End-User Software Engineering and Distributed Cognition

Margaret Burnett, Christopher Bogart, Jill Cao,
Valentina Grigoreanu, Todd Kulesza, Joseph Lawrance

Oregon State University, Corvallis, OR, USA
{burnett,bogart,caoch,grigorev,kulesza,lawrance}@eecs.oregonstate.edu

Abstract

End-user programmers may not be aware of many software engineering practices that would add greater discipline to their efforts, and even if they are aware of them, these practices may seem too costly (in terms of time) to use. Without taking advantage of at least some of these practices, the software these end users create seems likely to continue to be less reliable than it could be. We are working on several ways of lowering both the perceived and actual costs of systematic software engineering practices, and on making their benefits more visible and immediate. Our approach is to leverage the user's cognitive effort through the use of distributed cognition, in which the system and user collaboratively work systematically to reason about the program the end user is creating. This paper demonstrates this concept with a few of our past efforts, and then presents three of our current efforts in this direction.

1. Introduction

This workshop's call for papers stated its specific focus to be on "the software engineering that is required to make <end-user programming> a more disciplined process, while still shielding the end user from the complexities of greater discipline" (http://www.sei.cmu.edu/isis/workshops/seeup-2009/). This is an interesting issue from the perspective of the user's time and priorities. What might cause end-user programmers to become "more disciplined," and how would this impact their cost-benefit trade-offs of investing the time to do so versus the time/trouble they might save by doing so?

Our position is that we do not expect end-user programmers to voluntarily elect to become more disciplined unless doing so either (1) is perceived by users to have obvious pay-offs given their own priorities or (2) is so low in cost, they can afford to become more disciplined without worrying about the time cost.

To keep the cost of discipline low, end-user software engineering must be a collaboration between the system and the user[1]. The system's roles are to pay much of the cost of adding discipline and to make clear low-cost steps the user can perform to take advantage of that discipline and the benefits of doing so. We hypothesize that, if the user perceives reasonably low costs and useful benefits, the disciplined approaches suggested by the system will often seem more attractive than ad-hoc approaches, and users will follow them. Our previous work along these lines empirically supports this hypothesis.

*Distributed cognition "*extends the reach of what is considered cognitive beyond the individual to encompass interactions between people and with resources and materials in the environment" [8]. In system-user collaborations to support the direction we have just described, by definition, the user does some of the reasoning and the system does some of the reasoning. The system's contribution to this reasoning may be simple, such as helping users remember judgments they have made so far, or complex, such as performing static or dynamic analysis of source code to deduce possible errors. The rest of this paper provides examples as to how such distributed cognition approaches can be incorporated into end-user software development environments to encourage greater discipline by end-user programmers.

[1] unless there are professional software developers involved to provide the discipline, as in the work of Fischer and Giaccardi [6] and Costabile et al. [5].

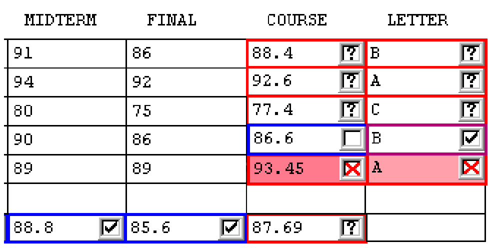
2. Examples from our Previous Work in End-User Software Engineering

Our *What You See Is What You Test* (WYSIWYT) methodology for testing spreadsheets [3, 16] demonstrates how distributed cognition can augment end users' abilities to use more disciplined approaches. In the case of WYSIWYT, the increased discipline is in testing and debugging.

With WYSIWYT, as a user incrementally develops a spreadsheet, he or she can also test that spreadsheet incrementally yet systematically. The basic idea is that, at any point in the process of developing the spreadsheet, the user can validate any value that he or she notices is correct. Behind the scenes, these validations are used to measure the quality of testing in terms of a test adequacy criterion. These measurements are communicated by visual decorations to reflect the new "testedness" state of the spreadsheet, to encourage users to direct their testing effort to the cells the system has systematically identified as needing the most attention.

For example, suppose that a teacher is creating a student grades spreadsheet, as in Figure 1. During this process, whenever the teacher notices that a value in a cell is correct, she can check it off ("validate" it). The result of the teacher's validation action is that the colors of the validated cell's borders become more blue, indicating that data dependencies between the validated cell and cells it references have been exercised in producing the validated values.

A red border means untested, a blue border means tested, and shades of purple (i.e., between red and blue) mean partially tested. From these border colors, the teacher is kept informed of which areas of the spreadsheet are tested and to what extent. Thus, in the figure, row 4's Letter cell's border is partially blue (purple), because some of the dependencies ending at that cell have now been tested. Testing results also flow upstream against dataflow to other cells whose formulas have been used in producing a validated value. In our example, all dependencies ending in row



MIDTERM	FINAL	COURSE	LETTER
91	86	88.4 ?	B ?
94	92	92.6 ?	A ?
80	75	77.4 ?	C ?
90	86	86.6 ☐	B ✓
89	89	93.45 ☒	A ☒
88.8 ✓	85.6 ✓	87.69 ?	

Figure 1: WYSIWYT supports systematic testing for end users, to help the user test and debug spreadsheet formulas [16].

4's Course cell have now been exercised, so that cell's border is now blue.

The border colors support distributed cognition by remembering (and figuring out and updating) a "things to test" list for the teacher. This distributed cognition allows the teacher to test in a more disciplined way than she might otherwise do, because it constructs its things-to-test statuses using a formal test adequacy criterion (du adequacy) that the user is not likely to know about.

The checkboxes further support distributed cognition by remembering for the user the specifics of testing that was done. Here the checkmark reminds the teacher that a cell's value has been validated under current inputs. As with the border colors, the distributed cognition goes further than just remembering things done directly—it also manages the "things tested" set by changing the contents of the checkboxes when circumstances change. For example, an empty checkbox indicates that the cell's value was validated, but the value was different than the one currently on display. Finally, the system helps the teacher manage her testing strategy by showing a question mark where validating the cell would increase testedness.

Checkmarks and border colors assist cognition about things to test and things tested successfully. There is also a fault localization functionality for things tested *un*successfully. For example, suppose our teacher notices that row 5's Letter grade is erroneous, which she indicates by X'ing it out instead of checking it off. Row 5's Course average is obviously also erroneous, so she X's that one too. As Figure 1 shows, both cells now contain pink interiors, 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 colorings are another example of distributed cognition. They make precise the teacher's reasoning/recollection about cell formulas that could have contributed to the bad value and direct her attention to the most implicated of these, thereby encouraging her to systematically consider all the possible culprits in priority order to find the ones that need fixing.

Recall our hypothesis from Section 1 that it is necessary to keep the cost of discipline low. WYSIWYT gives us a vehicle for considering the cost of discipline: Just why *would* 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?

To succeed at enticing the user to use discipline, we require a strategy that can motivate these users to make use of software engineering devices, can provide the just-in-time support they need to effectively follow up on this interest, and will not require the user to spend undue time on these devices.

We call our strategy for enticing the user down this path Surprise-Explain-Reward [21]. 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: research about curiosity [13], Blackwell's model of attention investment [2], and minimalist learning theory [4].

The red borders and the checkboxes in each cell, both of which are unusual for spreadsheets, are therefore intended to surprise the user, to arouse curiosity. These surprises are non-intrusive: users are 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, users can ask the colors or checkboxes to explain themselves at a very low cost, simply by hovering over them with their mouse. Thus, the surprise component delivers the user to the explain component.

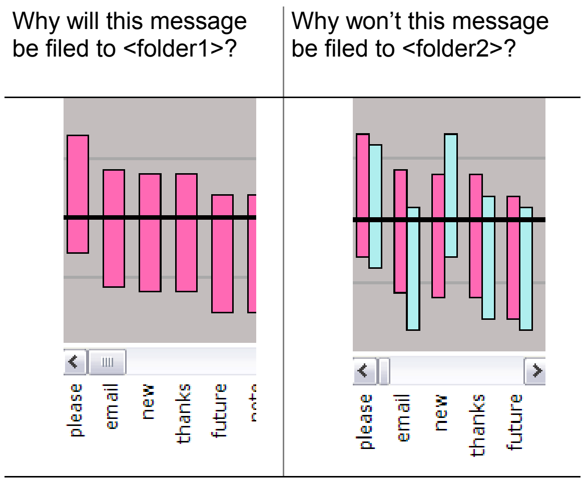

Why will this message be filed to <folder1>?
Why won't this message be filed to <folder2>?
please
email
new
thanks
future
note
please
email
new
thanks
future

Figure 2: An interactive visualization for showing end users the relative importance of different words to a machine-learned program's decision-making. For example, the word "email" is fairly neutral in classifying messages to folder1 (pink), but is forbidden for folder2 messages (blue). Users are able to drag any bar up or down to explicitly change the logic of the learned program [10].

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: "*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.

The main reward is finding errors, which is achieved by checking values off and X'ing them out to narrow down the most likely locations of formula errors. A secondary reward is 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 both motivating and that they lead to more effectiveness [3, 17].

3. Current Research Directions

3.1 More disciplined debugging of machine-learned programs

The recent increase in machine learning's presence in a variety of desktop applications has led to a new kind of program that needs debugging by the end user: programs written (learned) by machines. Since these learned programs are created by observing the user's data and reside on the user's machine, the only person present to fix them if they go wrong is the user.

Traditional methods for end users to improve the logic of machine-learned programs have been restricted to re-labeling the output of these programs. For example, imagine a movie recommendation system that uses machine-learning techniques to make intelligent recommendations based on a user's previously viewed movies. This system allows the user to label the suggestions by marking each as something they are either interested in or not interested in. Such debugging, however, is ad hoc. The user can neither know how many recommendations to label before the system will improve nor know how far-reaching an observed improvement is, and therefore, cannot plan or be strategic: the entire process is based solely on the luck of just the right inputs arriving in a timely way.

We are working on a more disciplined approach to debugging machine-learned programs to remove this complete reliance on fortuitous inputs arriving. In our approach, end users directly fix the logic of a learned program that has gone wrong, and the machine keeps them apprised of the greater impacts of their fixes. Two keys to our approach are (a) an interactive explanation of the learned program's logic and (b) a machine-learning algorithm that is capable of accepting and integrating user-provided feedback. The approach is more disciplined than traditional approaches in that it removes much of the element of luck in the arrival of suitable inputs, and also employs distributed cognition devices to enable users to predict the impacts of their logic changes.

We have designed and implemented our first prototype [10] aimed at meeting these goals. The domain we used for this prototype was automatically filing the user's incoming email messages to the correct folder. Our approach was inspired by the Whyline [9]. In our variant of why-oriented debugging, users debug through viewing and changing answers to "why will" and "why won't" questions. Examples of the questions and manipulable answers in our prototype are shown in Figure 2. As soon as the user manipulates these answers, the system not only updates the result of the input they are working with (in this case, the particular email message), but also apprises them of how other messages in their email system would be categorized differently given these changes, in essence running a full regression test.

Using this prototype, we analyzed barriers end users faced when attempting to debug using such an approach [10]. The most common obstacle for users was determining *which sections* of the learned program's logic they should modify to achieve the desired behavior changes on an ongoing basis. The complete set of barriers uncovered is being used to inform the design of our continuing work in enabling end users to debug programs that are learned from that particular user's data.

3.2 Strategies as agents of discipline for males and females

Given their lack of formal training in software engineering, end-user programmers who attempt to reason about their programs' correctness are likely to do so in an ad-hoc way rather than using a systematic strategy. *Strategy* refers to a reasoned plan or method for achieving a specific goal. It involves intent, but the intent may change during the task. Until recently, little has been known about the strategies end-user programmers employ in reasoning about and debugging their programs. We have been working to help close this gap, and to devise ways to better support end-user programmers' strategic efforts to reason about program correctness.

The WYSIWYT approach described in Section 2 promotes debugging strategies based on testing. One problem with testing-based strategies is that they do not seem to be equally attractive to male and female end-user programmers. In a recent study, we found males both preferred testing-based strategies more, and were more effective with them, than females [18]. This was also the case for dataflow strategies (Figure 3). On the other hand, the same study showed that code (formula) inspection strategies were more effective for females than for males. Gender differences in approaches to end-user software development have also been reported in debugging feature usage [1] and in end-user web programming [15].

In fact, of the eight debugging strategies we learned about in our study of spreadsheet work—Testing, Code Inspection, Specification Following, Dataflow, To-Do Listing, Color Following, Formula Fixing, and Spatial—seven (all but Spatial) had gender differences in ties to success at fixing spreadsheet formula errors [18]. In a follow-up study on strategies employed by a different population at the border between end-user programmers and professional developers, namely IT professionals debugging system administration scripts, the results on what debugging strategies were used were nearly the same, with a few additions due to differences in resources and paradigm [7]. The resulting list of ten end-user debugging strategies is shown in Table 1.

We are now beginning to explore how to explicitly support strategies that seem particularly attractive to one or the other gender, but are not yet well supported in end-user software development environments. For example, females' most effective strategies, namely Code Inspection, To-Do Listing, and Specification Checking, are not supported in spreadsheet software [18]. One example of an approach to supporting code inspection would be adding an "inspectedness" state to cell formulas, similar to the "testedness" state supported by WYSIWYT. This way, distributed

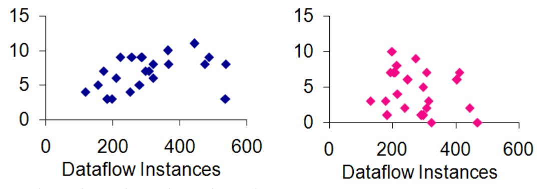


Figure 3: Correlation between total bugs fixed and number of dataflow following instances. Left: male (significant), right: female (not significant) [18].

Table 1. Strategies in finding/fixing bugs [7].

Strategy	Definition
Direct Matches	
Testing	Trying out different values to evaluate the resulting values.
Code Inspection	Examining code to determine its correctness.
Specification Checking	Comparing descriptions of what the script should do with the script's code.
Dataflow	Following data dependencies.
Spatial	Following the spatial layout of the code.
Generalized Matches	
Feedback Following	Using system-generated feedback to guide their efforts.
To-Do Listing	Indicating explicitly the suspiciousness of code (or lack of suspiciousness).
New Strategies	
Control Flow	Following the flow of control (the sequence in which instructions are executed).
Help	Getting help from people or resources.
Proceed as in Prior Experience	Recognizing a situation (correctly or not) as one experienced before, and using that prior experience as a blueprint of next steps to take.

cognition in the system environment could help users track which *formulas* have been inspected and judged correct or incorrect. (The spreadsheet auditing product described in [19] shows one possible approach for this.)

In the study of IT professionals' debugging, one interesting way in which females used Code Inspection effectively was by looking up examples of similar formulas to *fix* errors in the spreadsheet, after already having *found* the error [7]. This is a repurposing of code inspection for debugging purposes that has little support in debugging tools, either for professional developers or for end-user programmers. An idea along these lines that we are exploring is that part of the cognitive effort of searching for and memorizing related formulas could be reduced by offloading to the software the task of finding related formulas and displaying them (external memory).

Thus, the goal of this work is to encourage the use of disciplined, strategy-based problem solving by end-user programmers through distributed cognition approaches that support a variety of strategies. We hypothesize that such support will increase the discipline used in end-user programmers' problem-solving and, as a result, will increase male and female end-user developers' productivity and success at debugging their programs.

3.3 How information foraging theory can inform tools to promote discipline

The theme of this paper is that distributed cognition can help promote discipline for end-user programmers, but beyond this basic point, it would be helpful to developers of tools for end-user programmers to have guidance that is more concrete and prescriptive. We believe information foraging theory can provide such guidance.

Information foraging theory models a person's search for information as a hunt for *prey*, guided by *scent*. The *prey* is the information they are seeking. The *scent* is the user's estimate of relevance, which the user derives from cues in the environment. The hunt for relevant information then proceeds from location to location within that environment, each time following the most salient cue. Thus the *topology* of that information space and the *scent* of the cues predict how well the user will be able to navigate to the most needed information. Information foraging theory was proposed as a general model of information seeking [14], but has primarily been applied to web browsing. We have been researching the applicability of this theory to people's information-seeking behavior in software maintenance. In our studies to date on professional programmers working in Java, information foraging theory predicted people's navigation behavior when debugging as well as the aggregate human wisdom of a dozen programmers, and it was also effective at picking the right locations to focus on when debugging [11, 12].

Because cues are externalizations of scent (relevance), following cues is a strategic way to eliminate large portions of the code that must be considered in tracking down a bug. Cues can be found in both the GUI and in the software artifacts themselves. For example, variable names, component names, pictorial icons that seem to represent the relevant functionalities, are all cues. Tool feedback, such as the highlighted cells of WYSIWYT's fault localization, are the system's way of enhancing distributed cognition about where to navigate. The user can also enhance this distributed cognition through what Pirolli and Card term *enrichment*.

Pirolli defines enrichment as the extra work an information seeker does to enhance the information density or topology of the space they are working in.

Pirolli [14] studied an analyst flipping through a pile of magazines, cutting out articles to examine more closely later. The information density of this smaller pile of articles would make later information seeking go more quickly. Many software engineering tools and practices are about enrichment. Professional developers comment code, draw UML diagrams, write specifications, document changes, and link these all together to create an information topology that allows developers to more quickly get to useful information about the program. Some enrichment is informal as well, such as maintaining "to do" lists and sketching diagrams. These little notes and diagrams users create in a working space are in essence a new user-defined patch whose purpose is to help the user process information quickly. In these ways, enrichment adds to distributed cognition.

These constructs of information foraging theory—scent, cues, topology, and enrichment—thus suggest a design strategy for tools aimed at encouraging discipline in end-user programmers. First, identify what questions the tool is trying to support. Then, given these questions, choose the scent (form of relevance) that the tool will support in order to answer them. Then choose a topology and cues to allow following that scent.

For example, if the question being supported is "Why did…", the relevant scent could be the trail of dynamic state, the cues could be "invoked" edges between each called function/component as well as their names, and the topology could connect these cues to the function's states at the time they were called so that the user can step along the call sequence with each function's details (as in the Whyline [9]).

Topology and choice of cues are interdependent. The topology needs to allow a user to navigate to the relevant information called out by the cues, and the cues (and thus distributed cognition) need to emanate scent to attract the user down appropriate paths in the topology. Finally, the system should allow the user to easily enrich the cues and topology to further enhance their working environment's distributed cognition.

Tools based on information foraging theory could also promote program maintainability. For example, tools based on information foraging theory could evaluate and suggest improvements to words, labels, pictures, and explicit connections in programs and their associated artifacts, so that cues in these artifacts would emanate stronger and more precise scent.

In the service of reuse, tools based on information foraging theory could serve as distributed cognition elements connected to cues, topology, or enrichment, to help people get answers to reuse questions [20] such as: (1) I know generally what I want; which components are relevant? (2) There is a component that I've used before from this repository, but I forget the name and several other details of it; where is it? (3) This component is not quite what I need; which other components are similar?

As these examples show, information foraging theory provides a basic foundation from which design ideas can be derived on how to promote disciplined approaches to navigation-oriented needs in end-user software development.

4. Conclusion

As we have shown, tools based on distributed cognition can promote more disciplined behavior by end-user programmers. Distributed cognition works because it allows the system to contribute part of the reasoning and memory, so that users do not have to do everything in their own heads in order to follow disciplined approaches. Our empirical results over the years have provided encouraging evidence that this approach can not only encourage software engineering discipline, but can do so in a way that tears down some barriers that, in current tools, seem to disproportionately target female end-user programmers. One key to reaping these benefits is keeping the costs of using these tools low, and another is keeping the benefits to the targeted users high, so that users' perception of the costs/benefits/risks involved will make them *want* to use these devices.

5. Acknowledgments

This work was supported in part by an IBM Faculty Award, by the Air Force Office of Scientific Research, and by NSF ITR-0325273 and IIS-0803487. We are grateful to our colleagues in the EUSES Consortium for helpful discussions, feedback, and ideas, and to Laura Beckwith, Rachel Bellamy, Curt Cook, Paul ElRif, Xiaoli Fern, Mark Fisher, Cory Kissinger, Andrew Ko, Vaishnavi Narayanan, Ian Oberst, Kyle Rector, Stephen Perona, Gregg Rothermel, Joseph Ruthruff, Amber Shinsel, Simone Stumpf, Neeraja Subrahmaniyan, Susan Wiedenbeck, Weng-Keen Wong, and our former students and colleagues for their contributions to this work.

6. References

[1] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings, "Effectiveness of End-User Debugging Software Features: Are There Gender Issues?" *ACM Conference on Human Factors in Computing Systems*, Portland, Oregon, USA, 2005, pp. 869-878.

[2] A. Blackwell, "First Steps in Programming: A Rationale for Attention Investment Models," *IEEE Symposium on*

Human-Centric Computing Languages and Environments, Arlington, Virginia, USA, Sept. 2002, pp. 2-10.

[3] M. Burnett, C. Cook, and G. Rothermel, "End-User Software Engineering," *Communications of the ACM* 47(9), ACM Press, Sept. 2004, pp. 53-58.

[4] J. M. Carroll and M. B. Rosson, "Paradox of the Active User" (J. M. Carroll Ed.), *Interfacing Thought*, MIT Press, Cambridge, Massachusetts, USA, 1987, pp. 80-111.

[5] M. F. Costabile, D. Fogli, P. Mussio, and A. Piccinno, "End-User Development: The Software Shaping Workshop Approach" (H. Lieberman, F. Paterno, V. Wulf Eds.), *End-User Development*, Springer, Dordrecht, Netherlands, 2006, pp. 183-205.

[6] G. Fischer and E. Giaccardi, "Meta-Design: A Framework for the Future of End-User Development" (H. Lieberman, F. Paterno, V. Wulf Eds.), *End-User Development*, Springer, Dordrecht, Netherlands, 2006, pp. 427-457.

[7] V. Grigoreanu, J. Brundage, E. Bahna, M. Burnett, P. ElRif, and J. Snover, "Males' and Females' Script Debugging Strategies," *International Symposium on End-User Development*, Siegen, Germany, published as *Lecture Notes in Computer Science 5435* (V. Pipek et al., eds.), Springer-Verlag, Mar. 2009, to appear.

[8] J. Hollan, E. Hutchins, and D. Kirsh, "Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research," *ACM Trans. Computer-Human Interaction* 7, 2000, pp. 174-196.

[9] A. Ko, and B. Myers, "Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior," *ACM Conference on Human Factors in Computing Systems*, Vienna, Austria, Apr. 2004, pp. 151-158.

[10] T. Kulesza, W.-K. Wong, S. Stumpf, S. Perona, R. White, M. Burnett, I. Oberst, and A. Ko, "Fixing the Program My Computer Learned: Barriers for End Users, Challenges for the Machine," *ACM Conference on Intelligent User Interfaces*, Sanibel Island, Florida, USA, Feb. 2009, to appear.

[11] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks," *ACM Conference on Intelligent User Interfaces*, Florence, Italy, Apr. 2008, pp. 1323-1332.

[12] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Can Information Foraging Pick the Fix? A Field Study," *IEEE Symposium on Visual Languages and Human-Centric Computing*, Herrsching am Ammersee, Germany, Sept. 2008, pp. 57-64.

[13] G. Lowenstein, "The Psychology of Curiosity," *Psychological Bulletin* 116(1), American Psychological Association Press, Washington D.C., USA, 1994, pp. 75-98.

[14] P. Pirolli and S. Card, "Information Foraging," *Psychological Review* 106, American Psychological Association Press, Washington D.C., USA, 1999, pp. 643-675.

[15] M. B. Rosson, H. Sinha, M. Bhattacharya, and D. Zhao, "Design Planning in End-User Web Development." *IEEE Symposium on Visual Languages and Human-Centric Computing*. Coeur d'Alène, Idaho, USA, 2007, pp. 189-196.

[16] G. Rothermel, M. Burnett, L. Li, C. DuPuis, and A. Sheretov, "A Methodology for Testing Spreadsheets," *ACM Trans. Software Engineering and Methodology* 10(1), ACM Press, Jan. 2001, pp. 110-147.

[17] J. Ruthruff, A. Phalgune, L. Beckwith, M. Burnett, and C. Cook, "Rewarding Good Behavior: End-User Debugging and Rewards," *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, Sept. 2004, pp. 115-122.

[18] N. Subrahmaniyan, L. Beckwith, V. Grigoreanu, M. Burnett, S. Wiedenbeck, V. Narayanan, K. Bucht, R. Drummond, and X. Fern, "Testing vs. Code Inspection vs. What Else? Male and Female End Users' Debugging Strategies," *ACM Conference on Human Factors in Computing Systems*, Florence, Italy, Sept. 2008, pp. 617-626.

[19] N. Subrahmaniyan, M. Burnett, and C. Bogart, "Software Visualization for End-User Programmers: Trial Period Obstacles," *ACM Symposium on Software Visualization*, Herrsching am Ammersee, Germany, Sept. 2008, pp. 135-144.

[20] R. Walpole and M. Burnett, "Supporting Reuse of Evolving Visual Code," *IEEE Symposium on Visual Languages*, Capri, Italy, Sept. 1997, pp. 68-75.

[21] A. Wilson, M. Burnett, L. Beckwith, O. Granatir, L. Casburn, C. Cook, M. Durham, and G. Rothermel. "Harnessing Curiosity to Increase Correctness in End-User Programming," *ACM Conference on Human Factors in Computing Systems*, Ft. Lauderdale, Florida, USA, Apr. 2003, pp. 305-312.