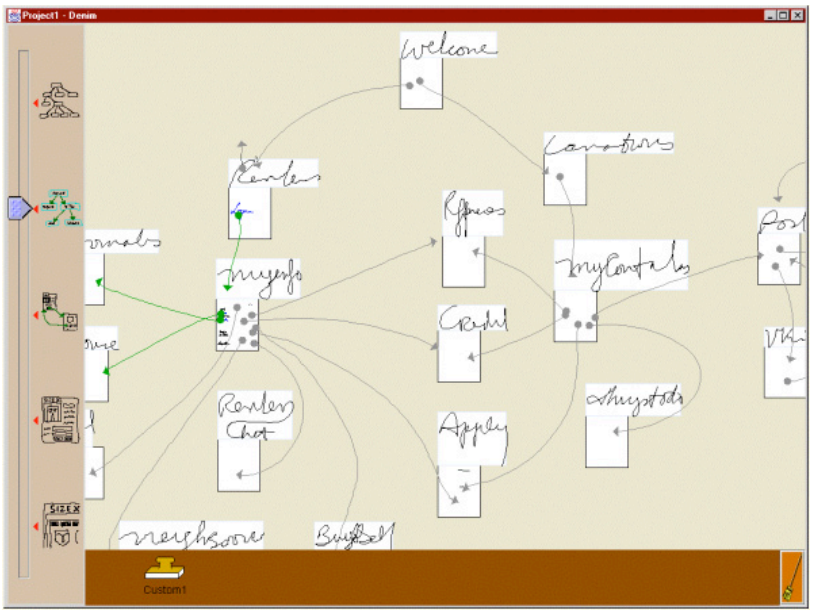

Figure 2. Links between web site content, sketched in DENIM, an informal web site sketching tool. Reprinted from [Newman et al. 2003] with permission from authors.

characteristic is called *provisionality* [Green et al. 2006], where elements of a design can be partially, and perhaps imprecisely stated.

Another approach to dealing with end users' emergent designs is to constrain what can be designed to a particular domain. The WebSheets [Wolber et al. 2002] and Click [Rode et al. 2005] environments both strive to aid users in developing web applications at a level of abstraction that allows the environment to *generate* database-driven web applications, rather than write the necessary code at a lower level.

Supporting emergent designs under changing ideas of requirements can also be done by supporting asynchronous or synchronous collaborations between professional software developers and end-user programmers. Approaches that emphasize synchronous aspects view professional developers and end-user programmers as a team (e.g., [Costabile et al. 2006, Fischer and Giaccardi 2006]). On the other hand, in strictly asynchronous approaches, the professional developer provides tailoring mechanisms for end-user programmers, thereby building in flexibility for end-user programmers to adjust the software over time as new requirements emerge [Bandini and Simone 2006, Dittrich et al. 2006, 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 and 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 against these specifications for inconsistencies. 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 Gencel [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.

Other researchers have developed end-user specification languages for privacy and

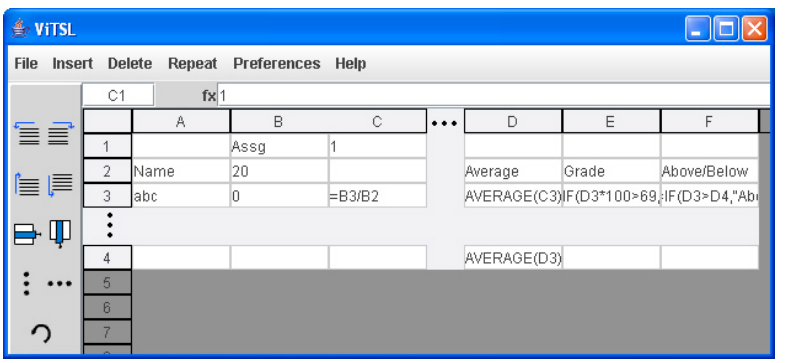

Figure 3. A ViTSL template, specifying the underlying structure of a grading spreadsheet. The names appear in rows and the assignments appear in columns, with the ellipses indicating repetition. Reproduced from [Abraham and Erwig 2006c] with permission from authors.

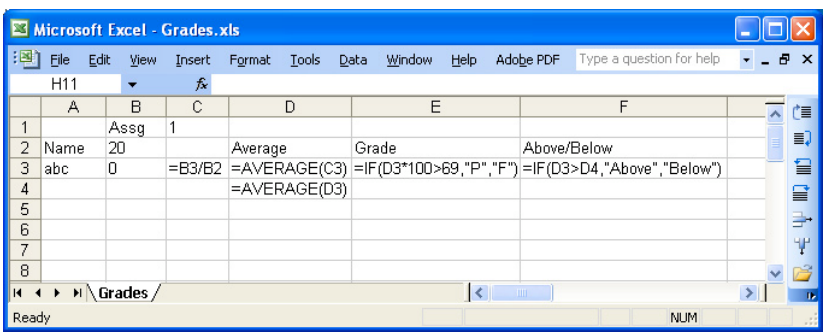

Figure 4. An instance of the grade sheet template from Figure 3 loaded into Excel. The operations in the toolbar on the right utilize the spreadsheet's underlying structure to help users avoid introducing errors into the structure. Reproduced from [Abraham and Erwig 2006c] with permission from authors.

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."

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 translate 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 express 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 a spreadsheet [Abraham and Erwig 2006a], allowing users more flexibility in redefining the spreadsheet template as requirements change. In the domain of 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. One problem with these 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 (though see [Smith et al. 2000] for a counter-example).

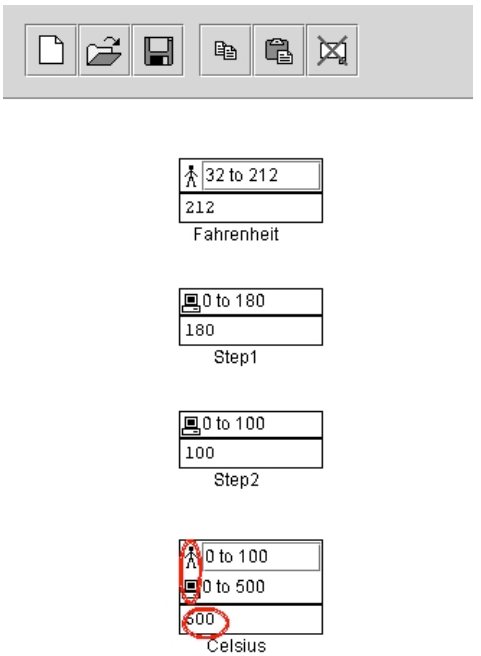

Figure 5. An assertion conflict in Forms/3. The user wrote the assertion on the Celsius cell (0 to 100), which conflicts with the computer generated assertion (0 to 500). This prompts the user to check for errors in the cells' formulas [Burnett et al. 2003]. Original figure obtained from authors.

Another way of inferring specifications is to elicit them 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 inference 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 they are familiar with and the system generates the code.

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

Reuse generally refers 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. The motivations for these various types of reuse depend on the goals of the software development activity. In professional software engineering, composition saves time and avoids the risk of writing erroneous new code [Ye and Fischer 2005, Ravichandran and Rothenberger 2003]. In practice, most of the code that professional developers write involves reuse of some sort, whether copying code snippets, adapting example code, or using APIs [Bellon et al. 2007].

In end-user software engineering, reuse is clearly 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 write new functionality [Blackwell 2002a]. This additional constraint leads to several unique reuse challenges, which we discuss in detail in this section.

3.3.1. Finding Code to Reuse. One 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 for professional programmers [Rosson and Carroll 1996, Stylos and Myers 2006] and for end-user programmers [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 search techniques may be useful in bringing similar support to end-user programmers.

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 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. For example 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. The most extreme types of templates are tailoring interfaces, where behavior can only be parameterized [Eagan and Stasko 2008].

Modifying APIs, libraries and other kinds of abstraction is more difficult, 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]).

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 substructure.
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 [Texier and Guittet 1999]	CAD	Formulas may drive values of attributes on cloneable objects.	Instances of classes may possess attributes and sub-objects.

Table 2. Behavioral and data abstractions in various end-user development environments.

Another issue for API and abstraction use is whether future versions of the abstraction will introduce mismatch because of internal changes. 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 development environments also provide ways for users to create new abstractions. *Table 2* 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].

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. 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].

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].

In other cases, choosing the right abstractions for a problem domain involves understanding the *data* used in the domain. 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 (http://pipes.yahoo.com).

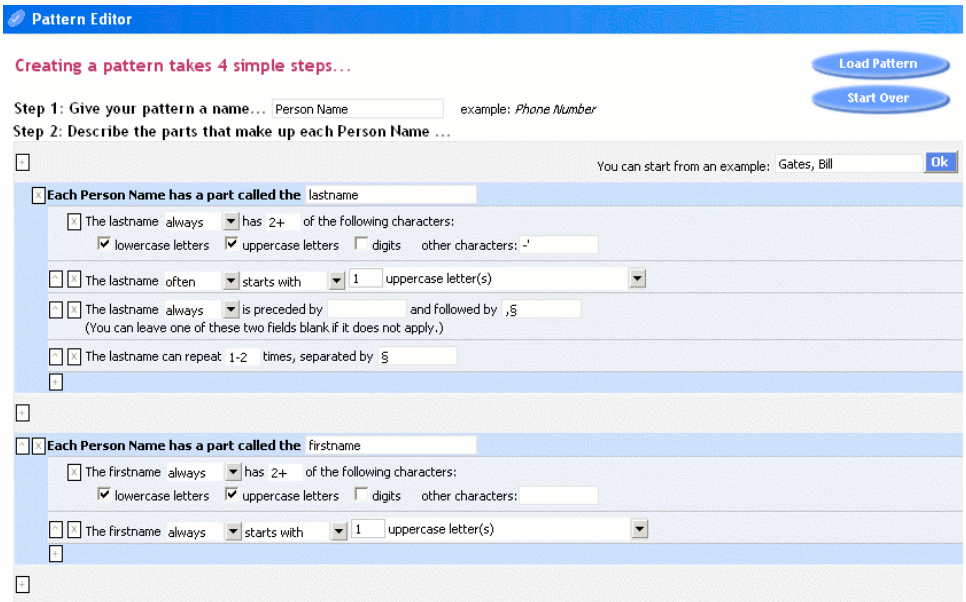

Figure 6. The Topes pattern editor [Scaffidi et al. 2008], allowing the creation of string data types that support recognition of matching strings and transformation between formats. Original figure obtained from authors.

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, whether through verification, testing, or a number of other approaches. The goal of much of the work on this problem is to enable people to have a more objective and accurate level of confidence than they would if they were left unassisted. In this section we first discuss issues of overconfidence about program correctness, and then describe technologies that help end-user programmers overcome such 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. Such oracles might be people, making decisions about the correctness of program behavior with varying degrees of formality, or can be stored known results.

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 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 from 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.

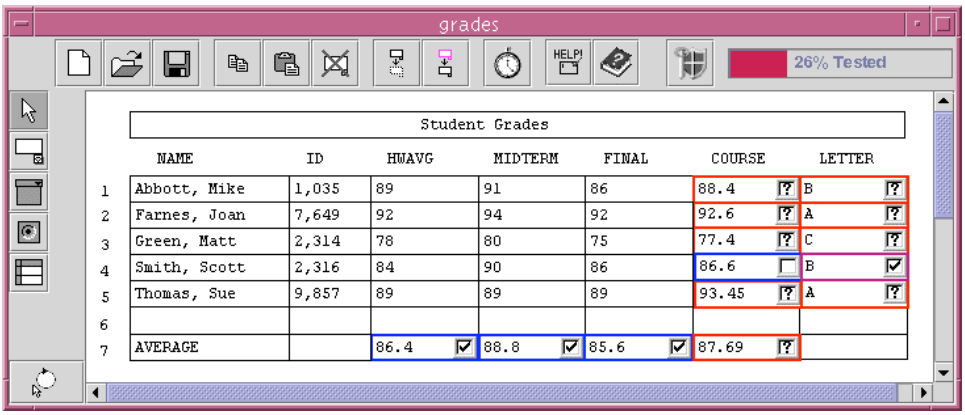

Figure 7. The WYSIWYT testing approach. Checkmarks represent decisions about correct values. Empty boxes indicate that a value has not been validated under the current inputs. Question marks indicate that validating the cell would increase testedness [Burnett et al. 2002]. Original figure obtained from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

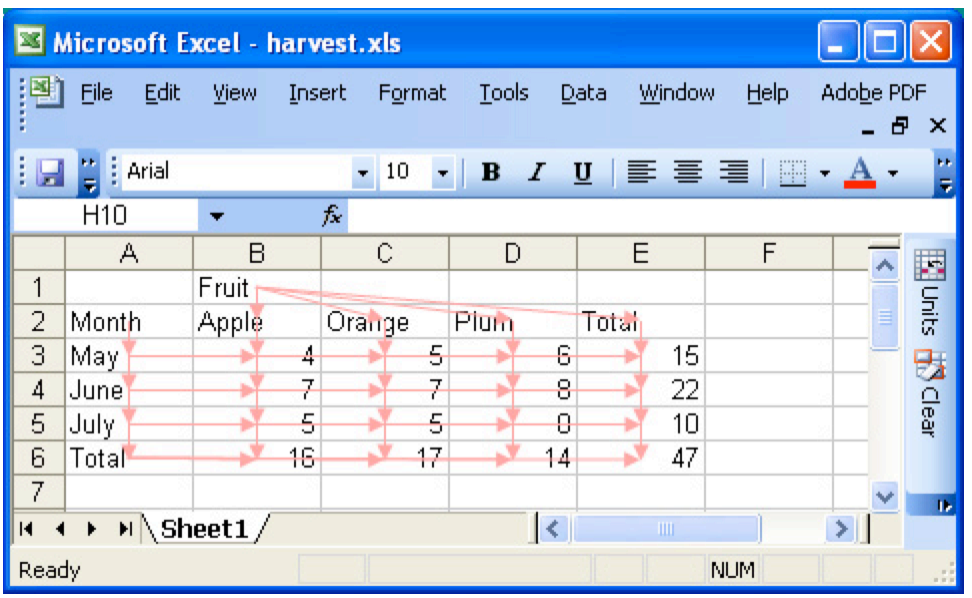

Figure 8. The UCheck system for inferring units from headers. The arrows represent unit inferences based on the column and row labels [Abraham and Erwig 2007b].

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 user 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 with higher precision and recall 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.

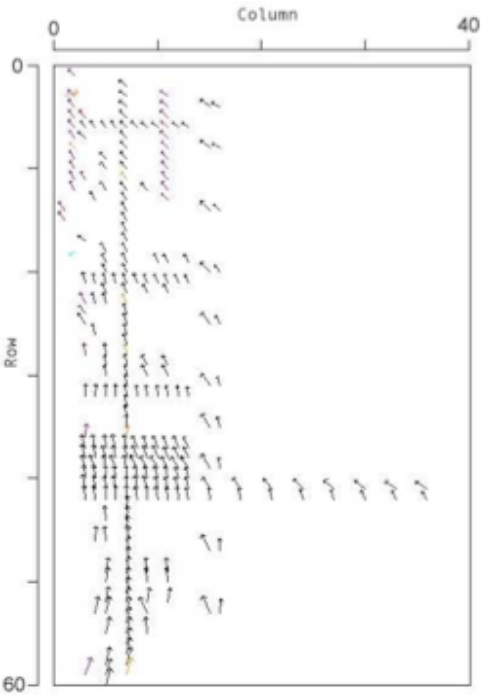

Figure 9. A "data dependency flow" of a spreadsheet's dependencies. Reproduced from [Ballinger et al. 2003] with permission from authors.

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 [Igarashi et al. 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]. All of these approaches support end-user programmers in their search for potential errors.

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].

Researchers have explored a number of techniques for addressing these challenges, beyond the traditional breakpoint debuggers and print statements common in both professional and end-user development environments. Although many of these techniques parallel research on debugging for professional programmers, there are a number of differences.

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. 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 contributed 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, users in empirical work 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.

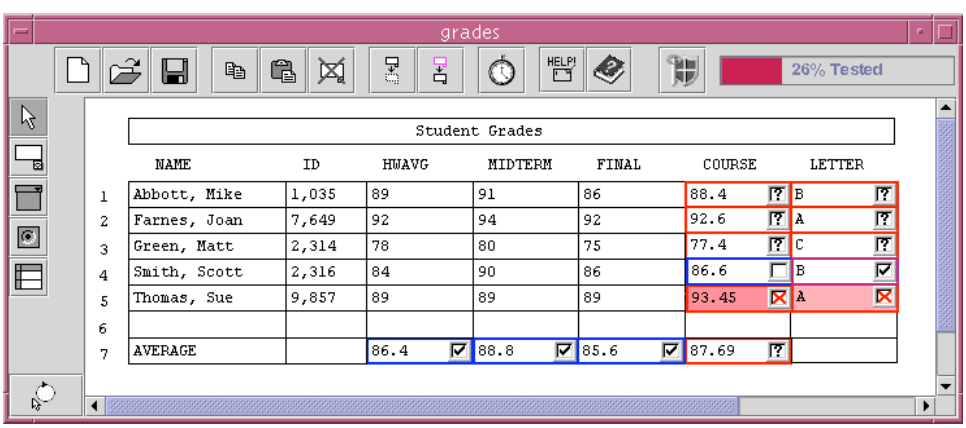

Figure 10. After the teacher marks a few successful and unsuccessful test values, the system helps her narrow down the most likely location of the erroneous formula [Ruthruff et al. 2005b]. Original figure obtained from authors.

A new class of tools based on *question asking* rather than on test outcomes has recently 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 by the 3D objects in the program and their properties and behaviors. 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.

3.5.2. Change Suggestions. An entirely different approach to debugging goes a step

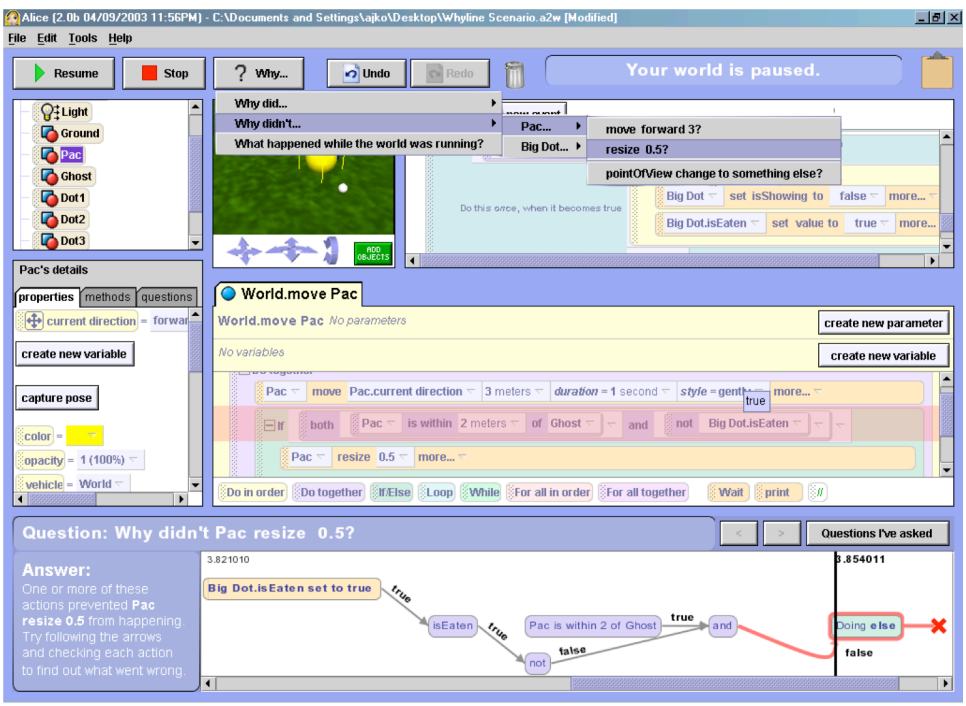

Figure 11. The Whyline for Alice. The user has asked why Pac Man failed to resize and the answer shows a visualization of the events that prevented the resize statement from executing [Ko and Myers 2004a]. Original figure obtained from authors.

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 are bound to 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. MOTIVATIONS AND PERCEPTIONS

One reason that end-user software engineering is unique is that the programming is often optional. If one needs to calculate a set of values, one can either do this manually, or one can write a program. Because of this difference, end-user programmers' perceptions about a problem and the tools that may be used to solve it, and even their perceptions about their ability to solve the problem, can influence how they use end-user software engineering tools. In this section, we discuss three areas of work covering these issues, in an effort to characterize some unique design decisions that must be made when designing end-user development environments.

4.1. Attention Investment

The first step in any end-user programming activity is deciding whether or not to undertake it. For example, web content developers may find that they need to master a style sheet language to add styles to their HTML files. For many computer users, programming does not seem to be an option. The costs involved in learning how to automate a task are so high that it always seems more economical to find a manual alternative (or to persuade someone else to write the program for you). However, for those who do undertake programming, they do so in the expectation that this investment will be rewarded and repaid through future automation. Unfortunately, this return on investment is contingent on a variety of risks. For example, it is possible that a program written to automate one task may be insufficiently general to automate the next. This is a failure of requirements engineering, of a kind that (in professional software engineering) should be avoided by more careful thought and analysis in advance.

Blackwell's Attention Investment model [Blackwell and Green 1999, Blackwell 2002] provides a cognitive model of these insights. It describes individuals' allocations of attention as cognitive "investments." According to the model, a user weighs four factors (not necessarily explicitly) before taking an action:

- *Perceived benefits*
- *Expected pay-off*
- *Perceived cost*, and
- *Perceived risks*

For example, an administrator in a small art museum might be considering adopting a spreadsheet enhancement 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. Her decision is based on a calculus of these factors.

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). End-user software engineers 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 being a valuable way to think about end-user programmers' decisions about programming, the Attention Investment model can also be useful for designers of end-user development environments and tools. Based on the four factors, the designer can reason about the tradeoffs from the viewpoint of the targeted user group (e.g., using a persona [Cooper and Reimann 2003]). A case study of using the model in this manner suggests that it is a viable design method [Blackwell and Burnett 2002].

4.2. The Surprise-Explain-Reward Strategy

One of the central challenges in designing end-user software engineering tools is motivating users to take full advantage of them. As the Attention Investment model suggests, 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].

Surprise-Explain-Reward [Wilson et al. 2003, Robertson et al. 2004, Ruthruff et al. 2004] is a tool design strategy aimed at changing end-user programmers' perceptions. 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].

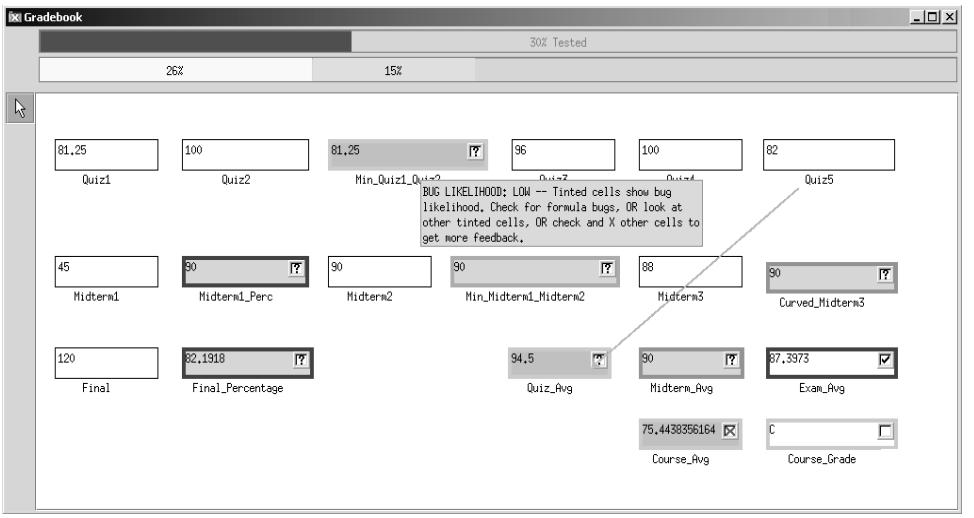

Figure 12. The Surprise-Explain-Reward strategy in Forms/3. The changing colors surprise users, the tooltips explain the potential rewards, and the further changes in colors and the percent testedness bar at the top are the rewards [Wilson et al. 2003]. Original figure obtained from authors.

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. However, empirical studies of WYSIWYT have shown that, whatever extent this behavior occurs is not enough to outweigh its effectiveness in helping users find errors [Burnett et al. 2004].

4.3. Gender and tool use

Gender has been used to describe differences in a computer use and computer science education [Busch 1995, Beyer et al. 2003, Margolis and Fisher 2003]. Researchers have now begun to look at gender differences relevant to end-user development activities [Beckwith and Burnett 2004, Grigoreanu et al. 2006, Beckwith et al. 2007, Subrahmaniyan et al. 2008, Grigoreanu et al. 2008]. The central question in these investigations is how gender and the design of end-user development environments interact to affect end-user programmers' effectiveness on programming activities.

Some investigations have considered *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.

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 debugging features less often and 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-Panheizen 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 statistical 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].

The implications of such findings on the design of end-user development 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, and that designs of 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. OPEN QUESTIONS

There has been significant progress in understanding the nature of end-user software engineering and inventing tools to support it, but as previous sections have made clear,

these areas are far from mature. In addition, there are a number of issues that remain almost completely unstudied. We highlight the latter here.

5.1. End users and professionals

First is an understanding of the population performing end-user software engineering activities. As we have shown, the population is extremely diverse, and is ephemerally tied with roles, not statically assigned to people. End-user software engineering research needs a better characterization of the roles and the situations in which end-user software engineering concerns arise. Some researchers have begun research in these directions [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].

There is also some question as to whether end-user software engineering and professional software engineering overlap so much that research on one group inevitably helps the other. For example, the Whyline [Ko and Myers 2004a], which began as a tool for end-user debugging, was successfully adapted for professional Java programmers [Ko and Myers 2008a]; it is possible that the primary challenges are not fundamental differences between the target populations of the tools, but merely issues of scale. However, even if this is the case, there are arguments for starting with end users. For example, because end users can have much higher expectations about what computers can help them achieve, researchers are forced to directly address end users' problems, rather than focusing on the issues of scale and formality, which pervade traditional software engineering research.

5.2. Formality and precision

Throughout all of the software engineering activities we have discussed, a central issue in the design of tool support has been the tensions that come from formality and precision. With more explicit requirements, tests, and verifications, come more precise analysis, but more difficulty in expression. Although this paper has demonstrated notations that are accessible and precise, there are only a few such examples. An important design concept related to formality and precision is the notion of *provisionality*. A concept proposed by Green et al. [1996], it is the ability in a tool or notation to express things tentatively or imprecisely. Thus far, it has been considered in only a few works (e.g., [Beckwith et al. 2005b, Gross and Do 1996]).

Another potential issue is whether *training* end-user programmers about software engineering and computer science principles could be more effective at improving depend-

ability than trying to design tools around end users' existing habits. While such training could always prove useful in the right circumstance, it is not always inexperience that leads end-user programmers to overlook software qualities, but a difference in priorities. Nevertheless, identifying concepts that any programmer should know is a legitimate and important goal.

5.3. Other factors influence EUSE

In our definitions, the key distinction between professional and end-user software engineering was the *motivation* for programming and not *experience*. Of course, experience and intent alone fail to capture the variety of factors that may influence success at end-user software engineering and the design of end-user software engineering tools.

For example, there are several domain factors that may lead to differences in end-user software engineering activities. 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.

There are also several contextual factors that may influence the role of software engineering in end users' software development. People can vary in their *toleration* and *perceptions* of risk and reward (as discussed in Section 4.1). For example, some teachers may not be willing to learn a new testing tool because they may not see the eventual payoff or are skeptical about their own success. A financial analyst faced with performing thousands of manual transactions may see the situation differently. 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].

Of course, these factors are not necessarily independent. Future work in end-user software engineering research should begin to identify these factors and explore their relationships as a way of predicting the software engineering needs of various end-user programming populations.

5.4. Broadening the scope

As is clear from the work discussed in this article, the most mature of the work has focused on the spreadsheet paradigm. This is a natural bias, as spreadsheets are the most broadly used end-user programming platform [Scaffidi et al. 2005]. However, as computer use and computer programming become more ubiquitous, other platforms may grow as large or larger. The web, for example, is probably already a larger platform, though there are no firm numbers on just how many people use it as a programming platform. As this article has pointed out, there is emerging work on supporting end-user programming for the web, but the end-user software engineering aspects of this work are still at very early stages.

Another issue of scope is the EUSE focus on dependability, when there are a number of other software quality attributes that end users may be concerned with. As the number of end-user programmers grows, they may want their programs to be more maintainable, usable, composable, efficient, secure, etc. Traditional software engineering researchers have investigated these and other qualities for decades, but how and whether these gains might be adapted to end-user software engineering remains an open question.

6. 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 organizes research on a number of approaches to making these programming efforts more productive and more successful. Throughout all of this work, however, 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.

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*, Edinberg, 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., LAWRANCE 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.

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*.

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.

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.

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.

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.

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.

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., LINDEBERG 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, Washingotn, 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 employ 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.

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.

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.

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., SUBRAHMANIYAN 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. 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.

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. 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.

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*.

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. 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, Herrsching 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.

NIESS, 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. 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 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., QUAISSER 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.

SUBRAHMANIYAN 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 dAlene, Idaho, Sept. 23-27, 127-134.

SUBRAHMANIYAN, 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.

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.

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 ENGEBRETSON 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., 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.