What Is End-User Software Engineering and Why Does It Matter?

Margaret Burnett

Oregon State University, School of Electrical Engineering and Computer Science,
Corvallis, Oregon, 97331 USA
burnett@eecs.oregonstate.edu

Abstract. End-user programming has become ubiquitous, so much so that there are more end-user programmers today than there are professional programmers. End-user programming empowers—but to do what? Make really bad decisions based on really bad programs? Enter software engineering's focus on quality. Considering software quality is necessary, because there is ample evidence that the programs end users create are filled with expensive errors. In this paper, I consider what happens when we add to end-user programming environments considerations of software quality, going beyond the "create a program" aspect of end-user programming. I describe a philosophy to software engineering for end users, and then survey several projects in this area. A basic premise is that end-user software engineering can only succeed to the extent that it respects the fact that the user probably has little expertise or even interest in software engineering.

Keywords: End-user software engineering, End-user programming, End-user development.

1 Introduction

It all started with end-user programming.

End-user programming enables end users to create their own programs. Researchers and developers have been working on empowering end users to do this for a number of years, and they have succeeded: today, end users create numerous programs.

The "programming environments" used by end users include spreadsheet systems, web authoring tools, and graphical languages for creating educational simulations (e.g., [6, 16, 18, 22, 23]). Using these systems, end users create programs in forms such as spreadsheets, dynamic web applications, and educational simulations. Some ways in which end users create these programs include writing and editing formulas, dragging and dropping objects onto a logical workspace, connecting objects in a diagram, or demonstrating intended logic to the system.

In fact, research based on U.S. Bureau of Census and Bureau of Labor data shows that there are about 3 million professional programmers in the United States—but over 12 million more people who say they do programming at work, and over 50 million who use spreadsheets and databases [28]. Fig. 1 shows the breakouts. Thus,

V. Pipek et al. (Eds.): IS-EUD 2009, LNCS 5435, pp. 15–28, 2009.

the number of end-user programmers in the U.S. alone probably falls somewhere between 12 million and 50 million people—several times the number of professional programmers.

Clearly then, end-user programming empowers—it has already empowered millions of end users to create their own software.

Unfortunately, there is a down side: the software they are creating with this new power is riddled with errors. In fact, evidence abounds of the pervasiveness of errors in software end users create. (See, for example, the EUSPRIG web site's 89 news stories recounting spreadsheet errors [9].) These errors can have significant impact. For example, one school faced a £30,000 shortfall because values in a budget spreadsheet had not been added up correctly [9 story # 67]. TransAlta Corporation took a $24 million charge to earnings after a bidding error caused it to buy more U.S. power transmission hedging contracts than it bargained for, at higher prices than it wanted to pay, due to a spreadsheet error [10].

Even when the errors in end-user-created software are non-catastrophic, however, their effects can matter. 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. Software resources linked by end users to monitor non-safety-critical medical conditions can cause unnecessary pain or discomfort for users who rely on them. Such problems are ubiquitous in two particularly rapidly growing types of software end users develop: open resource coalitions and dynamic web applications.

Thus, the problem with end-user programming is that end users' programs are all too often turning out to be of too low quality for the purposes for which they were created.

1.1 A New Area: End-User Software Engineering

A new research area is emerging to address this problem. The area is known as *end-user software engineering* [7], and it aims to address the problem of end users' software quality by looking beyond the "create" part of software development, which is already well supported, to the rest of the software lifecycle. Thus, *end-user programming* is the "create" part of end-user software development, and end-user software engineering adds consideration of software quality issues to both the "create" and the "beyond create" parts of software development.

More formally, Ko et al. 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" [14].

End-user software engineering is similar to the notion of *end-user development* [17], but not quite the same. According to Wikipedia, "end-user development (EUD) is a research topic within the field of computer science, describing activities or techniques that allow people who are not professional developers to create or modify a software artifact. A typical example of EUD is programming to extend and adapt an

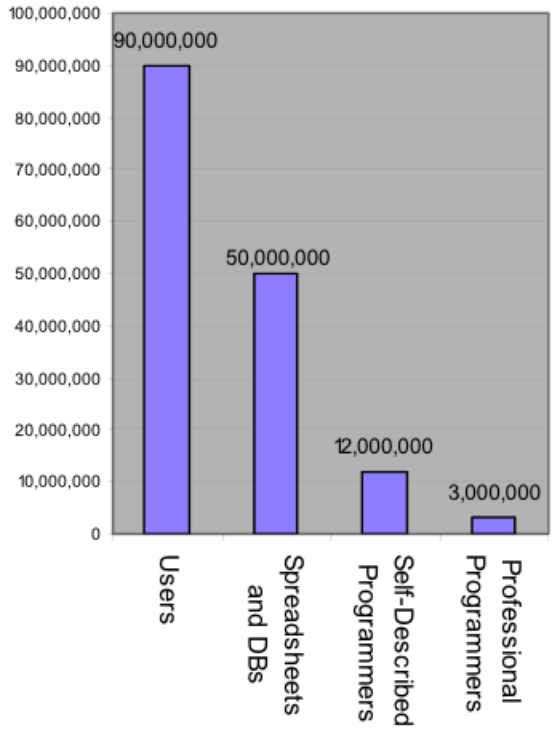


Fig. 1. U.S. users in 2006 and those who do forms of programming [28]

existing package (e.g. an office suite)" [30]. Thus, end-user software engineering is end-user development with the additional notion of the software's quality.

In my view, end-user software engineering (done well) is inherently different from traditional software engineering, because simply mimicking traditional approaches would not be likely to produce successful results. One reason is that end users often have very different training and background than professional programmers. Even more important, end users also face different motivations and work constraints than professional programmers. They are not likely to know about quality control mechanisms, formal development processes, modeling diagrams, or test adequacy criteria, and are not likely to invest time learning about such things. This is because in most cases, end users are not striving to create the best software they can; rather, they have their "real goals" to achieve: such as accounting, teaching, managing safety, understanding financial data, or authoring new media-based experiences.

The strategy my collaborators and I have used in our end-user software engineering research to support these users in pursuing their real goals has been to gently alert them to dependability problems, to assist them with their explorations into those problems to whatever extent they choose to pursue such explorations, and to work within the contexts with which they are familiar. This strategy represents a paradigm shift from traditional software engineering and end-user programming research, because it marries dependability with end-user software development. Thus, our end-user

software engineering projects combine in equal measures software engineering foundations with human-computer interaction foundations.

1.2 Organization of This Paper

I'll illustrate the end-user software engineering area with examples of projects that have been conducted by members of the EUSES Consortium (http://eusesconsortium.org), an NSF-funded collaboration of researchers working in the end-user software engineering area. The examples are:

- WYSIWYT and Surprise-Explain-Reward: WYSIWYT is a methodology for supporting systematic testing by end users. Surprise-Explain-Reward is a strategy for enticing end users to engage in software engineering practices such as the testing supported by WYSIWYT. Since WYSIWYT's success depends on Surprise-Explain-Reward, I'll discuss the two of these works together.
- Debugging Machine-Learned Programs: In recent times, a new kind of "programmer" has entered the mix—machines. These machines, through machine-learning algorithms, automatically create programs on the user's computer, deriving these programs from the user's interaction habits and data history. I'll discuss a debugging approach and early results for one type of program in this class.
- Gender in End-User Software Engineering: If end-user software engineering is to properly blend HCI-based people-oriented foundations with software engineering foundations, then it must attend to both 50%s of the people who are end users—both the males and the females. I'll discuss emerging information about gender differences' implications for the design of end-user software engineering tools.

2 WYSIWYT Testing and Surprise-Explain-Reward

WYSIWYT (What You See Is What You Test) [26] supports *systematic testing* by end-user programmers. It has mostly been implemented in the spreadsheet paradigm, so I'll present it here from that perspective. Its motivation is the following: empirical studies have shown that users often assume their spreadsheets are correct, but even if they try to consider whether there are errors, they do so by looking at the immediate value recalculations they see when they enter or change formulas. Empirical work has shown that this "one test only" feedback is tied to overconfidence about the correctness of their spreadsheets.

WYSIWYT helps to address this problem. With WYSIWYT, as a user incrementally develops a spreadsheet, he or she can also test that spreadsheet. As the user changes cell formulas and values, the underlying evaluation engine automatically evaluates cells, and the user (validates) checks off resulting values that are correct. Behind the scenes, these validations are used to measure the quality of testing in terms of a dataflow adequacy criterion, which tracks coverage of interactions between cells caused by cell references.

For example, in Fig. 2, the user has noticed that Smith's letter grade (row 4) is correct, so the user checked it off. The Average row's values under HWAVG, MIDTERM, and FINAL are also correct, so the user checks them off too. As a results, the cell borders turn closer to blue on a red-blue continuum, in which red means

untested, blue means tested, and colors between red and blue (shades of purple) mean partially tested.

But, pause to reflect: Why *should* a user whose interests are simply to get their spreadsheet results as efficiently as possible choose to spend extra time learning about these unusual new checkmarks, let alone think carefully about values and whether they should be checked off? Let's further assume that these users have never seen software engineering devices before. To succeed at enticing the user to use these devices, we require a strategy that will both motivate these users to make use of software engineering devices and provide the just-in-time support they need to effectively follow up on this interest.

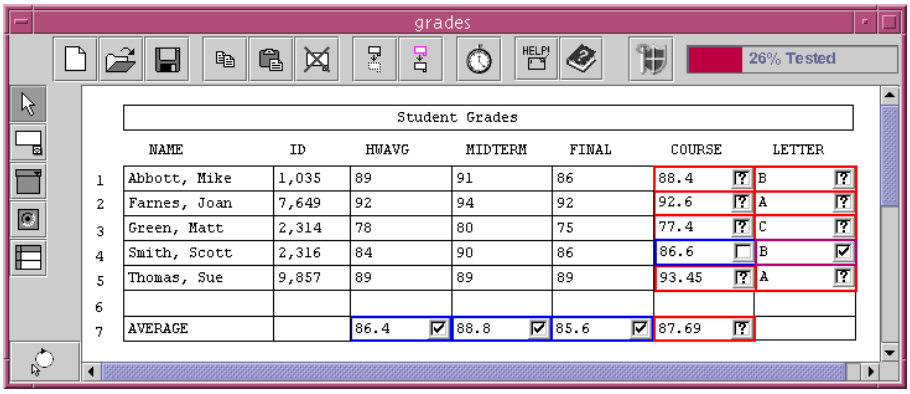


Fig. 2. At any time, the user can test by checking off a value that turned out to be correct, and this test causes borders of the cells involved to become more blue, reflecting coverage of the tests so far

We call our strategy for enticing the user down this path Surprise-Explain-Reward [31]. The strategy attempts to first arouse users' curiosity about the software engineering devices through surprise, and to then encourage them, through explanations and rewards, to follow through with appropriate actions. This strategy has its roots in three areas of research: (1) research about curiosity (psychology) [20], (2) Blackwell's model of attention investment [4] (psychology/HCI), and (3) minimalist learning (educational theory, HCI) [8].

Research into curiosity indicates that surprising by violating a user's assumptions can trigger a search for an explanation. The violation of assumptions indicates to the user the presence of something they do not understand. According to the information-gap perspective [20], a revealed gap in the user's knowledge focuses the user's attention on the gap and leads to curiosity, which motivates the user to close the gap by searching for an explanation.

This is why the first component of our surprise-explain-reward strategy is needed: to arouse users' curiosity enough, through surprise, to cause them to search for explanations. Blackwell's model of attention investment [4] considers the costs, benefits, and risks users weigh in deciding how to complete a task. For example, if a user's goal is to forecast a budget using a spreadsheet, then exploring an unknown feature has perceived costs, perceived benefits, and a perceived risk — such as that using the

new feature will waste time or, worse, leave the spreadsheet in a state from which it is difficult (and thus incurs more costs) to recover. The model of attention investment implies that the second (explanation) component of the surprise-explain-reward strategy must provide motivation by promising specific rewards (benefits). The third component must then follow through with at least the rewards that were promised.

For example, we instantiate the surprise-explain-reward strategy with the red borders and the checkboxes in each cell, both of which are unusual for spreadsheets. These surprises (information gaps) are non-intrusive: the user is not forced to attend to them if they view other matters to be more worthy of their time. However, if they become curious about these features, they can ask them to explain themselves at a very low cost, simply by hovering over them with their mouse. Thus, the surprise component delivers to the explain component.

The explain component is also very low in cost. In its simplest form, it explains the object in a tool tip. For example, if the user hovers over a checkbox that has not yet been checked off, the tool tip says (in one variant of our prototype): "*If this value is right, √ it; if it's wrong, X it. This testing helps you find errors.*" Thus, it explains the semantics very briefly, gives just enough information for the user to succeed at going down this path, and gives a hint at the reward.

As the above tool tip has pointed out, it is also possible for the user to "X out" a value that is incorrect. For example, in Fig. 3, the user has noticed two incorrect values. The system reasons about the backward slice (contributing cells and their values), taking correct values also into account, and highlights the cells in the data-flow path deemed most likely to contain the formula error. In the figure, two cells were X'd out, and those same two are highlighted, but one is highlighted darker than the other, because it was both identified as having a wrong value and also contributed to the other one that had the wrong value.

The main reward is finding errors through checking values off and X'ing them out to narrow down the most likely locations of formula errors, but a secondary reward is

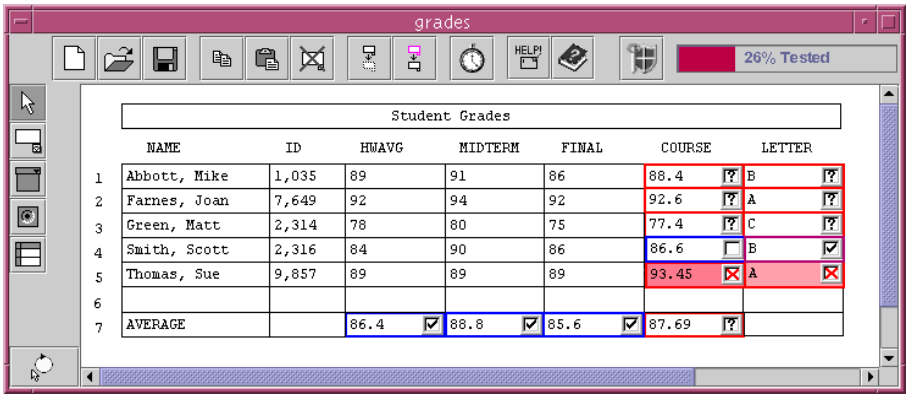


Fig. 3. If the user also notices that a value is incorrect, the user can X it out, and this causes the fault localization algorithm to suggest which cell formulas are most likely to contain the error

a "well tested" (high coverage) spreadsheet, which at least shows evidence of having fairly thoroughly looked for errors. To help achieve testing coverage, question marks point out where more decisions about values will make progress (cause more coverage under the hood, cause more color changes on the surface), and the progress bar at the top shows overall coverage/testedness so far. Our empirical work has shown that these devices are quite motivating, and further more lead to more effectiveness [27].

3 Debugging Machine-Learned Programs

But what if the program that has gone wrong was not written by a human at all? How do you debug a program that was written by a machine instead of a person?

This is the problem faced by users of a new sort of program, namely, one generated by a machine learning system that customizes itself to the user. For example, intelligent user interfaces, recommender systems, and categorizers of email use machine learning to adapt their behavior to users' preferences. This learned set of behaviors is a *program*. These learned programs do not come into existence until the learning environment has left the hands of the machine-learning specialist: they are learned on the user's computer. Thus, if these programs make a mistake, the only one present to fix them is the end user. These attempts to "fix" the system can be viewed as debugging—the user is aware of faulty system behavior, and wants to change the system's logic so as to fix the flawed behavior.

Sometimes correctness is not critical; "good enough" will suffice. For example, a spam filter that successfully collects 90% of dangerous, virus-infested spam leaves the user in a far better situation than having no spam filter at all. However, as the applications of machine learning expand, these programs are becoming more critical. For example, recommender systems that recommend substandard suppliers or incorrect parts, language translators that translate incorrectly, decision support systems that lead the user to overlook important factors, and even email classifiers that misfile important messages could cause significant losses to their users and raise significant liability issues for businesses.

My collaborators and I have begun to investigate how to support end-user debugging of machine-learned programs [15]. Inspired by the success of the Whyline's support of end-user debugging [13, 21], we designed a method to allow end users to ask Why questions of machine-learned software. Our approach is novel in the following ways: (1) it supports *end users* asking questions of machine-learned programs, and (2) the answers aim at providing suggestions for these end users to *debug* the learned programs.

We have built a prototype of our approach, so that we could investigate both barriers faced by end users when debugging machine-learned programs, and challenges to machine learning algorithms themselves. Our prototype was an e-mail application with several predefined folders. The system utilized a machine-learned program to predict which folder each message in the inbox should be filed to, thus allowing the user to easily archive messages. Our prototype answers the Why questions shown in Table 1.

Table 1. The Why questions [15]

Why will this message be filed to <Personal>?
Why won't this message be filed to <Bankruptcy>?
Why did this message turn red?
Why wasn't this message affected by my recent changes?
Why did so many messages turn red?
Why is this email undecided?
Why does <banking> matter to the <Bankruptcy> folder?
Why aren't all important words shown?
Why can't I make this message go to <Systems>?

For example, the answer to Table 1's second question (with dynamically-replaced text in <brackets>) is:

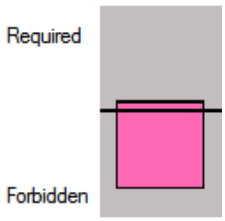
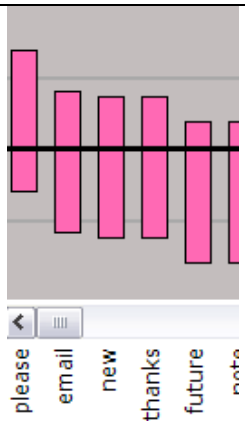
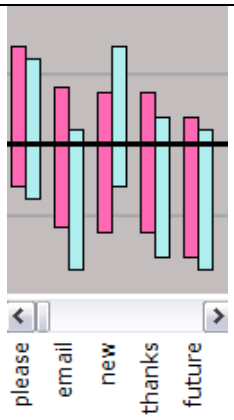
The message will be filed to <Personal> instead of <Bankruptcy> because <Personal> rates more words in this message near Required than <Bankruptcy> does, and it rates more words that aren't present in this message near Forbidden. (Usage instructions followed this text.)

In addition to the textual answers, three questions are also answered visually. These are shown in Table 2. The bars indicate the weight of each word for predictions to a given folder; the closer to Required/Forbidden, the more/less likely messages containing this word will be classified to this folder.

Fig. 4 shows a thumbnail of the entire prototype. The top half is not readable at this size, but it is simply a traditional email program. The bottom middle panel provides visual answers, shown at a readable size in Table 2.

Using this prototype, we conducted a formative empirical study to unearth barriers faced by the end user in debugging in this fashion, as well as challenges faced by machine-learning systems that generate the programs that ultimately will be debugged

Table 2. Visual explanations for three Why questions [15]

Why does <word> matter to <folder>?	Why will this message be filed to <folder>?	Why won't this message be filed to <folder>?
		

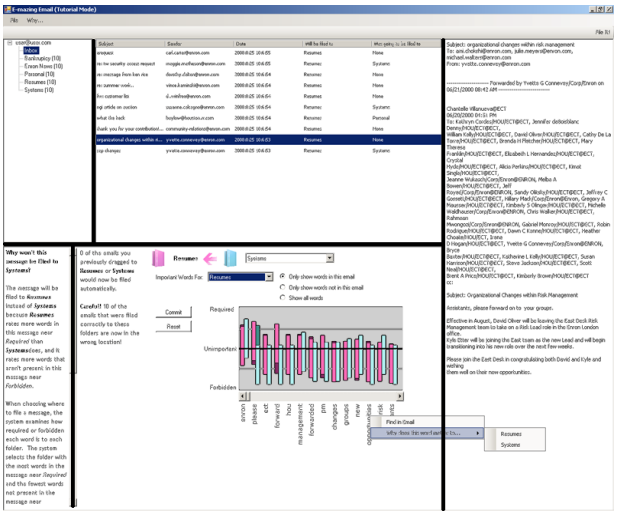


Fig. 4. A thumbnail view of the prototype [15]

by end users [15]. One of our primary results was that end users faced great difficulty in determining *where* would be the effective places to correct errors—much more so than in than in *how* to do so. The sheer number of these instances strongly suggests the value of providing end users with information about where to give feedback to the machine-learned program in order to debug effectively.

4 Gender in End-User Software Engineering

Another important result in the Kulesza et al. study was that gender differences were present in the *number* of barriers encountered, the *sequence* of barriers, and *usage* of debugging features. This is one of many studies conducted by EUSES Consortium collaborators in recent years that show gender differences in how male and female end-user programmers can best be supported in developing software effectively.

For example, evidence has emerged indicating gender differences in programming environment appeal, playful tinkering with end-user software engineering features, attitudes toward and usage of end-user software engineering features, and end-user debugging strategies [1, 2, 5, 12, 19, 24, 25, 29]. In essence, in these studies females have been shown to both use different features and to use features differently than males. Even more critically, the features most conducive to females' success are different from the features most conducive to males' success—and are the features least supported in end-user programming environments. This is the opposite of the situation for features conducive to males' success [29].

To begin to address this problem, we proposed two theory-based features that aimed to improve female performance without harming male performance [3]. We evolved these features over three years through the use of formative investigations, drawing from education theory, self-efficacy theory, information processing theory, metacognition, and curiosity theory.

Fig. 5. Clicking on the checkbox turns it into four choices whose tool tips say "it's wrong," "seems wrong maybe," "seems right maybe," "it's right." [3]

The first feature was to add "maybe" nuances to the checkmarks and X-marks of the WYSIWYT approach (Fig. 5) [3]. The empirical work leading to this change suggested that the original "it's right" and "it's wrong" checkmark and X-mark might seem too assertive a decision to make for low self-efficacy users, and we therefore added "seems right maybe" and "seems wrong maybe" checkmark and X-mark options. The change was intended to communicate the idea that the user did not need to be confident about a testing decision in order to be "qualified" to make judgments.

The second change was a more extensive set of explanations, to explain not only concepts but also to help close Norman's "gulf of evaluation" by enabling users to better self-judge their problem-solving approaches. We proposed it in [3] and then evolved that proposal, ultimately providing the strategy explanations of Fig. 6. Note that these are explanations of testing and debugging strategy, not explanations of software features per se.

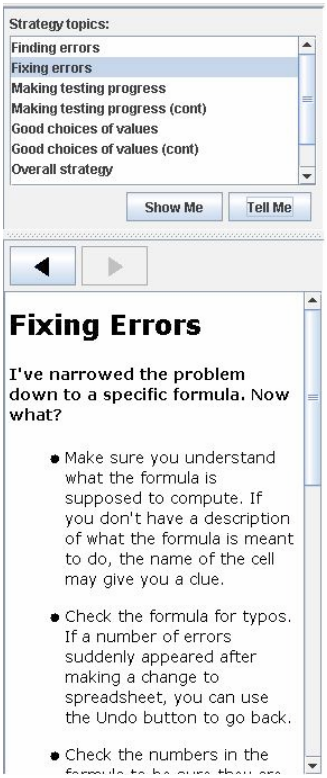


Fig. 6. (Top): 1-minute video snippets. (Bottom): Hypertext version [11].

The strategy explanations are provided as both video snippets and hypertext (Fig. 6). In each video snippet, the female debugger works on a debugging problem and a male debugger, referring to the spreadsheet, helps by giving strategy ideas. Each snippet ends with a successful outcome. The video medium was used because theory and research suggest that an individual with low self-efficacy can increase self-efficacy by observing a person similar to oneself struggle and ultimately succeed at the task. The hypertext version had exactly the same strategy information, with the obvious exception of the animation of the spreadsheet being fixed and the talking heads. We decided on hypertext because it might seem less time-consuming and therefore more attractive to users from an attention investment perspective [4], and because some people prefer to learn from text rather than pictorial content. Recent improvements to the video explanations include shortening the explanations, revising the wording to sound more like a natural conversation, and adding an explicit lead-in question to immediately establish the purpose of each explanation.

We evaluated the approach in a controlled laboratory study, in which a Control group used the original WYSIWYT system as described in Section 2 and a Treatment group used the system with the two changes just described in this system [11]. The Treatment females did not fix more bugs than Control females, but we would not expect them to: Treatment females had both lower self-efficacy than Control females and more things to take their time than Control females did. However, taking the self-efficacy and time factors into account reveals that the new features helped to close the gender gap in numerous ways.

First we found that our feature changes reduced the debugging feature *usage* gap between males and females. When we compared the males and females in the Treatment group to their counterparts in the Control group, the feature changes were tied to greater interest among the Treatment group. Compared to females in the Control group, Treatment females made more use of debugging features such as checkmarks and X-marks, and had stronger ties between debugging feature usage and strategic testing behaviors.

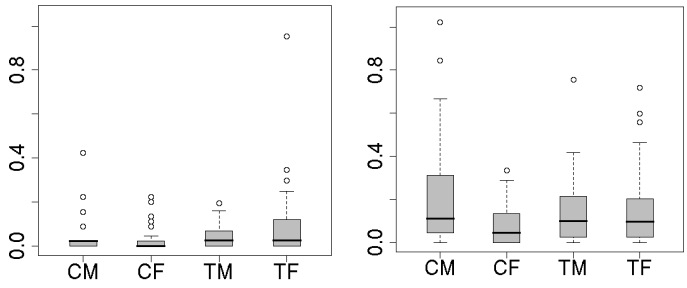


Fig. 7. Tinkering with X-marks (left) and √-marks (right), in marks per debugging minute. Note the gender gaps between the Control females' and males' medians. These gaps disappear in the Treatment group [11].

Second, we considered playful experimentation with the checkmarks and X-marks (trying them out and then removing them) as a sign of interest. Past studies reported that females were unwilling to approach these features, but that if they did choose to tinker, their effectiveness improved [1, 2]. Treatment females tinkered with the features significantly more than Control females, and this pattern held for both checkmarks and X-marks. Fig. 7 illustrates these differences.

Even more important than debugging feature usage per se was the fact that the feature usage was helpful. The total (playful plus lasting) number of checkmarks used per debugging minute, when accounting for pre-self-efficacy, predicted the maximum percent testedness per debugging minute achieved by females in both the Control group and in the Treatment group. Further, for all participants, maximum percent testedness, accounting for pre-self-efficacy, was a significant factor in the number of bugs fixed.

Finally, Treatment females' post-session verbalizations showed that their attitudes toward the software environment were more positive than Control females', and

Treatment females' confidence levels were roughly appropriate indicators of their actual ability levels, whereas Control females' confidence levels were not.

Taken together, the feature usage results show marked differences between Treatment females versus Control females, all of which were beneficial to the Treatment females. In contrast, there were very few significant differences between the male groups. Most important, none of the changes benefiting the females showed adverse effects on the males.

These results serve to reconfirm previous studies' reports of the existence of a gender gap related to the software environments themselves in the realm of end-user programming. However, the primary contribution is that they show, for the first time, that it is possible to design features in these environments that lower barriers to female effectiveness and help to close the gender gap.

5 Conclusion

End-user software engineering matters when software quality matters. End-user software engineering takes end-user programming beyond the "create" stage, expanding to consider other elements of the software lifecycle. It matters because sometimes end users' software creations have flaws, and it empowers the end users to do something about these flaws.

End-user software engineering's success rests on respecting end users' real goals and work habits. As the work in this paper illustrates, we do not advocate trying to transform end users into engineers, nor do we propose to mimic the traditional engineering approaches of segregated support for each element of the software life cycle, or even to ask the user to think in such terms. Instead, we advocate promoting systematic ways an end-user programmer can guard against and solve software quality problems through mechanisms meant especially for end-user programmers.

References

1. Beckwith, L., Burnett, M., Grigoreanu, V., Wiedenbeck, S.: Gender HCI: What About the Software? Computer, 83–87 (2006)
2. Beckwith, L., Inman, D., Rector, K., Burnett, M.: On to the Real World: Gender and Self-Efficacy in Excel. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 119–126. IEEE, Los Alamitos (2007)
3. Beckwith, L., Sorte, S., Burnett, M., Wiedenbeck, S., Chintakovid, T., Cook, C.: Designing Features for Both Genders in End-User Programming Environments. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 153–160. IEEE, Los Alamitos (2005)
4. Blackwell, A.: First Steps in Programming: A Rationale for Attention Investment Models. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 2–10. IEEE, Los Alamitos (2002)
5. Brewer, J., Bassoli, A.: Reflections of Gender, Reflections on Gender: Designing Ubiquitous Computing Technologies. In: Gender & Interaction: Real and Virtual Women in a Male World, Workshop at AVI, pp. 9–12 (2006)

6. Burnett, M., Chekka, S., Pandey, R.: FAR: An End-User Language to Support Cottage E-Services. In: Human-Centric Computing Languages and Environments, pp. 195–202. IEEE, Los Alamitos (2001)
7. Burnett, M., Cook, C., Rothermel, G.: End-User Software Engineering. Communications of the ACM 47(9), 53–58 (2004)
8. Carroll, J., Rosson, M.: Paradox of the Active User. In: Carroll, J. (ed.) Interfacing Thought: Cognitive Aspects of Human-Computer Interaction, pp. 80–111. MIT Press, Cambridge (1987)
9. EUSPRIG Spreadsheet Mistakes News Stories, http://www.eusprig.org/stories.htm
10. French, C.: TransAlta Says Clerical Snafu Costs It $24 Million. Globe and Mail (June 3, 2003)
11. Grigoreanu, V., Cao, J., Kulesza, T., Bogart, C., Rector, K., Burnett, M., Wiedenbeck, S.: Can Feature Design Reduce the Gender Gap in End-User Software Development Environments? In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 149–156. IEEE, Los Alamitos (2008)
12. Kelleher, C., Pausch, R., Kiesler, S.: Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. In: ACM Conference on Human Factors in Computing Systems, pp. 1455–1464. ACM, New York (2007)
13. Ko, A., Myers, B.: Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In: ACM Conference on Human Factors in Computing Systems, pp. 151–158. ACM, New York (2004)
14. Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Lawrance, J., Lieberman, H., Myers, B., Rosson, M.B., Rothermel, G., Scaffidi, C., Shaw, M., Wiedenbeck, S.: The State of the Art in End-User Software Engineering (submitted, 2008)
15. Kulesza, T., Wong, W., Stumpf, S., Perona, S., White, R., Burnett, M., Oberst, I., Ko, A.: Fixing the Program My Computer Learned: Barriers for End Users, Challenges for the Machine. In: ACM Conference on Intelligent User Interfaces. ACM, New York (to appear, 2009)
16. Lieberman, H. (ed.): Your Wish Is My Command: Programming By Example. Morgan Kaufmann Publishers, San Francisco (2001)
17. Lieberman, H., Paterno, F., Wulf, V. (eds.): End-User Development. Springer, Heidelberg (2006)
18. Little, G., Lau, T., Cypher, A., Lin, J., Haber, E., Kandogan, E.: Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In: ACM Conference on Human Factors in Computing Systems, pp. 943–946. ACM, New York (2007)
19. Lorigo, L., Pan, B., Hembrooke, H., Joachims, T., Granka, L., Gay, G.: The Influence of Task and Gender on Search and Evaluation Behavior Using Google. Information Processing and Management, 1123–1131 (2006)
20. Lowenstein, G.: The psychology of curiosity. J. Psychological Bulletin 116(1), 75–98 (1994)
21. Myers, B., Weitzman, D., Ko, A., Chau, D.H.: Answering Why and Why Not Questions in User Interfaces. In: ACM Conference on Human Factors in Computing Systems, pp. 397–406. ACM, New York (2006)
22. Pane, J., Myers, B., Miller, L.: Using HCI Techniques to Design a More Usable Programming System. In: Proc. IEEE Human-Centric Computing Languages and Environments, pp. 198–206. IEEE, Los Alamitos (2002)

23. Repenning, A., Ioannidou, A.: AgentCubes: Raising the Ceiling of End-User Development in Education through Incremental 3D. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 27–31. IEEE, Los Alamitos (2006)
24. Rode, J.A., Toye, E.F., Blackwell, A.F.: The Fuzzy Felt Ethnography - Understanding the Programming Patterns of Domestic Appliances. Personal and Ubiquitous Computing 8, 161–176 (2004)
25. Rosson, M., Sinha, H., Bhattacharya, M., Zhao, D.: Design Planning in End-User Web Development. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 189–196. IEEE, Los Alamitos (2007)
26. Rothermel, G., Burnett, M., Li, L., DuPuis, C., Sheretov, A.: A Methodology for Testing Spreadsheets. ACM Transactions on Software Engineering 10(1) (January 2001)
27. Ruthruff, J., Phalgune, A., Beckwith, L., Burnett, M., Cook, C.: Rewarding Good Behavior: End-User Debugging and Rewards. In: IEEE Symposium on Visual Languages and Human-Centric Computing, pp. 115–122. IEEE, Los Alamitos (2004)
28. Scaffidi, C., Shaw, M., Myers, B.: Estimating the Numbers of End Users and End User Programmers. In: IEEE Symp. Visual Lang. Human-Centric Computing, pp. 207–214. IEEE, Los Alamitos (2005)
29. Subrahmaniyan, N., Beckwith, L., Grigoreanu, V., Burnett, M., Wiedenbeck, S., Narayanan, V., Bucht, K., Drummond, R., Fern, X.: Testing vs. Code Inspection vs.. What Else? Male and Female End Users' Debugging Strategies. In: ACM Conference on Human Factors in Computing Systems, pp. 617–626. ACM, New York (2008)
30. Wikipedia, End-User Development, http://en.wikipedia.org/wiki/End_user_development
31. Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., Rothermel, G.: Harnessing Curiosity to Increase Correctness in End-User Programming. In: ACM Conference on Human Factors in Computing Systems. ACM, New York (2003)