AN ABSTRACT OF THE DISSERTATION OF

Valentina I. Grigoreanu for the degree of Doctor of Philosophy in Computer Science presented on December 1, 2009.

Title: Understanding and Supporting End-User Debugging Strategies

Abstract approved:

____

Margaret M. Burnett

End users' programs are fraught with errors, costing companies millions of dollars. One reason may be that researchers and tool designers have not yet focused on end-user debugging *strategies*. To investigate this possibility, this dissertation presents eight empirical studies and a new strategy-based end-user debugging tool for Excel, called *StratCel*.

These studies revealed insights about males' and females' end-user debugging strategies at four levels of abstraction (moves, tactics, stratagems, and strategies), leading to specific implications for the design of tools. Particular results include a set of ten debugging stratagems, which generalized across three environments: the research spreadsheet environment Forms/3, commercial Windows PowerShell, and commercial Microsoft Excel. There was also evidence of the stratagems' generalization to a fourth set of environments: interactive application design environments, such as Dreamweaver, Flash, and Blend. Males and females statistically preferred and were effective with different stratagems, and females' stratagems were less well-supported by environments than males' were. In addition to *what* stratagems our participants used, we also investigated *how* these stratagems were used to find bugs, fix bugs, and evaluate fixes. Furthermore, a Sensemaking approach revealed end-user debugging strategies and the advantages and disadvantages of each.

We then built StratCel, which demonstrates a strategy-centric approach to tool design by explicitly supporting several of the implications from our studies. StratCel's evaluation revealed significant benefits of a strategy-centric approach to tool design: participants using the tool found *twice* as many bugs as participants using standard Excel and fixed *four* times as many bugs (including two bugs which had not been purposefully inserted by researchers and had gone unnoticed in previous studies). Further, StratCel participants did so much faster than participants using standard Excel. For example, participants using StratCel found the first bug 90% faster and fixed it 80% faster than participants without the tool. Finally, this strategy-based approach helped the participants who needed it the most: boosting novices' debugging performance near experienced participants' improved levels. The fact that this approach helped everyone—males and females, novices and experts—demonstrates the significant advantage of strategy-centric approaches over feature-centric approaches to designing tools that aim to support end-user debugging.

Understanding and Supporting End-User Debugging Strategies

by

Valentina I. Grigoreanu

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented December 1, 2009
Commencement June 2010

Doctor of Philosophy dissertation of Valentina I. Grigoreanu presented on December 1, 2009.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

Dean of the Graduate School

Valentina I. Grigoreanu, Author

ACKNOWLEDGEMENTS

To my advisor, Margaret Burnett, the lessons I have learned from you are indispensible. Thank you especially for throwing me into diverse high-responsibility tasks from the very beginning. There was not a day that passed where I did not learn something new during the whirlwind past years. I will always remember EUD'09: what a great trip, paper, and talks.

Margaret Burnett, Paul ElRif, George Robertson, and Jens Mache: Thank you from the bottom of my heart for discovering me and for giving me the opportunity of working with your research groups. Working with you has provided me with a well-rounded view of research that I will continue to cherish as I move forward in my career.

I would also like to thank the rest of my committee (Carlos Jensen, Timothy Budd, Maggie Niess, and Bob Higdon) for their thoughtful feedback on my dissertation and, along the way, on my Qualifier and Dissertation Proposal. Thank you for your friendship over the past few years. Thank you also to Paul Cull (the PhD program would not be the same without Friday Coffee Hour) and my other Oregon State professors in the Computer Science, Electrical Engineering, Psychology, and Statistics departments who have all helped create the foundations for this work. Also would like to give many thanks to Ferne Simendinger, Jon Herlocker, Bella Bose, Terri Fiez, and Chris Bell for making me feel welcome at OSU from the very beginning of the program.

To all of my coauthors and collaborators: it has been great working with you and I hope we will keep collaborating for years to come. To my new friends (especially OSU EECS graduate students, EUSES consortium members, VLHCC attendants, Microsoft MAX team members, Microsoft Research VIBE and HIP team members, and fellow CHI student volunteers): I have many a fond memory of our chats and your friendship means so much to me.

Most importantly, I am eternally grateful to my amazing parents. Nothing would have been possible without your unconditional love during this entire process, and your kind words and deeds. Chester, your energy and enthusiasm have been inspiring.

This work was supported in part by Microsoft Research, by NSF grant CNS-0420533, by an IBM Faculty Award, by the EUSES Consortium via NSF grant ITR-0325273, by NSF 0917366, by AFOSR FA9550-09-1-0213, by NSF CCR-0324844, and by the Saturday Academy's Apprenticeships in Science and Engineering Program for the support of our interns.

CONTRIBUTION OF AUTHORS

George Robertson helped polish the papers presented in Chapters 5, 7, and 8 and also helped analyze the data for Chapter 5, as the second qualitative analysis researcher, and helped supervise the data collection and analysis for that chapter.

Jill Cao, Todd Kulesza, and Chris Bogart helped conduct some of the daily study sessions with participants for the study presented in Chapter 3. Jill and Todd also iterated on and shot the videos used in the new explanations for that study, and analyzed the data related to those new explanations for the Discussion for that chapter. Jill also analyzed the participants' background data. Chris played a big role during the rebuttal and rewriting phases of the paper submission process. Jill also iterated on an earlier version of Table 19 and created Figure 21. All three helped review the experimental setup and paper write-ups we worked on together.

Kyle Rector also helped collect data during daily participant sessions for the study presented in Chapter 3. She also helped as the second content analysis researcher for both that paper and for the study presented in Chapter 6; I used that same data in part in Chapters 7 and 8.

Susan Wiedenbeck helped write the related work sections of both Chapters 3 and 6. She also helped review and polish both of the papers in their entirety. Curtis Cook, Sherry Yang, and Vaishnavi Narayanan further reviewed Chapter 3 and gave input along the way about which constructs worked best.

Laura Beckwith helped with the write-up for the study presented in Chapter 2, particularly the Introduction and Self-Efficacy Results. The data we mined for the Data Mining Study came from an earlier study of hers. Also, the grade-book spreadsheet task for the studies presented in Chapters 6-8 came from one of her studies, and I harvested the ten bugs I introduced from that particular study's participants.

Xiaoli Fern, with help from Chaitanya Komireddy, led the sequential pattern mining side for the study presented in Chapter 2: chose the algorithm and applied it. They also helped write up the data mining parts of the paper, including the ones about the pattern mining process and the related work.

James Brundage was the second content analysis researcher for the study presented in Chapter 4. He also played a significant role in helping set up the study (the task was one of his real-world scripts), run participants, and write a first draft of the section about the script and the bugs.

Eric Bahna, Paul ElRif, and Jeffrey Snover also helped review and polish the paper presented in Chapter 4 before submission. Eric wrote about the interaction between stratagems and strategies. Paul helped in reviewing the paper and making sure it adheres to APA standards. He also wrote the first draft of the Participants section in Chapter 4. Jeffrey is PowerShell's architect and helped write the section of Chapter 4 about the environment.

Roland Fernandez and Kori Inkpen also helped with the paper reported in Chapter 5. They supervised the study and helped review and polish off the paper. Discussions with Roland also helped structure Chapter 8.

All of the papers which have been accepted to conferences went through a round of peer-review. The reviewers' comments were often quite insightful and I would therefore like to acknowledge their input here as well, in polishing off the papers.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF FIGURES (Continued)

LIST OF TABLES

LIST OF TABLES (Continued)

To mom and dad

UNDERSTANDING AND SUPPORTING

END-USER DEBUGGING STRATEGIES

1.

Introduction

Although twenty years ago, the idea of end users creating their own programs was still a revolutionary concept, today, end-user programming has become a widespread phenomenon. However, just like professional programmers' code, end-user programmers' code can contain costly errors.

One recent example that received media attention came following Lehman Brothers' collapse. Barclays Capital agreed to purchase some of Lehman's assets but, due to a spreadsheet error resulting from hidden cells, the company purchased assets for millions of dollars more than they had intended [Hayes 2008]. Factors playing into this included: several rushed and exhausted employees, without professional debugging training, making assumptions about the spreadsheet's hidden structure to make an imminent deadline. A few weeks later, Barclays filed a motion in court asking for relief due to the mistake.

These errors do not only haunt the industry, but also governmental budgets. For example, a Nevada city budget spreadsheet posted to the city's website and distributed to city council members falsely showed a $5 million dollar deficit in the water and sewer fund [Pokorny 2006]. This discrepancy delayed voting on the city's yearly budget. Upon closer examination, the finance director found several other errors in the budget spreadsheet.

This dissertation explores the ways in which end-user programmers, such as the clerks and government officials in two above stories, approach debugging their end-user programs, and how tools can be designed to support their effective strategies.

1.1. Problem Addressed and Definitions

End-user programmers were described by Nardi as people who program (create artifacts that instruct computers how to perform an upcoming computation), not as an end in itself, but as a means to more quickly accomplish their tasks or hobbies [Nardi 1993]. In fact, end-user programmers often do not have professional computer science training; examples include an accountant creating a budget spreadsheet, a designer building interactive web applications in Flash, a garage mechanic entering rules to sort email, or a teacher authoring educational simulations of science phenomena.

End-user programming has become pervasive in part due to research advances such as graphical techniques for programming, programming by demonstration, and innovative ways of representing programs (described, for example, in [Kelleher and Pausch 2005; Myers et al. 2006; Nardi 1993], and in part due to the popularity of spreadsheets [Scaffidi et al. 2005]. Other examples include teaching kids to create programs (e.g., [Cypher and Smith 1995; Kelleher et al. 2007]), programming for and over the web (e.g., [Kandogan et al. 2005; Rosson et al. 2007]), and even programming household appliances [Rode et al. 2004].

Along with the ability to create programs comes the need to debug them, and work on end-user debugging is only beginning to become established. The impact of end-user programming errors like the Lehman-Barclays and Nevada city examples is amplified by the quickly increasing complexity of end-user programs and by the large number of end-user programmers. The complexity of corporations' spreadsheets doubles in both size and formula content every three years [Whittaker 1999]. In addition, there are tens of millions more end-user programmers than there are professional programmers [Scaffidi et al. 2005].

The numerous reports of expensive errors in end users' programs, especially spreadsheets (e.g., [Boehm and Basili 2001; Butler 2000; EuSpRIG 2006; Panko 1998; Panko and Orday 2005]), make clear that supporting end users' debugging efforts is important. There has been recent work on tools for end-user debugging to fill this need (e.g., [Abraham and Erwig 2007; Ayalew and Mittermeir 2003; Burnett et al. 2003; Burnett et al. 2004; Ko and Myers 2004; Wagner and Lieberman 2004]). Much of this work has been about feature usage and mapping techniques from professional software engineering to this population.

However, we believe that a critical stone has been left unturned in the design of spreadsheet debugging tools: how tools can be designed to directly support end-user programmers' effective *debugging strategies* (users' plans of action for accomplishing a task). While, up to this point, we have used the term "strategy" to refer to any "reasoned plan of action for achieving a goal," from here on out, we differentiate between nuances of strategy items at different levels. To do so, we employ Bates' terminology for four strategy levels, which she defined in the context of online searching [Bates 1990].

Bates argues that search systems should be designed to make a good strategy easy to employ, by taking into account search behaviors which promote the strategic goals of searching for information [Bates 1990]. This advice also holds in designing debugging tools for end-user programming environments. Bates' four levels of strategy are moves, tactics, stratagems, and strategies. A *move* is at the lowest level of abstraction: "an identifiable thought or action" [Bates 1990]. A *tactic* is "one or a handful of moves made." A *stratagem* is "a larger, more complex, set of thoughts and/or actions." Finally, a *strategy* is "a plan which may contain moves, tactics, and/or stratagems for *the entire search*," or in our case, the entire debugging session. To refer to all four of these levels, we employ the phrase *strategy levels*, and to refer to an item at any of the four levels, we employ the term *strategy item*.

In essence, studying software features, as in previous research, allows consideration of only that which exists. This dissertation looks beyond features to debugging strategies, to consider that which may not yet exist, but should.

1.2. Methodological Approach

1.2.1. Research Goals

This dissertation investigates the claim that strategy items, and the possible gender differences in their usage, should be taken into account in the design of debugging tools for end-user programming environments. Our goals were, first, to understand the strategy items used by male and female end-user programmers, and second, to find ways the software can support these strategy items.

1.2.2. Research Methods

To accomplish our goals, we used a mix of research and analysis methods commonly employed in Human-Computer Interaction research in an iterative research paradigm. We began with hypotheses derived empirically from a Data Mining study (Chapter 2) and a Feature Usage study (Chapter 3), informed by theories in related domains (including Marketing, Communication, and Computer Science). After refining these hypotheses, we conducted a mixed-method lab experiment to test them in a Forms/3 Stratagems study, thereby generating results about *what* strategy items male and female end-user programmers employed and how they were tied to success for males and females [Subrahmaniyan et al. 2008]. We then conducted four qualitative studies to better understand the strategy items: *how* the discovered stratagems led to male and female success (Chapter 4), how the stratagems generalized across three other types of end-user programming environments and populations (Chapter 4, Chapter 5, and Chapter 7), what end-user debugging *strategies* were employed by our participants (Chapter 6), and examining all four levels of strategy to derive 21 detailed implications for the design of end-user debugging tools (Chapter 7). To evaluate the usefulness of a strategy-centric approach to tool design, we tested our implications for design by building a strategy-based tool for Excel, which we called StratCel (Chapter 8). Using a mixed-methods approach (triangulating between quantitative and qualitative data), we evaluated the resulting progress in supporting end-user programmers' debugging. The new strategy-centric approach succeeded at increasing male, female, experienced, and novice success. The outcomes also led to further theories used to generate new hypotheses for future work (Chapter 9).

Strategy items exist in the head, and in-the-head data are not easy to obtain reliably. Hence, our experiments were heavily triangulated, using different types of data to confirm findings from multiple sources: questionnaire data, audio/video session replays, interview data, focus groups, survey data, end-user programmers' artifacts (emails and forum entries), and logged behaviors from controlled lab experiments. Questionnaire, survey, interview, and audio data were used primarily to elicit participants' in-the-head strategy items, which are not directly observable, while directly observable behavioral data and analysis of their artifacts were used to corroborate the presence of these strategy items. To gather these data, six of the above studies were formative in nature (Data Mining Study, Forms/3 Stratagems Study, PowerShell Stratagems Study, Designers Grounded Theory Study, Sensemaking Study and Excel Strategy Items Study) and the remaining two were summative (Feature Usage Study and StratCel Tool Study). We also employed iterative design methods (requirement gathering and summarized in the form of scenarios and use cases, personas, a storyboard, and many walkthroughs) for creating and evaluating StratCel, our strategy-based tool for Excel. We analyzed our collected data using qualitative methods (such as coding transcripts), automated data mining methods (such as sequential pattern mining), and traditional statistical methods (such as linear regression, ANOVA, and the Mann-Whitney rank sum test).

1.2.3. Research Scope

We pursued three major threads in parallel. The first was to apply existing theories as well as our own research to hypothesize about how gender differences tie to preferred and effective adoption of the strategy items, and about the importance of researching debugging strategy items in addition to feature usage. The second followed a grounded theory approach to discover what debugging strategy items end-user programmers employ across multiple software environments and how those strategy items are used. The third was to consider how strategy-based end-user debugging tools can be built to address the theoretical findings, and what benefits arise from this new approach to the design of end-user debugging tools.

The combination of these three threads aimed to answer the following research question:

How can environments better support strategy items to improve end-user programmers' debugging success?

PART 1 – MOTIVATION:

Two Preliminary Studies

Overview of Part I of the Dissertation:

Chapter 2: Should we consider gender differences in our strategy items work?

Our initial investigation into end-user debugging strategy items was an exploratory data mining study to judge the importance of males' and females' data separately. The study's purpose was to generate new hypotheses about end-user debugging strategy items and gender differences in a ground-up manner via data mining. In this first study, we ran data mining algorithms on existing data. In particular, we estimated end-user debugging stratagems by mining short sequential patterns of moves from user logs, both to observe possible end-user debugging stratagems and also to decide whether it would be fruitful to look for gender differences in preferred and effective strategies throughout this work.

This approach provided an approximation of end-user debugging *stratagems* from earlier collected data on participants' *moves*: observing common patterns in sequences of moves. Our findings suggested four new hypotheses to frame future research on end-user debugging strategies. We tested the first of those (*"The debugging strategy items that help with males' success are not the right ones for females' success."*) by first finding out what debugging stratagems were employed by male and female end-user programmers in the Forms/3 spreadsheet environment, and second by seeing which of those led to success for males and females.

Chapter 3: Is a feature-based approach alone sufficient for closing the gender gaps in end-user debugging?

The goal of this study was to see whether designing tools which support gender differences in self-efficacy and feature usage would be enough to close the observed gender gaps (in end-user debugging confidence, feature usage, and debugging success).

Our results show, for the first time, that it is possible to design features in these environments that close the gender gaps in confidence and feature usage. In particular, females' usage of testing/debugging features increased, their confidence levels were improved and their self-judgments were roughly appropriate indicators of their actual ability levels, and females' attitudes toward the software environment were more positive. These gains came without disadvantaging the males.

However, there was one gap these features were unable to close: the gender gap in debugging success, as measured by the number of bugs found and fixed. One possible reason for this may be that the features push both males and females into males' effective strategy items, with testing features and testing strategy hints. To see whether this was the case, in Part 2 of the dissertation, we reveal males' and females' preferred and effective end-user debugging strategy items, so as to design tools to help increase both males' and females' debugging success.

Gender Differences in End-User Debugging, Revisited: What the Miners Found

Valentina Grigoreanu, Laura Beckwith, Xiaoli Fern, Sherry Yang, Chaitanya Komireddy, Vaishnavi Narayanan, Curtis Cook, Margaret Burnett

VLHCC'06 (Brighton, UK: September 04 - 08, 2006)

IEEE Computer Society, Washington, DC.

Issue manuscript appears in the *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 19-26.

2.

The DATA MINING STUDY: Approximating Stratagems and the Impact of Gender

2.1. Abstract

We have been working to uncover gender differences in the ways males and females problem solve in end-user programming situations, and have discovered differences in males' versus females' use of several debugging features. Still, because this line of investigation is new, knowing exactly what to look for is difficult and important information could escape our notice. We therefore decided to bring data mining techniques to bear on our data, with two aims: primarily, to expand what is known about how males versus females make use of end-user debugging features, and secondarily, to find out whether data mining could bring new understanding to this research, given that we had already studied the data manually using qualitative and quantitative methods. The results suggested several new hypotheses in how males versus females go about end-user debugging tasks, the factors that play into their choices, and how their choices are associated with success.

2.2. Introduction

Although there has been a fairly wide interest in gender differences in computing professions and education, as well as in gaming, there has not been much research on how gender differences interact with end users' use of *software features*, a research area we have begun to pursue which we term gender HCI. (While individual differences, such as experience, cognitive style, and spatial ability, are likely to vary more than differences between gender groups, research from several domains has shown gender differences that are relevant to computer usage [Beckwith et al. 2005; Busch 1995].) Our particular focus is on questions related to end-user software development. Our goal is to learn how to design end-user programming environments such that they support end-user programmers of both genders.

Most of our work so far in this area has followed a theory-driven approach, in which theories from psychology, education, and HCI have been used to generate hypotheses which have then been investigated via empirical studies. However, a disadvantage in deriving empirical hypotheses from only established theories is that these theories do not take into account the specific needs and issues that arise in end-user programming. Research situations such as this are often referred to as "ill-structured" problems [Simon 1973]. Such problems contain uncertainty about which concepts, rules, and principles are pertinent to the problem. Further, the "best" solutions to ill-structured problems depend on the priorities underlying the situation. In such problems, in addition to hypothesis testing and application, there is also the need for hypothesis generation. Such problems are candidates for ultimately deriving new theories from data and patterns.

Toward this aim, we have previously used manual qualitative analysis techniques [Strauss and Corbin 1998], inspecting data on software feature usage in search of useful patterns leading to hypotheses. Although the results of these efforts have been fruitful, still, as humans we are fallible, especially given large amounts of detailed data. We suspected that there may be important information that we were overlooking. Therefore, we employed a methodology change: turning to data mining techniques to find feature usage patterns that we may have missed.

In this paper we report the results of revisiting data we had already analyzed, but this time using a data mining approach. Using this approach, we focus on gender differences in *how* features are used, with the aim of gaining new insights into our previous reports of *when* and *how much*.

Our aim was to derive new hypotheses about females' and males' strategies, adding to the growing foundation for understanding the gender differences in end-user programming situations—by "listening" to the participants, through their data, from the ground up.

2.3. Background and Related Work

We began our gender HCI research by generating hypotheses [Beckwith and Burnett 2004] from relevant theoretical work from other domains, including self-efficacy theory [Bandura 1986], educational theories such as minimalist learning theory [Carroll 1998], and the model of attention investment [Blackwell 2002]. Several of these hypotheses also came from empirical results from others' investigations into gender differences in educational, work, and entertainment settings. We followed up on some of these hypotheses by conducting empirical studies of these gender differences, including both qualitative (e.g., [Beckwith et al. 2005]) and quantitative [Beckwith et al. 2005; Beckwith et al. 2006] results. Many of our findings have related self-efficacy to the way females interact with software. Self-efficacy is a form of confidence in one's ability, specific to the task at hand [Bandura 1986]. Gender differences regarding computer related confidence have been widely studied, revealing that females (both computer science majors and end users) have lower self-confidence than males in their computer-related abilities (e.g., [Busch 1995]). Previous work also found that tinkering with features can be helpful to both females' and males' success, but that males sometimes overdo it, which can interfere with their success [Beckwith et al. 2006].

In this paper we apply data mining to uncover additional patterns of interest. Data mining, also known as knowledge discovery in databases (KDD), is typically used to find hidden patterns in data to help understand data and make predictions about future behavior. In this study, we apply sequential pattern mining [Agrawal and Srikant 1995] to our HCI data.

Sequential Pattern Mining was first introduced in the context of retail data analysis for identifying customers' buying habits [Agrawal and Srikant 1995] and finding telecommunication network alarm patterns [Hatonen et al. 1996; Mannila et al. 1997]. It has since been successfully applied to many domains including some HCI related applications, such as web access pattern mining for finding effective logical structure for web spaces [Perkowitz and Etzioni 1998] and automatic web personalization [Mobasher et al. 2000], mining Windows processes data to detect unauthorized computer users [Cervone and Michalski 2002], and mining user-computer interaction patterns for finding functional usage scenarios of legacy software [El-Ramly et al. 2002].

Most work on applying data mining techniques to HCI data has focused on finding characteristic patterns of individual users [Cervone and Michalski 2002; Mobasher et al. 2000] or finding patterns that are common to the entire population [El-Ramly et al. 2002; Perkowitz and Etzioni 1998]. In contrast, in this study we are not interested in these two types of patterns. Instead, our research goal requires us to find patterns that are linked to subgroups of users, i.e., female users and male users. Another somewhat unusual aspect of our work is that we use the found patterns to generate gender-related hypotheses. To our knowledge, data mining has not been used before to generate hypotheses from HCI data. In particular, it has not previously been used to find gender differences in females' and males' interactions with software.

2.4. The Data and Environment

For this type of purpose, it is acceptable to mine from previous data, so we used data from one of our earlier studies [Beckwith et al. 2006]. The data was collected from 39 participants who used the "High-Support Environment" in our research prototype. The environment provided a number of features to participants who might feel in need of extra help. These features are detailed in [Beckwith et al. 2006]. The participants' task was to find and fix errors in two spreadsheets. Prior to the task, the participants were introduced through a tutorial to the environment features designed to aid in debugging.

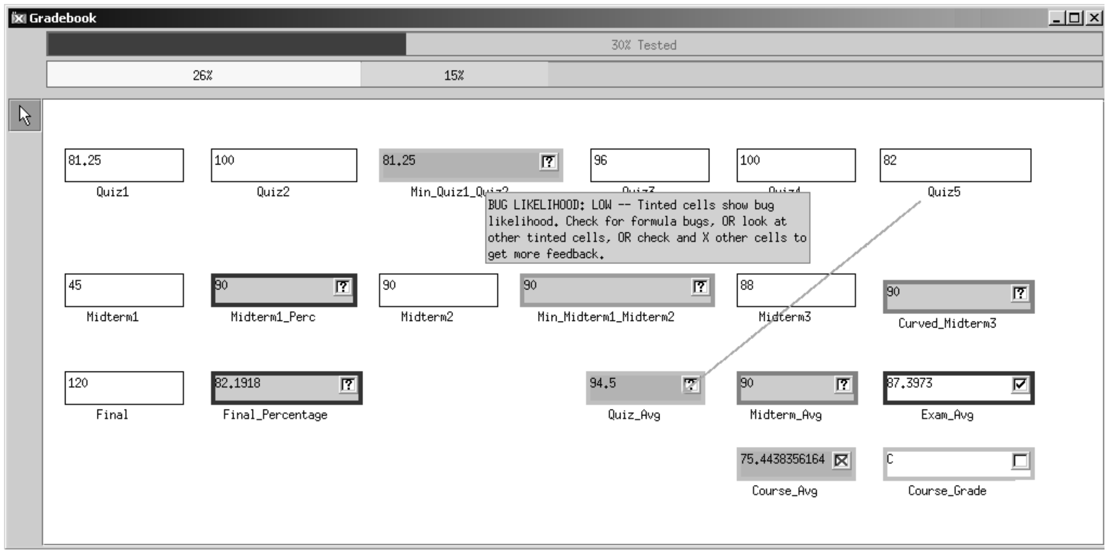
The debugging features present were part of WYSIWYT ("What You See Is What You Test"). WYSIWYT is a collection of testing and debugging features that allow users to incrementally "check off" (Checkmark) or "X out" (X-mark) values that are correct or incorrect, respectively [Burnett et al. 2004]. Cells initially have red borders, indicating that they are untested. The more a cell formula's subexpressions are covered by tests (checked-off values), the more blue its border becomes.

The environment also includes arrows, which participants can toggle on and off on a per-cell (or even per-arrow) basis. Arrows serve two purposes: First, they explicitly depict the dataflow dependencies between the cells and, when cells' formulas are open, even between subexpressions in the related cells. Second, arrows' coloring reflect WYSIWYT "testedness" status at a finer level of detail, following the same color scheme as the borders. A user can thus look for red arrows to find cell dependencies that still need to be tested. Figure 1 shows an example of these WYSIWYT features that were available to the participants.

Also present in the environment was the "Help Me Test" (HMT) feature. Sometimes it can be difficult to find test values that will cover the untested logic in a collection of related formulas, and HMT tries to find inputs that will lead to coverage of untested logic in the spreadsheet, upon which users can then make testing decisions.

Each of these features is supported through the Surprise-Explain-Reward strategy [Wilson et al. 2003]. This strategy relies on a user's curiosity about features in the environment. If a user becomes curious about a feature, the user can seek out explanations of the feature via tool tips. The aim of the strategy is that, if the user follows up as advised in the explanation, rewards will ensue.

Participants' actions were recorded in user action log files. A *user action* is defined as a participant's use of a debugging feature. The log files contained detailed information about every user action, including a time stamp for when the action was taken, on which cell it operated, and various related parameters. Here is an excerpt of a log file:



BUG LIKELIHOOD: LOW -- Tinted cells show bug likelihood. Check for formula bugs, OR look at other tinted cells, OR check and X other cells to get more feedback.

Figure 1. The user notices an incorrect value in Course_Avg—the value is obviously too low—and places an X-mark in the cell. As a result of this X and the checkmark in Exam_Avg, eight cells are highlighted as being possible sources of the incorrect value, with the darker shaded cells deemed more likely than others.

```
15:43:47, Tooltip Showing, CELL31567926-2332 …
15:44:12, Checkmark, CELL31567926-2342 …
15:44:57, Checkmark, CELL31567926-2332 …
```

In addition to the log files, we also collected information on participants' task success, background information, and pre-task self-efficacy. Only the log files were used in pattern mining. The additional information was used to help analyze the patterns we found.

2.5. The Pattern Mining Process

In this study, we applied sequential pattern mining to our log files to search for potentially interesting patterns.

To formulate the sequential pattern mining problem, we considered each user action as an *event*. Since we are most interested in *how* participants used features, we abstracted away detailed contextual information that distracted from this goal (such as the specific cell on which the features were being used, or the exact time of the actions). This abstraction transformed the data into *sequences* of events. For example, the log excerpt in the previous section translated into the sequence (Tooltip Showing, Checkmark, Checkmark).

2.5.1. Preprocessing into Debugging Sessions

Following the procedure of [Ruthruff et al. 2005], we used the notion of *debugging sessions* to break the sequence of events into subsequences. As with Ruthruff et al.'s definition, a debugging session ends with a formula edit (or at the end of the experiment), which presumably represents an attempt to fix a bug. However, unlike Ruthruff et al.'s definition, in which a debugging session began with the placement of an X-mark, our debugging sessions begin as soon as the previous one ended (or at the beginning of the experiment), so that all actions could be considered—not just the subset following an X-mark. In some cases participants edited the same formula multiple times consecutively. Since such edits were obviously a continuation of fixing the same bug, we included them in the preceding debugging session.

Based on this definition, we broke each log file into debugging sessions. The total number of debugging sessions for all 39 participants was 641. Thus the mean per participant was 16.47 debugging sessions. The mean number of events per debugging session was 24.45 events.

2.5.2. Sequential Pattern Mining

We used the SLPMiner program [Seno and Karypis 2002] to search for patterns of the form (A, B, C), where A, B, and C are events that happened in the specified order. A debugging session was considered to contain the pattern (A, B, C) if it had at least one occurrence of events A, B, and C in that order, but the events did not need to be consecutive. We refer to the percentage of all debugging sessions that contained a pattern as the *support* of the pattern.

SLPMiner searches for all sequential patterns whose support exceeds a pre-specified threshold, and these patterns are referred to as *frequent patterns*. To avoid redundancy due to the fact that any subsequence of a frequent pattern will also be a frequent pattern, the software output the maximal patterns, i.e., patterns that are not subsequences of other frequent patterns. We chose the support threshold to be 10%, i.e., a pattern had to be contained in more than 10% of the 641 debugging sessions to be output by SLPMiner. This threshold was chosen because it allowed us to find patterns that were common to multiple users while still containing some of the interesting but less frequently used features such as X-marks and Arrow operations. We further focused our attention on patterns of limited size, in particular of length between one and four, because without limitations there would simply be too many patterns to process, and longer patterns often contained cyclic behavior and were difficult to interpret.

2.5.3. Output and Post-Processing

From the 641 debugging sessions, SLPMiner found 107 patterns of length one through four. Note that SLPMiner (and other sequential pattern mining algorithms) can only find “frequent” patterns, i.e., those satisfying a minimum support criterion, which was 10% in our case. It was up to us to determine which of the found patterns were really interesting to our research goal.

Toward this aim, for each pattern we computed its occurrence frequency for each user as the percentage of that user’s debugging sessions that contained the pattern. For example, if user A had 20 debugging sessions and 10 of them contained pattern p, the occurrence frequency of pattern p for user A was 50%. As a result, we obtained a pattern occurrence frequency table, which provided a comprehensive description of the distribution of the pattern occurrence among all users. We then analyzed these pattern occurrence frequencies in relation to the gender, task performance, and self-efficacy of the participants who used them.

To help analyze the pattern occurrence frequencies in an organized manner and gain high-level understanding of the patterns, we categorized the found patterns such that each category contained patterns centered on a certain set of features. Figure 2 shows how the 107 patterns were distributed into nine non-overlapping categories. See Table 1 for examples of patterns and their categories. Our analysis described in the following sections will be presented based on these categories.

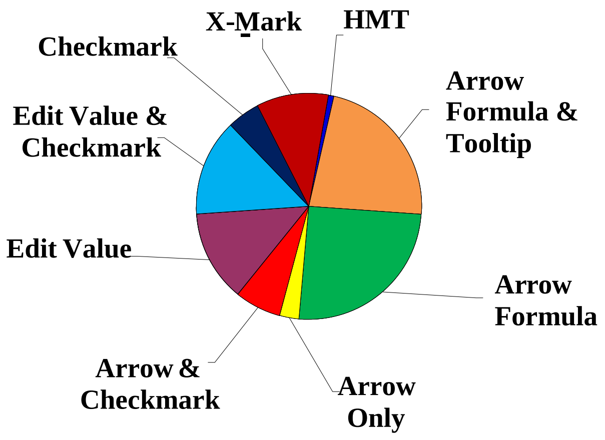


Figure 2. We grouped the 107 patterns into these 9 categories. The categories, based on the patterns' content, are focused on the debugging and other features available in the environment.

Table 1. Each of the nine categories contained patterns that only had the moves mentioned in the category name as a part of them.

Category	**Example Pattern**
Help Me Test (HMT)	(HMT)
Arrow, Formula & Tooltip	(Tooltip Showing, Arrow On, Arrow Off, Edit Formula)
Arrow & Formula	(Arrow Off, Post Formula, Hide Formula, Post Formula)
Arrow Only	(Arrow On, Arrow On)
Arrow & Checkmark	(Hide Formula, Checkmark, Arrow On)
Edit Value	(Edit Value, Edit Value)
Edit Value & Checkmark	(Post Formula, Edit Value, Checkmark, Hide Formula)
Checkmark	(Checkmark, Tooltip Showing, Tooltip Showing, Checkmark)
X-Mark	(Hide Formula, X-Mark, Post Formula, Edit Formula)

2.6. Results: How Each Gender Pursued Success

How did the successful versus unsuccessful females and males go about debugging? To investigate this question, we divided the 39 participants (16 males and 23 females) into four groups by gender and number of bugs fixed. We considered a participant "successful" if they fixed at least 7 of the 10 bugs, and "unsuccessful" otherwise. The groups and number of participants are displayed in Table 2.

2.6.1. Just Like Males: A Female Success Strategy?

Strikingly, in Figure 3 the unsuccessful females and successful males showed a similar frequency profile for each of the five categories on the left half of the graph (from Edit Value to HMT)—all of which are testing-oriented activities. We follow the software engineering definition of "testing" here: judging the correctness of the *values* produced by the program's *execution*.

This suggests that the ways males successfully went about their debugging task are the very ways that did not work out well for the females, leading to the following hypothesis:

Hypothesis: The debugging and testing strategies that help with males' success are not the right ones for females' success.

Table 2. What: The median number of arrows turned on and off during the experiment by gender and debugging success. Note especially the difference between the successful and unsuccessful males.

Group	Number of participants	Arrows
Successful Females	8	17.5
Unsuccessful Females	15	24
Successful Males	10	12
Unsuccessful Males	6	25.5

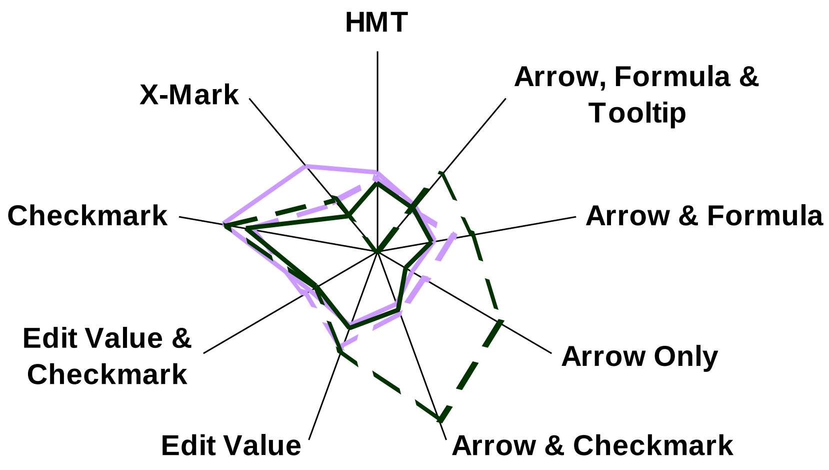


Figure 3. How (count of pattern usage) by success group. Successful: solid line, unsuccessful: dashed line, females: light, males: dark. (Each category is represented by an axis line radiating from the center. Where the polygon crosses an axis represents the frequency of that pattern.)

2.6.2. Unsuccessful Males Like Arrows

Turning to the right half of the graph, which represents arrow-oriented patterns, the successful and unsuccessful females converge with the successful males. Interestingly, regarding this "how" aspect of arrows, there was a striking difference between successful and unsuccessful males. This difference is further illustrated by Figure 4, which shows that, with all arrow patterns combined, the unsuccessful males used arrow patterns far more frequently than the successful males.

The higher frequency of arrow patterns for unsuccessful males coincides with a higher raw count of arrows used. As Table 2 shows, successful males used a median of 12 arrows, whereas unsuccessful males used more than twice as many, 25.5.

One of the most distinctive differences between the successful and unsuccessful males' arrow patterns occurred in the category Arrow & Checkmark. Within this category were two subcategories using the arrow and checkmark features: "Formula Checkmark Arrow", and "Arrow Checkmark Only." In the first, "Formula Checkmark Arrow", the patterns contain a formula-related action then a checkmark, followed by an arrow. For example: (Hide Formula, Checkmark, Arrow On) and (Hide Formula, Checkmark, Arrow Off). Both unsuccessful and

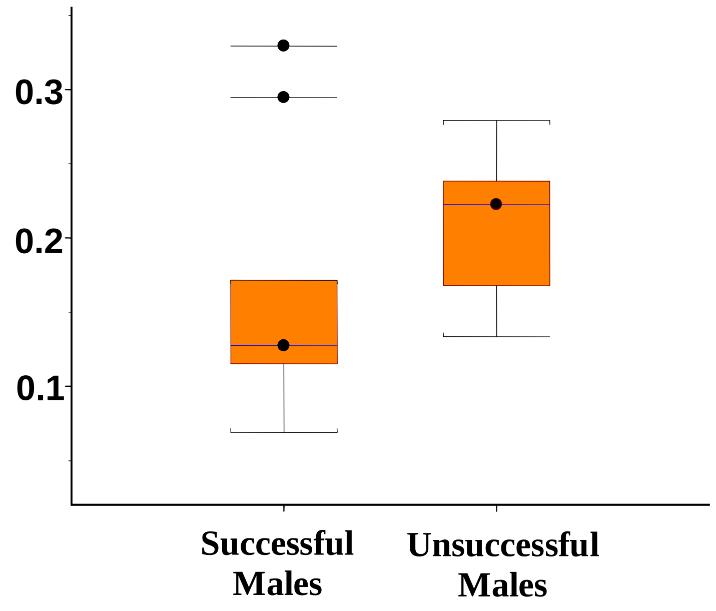


Figure 4. How: Percentage of debugging session that contained arrows by successful versus unsuccessful males.

successful males used these patterns frequently, in over one third of their debugging sessions.

On the other hand, “Arrow Checkmark Only”, while frequently used by the unsuccessful males (in one of every four debugging sessions), was rarely used by successful males (one of every ten debugging sessions). Examples of patterns in this subcategory included: (Arrow On, Checkmark) and (Arrow Off, Checkmark).

Although subtle, these two different strategies could have a large influence on task success. By basing testing decisions solely on the information provided by arrows, as may be the case in the “Arrow Checkmark Only” subcategory, participants may have neglected to take into account the current state of the program execution. In contrast, the “Formula Checkmark Arrow” subcategory is about making a testing decision and then using the arrows, perhaps to direct their next actions.

Hypothesis: Unsuccessful males overdo use of arrows—unlike successful males, successful females, or unsuccessful females.

2.6.3. Unsuccessful Males: Tinkering Addicts?

We suspected that gender differences in tinkering behavior may be a factor in observed pattern differences. In particular, the unsuccessful males' more frequent use of arrows and their greater variety of arrow-related patterns is suggestive of a larger picture of unsuccessful males tinkering with arrows, to their detriment.

In fact, in previous work, we reported results in which males were found to do more unproductive tinkering, using a different environment [Beckwith et al. 2006]. However, the definition of tinkering used in that paper was necessarily simple—and its simplicity prevented it from capturing the excessive exploring/playing the unsuccessful males did in the High-Support Environment. Based on patterns found via mining that data, we are now able to identify more complex tinkering behavior of unsuccessful males in this environment, which we failed to notice in our previous study.

For example, referring to Figure 3, notice the large differences in pattern frequencies for unsuccessful versus successful males in the Arrows Only category, which contains patterns that involve only arrow operations. Two representative patterns in this category were (Arrow Off, Arrow On) and (Arrow Off, Arrow Off). Unsuccessful males had more frequent occurrences of these patterns—one out of every four debugging sessions for the unsuccessful males versus only one out of 20 for the successful males.

Hypothesis: Unsuccessful males have a tendency to tinker excessively with the features themselves rather than using the features to accomplish their task.

2.7. Results: Self-Efficacy

How do high and low self-efficacy females and males go about debugging? Since self-efficacy did not give the same groupings of the participants as given by task success, it is useful to consider how self-efficacy related to pattern choices. Recall that *self-efficacy* measures a person's belief in his or her ability to perform a particular task [Bandura 1986].

To investigate the question of whether self-efficacy played a role in pattern usage, we divided the participants into four groups based on their self-efficacy score. In particular, we considered a participant to have high (low) self-efficacy if her or his score was higher (lower) than the median of all participants. See Table 3 for the grouping of the participants.

In previous studies [Beckwith et al. 2006], self-efficacy has been shown to be predictive of task success for females. That conclusion also holds for the data examined in this study. Half of the 12 high self-efficacy females were successful but only two out of 11 low self-efficacy females were successful. However, it was not true for males: seven out of 10 high self-efficacy males were successful and half of the low self-efficacy males were successful.

Patterns alone tell only part of the story. We turned to median raw counts of the number of features used to better understand the reasons behind the patterns that we were seeing. Low self-efficacy female feature counts (Table 3) revealed that low self-efficacy females were the highest usage group for all of the features—except the checkmark.

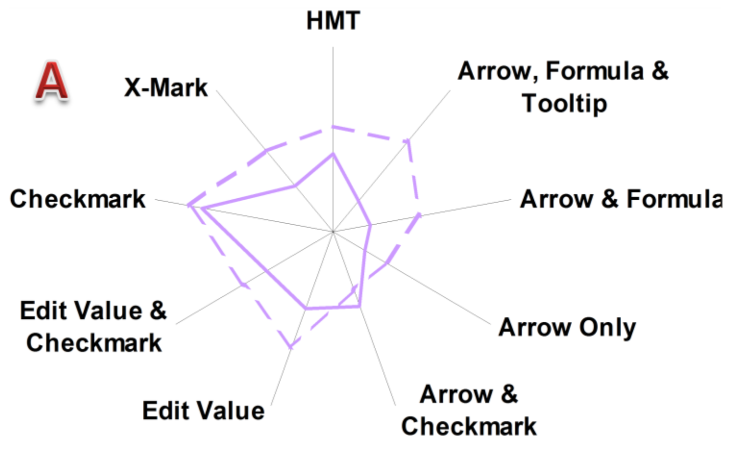
High feature usage by low self-efficacy females may at first seem to contradict our previous results, which showed that for females, high self-efficacy predicted more (effective) use of features, which in turn led to greater debugging success [Beckwith et al. 2005]. We

Table 3. What by self-efficacy group. These are about the number of moves, rather than the number of patterns, to supplement our understanding of the sequential patterns. We divided the participants into four groups based upon their gender and pre-task self-efficacy. The rest of the table shows median raw counts of the number of testing moves during the experiment.

Group	Number of participants	Arrow	X-mark	Checkmark	HMT
High Females	12	10.5	3	65.5	5
Low Females	11	24	8	45	8
High Males	10	20	2	52	1.5
Low Males	6	20	5.5	39	3

proposed that offering greater support in the environment would encourage low self-efficacy females to use the features more. The current study used the High-Support Environment, which included features designed to fix that very problem. Our results show that they worked—the low self-efficacy females did indeed use the features in this version! But our current study suggests that quantity of feature adoption is misleading in isolation: feature adoption must be considered in conjunction with *how* the features are used.

High and low self-efficacy females diverged in both counts and patterns. Notice in Figure 5 how many patterns the low self-efficacy females used compared to high self-efficacy



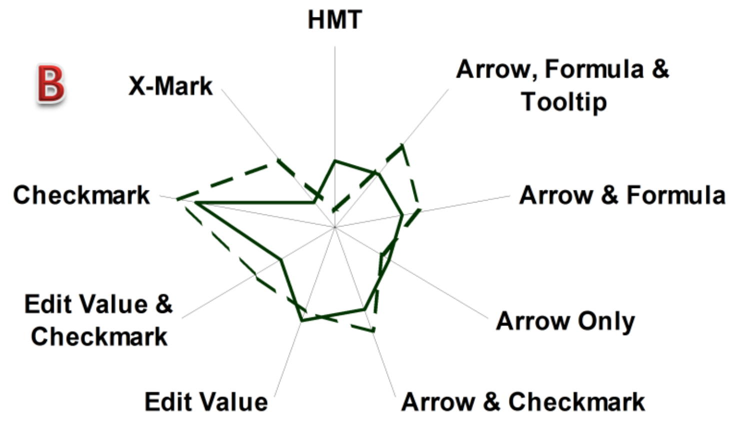


Figure 5. "How" by self-efficacy group for (a) females and (b) males. High self-efficacy: solid line, low self-efficacy: dashed line.

females, except for the checkmark. As suggested by self-efficacy theory, people with high self-efficacy are more likely to abandon faulty strategies faster than those with low self-efficacy [Bandura 1986]. Our results were consistent with this. For patterns other than the checkmark patterns, as suggested by Figure 6, the high self-efficacy females were willing to try out and quickly abandon many patterns in order to settle upon the ones they liked, whereas the low self-efficacy females were more likely to try a pattern again and again before ultimately moving on. For example, 54 patterns were used 5-10% of the time by high self-efficacy females, but only 16 were abandoned so quickly by the low self-efficacy females. This leads to the following hypothesis:

Hypothesis: Females with lower self-efficacy are likely to struggle longer to use a strategy that is not working well, before moving on to another strategy.

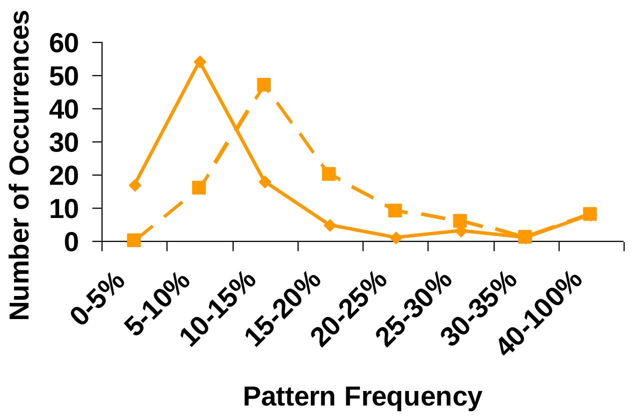


Figure 6. How: The high self-efficacy females (solid line) had more patterns fall in the frequency range of 5-10%, whereas the low self-efficacy females had more of their patterns fall in a higher number of debugging sessions (10-15%).

2.8. Some Lessons Learned

This research is the first exploration into using data mining in the investigation of possible gender differences in the way females and males went about problem-solving their programs. We briefly share the lessons we learned with the community so that others can gain from these lessons.

In short, these lessons were: (1) Data mining is no panacea. Many of us authors did not anticipate the amount of work required to identify useful patterns of human behavior and to interpret the results. (2) Data mining does not eliminate all bias. Human judgment can impact the results in many ways, such as in determining pattern definitions, thresholds and most importantly how to interpret patterns. Still, despite these somewhat rude awakenings, (3) it was worth it! Despite the fact that we had previously gone over the same data ourselves extensively, data mining turned up patterns for which we might never have thought to look.

2.9. Conclusions and Future Work

In this paper, we have reported new data-derived hypotheses about gendered patterns in the way females and males used our available spreadsheet debugging features. Because these data had already been analyzed, we have employed a conservative approach: we have used data mining solely to generate hypotheses that must be tested in later studies. These hypotheses are based on evidence in the data, as revealed by the data mining approach.

The new hypotheses related to the following results:

- Patterns used by the successful males were not a recipe for success for the females.
- Unsuccessful males used many more arrow patterns beyond those that appeared useful to the successful males.
- Self-efficacy, again, impacted females’ choice of patterns, but not males’. This is the fourth study showing ties between females’ self-efficacy and feature usage.

We caution that these are hypotheses and, although they are data-derived, they require future empirical study to accept or refute them.

CAN FEATURE DESIGN REDUCE THE GENDER GAP IN END-USER SOFTWARE DEVELOPMENT ENVIRONMENTS?

Valentina Grigoreanu, Jill Cao, Todd Kulesza, Christopher Bogart, Kyle Rector,

Margaret Burnett, and Susan Wiedenbeck

VLHCC'08 (Herrsching am Ammersee, Germany: September 15 - 19, 2008)

IEEE Computer Society, Washington, DC.

Issue manuscript appears in the *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 149-156.

3.

The FEATURE USAGE STUDY: What Did and Did Not Work in Supporting both Genders

3.1. Abstract

Recent research has begun to report that female end-user programmers are often more reluctant than males to employ features that are useful for testing and debugging. These earlier findings suggest that, unless such features can be changed in some appropriate way, there are likely to be important gender differences in end-user programmers' benefits from these features. In this paper, we compare end-user programmers' feature usage in an environment that supports end-user debugging, against an extension of the same environment with two features designed to help ameliorate the effects of low self-efficacy. Our results show ways in which these features affect female versus male end-user programmers' self-efficacy, attitudes, usage of testing and debugging features, and performance.

3.2. Introduction

Although there is a large body of literature about issues facing women in IT professions, until recently, researchers had not considered how gender differences interact with features and tools in programming environments. However, researchers are now beginning to report theory and empirical data pointing to gender differences in the use of end-user programming environments. Evidence of these differences has accumulated, indicating gender differences in programming environment appeal, playful tinkering with features, attitudes toward and usage of end-user programming features, and end-user debugging strategies [Beckwith et al. 2006; Beckwith et al. 2005; Brewer and Bassoli 2006; Kelleher et al. 2007; Lorigo et al. 2006; Rode et al. 2004; Rosson et al. 2007; Subrahmaniyan et al. 2008].

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 [Subrahmaniyan et al. 2008].

These studies have not delved into how to remedy this situation so that feature barriers to females' success can be removed. Although a few isolated solutions have been reported [Kelleher et al. 2007; Rosson et al. 2007], these solutions have not been tied to theory, and therefore do not provide designers of end-user programming environments the fundamental information they need about how to find and remove such barriers in their own environments.

In this paper, we tackle this issue. Our long-term research goal is to answer the following question:

Can theory be directly applied to end-user programming environment features to narrow the gender gap in end-user programmers' debugging success, by better supporting females?

The answer to this question holds the key to enabling designers of end-user programming environments to spot barriers in their own tool set, and further, to figure out what is needed to remove them.

In earlier research, we proposed two theory-based features that aimed to improve female performance without harming male performance: (1) adding "maybe" nuancing, inspired by self-efficacy theory, to features with which users input quality judgments and (2) integrated explanations geared toward gender differences in information processing and problem-solving styles [Beckwith et al. 2005]. 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 [Beckwith et al. 2005; Kissinger et al. 2006; Subrahmaniyan et al. 2007].

In this paper, we now face the challenge head-on. Namely, we committed these features to a robust implementation, and we investigated statistically whether they do indeed help to remove barriers to female end-user programmers' debugging efforts.

Our specific research questions were:

(RQ 1): Do these features help to close the gender gap in feature usage for female and male end-user programmers? If so, how does that relate to their success?

(RQ 2): Do these features help decrease the gap between females' and males' debugging self-efficacy? If so, how does that relate to their debugging success?

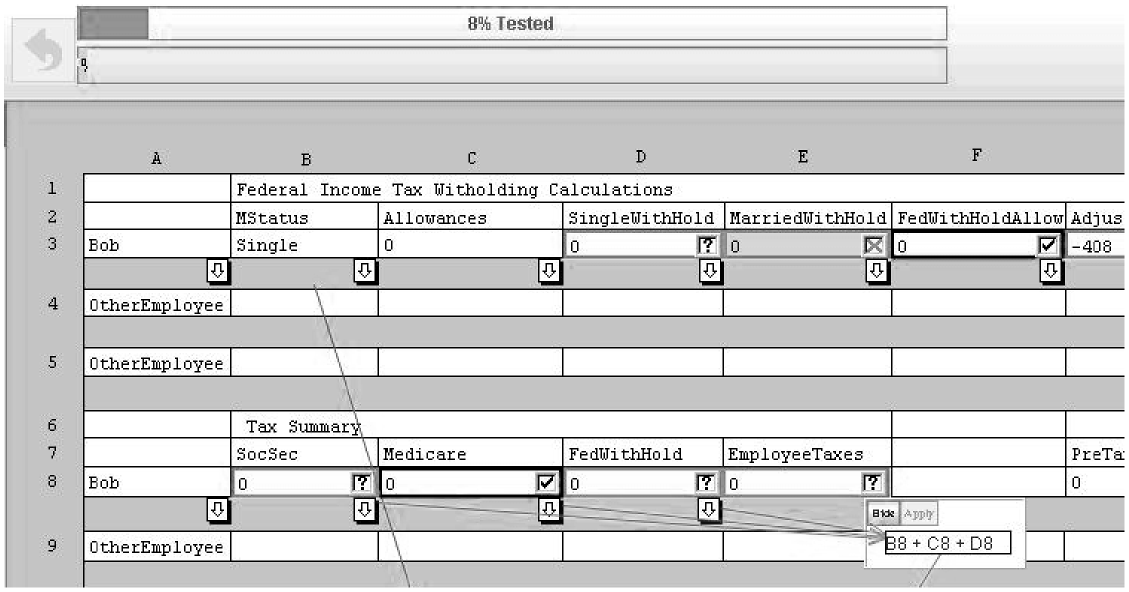
3.3. Background

3.3.1. Theoretical and Empirical Basis

Self-efficacy theory has had a strong influence on our work. Bandura's social-cognitive theory identifies self-efficacy (or confidence in one's ability to carry out a specific task) as a key element in performance outcomes. People with low self-efficacy for a task tend to expend less effort on the task, use simpler cognitive strategies, show less persistence when encountering difficulties, and ultimately finish with a lower success rate than people with high self-efficacy [Bandura 1977].

In software applications, studies have found gender differences in self-efficacy. Females generally have lower computer self-efficacy than males, and this has been tied to feature usage [Hartzel 2003]. Our own studies of both student and professional end-user debuggers have also found these effects [Beckwith et al. 2006; Beckwith et al. 2007]. For example, females' self-efficacy was predictive of successful use of debugging features, whereas males' feature usage was not tied to their self-efficacy. Further, unlike males, females were less willing to engage with, or even to explore, novel debugging features. This is unfortunate, because such features have been tied to improved debugging performance by both females and males.

For example, the Forms/3 spreadsheet environment, in which we prototype our ideas, includes WYSIWYT features (What You See Is What You Test) [Burnett et al. 2004] (Figure 7). This set of debugging features allows users to make decisions about the correctness of cells' output values, and then provides feedback to help them find and fix formula errors. When the user notices a correct cell value and puts a checkmark (√) in the cell's decision box (as in cell F3 in Figure 7, for example), the system increases the testedness of that cell and of the other cells that contributed to its value. These changes are reflected in the cell's border colors (red for untested, blue for tested, and shades of purple for partly tested; shown as shades of gray in this paper). Overall testedness of the spreadsheet is also reflected in the progress bar at the top of the screen. Instead of checking off a value, if the user sees an incorrect cell value and puts an X-mark in the cell (as in cell E3), fault-likelihood



	A	B	C	D	E	F	
1		Federal Income Tax Witholding Calculations					
2		MStatus	Allowances	SingleWithHold	MarriedWithHold	FedWithHoldAllow	Adjus
3	Bob	Single	0	0	0	0	-408
4	OtherEmployee						
5	OtherEmployee						
6		Tax Summary					
7		SocSec	Medicare	FedWithHold	EmployeeTaxes		PreTa
8	Bob	0	0	0	0		0
9	OtherEmployee						

B8 + C8 + D8

Figure 7. Forms/3 spreadsheet prototype with basic WYSIWYT features. (This is the version used by the Control group.)

calculations are triggered, which highlight the interior of cells that are likely to contain faulty formulas. The interior highlights follow a color continuum from light yellow for cells not likely to contain a formula error, to orange for cells with formulas very likely to contain an error (e.g., cell E3). Also, optional arrows can show dataflow relationships of cells; the arrows are colored using the same color scheme as the cell borders, to reflect testedness of the relationships. Tool tips briefly explain the semantics of each feature, suggest an action, and hint at a reward [Burnett et al. 2004].

We have reported in previous studies [Beckwith et al. 2006; Beckwith et al. 2007] females' lack of willingness to even try out the WYSIWYT features. Trying out new things by tinkering with them is a curiosity-based, playful behavior that leads to open-ended exploration in learning [Rowe 1978]. Such unguided, informal exploration has been shown to improve both motivation to learn and task performance [Martocchio and Webster 1992], but research in education suggests that females are less likely to tinker than males [Jones et al. 2000]. Our own investigations have confirmed females' lack of willingness to tinker [Beckwith et al. 2006]. Self-efficacy theory suggests that females' low computer self-efficacy decreased their willingness to tinker, denying them the benefits of exploratory learning and depressing the adoption of unfamiliar features. The solution, however, does not lie solely in encouraging tinkering—our previous results indicate that tinkering in overly complex environments can have the detrimental effect of decreasing females' self-efficacy [Beckwith et al. 2006]. Thus, if tinkering is to be helpful, it may need to involve features well-matched to the user's problem-solving style.

Another theory that has had a strong influence on our work is information processing theory [Meyers-Levy 1989]. This theory proposes that, as a group, females seek more comprehensive information than males do, although males tend to seek this too when problems become complex. We conducted formative empirical work to better understand what information end users seek when debugging [Kissinger et al. 2006]. Strategy information emerged as one of the top needs. Better ability to self-judge was another high-ranking information need, which is closely related to self-efficacy.

3.3.2. The Features

Given the results above, we looked for methods that might improve females' self-efficacy and willingness to try out, and ultimately to effectively use, the novel debugging features. The result was two feature changes. The first was adding "maybe" nuances to the checkmarks and X-marks (Figure 8), which we proposed earlier but did not statistically test [Beckwith et al. 2005]. 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. In the current paper we statistically investigate whether this feature does indeed help females, without hurting the males.

The second change, also proposed in [Beckwith et al. 2005], 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 evolved that proposal [Subrahmaniyan et al. 2007], ultimately providing the strategy explanations of the current study. Note that these are explanations of testing and debugging strategy, not explanations of software features per se.

The strategy explanations are provided as both video snippets and hypertext (Figure 9). 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

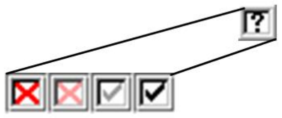


Figure 8. Clicking on the checkbox turns it into the four choices. The tool tips over the choices, starting with the leftmost X, are "it's wrong," "seems wrong maybe," "seems right maybe," "it's right."

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 [Bandura 1977; Compeau and Higgins 1995]. 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 [Blackwell 2002], 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.

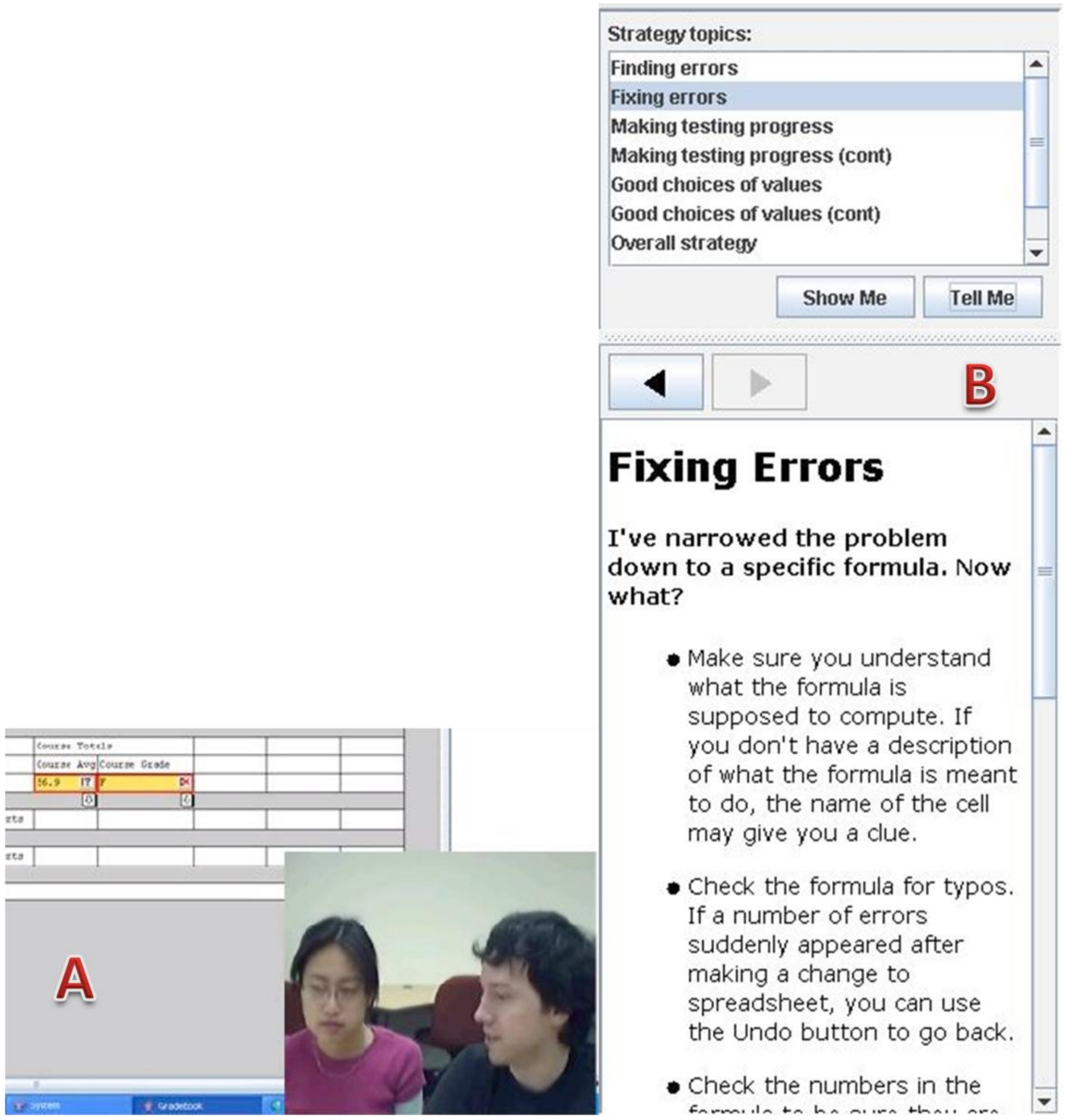
3.4. Experiment

3.4.1. Design

The experiment was a 2x2 between-subjects design with gender (female/male) and condition (Treatment/Control), carried out in a computer laboratory. The Treatment group's environment was the same as the Control group's (Figure 7) with the addition of the "maybe" marks and the strategy explanations features (Figure 8 and Figure 9). We did not isolate each feature in its own treatment because we feared that a single feature alone might not be enough to make a difference. This was important because the overall goal was to ascertain whether it is possible to make a difference in the gender gap through feature design.

3.4.2. Participants

The participants were 65 female and 67 male students from a variety of majors. Participants were required to have previous spreadsheet experience, including use of formulas, but could not have much programming experience. About half the participants were from non-technical majors. There were no significant differences across gender or group in GPA, age, year in school, programming experience, or spreadsheet experience.



Fixing Errors

I've narrowed the problem down to a specific formula. Now what?

- Make sure you understand what the formula is supposed to compute. If you don't have a description of what the formula is meant to do, the name of the cell may give you a clue.
- Check the formula for typos. If a number of errors suddenly appeared after making a change to spreadsheet, you can use the Undo button to go back.
- Check the numbers in the

Figure 9. (a) The 1-minute video snippets show two people working on a spreadsheet. (b) The content is also available in hypertext.

3.4.3. Materials and Task

A pre-session questionnaire gathered background information and self-efficacy based on a slightly modified version of Compeau and Higgins' validated scale [Compeau and Higgins 1995]; the modifications made the questionnaire task-specific to end-user debugging. In post-session questionnaires participants answered the same self-efficacy questions, rated the usefulness of the features, and answered an open-ended question about how parts of the software affected their confidence in finding and fixing bugs.

The hands-on portion began with a 30-minute “tour of features” tutorial, which explained what the features in the spreadsheet environment meant, but it did not imply any strategies for how best to use these features.

The participants were instructed to “test the updated spreadsheet and if [they] find any formula errors, fix them.” The spreadsheet had previously been seeded with six bugs that we have used in many previous experiments [Beckwith et al. 2006; Beckwith et al. 2005; Kissinger et al. 2006; Subrahmaniyan et al. 2007; Subrahmaniyan et al. 2008]; these were originally harvested from end users. Participants had 45 minutes to complete the task. The Treatment group was interrupted after 30 minutes and asked to view either a video or hypertext explanation of their choice. This interruption was necessary to ensure actual usage of the explanations; without usage, we could not have collected data on their usefulness.

3.5. Analysis Methodology

3.5.1. Measures

In this study, we were interested in only certain comparison groups: Treatment females vs. Control females to see if the Treatment helped females, Treatment males vs. Control males to see if males’ performance was affected, and “gender gaps”, i.e., Treatment female/male differences versus Control female/male differences.

In making comparisons, two potential confounds had to be taken into account: gender/group differences in *pre-self-efficacy*, and individual differences in *minutes* available for debugging.

Regarding pre-self-efficacy, analysis of pre-session questionnaires revealed that females in the Treatment group had significantly lower pre-self-efficacy than females in the Control group (Treatment females: *M*=36.82, *SD*=5.22; Control females: *M*=40.13, *SD*=4.28; ANOVA: F(1,63)=7.96, *p*<.0064). (Recall that there were no significant differences otherwise in participant background (e.g., academic areas, age, etc.) among the groups.) Consequently, except where noted, all statistical tests were done in ways that took pre-self-efficacy into account.

Regarding minutes available for debugging, the debugging time available to Treatment participants varied by individual, depending on how much time they spent on explanations. To account for the time differential, we calculated *debugging minutes* as ((45*60) – explanation seconds)/60, where 45 was the number of minutes given to debug the spreadsheet, and explanation seconds was the time a participant spent viewing the strategy explanation videos and/or text. We counted "explanation seconds" from the time they touched a strategy explanation until there was any event (even a simple mouse movement) in the spreadsheet area. Thus, except where noted, all statistical tests were done in ways that also took debugging minutes into account.

To take these covariates into account and to allow direct row-wise and column-wise comparisons (e.g., analyzing female differences by comparing Treatment females vs. Control females), we used one-way ANCOVAs and linear regressions, and in the few cases where ANOVAs were appropriate, we used one-way ANOVAs.

3.5.2. Coding Methodology

The post-session questionnaire's open-ended confidence question asked participants which parts of our software affected their confidence in their ability to fix bugs and how their confidence was affected. To analyze their responses, we began by deriving codes close to the words used by the participants in their answers. We then iteratively regrouped the low-level codes to generalize them. The result was two 3-level trees of codes, one tree for comments about positive effects on confidence (+) and a mirror image for comments about negative effects on confidence (-). The top level of the tree included six codes: environmental conditions, user features, software feedback, information given, software usability, and experiment setup.

Two researchers individually coded 20 participants' answers using this scheme, achieving a 91% agreement rate. Given the high agreement, a single researcher coded all remaining participants' answers.

3.6. Results

Although not our primary research question, the reader may be wondering whether males or females fixed more bugs. If we do not take into account self-efficacy and actual time spent debugging, males fixed significantly more bugs than females (females: *M*=2.88, *SD*=1.75; males: *M*=3.84, *SD*=1.46; ANOVA: F(1,130)=15.67, *p*<.00013) and achieved a significantly higher maximum percent testedness (females: *M*=0.49 (49.1% testedness), *SD*=0.24; males: *M*=0.64, *SD*=0.19; ANOVA: F(1,130)=15.05, *p*<.00017). This is not surprising given our setting. Previous studies have revealed numerous barriers to females' success in debugging spreadsheets, showing that males use spreadsheet testing and debugging features more than females [Beckwith et al. 2006; Beckwith et al. 2007], and that their strategies are better supported by the debugging features present in this environment [Subrahmaniyan et al. 2008].

The issue of interest to this paper is whether the new features in the Treatment group helped to close the gender gap. 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, as we have already discussed. However, as our analysis in the next few sections shows, taking the self-efficacy and time factors into account reveals that the new features helped to close the gender gap in numerous ways.

3.6.1. The Gender Gap and Debugging Features

First we consider the relationship between our feature changes and the debugging feature *usage* gap for males and females. When we compared the males and females in the Treatment group to their counterparts in the Control group, a clear answer emerged: our feature changes did lead 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. The mean and standard deviation for usage of each type of feature are detailed in Table 4.

As in prior studies, we took 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 [Beckwith et al. 2006; Beckwith et al. 2007].

Table 4. The mean features used per debugging minute by Control females, Control males, Treatment females, and Treatment males, as well as the standard deviations.

	CF	CM	TF	TM
Playful Checks	*M*=0.082 *SD*=0.091	*M*=0.20 *SD*=0.25	*M*=0.17 *SD*=0.19	*M*=0.15 *SD*=0.16
Playful X-marks	*M*=0.036 *SD*=0.062	*M*=0.036 *SD*=0.084	*M*=0.10 *SD*=0.18	*M*=0.046 *SD*=0.054
Lasting Checks	*M*=0.51 *SD*=0.40	*M*=0.76 *SD*=0.60	*M*=0.46 *SD*=0.39	*M*=0.66 *SD*=0.43
Lasting X-marks	*M*=0.047 *SD*=0.16	*M*=0.040 *SD*=0.081	*M*=0.069 *SD*=0.097	*M*=0.059 *SD*=0.087
”Maybe” Marks	na	na	*M*=0.32 *SD*=0.45	*M*=0.15 *SD*=0.21
Value Edits	*M*=0.29 *SD*=0.21	*M*=0.41 *SD*=0.30	*M*=0.18 *SD*=0.13	*M*=0.36 *SD*=0.24
Arrows	*M*=0.20 *SD*=0.31	*M*=0.45 *SD*=0.81	*M*=0.14 *SD*=0.19	*M*=0.29 *SD*=0.55
Formula Edits	*M*=0.48 *SD*=0.33	*M*=0.37 *SD*=0.13	*M*=0.40 *SD*=0.26	*M*=0.43 *SD*=0.25

Treatment females approached (tinkered with) the features significantly more than Control females, and this pattern held for both checkmarks and X-marks. Figure 10 illustrates these differences. Regarding these and other feature usage behaviors, Table 5 shows the statistics. For checkmarks, we used ANCOVA, to take self-efficacy into account; for X-marks

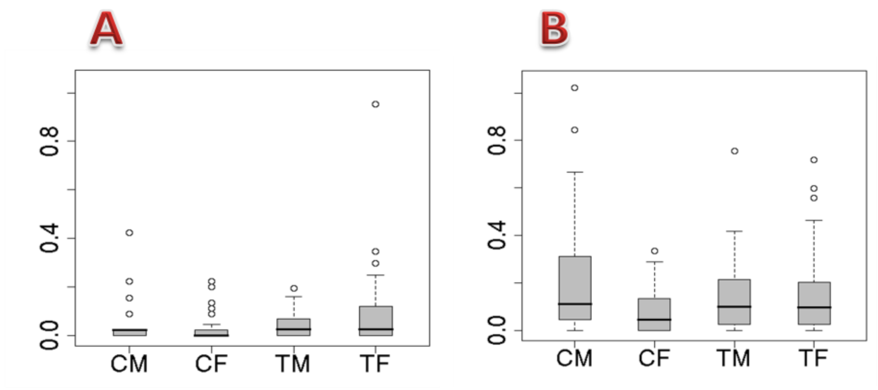


Figure 10. Tinkering with (a) X-marks and (b) checkmarks, in marks applied per debugging minute. Note the gender gaps between the Control females' and males' medians. These gaps disappear in the Treatment group.

Table 5. Tinkering and use of debugging features. 35 Treatment females, 30 Control females, 34 Treatment males, 33 Control males. **: p<0.01, *: p<0.05, ~: p<0.1.

	TF vs. CF	TM vs. CM	Test
Playful checkmarks/minute	TF more: F(2,62)=2.779 *p*<.022*	not significant	ANCOVA
Playful X-marks/minute	TF more: Z=-2.47 *p*<.014*	TM more: Z=-2.02 *p*<.044*	Wilcoxon
Lasting checkmarks/minute	not significant	not significant	ANCOVA
Lasting X-marks/minute	TF more: Z=-2.70 *p*<.070~	TM more: Z=-2.09 *p*<.037*	Wilcoxon
	TF vs. TM		
"Maybe" marks/minute	TF more: F(1,67)=3.98 *p*<.0503~		ANOVA

we used Wilcoxon rank-sum, since the number of ties at zero made the distribution non-normal. Since Wilcoxon does not facilitate a way to take self-efficacy into account, these results are especially noteworthy: *despite* significantly lower self-efficacy, Treatment females showed more interest in the features than Control females.

Why these effects? One reason may be the “maybe” checkmarks and X-marks, which were available in the Treatment environment (see Figure 7). (Recall that the features are designed to ameliorate differences between low self-efficacy and high self-efficacy users. Hence, we did not use self-efficacy as a covariate in this test because doing so makes allowances for lower self-efficacy users, and we wanted results for that test without such allowances.)

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 (total checkmarks per debugging minute: M=0.59, SD=0.43; ANCOVA: F(2,27)=18.04, β=0.0099, R^2=0.57, p<.000010) and in the Treatment group (total checkmarks per debugging minute: M=0.63, SD=0.53; ANCOVA: F(2,32)=31.11, β=0.0074, R^2=0.66, p<.000010). Further, for all participants, maximum percent testedness, accounting for pre-self-efficacy, was a significant factor in the number of bugs fixed (maximum percent testedness: M=0.57 (56.5% testedness), SD=0.22; bugs fixed: M=3.36, SD=1.65; ANCOVA: F(2,129)=8.88, β=2.47, R^2=0.12, p<.00024).

There is one very familiar feature that successful testers and debuggers *must* use: editing. However, overreliance on editing as one’s main problem-solving feature is time-wasting and can lead to introducing errors. Past studies have reported an overreliance by females on these familiar features [Beckwith et al. 2006], a finding consistent with their often low self-efficacy.

In the current study, Treatment females’ usage of editing was superior to Control females’. Specifically, value edits significantly predicted maximum percent testedness per debugging minute for Treatment females (ANCOVA: F(2,32)=4.18, β=0.014, R^2=0.21, *p*<.039), but not for Control females (ANCOVA: F(2,27)=0.77, β=0.0056, R^2=0.054, *p*<.28). Testing is trying values, but testedness advances based on *coverage* of formula interrelationships. Thus, in order to increase the spreadsheet's testedness, value edits must be done intelligently, such as by picking values that test each clause of an if/then expression. The fact that value edits were predictive of Treatment females' maximum percent testedness shows the participants were indeed choosing effective values, thus implying that they were using the testing strategy supported by the WYSIWYT features.

Treatment females' formula edits did not predict testedness (ANCOVA: F(2,32)=1.61, β=-0.0001, R^2=0.091, *p*<.98), which makes sense since testing is about values. However, Control females' did—but in a negative direction (ANCOVA: F(2,27)=2.39, β=-0.0064, R^2=0.15, *p*<.045)! Obviously, the greater proportion of time people devote to editing formulas, the less time they have to use the testing and debugging features. Overreliance on editing formulas thus interferes with real progress in testing, just as happened to the Control females.

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, except where noted above, there were no significant differences between the male groups. Most important, none of the changes benefiting the females showed adverse effects on the males.

3.6.2. The Gender Gap in Self-Efficacy

Self-efficacy has been found to be important to both feature usage and to various measures of debugging success for female end-user programmers [Beckwith et al. 2006; Beckwith et al. 2007]. Thus, reducing the gender gap in self-efficacy was one of the goals of the new features.

Debugging is a difficult task, and in our past studies, participants' confidence in their debugging has always decreased during the experiments. Consistent with this but encouraging, in the current study, measuring the change between pre- and post-self-efficacy revealed suggestive evidence that Treatment females' decrease was less than Control females' decrease (Control females' change in self-efficacy: *M*=-4.05, *SD*=5.00; Treatment females: *M*=-1.91, *SD*=5.14; ANOVA: F(1,63)=2.86, *p*<0.096); see Figure 11.

Low self-efficacy is appropriate when it judges one's abilities accurately. The problem is only when it is *inappropriately* low (underconfidence) or inappropriately high (overconfidence), either of which can lead to problems with persistence and strategy decisions. That is the reason one goal of the features was to improve people's ability to self-judge.

As Table 6 shows, for both groups of males, their post-self-efficacy was appropriate for (predicted by) their debugging performance. Evidence of this regarding Treatment females was suggestive. There was no evidence of it for Control females.

3.6.3. What Females Said About Confidence

We triangulated the suggestive evidence of positive effects on Treatment females' self-efficacy with evidence from what they said.

A post-session questionnaire asked participants which parts of the software affected their confidence in their ability to fix bugs. We analyzed their free-text responses, both

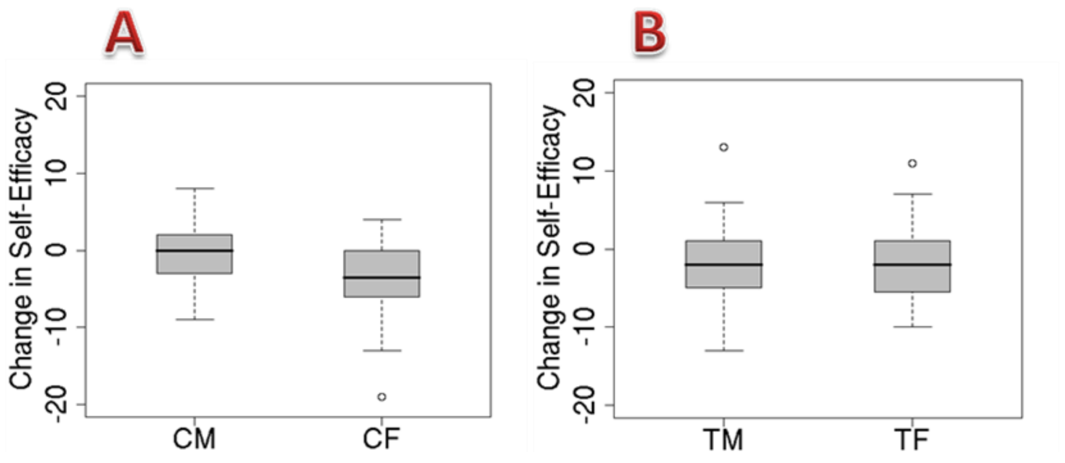


Figure 11. Change in self-efficacy for (a) Control participants and (b) Treatment participants.

positive and negative. An example of a positive statement from the Treatment group was: "I really liked the video. It was very helpful and easy to use. I also like the X-mark, checkmark system."

Treatment females said more positive things about the features and information provided to them. Table 7 summarizes the analysis results at or approaching significance for females. One Treatment participant wrote "I did not know how to test the cells until I

Table 6. Means, SD, and results of linear regression tests comparing whether bugs fixed per debugging minute is predictive of post-self-efficacy for Control females, Control males, Treatment females, and Treatment males. **:p<0.01, *: p<0.05, ~: p<0.1.

CF	CM	TF	TM
Bugs: *M*=0.067 *SD*=0.042 Post-SE: *M*=36.12 *SD*=5.55	Bugs: *M*=0.092 *SD*=0.026 Post-SE: *M*=39.50 *SD*=4.70	Bugs: *M*=0.065 *SD*=0.039 Post-SE: *M*=34.86 *SD*=6.49	Bugs: *M*=0.084 *SD*=0.040 Post-SE: *M*=39.12 *SD*=5.51
F(1,28)=2.07 β=34.46 R^2=0.069 *p<.17*	F(1,31)=6.92 β=76.79 R^2=0.18 *p<.014**	F(1,33)=3.62 β=52.86 R^2=0.099 *p<.066~*	F(1,32)=6.33 β=56.49 R^2=0.17 *p<.018**

Table 7. Means, SD, and Wilcoxon rank-sum tests results of 35 Treatment females' and 30 Control females' responses regarding what influenced their confidence. **:p<0.01, *: p<0.05, ~: p<0.1.

	CF	TF	Result
Positive statements overall	*M*=0.63 *SD*=0.93	*M*=1.26 *SD*=1.20	TF more: Z=-2.29, *p*<.022*
Negative statements about experiment	*M*=0.40 *SD*=0.50	*M*=0.20 *SD*=0.47	TF less: Z=1.92, *p*<.055~
Positive statements about information	*M*=0.03 *SD*=0.18	*M*=0.37 *SD*=0.65	TF more: Z=-2.70. *p*<.0070**
Positive statements about features	*M*=0.10 *SD*=0.31	*M*=0.29 *SD*=0.52	TF more: Z=-1.63, *p*<.103

watched the video." A Control participant wrote, "The x function and highlight of affecting boxes was helpful to find bugs."

Treatment males did not differ much from Control males in positive or negative statements. See Figure 12 for a comparison with females. However, the Treatment males did appreciate the availability of explanations: they spoke more positively about information than their Control counterparts did (Wilcoxon rank-sum test: Z=-2.21, n=67, *p*<.028). There were no other significant differences for males.

The significant differences in positive statements by Treatment females support the hypothesis that the Treatment environment supported females' confidence better than the Control environment.

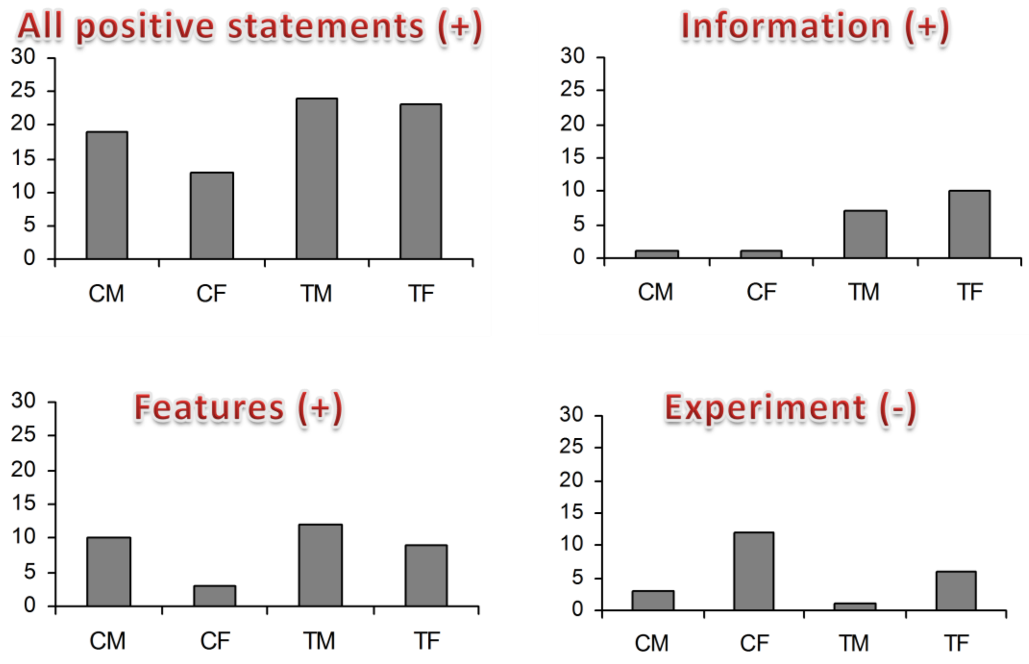


Figure 12. Selected counts of participants making positive or negative statements. Bars represent the number of participants in each group who made each type of comment.

3.7. Discussion

As we mentioned earlier, this experiment did not tease apart the impacts of the nuanced ("maybe") judgments feature from the impacts of the strategy explanations. We expected the use of strategy explanations to be a factor in closing the gap between females' and males' debugging success, as recent studies have shown the benefit of tutorial materials on females' performance in software development tasks [Hartzel 2003; Kelleher and Pausch 2005; Subrahmaniyan et al. 2007]. But we do not know whether this was the case; we did not find direct evidence of an impact of the strategy explanations alone on female success rates.

We had hoped to investigate this by analyzing relationships between the number of minutes spent viewing explanations and measures of each participant's success. Assessing ties to success involves interaction of participants' use of strategies and their self-efficacy. The relationships in our data appear to be both complex and non-linear, making it difficult to interpret their meaning. Further research will require in-depth qualitative investigation to better understand what individuals learned from the strategy explanations and how they applied this knowledge to their debugging task.

The combination of nuance and strategy explanations was tied to numerous quantifiable benefits for females. Gender differences in problem-solving strategies may explain this outcome. In a recent study [Subrahmaniyan et al. 2008], we found eight strategies that male and female participants used to debug spreadsheets, such as "code inspection," "dataflow," "fixing formulas," and "testing". Of the eight strategies, there were significant gender differences in seven. None of the females' most effective strategies (code inspection and specification checking) are well supported by spreadsheet software (ours or others'), whereas males' most effective strategies (testing and dataflow) are well supported.

The testing and dataflow-following strategies were not tied with success for females in the study of [Subrahmaniyan et al. 2008]. Yet, the new features in this study try to encourage their use, and succeeded: females indeed seem to have done just what we encouraged, namely applying the recommended strategies of testing and following dataflow, which worked to their advantage. It is not clear, however, whether nudging females in this direction is ultimately the ideal approach. In the future, we plan to investigate whether adding features supportive of strategies favored by females, such as code inspection, will be a better approach than encouraging them toward the strategies currently favored by males.

3.8. Conclusion

Our 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. Specifically, with the addition of nuancing to our testing judgment feature and video/text strategy explanation snippets:

- Regarding RQ1, females' usage of testing/debugging features increased, and this feature usage translated into testing and debugging improvements;
- Regarding RQ2, females' confidence levels improved and their self-judgments were roughly appropriate indicators of their actual ability levels; and
- Also regarding RQ2, females' post-session verbalizations showed that their attitudes toward the software environment were more positive.

Furthermore, these gains for females came without disadvantaging the males. Designers of end-user programming environments can therefore deploy these features into their own environments to help remove unintended barriers that currently impede females' full participation in end-user programming tasks.

PART 2 – UNDERSTANDING DEBUGGING STRATEGY ITEMS:

Four Empirical Studies

<u>Overview of Part II of the Dissertation:</u>

What end-user debugging stratagems do males and females use?

In chronological order, the second study I led on this topic was the Forms/3 Stratagems Study (Subrahmaniyan, Beckwith, et al. 2008). This was our first look inside our participants' heads to find out what higher level strategy items they were attempting to use. Recall that *stratagems* are systematic plans for subparts of the task, as opposed to *strategies*, which are about the entire task. This study uncovered eight debugging stratagems employed by end-user programmers and their ties to both gender and debugging success:

- Participants described eight stratagems: dataflow, testing, code inspection, specification checking, color following, to-do listing, fixing formulas, and spatial.
- Males engaged in dataflow and testing to much greater extents than females did and these two stratagems were successful primarily for males, but not for females. Color following was also a male stratagem.
- Instead, the females' major successful stratagems were code inspection and specification checking. To-do listing was also a female stratagem, while fixing formulas was detrimental to female success.

While this study revealed *what* stratagems were used effectively by males and females, the statistical experimental setup does not allow us to see *how* those stratagems were used successfully. The next two chapters address this gap in our current understanding of stratagems and test whether the stratagems generalize across other environments and populations.

Chapters 4 and 5: Do these stratagems generalize across to IT Professionals debugging PowerShell scripts? How are the stratagems used successfully by males and females? Do these findings also generalize to interactive application designers?

Studies using academic prototypes, populations, and tasks (such as the Data Mining Study, the Feature Usage Study, and the Forms/3 Stratagems Study reported earlier) have clear advantages: the features presented in those studies can be very carefully controlled, participants are fairly easy to come by, and there are very few limitations on how the software can be modified for future studies. However, a tradeoff of such studies is that the results also pertain only to that very specific academic setting: they have low external validity. Thus, in the studies reported in this second part of the dissertation, we first explore how the findings from the statistical Forms/3 Stratagems Study generalize across different populations, experimental setups, and commercial environments.

The PowerShell Stratagems Study (Chapter 4): Since our end-user programmers in the Forms/3 Stratagems Study might have missed some stratagems in their post-session answers, we first switched to a qualitative think-aloud study to observe stratagem usage as it occurs. This study was of IT Professionals using Windows PowerShell, which allowed us to observe in detail *how* stratagems are used during debugging, in addition to finding out *what* stratagems are used as previously.

Designers Needs Study (Chapter 5): The second study was a grounded theory study in which we analyzed existing designers' artifacts (e.g., emails) augmented by newly collected data (e.g., interviews) to classify professional designers' end-user programming problem-solving needs in several of the most popular design environments. This is the only generalization study in which we derived a completely new set of needs from the ground up and compared them to the final set of stratagem codes from the PowerShell Stratagems Study. This is important, since starting with an earlier code set might have biased us toward those stratagems.

The PowerShell Stratagems Study and the Designers Grounded Theory Study allowed us to see whether the stratagems revealed by our Forms/3 Stratagems Study generalized across environments, populations, and experimental setups. In general, they did. Some highlights of these studies' findings include:

- All but one of the stratagems found with the Forms/3 student participants also applied to IT professionals debugging PowerShell scripts, along with three more that emerged. Furthermore, all but two of the stratagems mapped to designers' debugging needs in many popular interactive application design environments.
- The seven stratagems we observed in both the Forms/3 Stratagems and the PowerShell Stratagems Studies were: *testing*, *code inspection*, *specification checking*, *dataflow*, *spatial*, *feedback following* (a generalization of the strategy previously termed *color following)*, and *to-do listing.* In addition, we observed the following three stratagems that had not been present in the spreadsheet study: *control flow*, *help*, and *proceeding as in prior experience*.
- An analysis of designers' debugging needs revealed a set of 20 design creation, iteration, and communication needs (in order of importance): flow, feel, look, usability, reuse, testing, communication, themes, training, optimization, external factors, granularity, automation, extensibility, bugs, cleanup, compatibility, jargon, propose, and settings.
- Examining the ties between *bugs* (or having bug-free code) and the other needs, the Designers Needs Study revealed that most of the PowerShell Stratagems Study debugging stratagems also apply to debugging in interactive application design environments (e.g., Microsoft Blend, Adobe Dreamweaver, and Adobe Flash). *Spatial* and *to-do listing* were the only stratagems which did not come up as needs for designers as well. The stratagems which did map were *code inspection* (to bugs), *control flow* (to flow), *dataflow* (to flow), *feedback following* (to jargon), *help* (to training), *proceeding as in prior experience* (to reuse), *specification checking* (to external constraints), and *testing* (to testing, compatibility, and settings).
- This study also adds three new stratagem candidates to the generalized list of stratagems: *cleaning up automatically generated code*, *switching levels of granularity*, and *usability testing*. While these stratagems appear to be more applicable to designers, they are likely to also apply to our other populations (e.g.,

spreadsheets are often highly formatted, which might be a sign that the author's thinking about usability).

- In the PowerShell Stratagems Study, we also observed some of the other levels of strategy items. In particular, we listed several moves and tactics employed by scripters and the need for support for *systematic* incremental testing, for easy inspection of large amounts of code and of code mini-patterns, for "drill down" into related testing information during code inspection and into related code information during testing, for informal specification checking, and for to-do listing.
- Most interestingly, all of the findings from the PowerShell Stratagems Study were consistent both in terms of gender ties and effectiveness ties to the results from the Forms/3 Stratagems Study. Observations about *how* stratagems were used successfully included a female's effective code inspection usage and males' effective testing and dataflow usage.

The PowerShell work also raised a significant new open question: whether males' and females' uses of debugging stratagems differ not only in *which* stratagems they use successfully, but also in *when* and *how* they use those stratagems. The Designers Grounded Theory Study, on the other hand, warns us about the list of stratagems' completeness, but validates the generalizability of the stratagems we observed.

Chapter 6: Can the Sensemaking Model reveal end-user debugging strategies?

In the studies presented up to here, we have uncovered and examined three levels of strategy items: moves, tactics, and stratagems. However, they have all come short of revealing end-user debuggers' *strategies.* Next, we therefore explored a new methodological approach to uncovering debugging strategies: viewing end-user debugging through a sensemaking lens.

Sensemaking is a term used to describe how people make sense of the information around them, and how they represent and encode that knowledge, so as to answer task-specific questions [Russell et al. 1993]. Thus, just like *strategies*, the model covers the entire problem-solving process: from the data to the presentation of the results.

Indeed, patterns in how participants traversed the sensemaking model revealed two successful strategies for systematic sensemaking. These findings were consistent with gender difference literature in information processing styles. Application of sensemaking research to end-user debugging revealed several other interesting findings as well, including: a new model of sensemaking by end-user debuggers, the dominance of information foraging in our end users' debugging efforts, a detailed account of transitions among sensemaking steps and among the three sensemaking loops in our model, the sensemaking sequences that were tied to successful outcomes versus those that identified trouble, and specific sensemaking trouble spots and consequent information losses.

MALES' AND FEMALES' SCRIPT DEBUGGING STRATEGIES

Valentina Grigoreanu, James Brundage, Eric Bahna,

Margaret Burnett, Paul ElRif, and Jeffrey Snover

IS-EUD'09 (Siegen, Germany: March 2-4, 2009)

Springer-Verlag, Berlin, Heidelberg.

Issue manuscript appears in *Lecture Notes in Computer Science, Programming and Software Engineering*, Vol. 5435, pages 205-224.

4. Generalizing Across: The POWERSHELL STRATAGEMS STUDY

4.1. Abstract

Little research has addressed IT professionals' script debugging strategies, or considered whether there may be gender differences in these strategies. What strategies do male and female scripters use and what kinds of mechanisms do they employ to successfully fix bugs? Also, are scripters' debugging strategies similar to or different from those of spreadsheet debuggers? Without the answers to these questions, tool designers do not have a target to aim at for supporting how male and female scripters want to go about debugging. We conducted a think-aloud study to bridge this gap. Our results include (1) a generalized understanding of debugging strategies used by spreadsheet users and scripters, (2) identification of the multiple mechanisms scripters employed to carry out the strategies, and (3) detailed examples of how these debugging strategies were employed by males and females to successfully fix bugs.

4.2. Introduction

At the border between the population of professional developers and the population of end-user programmers, lies a subpopulation of IT professionals who maintain computers, and they accomplish much of their job through scripting. As Kandogan et al. argue, this population has much in common with end-user programmers [Kandogan et al. 2005]: as in Nardi's definition of end-user programmers, they program as a means to accomplish some other task, not as an end in itself [Nardi 1993]. Scripting is also becoming much more common by end-user programmers themselves through the advent of end-user oriented scripting languages for the desktop and the web. However, despite the complexity of some scripting tasks, little attention has been given to scripters' specific debugging needs, and even less to the impact that gender differences might have on script debugging strategies and the mechanics used to support them.

We therefore conducted a qualitative study to address this gap by identifying the debugging strategies and mechanisms scripters used. *Strategy* refers to a reasoned plan or method for achieving a specific goal. *Mechanisms* are the low-level tactics used to support those strategies: through environment and feature usage. Our work was guided by the debugging strategies reported in an earlier end-user debugging study with spreadsheet users [Subrahmaniyan et al. 2008].

There are several reasons to ask whether strategies used by scripters working with a scripting environment might be different from strategies used by end-user programmers with a spreadsheet system. First, the populations are different; for example, one might expect scripters to have more experience in debugging per se than spreadsheet users. Second, the language paradigms are different: scripting languages are control-flow oriented, in which programmers focus primarily on specifying sequence and state changes, whereas spreadsheet languages are dataflow oriented, in which programmers focus primarily on specifying calculations (formulas) that use existing values in cells to produce new values. The language paradigm differences lead naturally to a third difference: the environments' debugging affordances themselves are different, with scripting environments tending toward peering into sequence and state, whereas spreadsheets' affordances tend more toward monitoring values and how they flow through calculations.

Therefore, the research questions we investigated were:

RQ1: What debugging strategies do scripters try to use?

RQ2: What mechanisms do scripters employ to carry out each strategy?

RQ3: How do our findings on scripters' strategies relate to earlier results on strategies tied with male and female spreadsheet users' success?

Thus, the contributions of this paper are in (1) identifying the strategies scripters try to use in this programming paradigm, (2) identifying the mechanisms scripters use to carry out their different strategies, and (3) exploring details of successful uses of the strategies by males and females.

4.3. Background and Related Work

Although there has been work in how to effectively support system administrators in *creating* their scripts [Kandogan et al. 2005], we have been unable to find work addressing scripters' debugging *strategies*. Instead, most of the work on script debugging has been on tools to automatically find and fix errors (e.g., [Whitaker et al. 2004; Yuan et al. 2006]).

However, there has been considerable work on professional programmers' debugging strategies, and some work on end-user debugging strategies. One study on professional programmers' debugging strategies classified debugging strategies as forward reasoning, going from the code forward to the output, and backward reasoning, going from the output backward through the code [Katz and Anderson 1988]. See Romero et al. for a survey of professional programmers' debugging strategies [Romero et al. 2007].

End-user programmers have elements in common with novice programmers, so the literature on how novice programmers differ from experts is relevant here. For both novices and experts, getting an understanding of the high-level program structure before jumping in to make changes relates to success [Littman et al. 1986; Nanja and Cook 1987]. However, experts have been found to read programs differently from novices: reading them in control flow order (following the program's execution), rather than spatial order (top to bottom).

In end-user programming, gender differences have been found in attitudes toward and usage of end-user programming and end-user programming environment features [Beckwith et al. 2005; Beckwith et al. 2006; Beckwith et al. 2007; Grigoreanu et al. 2008;

Ioannidou et al. 2008; Kelleher et al. 2007; Rode et al. 2004; Rosson et al. 2007]. Especially pertinent is a series of end-user debugging studies reporting gender differences in debugging strategies for spreadsheets [Grigoreanu et al. 2006; Subrahmaniyan et al. 2008]. The first of these studies pointed to behavior differences that suggested strategy differences, and the second reported a set of eight strategies end users employed in their debugging efforts. In both of these studies, the strategies and behaviors leading to male success were different from those leading to female success. For example, in [Subrahmaniyan et al. 2008], dataflow strategies played an important role in males' success, but not females'. Prabhakararao et al.'s spreadsheet debugging study with end users also reported a strong tie between using a dataflow strategy and success [Prabhakararao et al. 2003], but participant gender was not collected in that study.

In fact, gender differences that relate to processing information and solving problems have been reported in several fields. One of the most pertinent works is the research on the Selectivity Hypothesis [Meyers-Levy 1989; O'Donnell and Johnson 2001]. It proposes that females process information in a comprehensive way (e.g., attending to details and looking for multiple cues) in both simple and complex tasks. Males, on the other hand, process information through simple heuristics (e.g., following the first cue encountered), only switching to comprehensive reasoning for complex tasks.

Self-efficacy theory may also affect the strategies employed by male and female debuggers [Bandura 1986]. Self-efficacy is a person's confidence about succeeding at a specific task. It has shown to influence everything from the use of cognitive strategies, to the amount of effort put forth, the level of persistence, the coping strategies adopted in the face of obstacles, and the final performance outcome. Regarding software usage, there is specific evidence that low self-efficacy impacts attitudes toward software [Beckwith et al. 2005; Hartzel 2003], that females have lower self-efficacy than males at their ability to succeed at tasks such as file manipulation and software management tasks [Torkzadeh and Koufteros 1994], and that these differences can affect females' success [Beckwith et al. 2005].

Gender differences in strategies also exist in other problem-solving domains, such as psychology, spatial navigation, education, and economics (e.g., [Gallagher et al. 2000]). One goal of this paper is to add to the literature on gender differences in problem solving relating to software development, by considering in detail the usage of different strategies by male and female scripters.

4.4. Study

4.4.1. Participants

Eleven IT professionals (eight males and three females) volunteered to participate in the study by responding to invitations on an IT-related internet forum and on a PowerShell email discussion list. Participants received software as gratuity. Although we had hoped for equal participation by females and males, female IT professionals are in short supply, and only three signed up. Almost all participants had a technical college degree (in computer science, engineering, or information systems), with the exception of two males whose education ended after high school. Despite their technical degrees, six of the eleven participants rejected the label of "software developer." Those who did classify themselves as software developers were the three females and two of the eight males; the remaining six males described themselves as IT professionals or scripters. All participants reported that, in their everyday jobs, they accomplished their IT professional tasks using PowerShell. As examples of these regular tasks, participants mentioned moving packages, moving machines out of a domain, modifying the registry, initializing software, automating IT tasks, automating tests, and creating test users on servers.

All participants had written two or more scripts within the past year, using Windows PowerShell. The females had written fewer scripts in the past year than the males (number of scripts written by females: 2, 3, and 5; number of scripts written by males: 6, 6, 7, 20+, 30, 30+, 50, 100+).

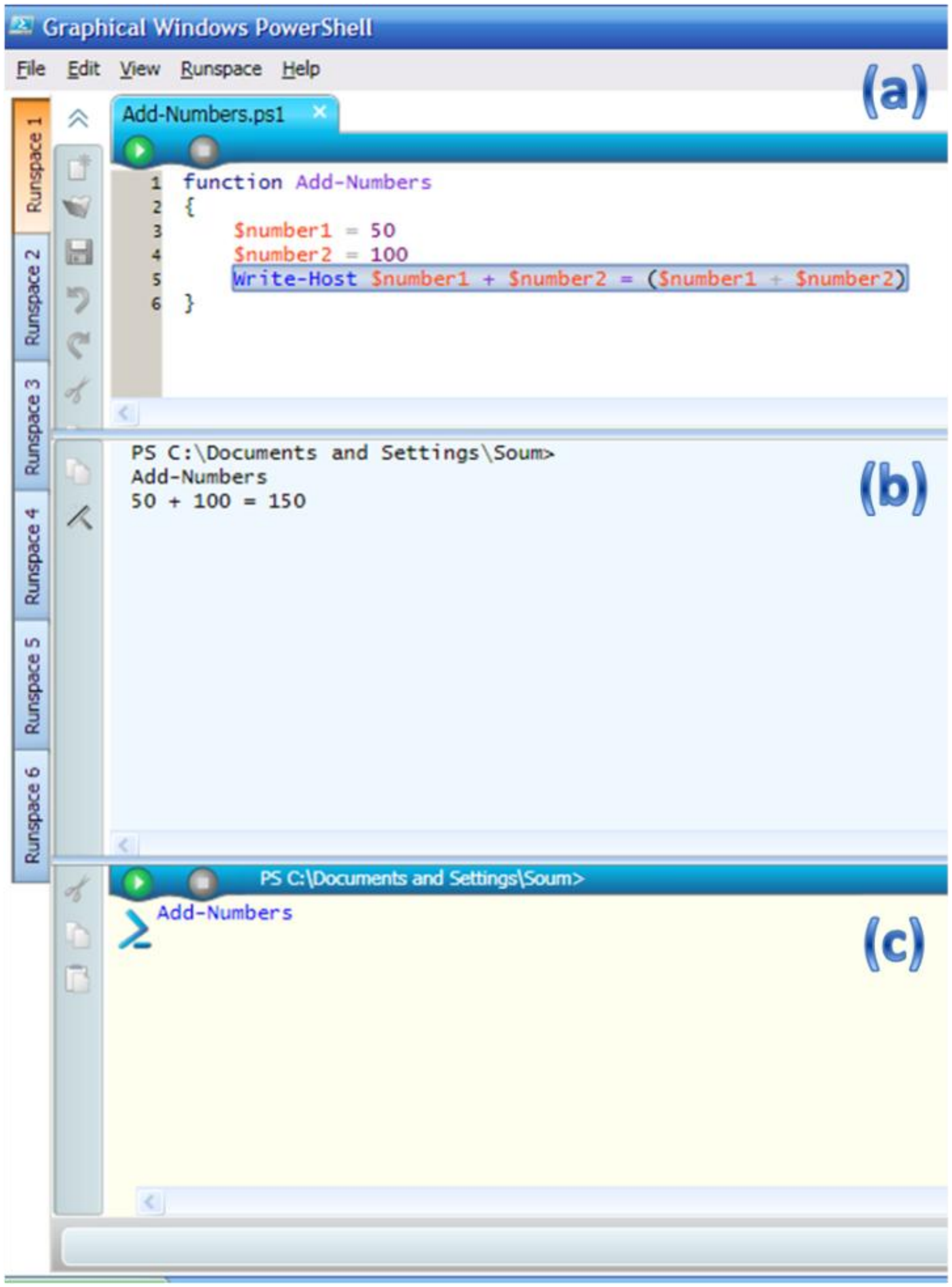
4.4.2. Scripting Language and Environment

PowerShell is a new implementation of the traditional Command Line Interface and scripting language developed by Microsoft, which aims to support both IT professionals' and developers' automation needs. We used an as-yet unreleased version of this language and environment in our study. PowerShell supports imperative, pipeline, object-oriented, and functional constructs. Its pipelining, unlike traditional UNIX commands that pipeline *text* to one another, pipelines *objects* to one another. PowerShell has both a command line shell and a graphical scripting environment, and participants used both. Both the command line and graphical scripting environment provide common debugging features such as breakpoints, the ability to step into, error messages, and viewing the call stack. Figure 13 shows a version of the graphical scripting environment that is similar to the one our participants used.

4.4.3. Tasks, Procedure, and Data Collection

We instructed the participants to debug two versions of a PowerShell script, which included a "main" section and eight called functions, each of which was in a separate file within the same directory. We used the same script (two versions) for both tasks in order to minimize the amount of time participants spent getting an understanding of the scripts so as to maximize the amount of time they spent actually debugging.

The script was a real-world script that one of us (Brundage) had previously written to collect and display meta-data from other PowerShell scripts. We introduced a total of seven bugs, which we harvested from bugs made by the script's author when he originally wrote the script. Each version of the main script contained one different bug. The eight functions called by both versions contained five other bugs. The seven total bugs fell into two categories: three errors *in data*: using an incorrect property, allowing the wrong kind of file as input, and omitting a filter; and four *errors in structure*: an assignment rather than a comparison, an off-by-one error, an infinite loop, and omitting the code that should have handled the last file.



```
function Add-Numbers
{
    $number1 = 50
    $number2 = 100
    Write-Host $number1 + $number2 = ($number1 + $number2)
}
```

```
PS C:\Documents and Settings\Soum>
Add-Numbers
50 + 100 = 150
```

```
PS C:\Documents and Settings\Soum>
Add-Numbers
```

Figure 13. This is the graphical version of the Windows PowerShell environment. (a) Scripts are written in the top pane, and the example shows a function that adds two numbers. (b) The output pane displays the result of running the script or command. (c) The command line pane is used exactly like PowerShell's command line interface. In this case, it is running the function to add two numbers.

After participants completed a profiling survey, we gave them a description of what the script was supposed to do. Participants then debugged one version of the script using a command line debugger and a second version using a graphical debugger. The order of the script versions and environments was randomized to control for learning effects.

Participants were instructed to talk aloud as they performed their debugging tasks. Data collected included screen captures, video, voice, and measures of satisfaction. After completion of each task, participants were given a post-session questionnaire that included an interview question about the debugging strategies they used.

4.4.4. Analysis Methodology

We analyzed the data using qualitative content analysis methods. We analyzed two data sources: participant responses to the questionnaire, and the videos. Because we wanted to measure the extent to which males' and females' debugging strategies identified in previous work [Subrahmaniyan et al. 2008] would generalize to our domain and population, we began by mapping that code set to the Powershell domain of the current study.

One researcher applied these codes to participants' post-session open-ended interview responses about strategies. We asked the same strategies question as in the earlier work mentioned above, but it was asked verbally, rather than on paper. This led to generalizations of a few of the codes and the introduction of a few new codes.

Two researchers then used this revised code set as a starting code set for the videos. They independently coded 27% of the videos (six tasks from various participants), achieving 88% agreement. The dually coded set included the three participants described in 4.5.2, for which the two researchers also collaboratively analyzed the circumstances, sequences, and mechanisms, resolving any disagreements as they arose. This also resulted in our final code set, which is given in Section 4.5. The first researcher then analyzed and coded the remaining videos alone.

4.5. Results

4.5.1. Scripters' Debugging Strategies

Our scripters used variants of seven of the eight strategies the spreadsheet users used [Subrahmaniyan et al. 2008], plus three others. The spreadsheet strategy termed "fixing formulas" (in which participants wrote only of editing formulas, but not of how they figured out what, where, or how to fix the bug or of validating their changes) was not found in our data.

As Table 8 shows, five of the strategies they used were direct matches to the earlier spreadsheet users' strategies, two were matches to generalized forms of the spreadsheet users' strategies, and three arose that had not been viable for the spreadsheet users. However, even for the direct matches, the mechanisms scripters used to pursue these strategies had differences from those of the spreadsheet users.

Direct Matches. *Testing* is trying out different values to evaluate the resulting values. Some of the mechanisms used by these scripters would not traditionally have been identified as testing, yet they clearly are checking the values output for correctness—but at finer levels of granularity than has been possible in classic software engineering treatments of the notion of testing.

Specifically, we noticed three types of testing mechanisms used by participants: testing different situations from a whole-program perspective, incrementally checking variable values, and incrementally testing in other ways. The first type is classic testing methodology to cover the specifications or to cover different parts of the code (testing both the antecedent and the consequent parts of an if-then statement, for example). The latter two are informal testing methods to see whether, after having executed part of the code, the variables displayed reasonable values. For example, from a whole-program perspective:

Female P0721081130 ran the code and examined both the error messages and the output text in order, rather than focusing on either one or the other.

Female P0718081400 tried different contexts by cd-ing back to the root directory in the command line before running the file again using the menu.

Male P0717080900 changed the format of the output so that he could understand it more easily when he ran the program.

Table 8. Participants' responses when asked post-session to describe their stratagems in finding and fixing the bugs (listed in the order discussed in the text).

Stratagem	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 Stratagems	
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.

However, incrementally checking variable values was much more common, and participants did it in many different ways:

Male P0718081030 hovered over variables to check their values.

Male P0117081130 also hovered over, but in conjunction with breakpoints to stop at a particular line to facilitate the hover.

Female P0718081400 ran the code by accessing it through the command line interface. Others preferred reaching it through the menu.

Female P0718081400 typed the variable name in the command line.

Male P0717080900 added temporary print statements to output variable values at that point in time.

Male P0721081330 added temporary print statements to check whether a particular part of the code was reached/covered by the input.

Female P0718081400 would have liked to use a watch window to examine variable values.

Male P0718081030 examined an entire data structure using tabs (auto-complete) to determine the correctness of its property values.

Other forms of incremental testing focused on running part of the code to check its output. For example:

Female P0718081400 did not understand why she was getting an "access denied" message and therefore tried performing the action manually with Windows Explorer and navigating to that directory (to see if she would get the same message in that different context).

Female P0718081400 first ran a variable to see its output, and then started adding the surrounding words to get more information about that variable and how it is being used.

Female P0717081630 used "stepping over" to see the script's output appear incrementally as she passed the line in the code that produced it.

Male P0721080900 wanted to break into the debugger once a variable had a particular value.

Primarily only the first category of testing is supported by tools aiming to support systematic testing for professional programmers or end-user programmers (e.g., WYSIWYT [Burnett et al. 2004]). However, our scripters were very prone to incremental testing, and although the scripting environment gives them good access to checking these values, there is no support in that environment or most others for using this incremental testing to *systematically* track which portions of the code are tested successfully, which have failed, and which have not participated in any tests at all.

Code inspection is examining the code to determine its correctness. Code inspection is a counterpart to testing with complementary strengths [Basili and Selby 1987]. It is heavily relied upon in the open source community [Rigby et al. 2008]. Not surprisingly, as in the spreadsheet study, testing and code inspection were the most common strategies. Participants' mechanisms for code inspection revealed a surprisingly large set of opportunities for supporting code inspection better in scripting environments, spreadsheets, and other end-user programming environments.

Besides simply reading through the code, some of the basic mechanisms the participants used were:

Female P0718081400 opened up all of the files in the same directory as the script (to view functions the main script was calling), and quickly scanned through all of them one after the other.

Male P0718081030 resized the script pane to show more of the script.

Male P0717080900 used the "Find" function to jump to the part of interest.

Male P0718081030 used the command line to find out all the contextual information he could about a variable he was inspecting (its type, for example).

Male P0717080900 used the integrated scripting environment as a code editor for the command line task because he disliked inspecting the code without syntax highlighting.

The above five mechanisms may seem obvious, but many end-user programming environments do not support these functionalities. For example, they are not well supported in spreadsheets; in that environment performing these actions is awkward and modal. Given the heavy reliance on code inspection by the participants in both this study and the previous spreadsheet study, a design implication for end-user programming and scripting environments is to provide support for the flexible and easy ability to inspect large amounts of the code when desired.

Finally, there were many instances of integration between testing and code inspection, such as this participant's fine-grained mixing of the two:

Female P0718081400 hovered over variables in the code view for simultaneously seeing both the code and output values.

Most participants in the earlier spreadsheet study also used testing and code inspection together. The preponderance of mixing these strategies suggests that programming environment designers should strive to support this mixture, allowing "drill down" into related testing information during code inspection, as in the example above, and conversely allowing drill down into related code information during testing. Getting directly to the code that produced certain values is well supported in spreadsheets and in some end-user languages and environments such as KidSim/Cocoa/Stagecast [Heger et al. 1998] and Whyline [Ko and Myers 2004], but is rarely present in scripting environments.

Specification Checking is somewhat related to code inspection, but involves comparisons: namely, comparing descriptions of what the script should do with the script's code. This strategy is not well supported in any scripting or end-user programming environment—code comments are the primary device to which users in these environments have access for the purpose of specification checking.

Both the spreadsheet study and this one provided (informal) specifications in the form of written descriptions of the intended functionality, and these were widely used by both the previous study's spreadsheet users and the current study's scripters. In addition, they relied on comments and output strings embedded in the code for this purpose, as in the examples below.

Male P0717080900 read the informal description handout related to the script.

Male P0717080900 read the comments in the code related to what that part of the code was supposed to do.

Male P0117081130 looked in the code for the places producing constant string outputs, with the view that those string outputs helped describe what nearby code what supposed to do.

Female P0718081400 read the comments one-by-one, as she was reaching the parts that they referred to in control flow order.

Thus, specification checking is an under-supported strategy for both spreadsheet users and scripters.

Dataflow means following data dependencies through the program. Following dataflow is a natural fit to the dataflow-oriented execution model of spreadsheets, and some spreadsheet tools provide explicit support for it such as dataflow arrows and slicing-based fault localization tools [Burnett et al. 2004]. Even in imperative programs, dataflow mixed with control flow (i.e., "slicing") is commonly used [Subrahmaniyan et al. 2008], and ever since Weiser's classic study identified slicing as an important strategy for debugging [Weiser 1982], numerous tools have been based on slicing. Our scripters followed dataflow a little, but it was not particularly common, perhaps because the scripting environment did not provide much explicit support for it:

Female P0717081630 said, "Wish I could go to where this variable is declared."

Female P0717081630 tried to "find all references" to a variable, in any file.

Male P0718081030 wanted to know how a particular variable got to be a certain value, and therefore followed the flow of data to see what other variables influenced this variable, and how it got to be the value it was.

Spatial is simply following the code in a particular spatial order. For example, scripts can be read from top to bottom. (This is different from following execution order; execution order deviates from top to bottom at procedure calls, loops, etc.) Most participants demonstrated a little of this strategy, but nobody relied on it for very long. It was fairly uncommon in the spreadsheet study as well, in which fewer than 10% of the participants mentioned that strategy.

Generalized Matches. The two strategies that matched generalizations of strategies observed in the spreadsheet study were Feedback Following and To-Do Listing.

Feedback Following is using system-generated feedback to guide debugging efforts. This is a generalization of the strategy "Color Following" in the spreadsheet study. To draw users' attention to them, the spreadsheet system colored cells' interiors to show their likelihood of containing errors (based on the judgments made by users about the correctness of each cell's value). The users who followed this type of feedback directly were considered to be color following. The scripting environment used certain messages (not colors alone) to draw users' attention to code with possible bugs, a generalization upon following colors toward possible bugs.

Our script participants paid particular attention to the feedback messages, including reading them, navigating backward and forward in them, looking at more or fewer of them, and drilling down to get more information about them. For example:

Male P0117081130 looked at the last error message.

Male P0718080800 changed the display settings so as to show only the first error message.

Male P0717080900 cleared the command line screen so he could easily scroll up and stop at the first error message.

Female P0718081400 resized the output window to see more of the messages at once.

Female P0718081400 opened up Windows Explorer to better understand what path the error message is talking about.

To-Do Listing is indicating explicitly the suspiciousness of code (or lack of suspiciousness) as a way to keep track of which code needs further follow-up. Some spreadsheet users did this by checking cells off or X-ing them out. (These features were designed for another purpose, but some participants repurposed them to keep track of things still to check.) Like the spreadsheet users, our scripters found mechanisms to accomplish to-do listing, such as:

Male P0117081130 put a breakpoint on a line to mark that line as incorrect, and to stop on it whenever he ran the code.

Female P0721081130 closed files that she thought to be error-free, leaving possibly buggy ones open.

Male P0718081030 used pen and paper to keep track of stumbling points.

The same male, P0718081030, also mentioned sometimes using bookmarks to keep track of stumbling points.

Keeping track of things to do and things done is a functionality so dear to computer users' hearts, they have been reported to repurpose all sorts of mechanisms to accomplish it, such as appropriating email inboxes [Danis et al. 2005] and code commenting [Storey et al. 2008] for this purpose. Yet, other than bug trackers (which do not work at the granularity of snippets of code), few programming environments support to-do listing. A clear opportunity for designers of end-user programming environments and scripting environments is providing an easy, lightweight way to support to-do listing.

New Strategies. Finally, there were three strategies that had not been present in the spreadsheet study: control flow, getting help, and proceeding as in prior experiences.

Control Flow means following the flow of control (sequence in which instructions are executed). Pennington found that expert programmers initially represent a program in terms of its control flow [Pennington 1987]. Since spreadsheets do not provide a view of execution flow, it is not surprising that following control flow did not arise in the spreadsheet study.

The scripting environment, however, provided multiple affordances for viewing control flow, and participants used them. For example:

Male P0717080900 used the call stack to see what subroutines were called and in what order.

Female P0718081400 placed a breakpoint on the first line to run the script in control flow from there in order to understand it.

Male P0117081130 stepped over and into to examine and execute the code in the order it was run.

Providing support for following control flow is relatively widespread in programming environments for professional programmers, but less so for end-user programming environments. A notable exception is the approach for allowing control flow following in the rule-based language KidSim/Cocoa/Stagecast [Heger et al. 1998], which features the ability step through the program to see which rules fire in which order.

Help means getting help from other people or resources, a common practice in real-world software development. For example, Ko et al. reported that developers often sought information in hard-to-search sources, such as coworkers' heads, scanned-in diagrams, and hand-written notes [Ko et al. 2007]. In our study, examples of following help included searching for help on a bug using Google Search, consulting the internal help documentation in order to set a breakpoint, or asking the researchers what a particular function does. This strategy was not available in the spreadsheet study but our script participants used it extensively.

Female P0717081630 sought help from the observers.

Male P0718081030 sought online help.

Male P0117081130 used the interface's help menu item.

Female P0718081400 used the command line's "-?" and "/?" commands.

Male P0718081030 used the function key to bring up the internal help. Later, he also brought help up on a particular word by first highlighting it and then hitting the function key.

Finally, one participant attempted to integrate external help with code inspection:

Male P0718081030 restored down the help window, to be able to look at the code and still have the help in an open window next the code.

Proceeding as in Prior Experience was recognizing a situation (correctly or not) as one experienced before, and using that prior experience as a blueprint of next steps to take. Sometimes the recognition was about a feature in the environment that had helped them in the past and sometimes it was about a particular type of bug. Once recognition struck, participants often proceeded in a trial-and-error manner, without first evaluating whether it was the right path. For example:

Male P0717080900: "Ah – I've seen this before. This is what must be wrong."

Female P0718081400: "It obviously needs to go up one directory."

Male P0721080900 said: "Just for kicks and giggles, let's try this."

Male P0718081030 felt something strange was going on and, from an earlier experience, decided that it was PowerShell's fault. He therefore closed the environment and opened it up again.

We suspect that proceeding as in prior experience is quite widespread, but it has not been reported in the literature on debugging. Given humans' reliance on recognition in everyday life, this strategy could be having a powerful influence on how people debug. It is an open question whether and how designers of debugging tools might leverage the fortunate aspects of this and take steps to help guard against the unfortunate aspects.

4.5.2. Sequential Usage of Strategies: Three Participants

To investigate how the participants used these strategies when succeeding, we analyzed three participants in detail. The first two were the most successful male (who fixed four bugs in one task) and the most successful female (who fixed one bug in one task). We then analyzed a male with the same scripting experience as the female (who also fixed one bug in one task). Each of these participants thus provided at least one successful event to analyze, in addition to several failed attempts. Figure 14 shows the sequence of strategies used in one of the two tasks by these participants.

As an aside, the overall low success rate on the number of bugs fixed was expected, because we deliberately designed the tasks to be difficult, so that strategizing would occur even with expert scripters. For example, one of our participants (the most successful male) was extremely experienced, having written more than 100 PowerShell scripts in the past year. He fixed all seven bugs in the two tasks. (Reminder: the figure shows only one of those two tasks.)

The most successful female described herself as a software developer. She was about 30 years old, and had nine years of scripting experience (in JavaScript, PowerShell, Perl, and Bash/UNIX Shell Scripting). Within the past year, she had written about five PowerShell scripts and was a frequent PowerShell user, normally using it about two to three times per week.

As Figure 14 shows, after reading the task description, this female began by running the script: “First thing I’m going to do is to try to run it to see what the errors are.” Using the error message which stated there was an error at a line which contained “Type = ‘NewLine’” because “Type” is a read-only property, she navigated directly to that line of the script. She right away noticed that the equal sign was doing an assignment instead of a comparison, thereby finding the first bug (the dashed bar at the beginning of her session in Figure 14).

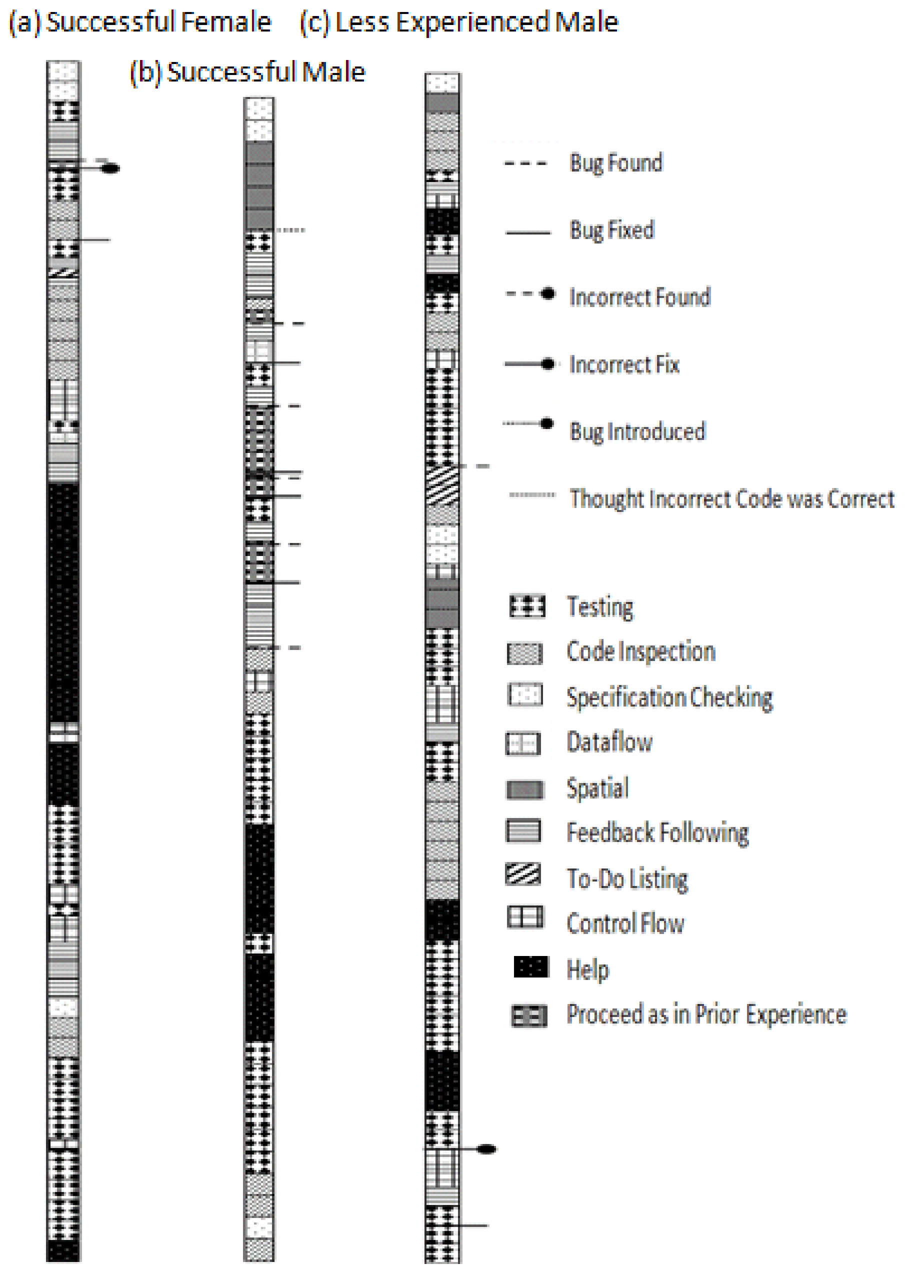


Figure 14. 30-second use of stratagems (each square) by three participants during one of the two tasks, as well as when bugs were found and fixed (lines). The start of the session is at the top.

But, although she knew what the error was, she fixed it incorrectly based on her prior experience with other languages (the solid line with a dot followed by 45 seconds of testing). Fortunately, testing her change made her realize that her fix was incorrect: “Ok, perhaps it was wrong...” Despite her experience with scripting and using PowerShell, she said she felt silly about not remembering what the correct syntax was, but that it is due to her not writing scripts from scratch in PowerShell often, but rather reusing and extending existing scripts.

Knowing what she wanted the program to do but not the syntax to accomplish it, she started to use code inspection to find a suitable fix by looking for examples in related code: “That’s why I usually start looking at other files, to see if there’s an ‘equal’ type thing.” She went on to skim two other PowerShell files, rejecting two Boolean operators she did not believe would fix her problem. However, the second one, even though it was not exactly what she needed, was close enough to enable her to fix the bug by patterning her change after that code: “Aha! ‘–like’ isn’t it because that would be like a ‘starts with’ type thing. So, maybe I need to do ‘-eq’?” This use of code inspection is what enabled her to actually fix the bug, and is a good example of *how* increased use of this strategy might have led to greater female success in [Subrahmaniyan et al. 2008)].

The female’s use of code inspection to actually *fix* the bug above, rather than just to *find* it, is interesting. It suggests a possible new debugging functionality, code mini-pattern recognition and retrieval, to support searching and browsing for related code patterns to use as templates. The female’s beneficial use of code inspection in this study is consistent with the results from [Subrahmaniyan et al. 2008] that code inspection was statistically tied to female spreadsheet users’ success. These combined results suggest the following hypotheses to more fully investigate the importance of code inspection to female debuggers:

Hypothesis 1F: Code inspection is tied to females’ success in finding bugs.

Hypothesis 2F: After a bug has been found, code inspection is tied to females’ success in correctly fixing the bug.

Hypothesis 3F: Environments that offer explicit support for code inspection strategies in fixing bugs will promote greater debugging success by females than environments that do not explicitly support code inspection strategies.

In contrast to the female, for males, code inspection did not appear to be tied to success, either in the earlier spreadsheet study or in this one. As the figure shows, the successful male used very little of it, and used none in the periods after finding, when working on actually fixing the bugs. Although the low-experience male did use code inspection, it did not seem to help him very much. Thus, we predict that a set of hypotheses (which we will refer to as 1M, 2M, and 3M) about males like the female-oriented Hypotheses 1F, 2F, and 3F will produce different results in follow-up research, because instead of emphasizing code inspection, the periods near the low-experience male's successful finding of a bug and near his successful fixing of the bug contained a marked emphasis on testing. (We will return to this point shortly.)

The successful male, whose sequence of strategies is also shown in Figure 14, was a very experienced scripter. He described himself as a scripter (not as an IT professional or developer) and had 20 years of experience writing scripts in languages such as Korn Shell, BIN, PowerShell, Perl, and Tcl. He had used PowerShell since its inception and had written over 100 PowerShell scripts within the past year alone.

After reading the task instructions, the successful male did not begin as the female did by running the script, but instead first began by reading the main script code from top to bottom for a couple of minutes, "The first thing I'll do is to read the script to find out what I believe it does." Once he got to the bottom of the script, he stated that "this code didn't seem to have anything wrong with it," denoted by the dotted line in Figure 14. He was incorrect about this.

After the dotted line, this successful male switched to running the script to see its outputs (testing) and to consider the resulting error messages (feedback following). The first error message this male pursued was the second error message that the successful female had also tried addressing: “cannot find path because it does not exist.” Without even navigating to the function to which that the error referred, the successful male was able to draw from his prior experience, immediately hypothesizing (correctly) that the error was caused by a function call in the main script that used the “name” property as a parameter, rather than the “full name” property of a file. He stated, “I know that the file type has a ‘full name’ property, so that’s what we need to do.” After changing the code, to check his change before really declaring it a fix, he opened the function that the error message referred to, checking to see how the file name that was being passed as a parameter was being used (dataflow). At this point, he declared the first bug fixed, and reran the script to see what problem to tackle next. He used a similar sequence of testing, feedback following, and prior experience for the next three bugs he found and fixed.

But when the successful male found the fifth bug (see the fifth dashed line in Figure 14), he did not have prior experiences relevant to fixing it. As the right half of the figure shows, he spent the rest of the session trying to fix it, mainly relying on a combination of fine-grained testing (checking variable output values) and help (documentation internal to the product on debugging PowerShell scripts), with bits of code inspection, control flow, and specification checking also sprinkled throughout.

Thus, the successful male provided interesting evidence regarding code inspection, testing, prior experience, and dataflow. We have already derived hypotheses about code inspection, and we defer hypotheses about testing until after discussing the second male. Regarding prior experience, both the successful male and the successful female drew on prior experience in conjunction with feedback following, but the female’s prior experience had negative impacts when she tried to fix a bug by remembering the syntax from a different language. The interplay between feedback following and proceeding as in prior experience is thought-provoking, but there is not as yet enough evidence about this interaction and gender differences for us to propose hypotheses for follow-up.

Dataflow, however, was also a successful strategy for the males in [Subrahmaniyan et al. 2008], and this successful male’s experience with it suggests exactly *where* it might be contributing to males’ success:

Hypothesis 4M: Dataflow is tied to males' success in finding bugs.

Hypothesis 5M: After the bug has been fixed, dataflow is tied to males' success at checking their fix.

Hypothesis 6M: Environments that offer explicit support for dataflow will promote greater debugging success by males than environments that do not explicitly support dataflow.

We do not expect the corresponding female Hypotheses 4F, 5F, and 6F to show significant effects, since we have seen no evidence of it in either study.

The successful female was much less experienced than the successful male, so we also compared her strategies to those of a less experienced male to obtain insights into strategy differences due solely to experience. This male had nearly identical experience to the female: 10 years of scripting experience (in CMD, VBScript, PowerShell, T-SQL, and SSIS). In the past year, he had written about six PowerShell scripts, and used PowerShell about three times per week.

Like the successful male, this less experienced male also started out with inspecting the code from top to bottom. The less experienced male examined most of the script very carefully, highlighting the lines he read as he went along. He used several strategies (including testing, feedback following, control flow, and help) to better understand a construct he had never run into before. After about four minutes of trying, he noted not completely understanding that part of the code and assumed that it was correct (which was true), stating that the part he had been studying seemed like a "red herring" and "a no-op". He therefore went on to examine the next line.

Directly following about three minutes of incremental testing (running only fragments of the code at a time to see what they output), the lower-experience male found a bug (dashed line in Figure 14). At that point, he stated "I'm making a note of a bug that's here; that we're not making a path here… And we're going to fail, because the system is simply not going to find those files." After having made the note, he went on trying to use several strategies (mainly testing and code inspection) to understand the rest of the code.

In the earlier study, we saw some evidence pointing in the direction of to-do listing being a strategy used more by females [Subrahmaniyan et al. 2008], and two of the three females used it in this study too. This male employed a pen-and-paper version of to-do listing, but to-do listing was so scarcely used overall in this study (perhaps since it was not supported by the environment) that we could not derive hypotheses based on these data alone.

By inspecting the code in control flow order, the less experienced male realized that an incorrect property used for one of the variables was the cause of the faulty output. Returning to the first bug he had written down on paper, he succeeded at fixing the bug through the use of testing. Specifically, he copied that variable and its property into the command line and ran the command. He stated that the output was incorrect, since it was the name of the file instead of its full path. Using tab-completion in the command line, he deleted the property, and tabbed through the list of all properties. He then also used a command to output a list of all properties and skimmed through them, wondering, "Is there a FullPath property?" There, he found a "FullName" property. He tried it out by typing the variable name and property in the command line again. The output was exactly what he wanted, so he put that small code fragment into the script's code, thereby fixing the bug. This suggests a possibility that a programming environment that supports *systematic debugging-oriented testing* mechanisms, such as tracking incremental testing and testing of small fragments of code, may be helpful to testing-oriented debuggers.

The testing evidence from both males above, combined with that of the previous study, suggests the following hypotheses for follow-up investigation.

Hypothesis 7M: Testing is tied to males' success in finding bugs.

Hypothesis 8M: After a bug has been found, testing is tied to males' success at correctly fixing the bug.

Hypothesis 9M: After the bug has been fixed, testing is tied to males' success at evaluating their fix.

Hypothesis 10M: Environments offering explicit support for incremental testing and testing of small code fragments will promote greater debugging success by males than environments that do not explicitly support incremental testing strategies.

We are also proposing identical hypotheses for testing with females (7F, 8F, 9F, and 10F). Our prior study provided no ties between testing and success by females, so we do not predict significant effects for 8F-10F. However, the successful female in this study used testing in conjunction with feedback following to successfully find a bug; 7F might therefore also hold true for females.

As we have been bringing out in our hypotheses, the above evidence from all three participants suggests that the debugging stage at which a strategy is used (finding a bug, fixing a bug, or evaluating a fix) might have an influence on females' and males' success with the strategy, and we consider this to be an interesting new open research question. For example, although everyone successfully found at least one bug by incorporating testing, only the lower experience male *fixed* a bug using that strategy. One concrete instance of this open question is, therefore, whether there is a difference in *how* males and females use testing. For example, might males incorporate testing into both finding and fixing, whereas females use it for only in the finding stage? We express this open question as a general hypothesis:

Hypothesis 11MF: Males' and females' success with a strategy differs with different debugging stages (finding a bug, fixing a bug, or evaluating a fix).

4.6. Conclusion

This paper presents the results from a think-aloud study we conducted to see how well end-user programmers' spreadsheet debugging strategies generalize to a different population and a different paradigm: IT professionals debugging Windows PowerShell scripts. Our results show that:

- All but one of the strategies found with the spreadsheet users also applied to IT professionals debugging scripts, along with three more that emerged. The seven strategies we observed in both studies were: *testing*, *code inspection*, *specification checking*, *dataflow*, *spatial*, *feedback following* (a generalization of the strategy previously termed *color following)*, and *to-do listing.* In addition, we observed the following three strategies that had not been present in the spreadsheet study: *control flow*, *help*, and *proceeding as in prior experience*.
- The mechanisms scripters used revealed several opportunities for new features in scripting environments, such as support for *systematic* incremental testing, for easy inspection of large amounts of code and of code mini-patterns, for "drill down" into related testing information during code inspection and into related code information during testing, for informal specification checking, and for to-do listing.
- The evidence of the earlier statistical study on spreadsheets combined with the qualitative analysis of this study's participants produced several detailed hypotheses on gender differences in successful strategy usage.

Perhaps the most important contribution of this study is that it raised a significant new open question: whether males' and females' uses of debugging strategies differ not only in *which* strategies they use successfully, but also in *when* and *how* they use those strategies.

WHAT DESIGNERS WANT: NEEDS OF INTERACTIVE APPLICATION DESIGNERS

Valentina Grigoreanu, Roland Fernandez, Kori Inkpen, and George Robertson

VLHCC'09 (Corvallis, OR, USA: September 20-24, 2009)

IEEE Computer Society, Washington, DC.

Issue manuscript appears in the *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 139-146.

5.

Generalizing Across: The DESIGNERS GROUNDED THEORY STUDY

5.1. Abstract

Designers' extensive software needs have not been adequately documented in the research literature, and are poorly supported by software. Without appropriate tools to support their needs, designers have difficulty knowing the best way to evolve the look and feel of interactive applications they are designing.

In order to inform the design of new tools for interactive application design, we used a grounded theory approach to find out what designers' needs are when designing such applications. This paper reports our findings (20 designer needs) from content analysis of five types of artifacts: surveys, Blend discussion list emails, Dreamweaver forum entries, Flash forum entries, and interviews with ten designers. These 20 needs were then validated in follow-up interviews and focus group sessions. The results of this work revealed trends regarding the importance of each need and show that flow is one of the most important needs.

5.2. Introduction

With the advent of tools such as Adobe Dreamweaver and Microsoft Expression Blend, designers are acquiring increased control in creating both the look and feel of interactive desktop and web applications. While much work exists on designers' processes and tools, we know little about the extent of the problems designers encounter with this software, and how design tools can support those needs. In this paper, we describe two grounded theory studies conducted to explore interactive application designers' needs and to validate our findings.

Jeanette Wing states "Computational methods and models give us the courage to solve problems and design systems that no one of us would be able of tackling alone" [Wing 2006]. The power of computers allows designers to offload some of the computational complexity of the design process onto the computer. We believe that there is much room for improvement in software support of designer's tasks. These tasks might involve acquiring and reusing earlier work for a new purpose, or optimizing users' interactions.

Consistent with the research method of grounded theory [Glaser and Strauss 1967], we conducted content analysis of data from several sources to identify these needs: (1) a small open-ended qualitative survey, (2) emails to a Blend discussion list, (3) posts to a Dreamweaver forum, (4) posts to a Flash forum, and (5) 1.5-hour interviews with ten designers. This study helped us better understand designers' needs.

Our resulting theory is a set of 20 needs experienced by designers while creating, iterating, and communicating their designs. We validated our findings through an interview with two experienced design managers and two focus group sessions with nine designers. The results demonstrate that designers have many unsupported needs. Each need is a starting point for the creation of better design tools. We also report which needs are likely to appear together, as well as the relative importance of each need.

5.3. Related Work

A *designer* is a professional who creates plans to be used in making something [Designer 2009] (in our case, interactive software applications). Research on the design process suggests that many designers start out with sketches and continue with intermediate representations (e.g., maps, storyboards, schematics, prototypes, mock-ups, and specifications) [Newman and Landay 2000]. We also know that designers find behavior more complicated to design than visual aspects of applications, and that communication plays a vital role in the design process [Myers et al. 2008]. However, we do not have an overarching view of the kinds of needs these designers have, which software should address.

Since the two most popular interactive application design tools are Dreamweaver and Flash [Myers et al. 2008], it is not surprising that much of our knowledge of interaction design stems from the web world. *Interactive websites* include online forms, surveys, and databases, and their design often requires end-user programming when they are not created by professional programmers (e.g., [Myers et al. 2008; Rode et al. 2005]). Early work in this area found that end-user web developers understood the roles of visible components (such as fields and buttons), and used technical terms (such as “file”, “database”, “page link”, and “member”) [Rode et al. 2005]. However, they were vague in the implementation of these terms, did not understand how hidden operations worked, and did not mention constructs such as “variables” or “loops.”

Designers use a variety of tools for different phases of the design, including (from most to least popular): Adobe Photoshop, Adobe Dreamweaver, Microsoft PowerPoint, Adobe Illustrator, Adobe Flash, Microsoft Visio, Adobe InDesign, Omni Group’s Omnigraffle, Microsoft Visual Studio, Adobe Fireworks, Adobe Director, Microsoft FrontPage, Adobe After Effects, Axure RP, Adobe Flex, Adobe GoLive, and Microsoft Expression Blend [Myers et al. 2008]. Research platforms for web designers include: CLICK [Rode et al. 2005], WebFormulate [Ambler and Leopold 1998], FAR [Burnett et al. 2001)], DENIM [Newman et al. 2003], and WebSheets [Wolber et al. 2002]. Even with this multitude of tools, a designer trying to create an interactive web or desktop application still faces many challenges. We believe a possible reason for this is that little is known about the overall extent of designer needs and their relative importance, to be explicitly addressed in design tools.

The design task has been defined as a “knowledge-based problem-solving activity” by some researchers, specified by a set of functions (constraints stated by the intended users of the design and the domain itself) and a technology (the components available for design creation and the relationships between them) [Chandrasekaran 1990]. Chandrasekaran [Chandrasekaran 1990] reviews existing literature on designers to get a better understanding of the steps involved in the problem-solving task of design, and proposes the following task structure: design, verify, critique, and modify. Smith and Browne [Smith and Browne 1993] break up the design problem into five elements: goals, constraints, alternatives, representations, and solutions.

Unlike these two works, we take a grounded theory approach to determining designers' needs. Relying on literature alone in determining designers' needs might lead to overlooking facets particular to designers' tasks. This paper therefore generates theory from the ground up, augmenting knowledge from related literature by collecting and analyzing designers' anecdotal evidence. In doing so, we focus on informing the design of tools for designers.

5.4. Grounded Theory Study Setup

5.4.1. Methods

We used the objectivist grounded theory methodology [Glaser and Strauss 1967] to identify designers' needs, since it allowed us to reduce the bias of our previous experience. Our methodology was also highly influenced by Charmaz' [Charmaz 2000] constructivist grounded theory approach. Unlike objectivists, constructivist grounded theorists pay special attention to the context of the research (the participants' and researchers' assumptions, implicit meanings, and tacit rules).

We triangulated our data and analyses on several fronts. First, since particular data sources could be biased toward revealing particular needs, we collected data (282 artifacts) from five different sources which are described in further detail in the next section. Second, since one researcher might notice different needs from another, we made sure that two interviewers were present at three of the twelve interviews. Finally, since bias may affect the results derived from the collected data, two researchers coded 40% of the data together (reaching an inter-coder reliability of 81%), before individually coding the rest.

While collecting data, we also iteratively analyzed them. Unlike traditional methods for generating theory, grounded theory calls for emerging theory to be integrated with further data collection and data analysis [Glaser 1978]. It also calls for the researcher to remove preconceived ideas in the early data collection phases. We therefore started with very open-ended survey and interview questions, generally asking about designers' troubleshooting, problem solving, and how they find and fix errors in designs. Our questions became more focused with every iteration thereafter, culminating in one last interview with two experienced designers which was structured around our final 20 needs, to verify and learn more about our emerging results.

5.4.2. Procedures and Participants

In grounded theory, data collection and data analysis are a part of the same iterative process. We describe all of the data collection methods here, and present our results in the next section.

Survey. We began with a small open-ended survey to determine designers' needs. The survey asked four open-ended questions: what tools they used, the type of design they did (visual, interaction, etc.), problem-solving or troubleshooting needs they had, and their prior experience with creating functional prototypes.

Six designers (three females) on a user experience team at Microsoft completed the survey. All six of the participants designed both the look and feel of the applications they worked on. Also, all but one had end-user programming experience: developing at least a website using languages such as JavaScript and Flash. The software used by this small group of designers varied widely, but was consistent with findings from larger surveys of designer populations [Myers et al. 2008].

Blend Distribution List. We next harvested questions from a mailing list where designers discussed design issues. Starting with 7524 emails, we narrowed the scope down to 236 by counting only initial inquiries, not the replies, and by only including emails that contained the word "designer." Finally, from that set, we analyzed the 25 emails sent within the past year, related to problems designers were having.

Dreamweaver and Flash Forums. We also harvested questions asked by designers on the Adobe public web forums for Dreamweaver and Flash, the two most frequently used interactive design application tools [Myers et al. 2008]. All forum entries were original design inquiries. We took the 25 most recent such entries from each forum, to match the number of artifacts we had collected from the Blend distribution list.

Formative Interviews. Informed by what we had learned thus far, we conducted ten qualitative interviews with designers across several product teams (four females). Interviews lasted 1.5 hours each, and participants received a lunch voucher for participating.

To encourage participants to show examples of designs they had created and problems they had encountered, the interviews were conducted at the designer's desk when possible (for eight of the ten interviews). The two others were conducted in a cafeteria and the interviewees sketched in a notebook to clarify the anecdotes they were giving.

The qualitative interviews provided us with an open-ended, yet in-depth, exploration of a topic of interest to the interviewees. As Charmaz [Charmaz 2000] points out, many grounded theory studies use a one-shot interviewing approach, when sequential interviews provide interviewers with the ability to follow up on earlier leads. To collect a richer understanding of our designers' needs, we invited the survey respondents to also be interviewed (we interviewed all three males and one of the females). We also interviewed six designers who were new to our research to get a fresh perspective on our emerging results.

5.5. Grounded Theory Results

5.5.1. Do Designers Need Tool Support?

Responses from the survey showed that our designers did indeed need interactive application design tools to better support their problem-solving activities. This lack of support sometimes had a negative impact on the designs they created. (For all the quotes in this paper: F=female, M=male, U=unknown, S=survey, O=forums, and I=Interviews.)

FS: Not having tools for the kind of visual debugging help I was looking for changed my design in a negative way; finding tweaks, workarounds, giving up…

While some environments offer some support for one type of design problem solving (end-user debugging and troubleshooting), those tools are often only about the underlying code, and are based on debugging tools used by professional developers:

MS: When I am in Blend/Visual Studio, I use the code debugging tools, though there is no such thing as something that helps me find out why something is showing up two pixels off.

Designers need problem-solving tools to support their design activities at a higher level than the code (i.e., at the same level of abstraction as their design representations), and even the code features needed to be geared toward designers themselves.

Designers encounter many problems which software environments could support. We identified 20 non-orthogonal needs through content analysis of the artifacts (see Table 9), as well as the relationships between them (see Figure 15). Each need is a starting point, rather than an endpoint. The edges between two codes give a sense of the frequency with which the needs appeared together (the more often they appear together, the shorter and thicker the edge between them is). Generally, the problems fell into three categories: problems in *creating* the first version of the design, in *iterating* on it, and in *communicating* it.

5.5.2. Designers' Creation Needs

The codes which strongly related to "creating the first design" (and therefore perhaps also to creativity) were *propose* (and its neighbors: *look*, *feel*, and *training*) and *reuse* (and its neighbors: *extend*, *themes*, and *flow*). Some of these needs also came up during the iteration and communication steps.

Table 9. Designers' needs and their definitions (entries are alphabetized).

Automation	Automating redundant steps of design by applying one action to many parts of the design.
Bugs	Creating bug-free code by getting help in fixing bugs, or by generating correct code.
Cleanup	Having to clean up code or take out unused generated code.
Communication	Communicating the design to developers and others through sketches, speech, prototypes, etc.
Compatibility	Ensuring that the same application works in multiple environments.
External Constraints	Keeping track of design constraints, and helping ensure that the constraints are met.
Extensibility	Extending or customizing the functionality of the design environment.
Feel	Ensuring that the feel of the design is correct.
Flow	Representing how data, events, and other resources flow through the design.
Granularity	Switching between levels of abstraction of the application's design.
Jargon	Understanding jargon.
Look	Ensuring that the look of the design is right.
Optimization	Optimizing either the look or feel of the application.
Propose	Proposing the first version of a design.
Reuse	Reusing someone else's or one's own code and designs.
Settings	Identifying incorrect or inexistent software, hardware, or other settings external to the design.
Testing	Evaluating the design's correctness for different situations.
Themes	Creating a design theme or to applying a theme from somewhere else.
Training	Getting training resources or other help.
Usability	Evaluating usability issues in a design.

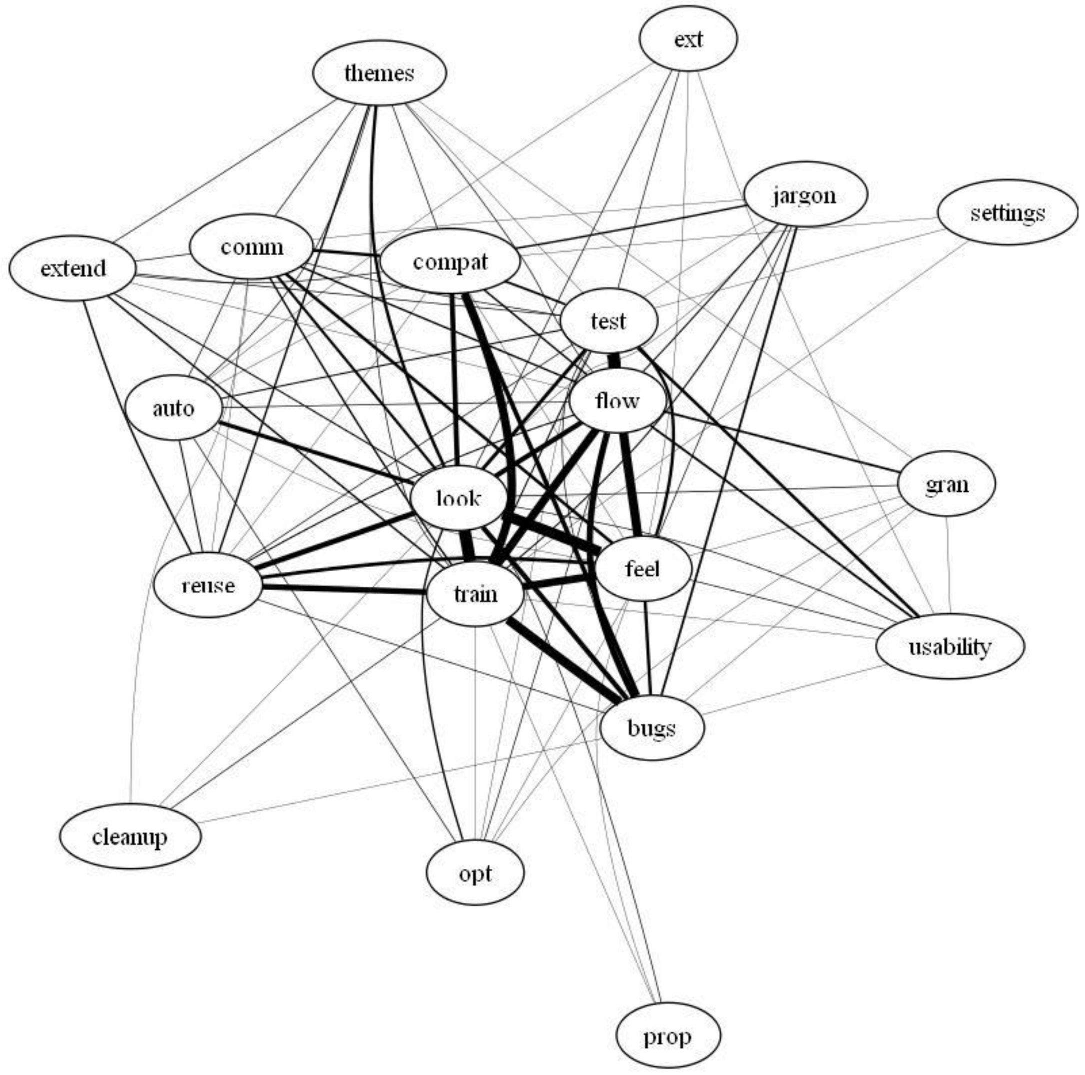


Figure 15. A constellation graph showing the relationships between the codes: the thicker and shorter an edge, the more artifacts those two codes (end-points of the edge) appeared together for.

Propose. Designers need tools to support *proposing a design*, since they sometimes need help coming up with that first design idea; "the right representation."

FI: How do you show that some items in the list are different from the other ones?

Look. This was a very general category encompassing all needs that were about the *look* of the designed application. Designers need to be able to problem solve the look of an application, when the desired visual aspects are not achieved or could be better. The look that a designer might have in mind is not always easy to create an artifact around.

MI: Developers often don't get the colors right, so it takes the detailed eye of a designer to make sure the design's right (generating bits of code helps with this).

Feel. Proposing the initial *feel* of an application is also hard. Like *look*, this was a very general need, referring to a wide range of problems having to do with a design flaw in the behavior. Examples included navigations and other interactions that did not behave as intended, or communicating the feel of an application to other team members.

MI: Designers have to make sure to always show some kind of feedback when the program's doing something.

Flow. Designers need to create, iterate on, and communicate the *flow* of data, events, and other resources through the application they are designing (dataflow, event flow, workflow, timelines). Flow involves designing the structure of an application using a diagrammatic representation. Terms used by designers to refer to this view included: schematics, information architecture, wireframes, Visio documents, tree diagrams, timelines, etc. As shown by the strong link between *feel* and *flow* in Figure 15, one common way of representing feel is in a flow diagram.

FI: I usually use PowerPoint by creating hyperlinks between slides, but there's no way to tell what slides connect to which other slides. I would love a way to visualize storyboard trees like that.

Training. Designers need *training* resources (classes, online training, etc.) and other help. Most of the emails and forum entries analyzed were coded as training since they asked for help with a particular problem. Training topics ranged across a variety of issues including bugs in the code, external settings, and asking about courses on particular software. Sometimes, training was needed to overcome a selection barrier [Ko et al. 2004]:

MI: Sometimes you don't know if it's a lack in your own knowledge, or something the software can't do. […] It costs knowledge and time to figure it out.

Reuse. One way in which designers first create their designs, or inspire themselves, is through reuse. Designers need tools to help support *reuse* their own (or others') code, image files, PowerPoint presentations, look, feel, etc., in creating a design.

MI: Snippet libraries to build your own code by taking code as is and using it as a Lego piece would be nice.

Designers also need to sometimes recreate a look or feel seen elsewhere, such as "recreate the flash effect on this banner" [UO].

Themes. One need related to *reuse,* though not always overlapping with it, was *themes.* When themes and reuse overlapped, designers wanted to apply previously-created themes to their own designs.

FI: Just like PowerPoint has deck templates, Blend should have application templates (Office, web, etc.).

Other times, creating a theme from their designs was in expectation of reuse.

Extensibility. Designers need to *extend* (and customize) software components, behaviors, and capabilities for their own needs and preferences.

FI: I would like better text control for making new fonts. Some software uses this squishing thing, rather than ultra-condensed weight.

Sometimes, extensibility overlapped with themes and reuse. One designer wanted to extend her software's functionality by reusing someone's code for testing different WPF themes and skins [FO].

5.5.3. Designers' Iteration Needs

Once the initial version of one or more designs has been created, designers iterate on them. Since design verification often sparked new iterations, the codes we address in this section are: *usability* (and its neighbors *external constraints* and *granularity*) and *testing* (and its neighbors *automation, optimization, bugs, settings,* and *compatibility*).

Usability. Designers need tool support for evaluating *usability* issues in their design. These evaluations often result in further design alterations (e.g., "creating multiple prototypes and have the users pick the one they like best" [FO]). Another way of evaluating the usability of a design was by seeing how well it followed standard design rules.

MI: A tool would be nice where all of the usability rules you needed to follow were given in a table; to make sure you haven't overlooked any of them.

As this last quote shows, usability sometimes overlaps with fixed external constraints.

External Constraints. Designers need to *keep track of external constraints* that limit the design: specifications for the design, the schedule, the size of the files, the performance of the application, additional software that needs to be downloaded before the application can be run, and the memory footprint of the application that is being designed. With design, some of the constraints are highly subjective, but the software could give recommended outputs.

FI: The design has to meet certain guidelines (spacing around images, contrast ratio, location, etc.). Maybe software could help make sure that these are met.

Granularity. To more easily evaluate the usability of their prototypes, designers need to easily switch *levels of granularity*: moving from high-level schematics, to a detailed view of each screenshot, to the code behind the application, to the preview of the application, and back.

MI: I do a lot of switches like that (high level to lower levels) to find and fix errors.

Testing. *Testing*, in its purest form, means to evaluate whether the design gives the correct output for particular inputs. Testing is an important need since interactive application designs are often highly complex and data-dependent.

MI: Tools should provide a mock service to generate data to hook up to the designed application. It's very hard to design a data-centric application without data.

As Figure 15 shows, the two most highly related needs were *testing* and *flow*: seeing how an output resulted from the flow of elements through the design. This is a concept similar to slicing, which has been found to be useful in end-user programming environments (e.g., [Rothermel et al. 2001]). Such tools should not be limited to code, but also lower-fidelity prototypes.

Automation. Designers need *automation* tools in order to quickly perform one action on many different parts of the design, or to otherwise automate (record and replay) redundant steps of the design process.

MI: If the computer could create a style-guide-in-a-box specification template, you could just define what you'd like your specifications to look like, and the software would spit them out for a designed application.

Optimization. Some designers wanted automation capabilities to quickly *optimize* the look or feel of the design. Optimization did not come up often, but was reported to be a very hard problem to solve. In perfecting visual aspects of the application:

MI: One way of quickly visualizing the best option is to generate sample previews of what the image would look like with the different number of colors.

Another commented about optimizing the user's interaction with the application:

MI: Workflows with fairly detailed screenshots are useful for making decisions about unnecessary pages.

Bugs. Designers need *debugging* tools to help them fix bugs. Bugs addressed by the users were sometimes specific errors in their code, and sometimes more general pain points. Some of the bugs were *usability* bugs, while others were just *incompatible* software.

MI: Things like case-sensitivity and overwrites are tough bugs to avoid and fix.

Settings. Incorrect output was sometimes not caused by a bug, but instead by incorrect settings on the designers' computer. These two instances were hard for designers to discern. Even when they were distinguishable, figuring out the wrong setting or missing software to download was a daunting task.

MO: Images that are posted live online appear as broken images in the design mode. [...] I've seen this crop up a few times on these boards, but still can't seem to figure out what settings I'm missing on my end.

Compatibility. Problems related to settings are exacerbated by the amount of switching designers do between different environments in both creating their designs (e.g., Visual Studio and Blend), and predicting environments used by the consumers (e.g. Internet Explorer and Firefox). *Software compatibility* tools are needed to help when a design is working in one environment, but not in another.

MI: It would be nice to render one web page in three different browsers with one click.

5.5.4. Designers' Communication Needs

Designs are boundary objects, shared by designers, developers, users, testers, and usability researchers, among others. It is therefore not surprising that many have considered *communication* the ultimate goal of design (e.g., [Myers et al. 2008]). *Jargon* and *clean code* are related.

Communication. Designers need *communication* tools to help them report their detailed vision of the design to developers, program managers, and others. Without them, the wrong look or feel can be introduced into the design by those who create the working code. As we mentioned earlier, a flow diagram was often used to design the application's interaction. It was also the representation of choice for communicating that interaction,

FI: Another thing that the diagram view helps with is that sometimes developers won't see the forest for the trees, and will focus on the details. If the interface is *instead explored at a step above the screenshots, then the presentation and the design meeting become more straightforward and focused on the interaction.*

It is also important to provide communication from developers back to designers.

FI: Designers should sometimes be given guidance on what controls to use by the developers.

Jargon. Unfortunately, when interacting with developers and with code, designers need to understand or somehow translate *developer jargon* (written or verbal) they encounter from discussions, error messages, warnings, etc. While sometimes error messages are useful, other times, they are not:

*UO: **Error** Scene=Scene 1, layer=Actions, frame=3:Line 3: Syntax error. function () {*

Cleanup. Some tools automatically generate code; however, in those situations, designers need *code cleanup* tools to remove unused (filler) code.

When generating code, a lot of 'junk code' is produced that developers complain about. If there is an error in the code, the designer then needs to do a code review and strip the unnecessary parts, which is hard!

5.6. Validating Needs: A Focus on Flow

To validate our grounded theory work (the 20 needs presented in the previous section), we conducted an interview with two senior designers and two focus group sessions with nine designers. Our goals during those sessions were to: (1) get designers' opinions on whether the needs we found are indeed real to their work, and (2) gauge the relative importance of each need for the creation of tools to support designers.

The first validation session was a two-hour unstructured interview with two senior designers (one female). Both designers received a lunch voucher for their participation. These designers were managers, so their anecdotes about designer needs not only spanned their own experiences, but also those of less experienced designers on their team. When asked to describe the top five needs that should be supported through software, one of the designers yelled “Flow!” and the other agreed. Having a hard time narrowing the list down to five needs, they mentioned their top eight: flow, training, testing, reuse, communication, usability, external constraints, and optimization.

The second and third validation sessions were each a two-hour lunch-time focus group session. Pizza was provided for the session and each participant received a lunch voucher. Nine designers (two females), from eight product teams, attended the sessions. The types of designers included: interaction designers, graphic designers, web designers, design lead/managers, systems designers, and motions designers.

Focus group participants were first asked to go through the list of 20 needs individually and mark the needs which they had personally encountered in their design work. Table 10 shows the number of designers who experienced each need (one participant forgot to answer this part of the question). Even with this small sample, we can see that every need was experienced by at least one designer and 12 of the needs were experienced by more

Table 10. Number of designers who reported experiencing each need.

Flow	8	Propose	6	External Constraints	4
Look	8	Communication	6		
Feel	7	Reuse	6	Automation	3
Optimize	7	Extensibility	5	Settings	2
Testing	7	Compatibility	5	Jargon	2
Themes	7	Granularity	4	Cleanup	1
Usability	6	Training	4	Bugs	1

than half of the designers.

The participants were then asked to rank the top five most important needs in their own work. As Figure 16 shows, flow and feel were mentioned as being the most important needs. Flow involves creating, iterating on, and communicating the higher level structure of

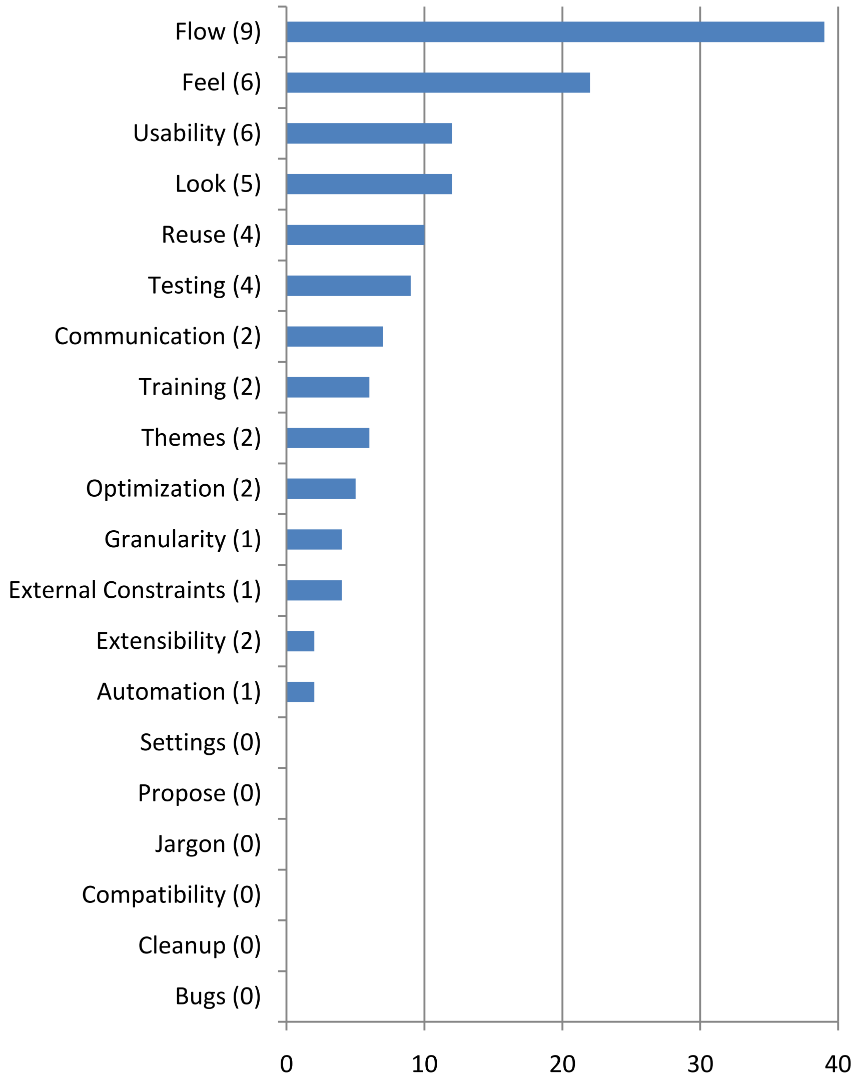


Figure 16. Importance rating (x-axis) for each need (y-axis). Ratings were calculated by summing the ranks assigned by each designer (rank 1: 5 points, 2: 4 points, etc.). The number in parentheses indicates how many designers ranked this need in the top 5.

the application. Feel involves creating, iterating on, and communicating the users' interaction with the application. Additionally, all of the designers ranked flow as being one of the top five needs.

Both interview and focus group sessions validated the needs we derived through our grounded theory work as indeed being needs that designers encounter in their everyday work. Furthermore, even with this small number of participants, clear patterns emerged: (1) Each of the eight top needs the interviewed designers wanted tool support for was a top-five need for at least one of the nine focus group designers, and (2) Flow was identified as being the top need from both the interview and the focus group sessions.

5.7. From Needs to the Design of Tools

The list of 20 needs, and each need's tie to other needs, acted as a useful discussion facilitator which led to the design of several unique software tools. Designers' initial survey, email, forum, and interview tool requests were often incremental improvements to existing software. However, during the focus group sessions, several completely new designs for designer tools arose. Two examples are briefly described here.

(1) A "design-centric versioning system" that allows designers to *1*-keep track of alternate latest design versions, *2*-show the hierarchical ties between designs, and *3*-keep a list of design decisions as a part of the versioning system (e.g., attaching the scenario which led to the decision).

(2) A "multi-view software development environment based on team members' role" to provide all software professionals a common artifact to work on, where each view would be a different facet of the software, directly connected to all the others.

Thus, the results presented here can be used to help in brainstorming about new design tools in addition to incremental improvements to existing software.

5.8. Discussion and Conclusion

This paper presents grounded theory findings from two studies to determine interactive application designers' needs. A content analysis of 282 designers' artifacts identified 20 design creation, iteration, and communication needs (in order of importance): flow, feel, look, usability, reuse, testing, communication, themes, training, optimization, external factors, granularity, automation, extensibility, bugs, cleanup, compatibility, jargon, propose, and settings.

Follow-up interview and focus group sessions validated these needs and further explored the importance of each. All of the needs were validated by at least one of the focus group participants, with flow and feel being the most commonly reported. Flow was ranked as being the most important need.

Our data collection and analysis processes were strongly triangulated, to reduce bias. The fruits of triangulating were evident. For example, collecting artifacts only from online sources (emails and forums) would have biased our findings toward "training" and "bugs," since the majority of those artifacts contained a question about why their code did not work. However, collecting data from several other sources (such as the survey and interviews), helped identify and avoid this bias. A possible limitation of our study was that the validations were based on Microsoft employees' feedback alone, and therefore may not generalize to other designer populations. However, our population was diverse within the company, spanning several types of designers on thirteen different product teams.

The most surprising results of this work are the number of needs expressed by designers, their ties to each other, and their low software support. Based on these results, it is clear that interactive application design environments should provide explicit support for the 20 needs presented in this paper, and especially for *flow*. Recent HCI work on trajectories, such as the conceptual framework for trajectories presented in [Benford et al. 2009]**Error! Reference source not found.** is likely to be useful in the design of tools for supporting flow in designers' software.

Each of the needs reported here is a starting point for designing one or more features of a future design tool. Carefully designing a set of features to meet these needs will require a deep understating of how the needs overlap and interact with each other. This has strong implications for future work. Figure 15 can be used in framing the structure of future studies about the needs and the design of software to support them. Two novel designer tool ideas were already presented here as a result of employing this approach.

FEMALES' AND MALES' END-USER DEBUGGING STRATEGIES: A SENSEMAKING PERSPECTIVE

Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, and Kyle Rector

Journal Paper under Review

Submitted July 16, 2009

22 pages

6.

The SENSEMAKING STUDY:
A Context for End-User Debugging Strategies

6.1. Abstract

Although there have been decades of research into how professional programmers debug, only recently has work begun to emerge about how end-user programmers attempt to debug their programs. Without understanding how end-user programmers approach debugging, we cannot build tools to adequately support their needs. To help fill this need, this paper reports the results of a qualitative empirical study that investigates in detail female and male end-user programmers' sensemaking about a spreadsheet's correctness. Using our study's data, we derived a sensemaking model for end-user debugging and then categorized participants' activities and verbalizations according to this model. We then used the categorized data to investigate how our participants went about debugging. Among the results are identification of the prevalence of information foraging during end-user debugging, two successful strategies for traversing the sensemaking model, ties to gender differences in the literature, sensemaking sequences leading to debugging progress, and sensemaking sequences tied with troublesome points in the debugging process. The results also reveal new implications for the design of spreadsheet tools to support female and male end-user programmers' sensemaking as they debug.

6.2. Introduction

Although twenty years ago, the idea of end users creating their own programs was still a revolutionary concept, today, end-user programming has become a widespread phenomenon. In fact, in the U.S., there are now more end-user programmers than professional programmers [Scaffidi et al. 2005]. Today's end-user programmers include anyone who creates artifacts that instruct computers how to perform an upcoming computation. Examples include an accountant creating a budget spreadsheet, a garage mechanic entering rules to sort email, or a teacher authoring educational simulations of science phenomena. The pervasiveness of end-user programming today is in part due to research advances such as graphical techniques for programming, programming by demonstration, and innovative ways of representing programs (described, for example, in [Kelleher and Pausch 2005; Myers et al. 2006; Nardi 1993]), and in part due to the popularity of spreadsheets [Scaffidi et al. 2005].

Along with the ability to create programs comes the need to debug them, and work on end-user debugging is only beginning to become established. The numerous reports of expensive errors in end users' programs, especially spreadsheets (e.g., [Boehm and Basili 2001; Butler 2000; EuSpRIG 2006; Panko 1998; Panko and Orday 2005]), make clear that supporting end users' debugging efforts is important. There has been recent work on tools for end-user debugging to fill this need (e.g., [Abraham and Erwig 2007; Ayalew and Mittermeir 2003; Burnett et al. 2003; Burnett et al. 2004; Ko and Myers 2004; Wagner and Lieberman 2004]), but a key issue that remains largely unanswered is *how* end-user programmers go about debugging. We believe that knowing more about end users' debugging strategies is important to informing the design of better tools to support their debugging.

This paper helps to fill this gap in knowledge by considering end-user debugging from a sensemaking perspective. *Sensemaking* is a term used to describe how people make sense of the information around them, and how they represent and encode that knowledge, so as to answer task-specific questions [Russell et al. 1993]. As we discuss in more detail in Section 6.3, sensemaking models provide a detailed view of strategies people use when trying to make sense of information they need, in situations in which much of the information available may be irrelevant to the problem at hand. Since such situations are precisely the sort encountered by debuggers—perhaps especially so by end-user debuggers with little training in debugging techniques—we posit that sensemaking is a suitable lens from which to gain new insights into debugging.

To understand how end-user programmers solve debugging problems, it is important to take into account individual differences in problem-solving styles. To gain insights into individual differences, it is often useful to consider identifiable subpopulations, and that is our approach here. One such division that has reported important differences in end-user debugging is gender [Beckwith et al. 2005; Beckwith et al. 2006; Grigoreanu et al. 2006; Grigoreanu et al. 2009; Subrahmaniyan et al. 2008], and as a result, debugging tool improvements have begun to emerge that have been helpful to both males and females [Grigoreanu et al. 2008].

Therefore, in this paper, we investigate the following research question:

How do male and female end-user programmers make sense of spreadsheets' correctness when debugging?

To answer this question, we collected detailed activity logs and think-aloud data from end users debugging a spreadsheet, and used the results to derive from earlier sensemaking research into intelligence analysts [Pirolli and Card 2005] a new model of sensemaking for end-user debuggers. Our model is particularly suited for use with empirical work, because it is expressed solely in terms of the *data* being processed by the user rather than on internal mental activities that do the processing. We use it in this paper to characterize our end-user debugging data in terms of sensemaking sequences, sensemaking subloops, and relationships between sensemaking and debugging. Among the results were the identifications of: the prevalence of information foraging during end-user debugging, two successful ways of traversing the sensemaking model and their ties to literature on gender differences, sensemaking sequences leading to debugging progress, and sensemaking sequences tied with troublesome points in the debugging process. Finally, we discuss how these results can be taken into account to build future tools for end-user debuggers.

6.3. Background and Related Work

6.3.1. Debugging by End-User Programmers

There have been decades of research on professional programmers' debugging strategies (see [Romero et al. 2007] for a summary), but the works most related to this paper are those on novice programmers' debugging strategies, end-user programmers' debugging feature usage, and end-user programming. Note that novice programmers are not necessarily the same as end-user programmers. Novice programmers program in order to learn the profession so that they can become professional developers. End-user programmers usually do not aspire to become professional developers; instead, their programming interest is more likely to be in the result of the program rather than the program itself [Nardi 1993]. Such differences in goals can play out in differences in motivation, in degree of intent to achieve software quality, and in the importance placed on understanding the fine points of the program. However, like novices, end-user programmers usually do not have professional programming experience, and therefore research into novice programmers' debugging is pertinent to end-user debugging.

Given that novice programmers program in order to learn, a number of researchers have looked into how novice programmers gain skill. One recent effort in this direction was research into the learning barriers faced by this population [Ko et al. 2004], which reported barriers of six types: design, selection, coordination, use, understanding, and information. Although the discussion of these relate mainly to the context of creating a new program from scratch, the barriers also tie to debugging, since difficulties with understanding a program's behavior can lead the programmer to a debugging mode. In fact, research on novice programmers shows that program comprehension is key to successful debugging (e.g., [Jeffries 1982; Nanja and Cook 1987]). For example, fixing a program with multiple modules can become intractable if the programmer does not understand the dependencies of the modules [Littman et al. 1986]. Also, *how* a programmer goes about comprehending a program matters. For example, reading a program in the order in which it will be executed has been empirically shown to be superior to reading the program from beginning to end like text [Jeffries 1982]. The essence of previous research into novice programmers points to their need for a sound understanding of the high-level structure of the program and the interactions among parts of the program in order to debug or maintain software effectively. The literature on novice programming was summarized in [Kelleher and Pausch 2005; Pane and Myers 1996].

To be precise about understanding strategies, we first introduce the nuances between two related terms employed in this paper: stratagems and strategies. A *stratagem* is a complex set of thoughts and/or actions, while a *strategy* is a plan which may contain stratagems for the entire task [Bates 1990]. For the remainder of this section, we focus on empirical findings and theories about end-user programmers' debugging behaviors and stratagems.

Empirical research of end users' debugging has begun to appear in recent years [Abraham and Erwig 2007; Beckwith et al. 2005; Beckwith et al. 2006; Beckwith et al. 2007; Fern et al. 2009; Grigoreanu et al. 2006; Grigoreanu 2007; Grigoreanu et al. 2008; Grigoreanu et al. 2009; Kissinger et al. 2006; Ko and Myers 2004; Phalgune et al. 2005; Prabhakararao et al. 2003; Rode and Rosson 2003; Subrahmaniyan et al. 2008]. One useful finding relating to how end users make sense of their programs' flaws is Rode and Rosson's empirical work showing how users "debugged their programs into existence" [Rode and Rosson 2003]. That is, they began with an initial solution, then iteratively found flaws with it, and expanded their solution to correct those flaws, which required adding new functionalities at the same time, debugging that new functionality to uncover more flaws, and so on. One difficulty in such debugging efforts has been end users' difficulties judging whether the program is working correctly [Phalgune et al. 2005]. To inform supporting end-users' debugging, studies investigating end users' information needs during debugging revealed that much of what end-user programmers wanted to know during debugging was "why" and "why not" oriented [Ko and Myers 2004], and that they also wanted to know more about strategies for debugging, not just features for doing so [Kissinger et al. 2006].

A reason this paper explicitly considers the gender of participants in our analysis is the recent body of research suggesting that females and males may go about debugging (and other software development tasks) differently [Beckwith et al. 2005; Beckwith et al. 2006; Beckwith et al. 2007; Grigoreanu et al. 2008; Ioannidou et al. 2008; Kelleher et al. 2007; Rosson et al. 2007]. For example, in spreadsheet debugging, females' self-efficacy (confidence about spreadsheet debugging) [Bandura 1986] has been lower than males', as has their willingness to use new debugging features, which in some cases led to lower performance outcomes [Beckwith et al. 2005; Beckwith et al. 2006]. In contrast to the females in these studies, the males' self efficacy was not correlated with their willingness to use the same new features. These studies are consistent with other studies of females' and males' technical and quantitative tasks revealing females to have lower self-efficacy than males in such tasks [Torkzadeh and Koufteros 1994; Busch 1995; Gallagher et al. 2000; Hartzel 2003]. Research also reports that females tend to be more risk-averse than males [Byrnes et al. 1999; Finucane et al. 2000; Powell and Ansic 1997]. The attributes of risk averseness and low self-efficacy are related, and may snowball. For example, risk-averse females who try out new debugging features and are not immediately successful may experience further reduced self-efficacy as a result, thereby enhancing the perception of risk in adopting the features.

Pertinent to end users' sensemaking about program bugs, Meyers-Levy's Selectivity Hypothesis describes two strategies in how people process information [Meyers-Levy 1989]. According to the Selectivity Hypothesis, males prefer to use a heuristic (or selective) approach that involves striving for efficiency by following contextually salient cues, whereas females process information comprehensively, seeking a more complete understanding of the problem. Empirical work supports this theory [Meyers-Levy 1989; O'Donnell and Johnson 2001], some of which has taken place in the context of an end-user programming tasks. In a study of spreadsheet auditing, female auditors were statistically more efficient (completed the task in less time and used fewer information items) than males in a complex analytical procedures task through use of comprehensive information processing, whereas males were statistically more efficient (used fewer information items) than females in a simple task through use of selective information processing [O'Donnell and Johnson 2001]. Research into female and male effective end-user debugging stratagems also seem related to their preferred information processing styles: females' (but not males') success has been tied to the use of code inspection, whereas males' (but not females') was tied to dataflow [Grigoreanu et al. 2009; Subrahmaniyan et al. 2008]. In both of these studies, females were observed to use the elaborative information processing with code inspection to examine formulas broadly and in detail to get the big picture of the spreadsheet. Males' use of dataflow, on the other hand, was to follow a particular formula's dependencies, in essence being selective about the information being pursued by going for depth early, but at the expense of a comprehensive understanding of the spreadsheet.

6.3.2. Sensemaking

Dervin began developing human-centric models in 1972, creating the Sensemaking methodology for the design of human communication systems. This methodology was grounded in her model of how a person makes sense of his or her situation, referred to as the *sensemaking triangle.* The most complete presentation of Dervin's early work on the sensemaking triangle model is [Dervin 1984], and a more modern overview of her sensemaking work is [Dervin et al. 2003]. According to Dervin's triangle model, an individual trying to make sense of a complex situation steps through a space-time context. Beginning with the current situation (what he or she knows now), the individual then moves through the space to recognizing gaps in understanding (questions, confusions, muddles) that must be "bridged" via resources (ideas, cognitions, beliefs, intuitions), leading to analysis and outcomes (effects, consequences, hindrances, impacts). Although Dervin's work mostly appeared in Communications literature, her goals align with those of Human-Computer Interaction (HCI) research that aims to design and implement human-centric communication systems [Dervin et al. 2003].

A series of sensemaking models later began to appear in Computer Science literature, particularly HCI. These new models focused primarily on Dervin's "bridge" aspect of sensemaking. An early effort in this regard, before the term "sensemaking" had been adopted by the HCI community, was Shrager and Klahr's experiment on instructionless learning [Shrager and Klahr 1986]. Non-technical participants were given a programmable toy and told to figure it out, without any manual or help. The study revealed that participants went through an orientation phase followed by a hypothesis phase: after direct attempts to control the device failed, participants systematically formulated hypotheses about the device by observation and tested them by interacting with the device. If hypotheses were not confirmed, the participants refined and tested them iteratively. The authors used these results to form a theory of how users make sense of a program based on its outputs, with a focus on participants' partial schemas developed through observation and hypothesis testing.

One of the earliest appearances of the term "sensemaking" in HCI was Russell et al.'s work on a model of the cost structure of sensemaking [Russell et al. 1993] known as the Learning Loop Complex. In this work, the authors observed how Xerox technicians made sense of laser printers: they entered a learning loop to seek suitable representations of the problem, then instantiated those representations, then shifted the representations when faced with residue (e.g., ill-fitting data, missing data, unused representations), and finally consumed the encodons (i.e., instantiated schemas) and representations they created [Russell et al. 1993].

Since that time, other versions of sensemaking models have emerged for different domains, of which Stefik et al. provide a good overview [Stefik et al. 2002]. Most of these models depict roughly the same sort of process, each providing a progression by which existing information or knowledge is turned into new knowledge useful to a target task. Even so, there remains controversy over what exactly sensemaking is and how sensemaking research should be performed. These differences in perspective are summarized in [Leedom 2001; Klein et al. 2006]. As these summaries point out, the psychological perspective focuses on creating a mental model, and takes into account elements that are not sensemaking per se, but may contribute to sensemaking, including creativity, curiosity comprehension, and situation awareness. The naturalistic decision-making perspective on sensemaking is empirically based and keeps expert analysts in the center of the sensemaking process, but uses decision aids as needed to improve the process. The human-centered computing perspective critiques intelligent systems intended to automatically solve the problem of sensemaking. For example, in intelligent systems, fused data from multiple sources reduce information overload, but hide the data from the analyst, which negates the analysts' expertise and prevents the development of their own interpretations [Leedom 2001; Klein et al. 2006].

Of particular interest to this paper is the relatively recent Pirolli/Card Sensemaking Model for Analysts [Pirolli and Card 2005]. As with other sensemaking research, Pirolli and Card pointed out that sensemaking is not based on direct insights or retrieving a final answer from memory, but rather a process that involves planning, evaluating, and reasoning about alternative future steps [Pirolli and Card 2005]. Their sensemaking model (Figure 17) can be viewed as a more detailed version of Russell et al.'s Learning Loop Complex. Like the Russell

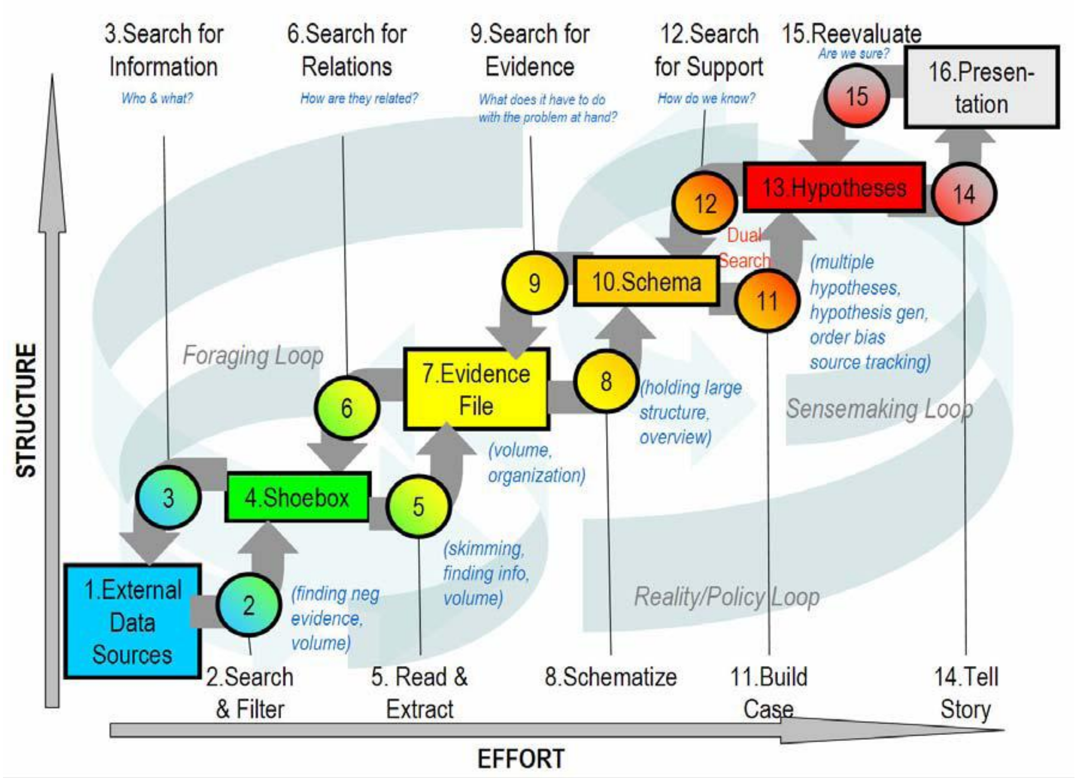


Figure 17. This figure was created by Pirolli and Card: it represents their sensemaking model for intelligence analysts [Pirolli and Card 2005]. The rectangles represent how the data representation changes at each step, while the arrows represent the process flow.

et al. model, the Pirolli/Card model focuses on how users' representations of data change during sensemaking. Pirolli and Card derived this model through cognitive task analysis and think-aloud protocol analysis of intelligence analysts' sensemaking data. The derived overall process is organized into two major loops of activities: first, a foraging loop for searching, filtering, reading, extracting information, and second, a sensemaking loop for organizing the relevant information into a large structure that leads to knowledge products. (These loops are coupled in Russell et al.'s works too [Russell et al. 1993; Furnas and Russell 2005].) The Pirolli/Card model can be viewed as a model with a low-level focus on Dervin's "bridge" component and Russell et al.'s learning loop complex," and this low-level focus made it a good fit for our interest in investigating end-user debugging in a fine-grained way. We will thus return to the Pirolli/Card model in Section 6.6.

6.4. The Sensemaking Study

6.4.1 Procedure

The study, conducted one participant at a time, used the think-aloud method. Each participant first read and signed the informed consent paperwork. We then conducted a pre-session interview about their spreadsheet background, which covered whether the participant had previously used spreadsheets for school, work, or personal reasons, and about the types of formulas he or she had written (Table 11). We then gave the participant a brief tutorial and warm-up exercise, described in Section 6.4.4., to ensure familiarity with Excel's audit features and with verbalizing during the task.

Table 11. Study participants' spreadsheet background.

Participant	Major	Spreadsheet Experience	Computer Science Background
P05200830 (Male)	Biochemistry / Biophysics (undergraduate)	School: chemistry labs. Formula experience: standard deviations, quadratic formulas, and basic arithmetic.	No programming experience.
P05211130 (Female)	Business Administration (undergraduate)	School, work, personal: checking paperwork (bank statements, personal records). Formula experience: SUM, IF, SUB, other basic functions.	An undergraduate business class on Business Application Development.
P05211600 (Male)	Animal Science (graduate)	School, work, personal: in classes, manages his own and his dorms' finances. Formula experience: basic arithmetic, statistics, and calculus.	An introductory computing class with some HTML.
P05220830 (Male)	Mechanical Engineering (undergraduate)	School, work, personal: created a spreadsheet for timing races and others for running a club. Formula experience: basic formulas, VLOOKUP, embedded IF, etc.	A Q-BASIC class in high school and is familiar with MATLAB.
P05221230 (Male)	Mechanical Engineering (undergraduate)	School and work: works as an accountant. Formula experience: AVERAGE, MIN, MAX, COUNT, COUNTA.	No programming experience.
P05270830 (Female)	Rangeland Ecology and Management (graduate)	School, work, personal: was an accountant six years, and now uses it for her research and labs. Formula experience: basic Excel formulas, IF, statistics formulas.	No programming experience (wrote a few macros years ago, but did not think she remembered how).
P05281130 (Female)	Pharmacy (undergraduate)	School: spreadsheets for labs and graphs/charts for reports. Formula experience: SUM, AVERAGE, MAX, MIN, and basic arithmetic.	An introductory CS class.
P05290830 (Female)	Animal Science (undergraduate)	School, work: was a club treasurer and calculated interest rates for classes. Formula experience: basic arithmetic, statistical formulas, and also financial formulas (N, PMT, FV, PV, I).	No programming experience.

Each participant was asked to “make sure the spreadsheet is correct and, if [you] find any errors, fix them.” We also provided the paper handout, shown in Figure 18, to give an overview of the way the spreadsheet was supposed to work. The participants had 45 minutes to debug the spreadsheet. The data we captured during the sessions were video showing their facial expressions, audio of their think-aloud verbalizations, synchronized screen recordings of the entire session (including pre-session background interviews and the task itself), and their final Excel spreadsheets.

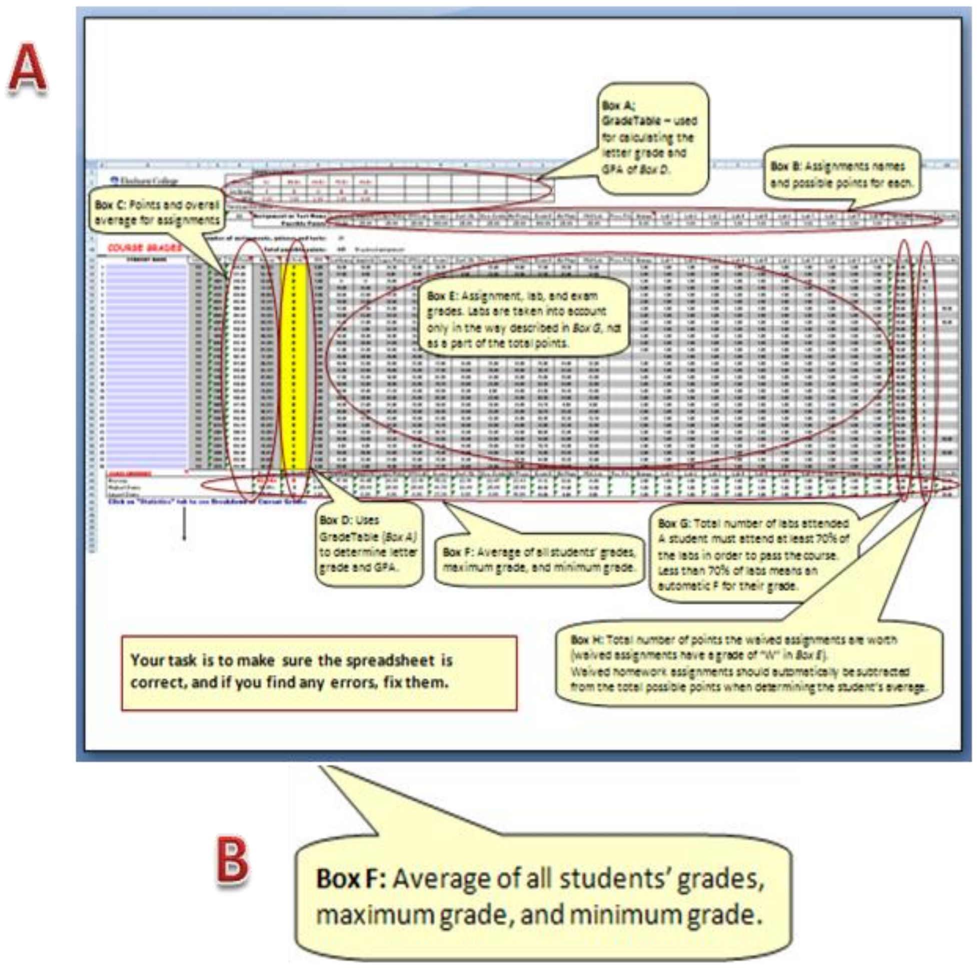

A
Box A; GradeTable – used for calculating the letter grade and GPA of Box D.
Box B: Assignments names and possible points for each.
Box C: Points and overall average for assignments
COURSE GRADES
Box E: Assignment, lab, and exam grades. Labs are taken into account only in the way described in Box G, not as a part of the total points.
Box D: Uses GradeTable (Box A) to determine letter grade and GPA.
Box F: Average of all students' grades, maximum grade, and minimum grade.
Box G: Total number of labs attended A student must attend at least 70% of the labs in order to pass the course. Less than 70% of labs means an automatic F for their grade.
Your task is to make sure the spreadsheet is correct, and if you find any errors, fix them.
Box H: Total number of points the waived assignments are worth (waived assignments have a grade of "W" in Box E). Waived homework assignments should automatically be subtracted from the total possible points when determining the student's average.
B
Box F: Average of all students' grades, maximum grade, and minimum grade.

Figure 18. (a) A thumbnail of the description handout of the grade-book spreadsheet. (b) Blowup of description Box F.

6.4.2. Participants

Ten participants (five men and five women) participated in our study. To take part in our study, participants were required to have experience with Excel spreadsheet formulas. Background in computer science was not allowed beyond the rudimentary requirements of their majors. Participants received a $20 gratuity for their participation.

We discarded the data of a participant whose verbalizations were inaudible and the data of a participant whose Excel experience was so much lower than he had claimed during recruitment that he was unable to proceed with the debugging task. The remaining eight participants were undergraduate and graduate students at Oregon State University majoring in animal science, biochemistry, business administration, mechanical engineering, pharmacy, and rangeland ecology/management. Table 11 details participant backgrounds.

6.4.3. Task and Materials

The spreadsheet the participants debugged was fairly complex. It consisted of the worksheet thumbnailed in Figure 18 and a small second sheet that produced frequency statistics and a chart of the grades calculated in the main sheet. We obtained the spreadsheet from the EUSES Spreadsheet Corpus of real-world spreadsheets [Fisher II and Rothermel 2005], most of which were obtained from the web. We chose the spreadsheet for the following reasons. First, it was complicated enough to ensure that we would obtain a reasonable amount of sensemaking data. Second, its domain (grading) was familiar to most participants, helping to avoid intimidating participants by the domain itself. Third, its real-world origin increased the external validity of our study. Finally, it has been used successfully in other studies (e.g., [Beckwith et al. 2007]), which not only provided evidence as to suitability for spreadsheet users in a lab setting, but also allowed us to harvest the bugs made by previous participants for use in the current study. We seeded the spreadsheet with a total of ten real bugs harvested from the spreadsheets of Beckwith et al.'s participants, who were Seattle-area adult spreadsheet users from a variety of occupations and age groups. None had computer science training or occupations, but all were experienced Excel users. Hence, the bugs we harvested from their work were realistic in terms of the kinds of bugs real end users generate.

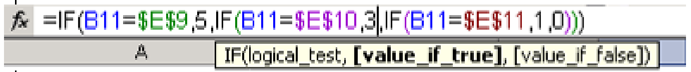
We harvested a variety of bug types from these previous users. Six of the harvested bugs were formula inconsistencies with respect to other formulas in the same row or column. For example, some cells' formulas omitted some students' grades in calculating the class average. Three more bugs were logic errors that had been propagated by their original authors over the entire row or column (e.g., using the ">" operator instead of ">="). The last bug was not part of any group of similar formulas: it counted lab attendance as a part of the total points, but should not have done so. This collection of ten real end-user bugs provided variety in the types of bugs our participants would need to track down and fix.

6.4.4. Spreadsheet Environment and Tutorial

The environment for the study was Excel 2003. To make sure participants were familiar with some of Excel's debugging features, we pointed the "auditing" collection out to them in a pre-session tutorial and suggested that these features might help them make debugging progress. However, participants were free to use any Excel features they wanted.

The tutorial's wording was about features per se, and carefully avoided hints about how to use these features strategically. Table 12 summarizes the features we presented during the tutorial. The tutorial was hands-on: the participant used the features as the tutorial went along. In addition, since most participants had not used these features prior to this experiment, everyone had five minutes after the tutorial to try the features out on their own before moving on to the main task.

Table 12. The Excel 2003 debugging features, namely Excel auditing tools plus formula tool tips, covered in the "tour of features" tutorial.

Feature	Description
	The *Arrow Features* show the relationships between spreadsheet formulas. Left to right: Trace/Remove Precedent Arrows, Trace/Remove Dependent Arrows, and Remove All Arrows.
 =IF(B11=$E$9,5,IF(B11=$E$10,3,IF(B11=$E$11,1,0))) A IF(logical_test, **[value_if_true]**, [value_if_false])	*Tool tips* can be brought up by hovering over formulas to aid in their understanding.
	The *Evaluate Formula* tool allows users to see intermediate results, by observing how nested parts of a formula are calculated step by step.
	The *Error Checking* tool points suspicious formulas with a green triangle. These cells can either be stepped through in order or examined directly within the spreadsheet.
	The *Watch Window* tool allows users to watch one or more formulas and their results, which might be helpful when inspecting cells of interest which might have scrolled off the screen.

6.4.5. Analysis Methodology

We began by labeling the debugging state changes ("bug found", "bug fixed", and "reevaluating fix") in all eight of the participants' videos. Since these identifications did not require subjective judgments, only one researcher was needed for this part. Sometimes participants believed they had fixed a bug when in fact they had not, so we also labeled these state changes as correct/incorrect. For example, editing a formula incorrectly was labeled "incorrect bug fix". Labeling these debugging state changes had two purposes. First, they pointed out milestones in the participants' success at the task as time progressed. Second, the count of participants' successful bug fixes was used to identify the corner cases for further analysis, namely the most successful and least successful female and male, a total of four participants.

For the four selected participants, two researchers then independently coded the videos according to the sensemaking codes to be described in Table 14 (which will be presented in Section 6.6.), using the following procedure. First, each researcher independently coded 20 minutes of one of the four transcripts (about 10% of the total video data), using videos as well as written transcripts in order to have full access to the context in which actions were performed and words were spoken. The coders reached an 84% inter-rater reliability, calculated using the Jaccard index. Given this level of agreement, the two researchers then split up the remaining videos and coded them independently.

6.5. Results: Participants' Success at Debugging

To provide context for the remainder of the results, we begin with the participants' success levels. Table 13 shows each participant's number of bug-find, bug-fix, and fix-reevaluation actions. We defined a bug-find action as identifying a seeded incorrect formula as being faulty, a bug-fix action as changing a faulty formula, and a fix-evaluation action as checking a bug-fix action. We also used an "incorrect" modifier for the bug finds and bug fixes. Specifically, when the participant mistakenly identified a correct formula as being faulty, we labeled it as an incorrect bug-find, and when a participant edited a formula in a way that left the formula incorrect, we labeled it an incorrect bug-fix.

As Table 13 shows, participants' sensemaking about where the bugs lurked was more successful than their sensemaking about how to fix those bugs. Specifically, in finding the bugs, six out of eight of the participants made no mistakes, and the remaining two made only one or two mistakes. When it came to actually fixing the bugs, only three of the participants made more correct fixes than incorrect ones, and one participant's incorrect fix count was as high as eight.

The number of reevaluations averaged to less than one reevaluation per fix (51 fixes and 33 reevaluations). This is consistent with prior work that suggested reevaluation in debugging tends to be undersupported in spreadsheets, and users may believe that the immediate recalculation feature is sufficient for reevaluation purposes (a "one test proves correctness" view) [Wilcox et al. 1997].

To investigate end users' sensemaking about spreadsheet correctness, we selected a subset of the participants to examine in detail. We chose the four most extreme participants two with the most bugs fixed (one female and one male), and the two with the fewest bugs

Table 13. For the bugs (10 in total) that needed fixing, these are the participants' successful and unsuccessful find and fix actions, as well as how many times they reevaluated the fixes they attempted. The top-scoring and bottom-scoring participants we selected for further detailed analysis are highlighted.

Participant	Bug Finds		Bug Fixes		Evaluations of Fixes
	Correct	Incorrect	Correct	Incorrect	
P05211130 (Female)	9	0	6	0	5
P05220830 (Male)	8	0	6	8	8
P05281130 (Female)	6	2	5	2	5
P05270830 (Female)	5	0	4	1	4
P05211600 (Male)	5	0	2	2	1
P05221230 (Male)	4	0	2	4	3
P05290830 (Female)	3	0	1	7	2
P05200830 (Male)	1	1	0	1	5

fixed (one female and one male). These participants correspond to the shaded rows in Table 13, and the remainder of this paper focuses on them. From now on, we will refer to these four as SF (successful female, participant P05211130), SM (successful male, participant P05220830), UF (unsuccessful female, participant P05290830), and UM (unsuccessful male, participant P05200830).

6.6. Results: The Sensemaking Model for End-User Debuggers

6.6.1. Three Sensemaking Loops

Pirolli and Card characterized intelligence analysts' sensemaking in terms of a major loop and its subloops, reflecting the iterative nature of sensemaking. Our data echoed this iterative character, but our participants' subloops were organized under not one but *three* major loops that corresponded to different classes of challenges in our participants' work. Our model is presented in Table 14.

We term the major loop, in which participants reasoned about the bugs and spreadsheet formulas/values, the Bug Fixing Sensemaking Loop; the loop in which they reasoned about the environment, the Environment Sensemaking Loop; and the loop in which they reasoned about common-sense topics and/or the domain, the Common Sense and Domain Sensemaking Loop. Considering these loops separately provided the conceptual benefit of focusing on the challenges introduced by a particular programming environment or domain separately from the challenges introduced by the difficulties of debugging that are independent of the environment or domain.

Table 14. A side-by-side view of how Pirolli and Card's Sensemaking model and definitions compare with our node-oriented sensemaking model of end-user debugging. Node numbers refer to those in Figure 17. Our model elements (the right hand column) were also our sensemaking code set; see text for details.

The Sensemaking Model for Analysts [Pirolli and Card 2005]	The Sensemaking Model for End-User Debuggers
External Data Sources (node 1)*:* All of the available information.	*External Data Sources:* Same as Pirolli/Card.
Shoebox (node 4)*:* Much smaller set of data, relevant for processing.	*Shoebox:* Data that a participant deemed relevant enough to "touch" in the spreadsheet or study environment. Examples: Particular cells selected, spreadsheet description handouts read, menus of features perused, help documents accessed, etc.
Evidence File (node 7)*:* Even smaller set of data extracted from the shoebox items.	*Evidence File:* Extracted from the shoebox, data that attracted a participant's interest enough for follow-up. Example: Wanting to find out more information about a suspicious cell.
Schema (node 10)*:* A large structure or overview of how the different pieces of data from the evidence file fit together: a re-representation of the data.	*Schema:* A structure or pattern a participant noticed as to how cells or information related. Examples: Declaring that all cells in an area were behaving properly or that a cell(s) did not fit the pattern.
Hypotheses (node 13)*:* A tentative representation of the conclusions with supporting arguments.	*Hypothesis:* A tentative idea about how to fix a particular bug based on the participant's schema. Example: "So it's saying that the group average is higher than it really was. I would say that is a mistake, since the formulas below it include all of them, this formula should include all" (Participant SF).
Presentation (node 16)*:* The work product.	*Presentation:* The work product. Example: An edit to fix a formula (the edit could be right or wrong).
Reevaluate (edge 15, from node 16 to 13)*:* After the presentation has been created, checking to make sure that the Presentation is indeed accurate.	*Reevaluate:* After changing a formula, making sure that the change was in fact correct. Examples: Trying to input different value to see the result of the newly edited formula, reviewing the formula to evaluate its correctness.

Table 154 (Continued).

	Environment Sensemaking: A sensemaking loop for figuring out the environment. Examples: Trying to understand Excel's error triangle feature or formula syntax.
	Common Sense and Domain Sensemaking: A sensemaking loop for answering questions about common-sense and domain questions. Example: Trying to figure out how weighted averages are normally computed.

The Bug Fixing Sensemaking Loop was where our participants spent most of their time. We devote the next subsection to it.

The Environment Sensemaking Loop arose multiple times for all four participants. It was triggered when participants tried to make sense of Excel features or syntax. For example:

SF: "So, I'm clicking on the trace dependents. [Excel displays a small thumbnail of a table with an arrow pointing from it to the formula.] [Participant hovers over the little table and then tries clicking on it. Nothing happens.] And it goes to wherever… There's a little box, but I don't know what that means."

These visits to the Environment Sensemaking Loop were sometimes disadvantageous, but other times led to leaps forward. We will point to examples of both in upcoming sections.

The third sensemaking loop was the Common Sense/Domain Sensemaking Loop. This loop involved reasoning about general knowledge items, such as trying to remember mathematical principles, or conventions used in the domain such as trying to recall how grades are usually computed. This loop was less common with our participants, perhaps because, as college students, they were very familiar with grade computations. Here is an example of accessing this loop:

SM: "In the grading, it says students must attend 70% of the labs in order to pass the course. Are, uh, is it possible to be waived from the labs?"

Because they are peripheral to our main research questions, we did not perform detailed analyses of the Environment and Common Sense/Domain Sensemaking Loops. However, we did code the instances of these loops' presence so that we could see the interactions between these loops and the Bug Fixing Sensemaking Loop.

6.6.2. The Bug Fixing Sensemaking Loop

In the Bug Fixing Loop, participants gathered information about the spreadsheet logic and made sense of it in order to create the final product, namely a bug fix. We derived the elements of our Bug Fixing Loop directly from the Pirolli/Card model for intelligence analysts [Pirolli and Card 2005]. We chose the Pirolli/Card model over the other sensemaking models presented in Section 6.3.2 because of Pirolli/Card's low-level focus on how data are used to bridge a gap. This low-level focus on Dervin's "bridge" aspect, combined with the high-level overview of the entire problem-solving process, mapped well to our investigation of end-user programmers' debugging processes.

Pirolli and Card characterized their model as consisting of four high-level sensemaking steps: information gathering, schematic representation of the information, development of an insight through manipulation, and the creation of a knowledge product. These steps clearly apply to the end-user debugging task. *Information gathering* involves finding data relevant to the task at hand by, for example, identifying relevant information on the handout or locating formulas and values relevant to a bug. An example of *schematic representation* is building a comprehensive picture of how multiple parts of the spreadsheet work together. An example of *development of an insight* is realizing the significance of a particular unexpected output value. Finally, the primary *knowledge product* is a formula modification intended to fix the bug.

Given this correspondence, the elements of the Bug Fixing Loop in our model mapped directly from the nodes from the Pirolli/Card model. All the data representation steps of the Pirolli/Card model are nodes; these are steps 1, 4, 7, 10, 13, and 16 in Figure 17. Given the complete set of nodes, the Pirolli/Card edges connecting neighboring nodes (representing mental activities that connect these nodes) are implicit. Therefore, the only edge we included explicitly was step 15 (Reevaluate), since reevaluation of changes has long been reported to be fundamental to debugging (e.g., [Nanja and Cook 1987]).

Table 14 shows our model's correspondence with Pirolli/Card's model. Note that the exclusion of the edges (except for step 15) simplifies our model. Excluding the edges had no real disadvantage because edges are implicit in node changes—to get from one node to another, one must traverse the edge connecting them. The nodes represent *data* with which a user works (such as the "shoebox"), not the *process* by which the user works with that data (such as "skimming"). The advantage of this data-oriented model was that the resulting code set greatly facilitated analysis: it was much easier for researchers to reliably (i.e., with high agreement) identify the data representation with which a participant was working than to reliably identify the process a participant was using. The right column of Table 14 thus served as our code set (except the top row which was participant-independent and therefore not of interest to our research questions). We used this code set according to the methodology previously described in Section 6.4.5.

Figure 19 shows thumbnails of the participants' progressions through the sensemaking steps up and down the Bug Fixing Sensemaking Loop. (Full-sized versions of these graphs will be shown later in Figure 21.) The graphs show participants "climbing" the Bug Fixing Sensemaking Loop, and then dropping down to earlier steps of the model. Note the prevalence of traversing adjacent nodes in the Bug Fixing Sensemaking Loop upward in direct succession. For example, participants often advanced from adding to the evidence file (yellow) to structuring that information into a schema (orange).

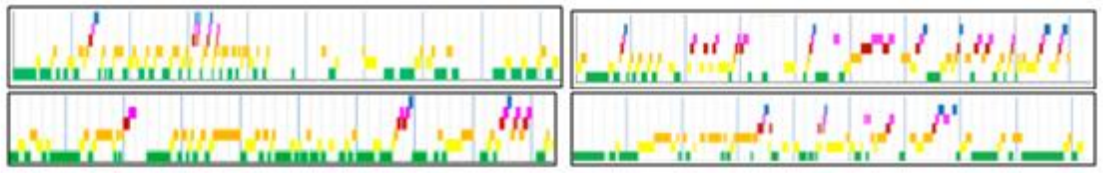


Figure 19. These thumbnails show the participants' upward climbs and downward drops in Bug Fixing sensemaking steps. X-axis: time. Y-axis: the step in the Bug Fixing sensemaking loop, from Shoebox to Reevaluate. (Time spent in the Environment and Common Sense/Domain Loops appear as horizontal gaps.) Top left: Participant SF. Top right: Participant SM. Bottom left: Participant UF. Bottom right: Participant UM.

Exceptions to the forward progressions through consecutive steps of the Bug Fixing Loop were sometimes due to switches to the other two loops (Environmental or Common Sense/Domain). In the thumbnails, these loop switches are simply shown as gaps (white space in Figure 19), such as in Participant SF's second half. Another exception was steps backward through the sensemaking model, sometimes returning to a much earlier step in the process, as we shall see in more detail shortly.

6.7. Results: Sensemaking Model Traversal Styles and strategies

6.7.1. Dominance of Foraging during Sensemaking

Figure 20 shows the sensemaking traversal frequencies for each sensemaking node in the Bug Fixing Loop, with separators marking the major subloops of the model. Left of the separators are the nodes Pirolli and Card grouped into the "foraging subloop," in which people search for information and classify information related to their task at hand. Right of the separators are the nodes of the Pirolli/Card "sensemaking subloop," in which people organize and make inferences from the information they have collected [Pirolli and Card 2005].

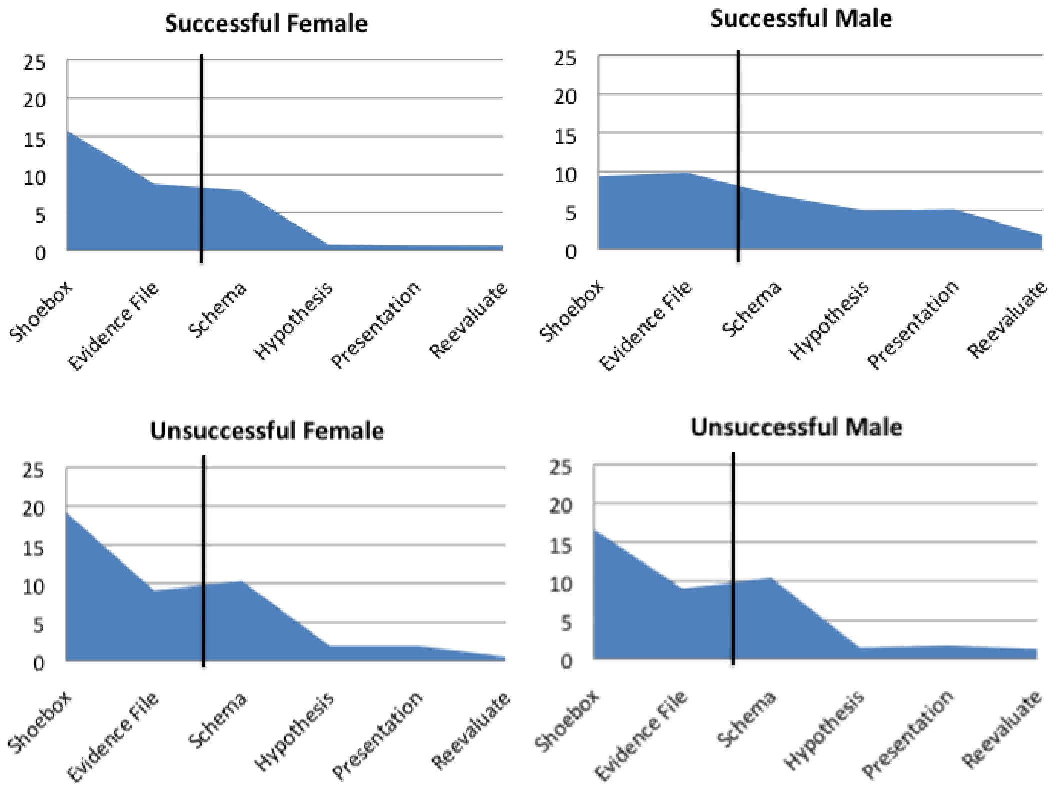


Figure 20. The amount of time in minutes (y-axis) spent at each Sensemaking model step (x-axis) by the participants. The vertical bars separate the information foraging subloop (left of the bars) from the sensemaking subloop (right of the bars).

Information foraging has an associated theory of its own, termed information foraging theory [Pirolli and Card 1999]. The theory is based on optimal foraging theory, which describes how predators (animals in the wild) follow scent to a patch where the prey (food) is likely to be. Applying these notions to the domain of information, information foraging theory predicts that predators (people in need of information) will follow scent through cues in the environment to the information patch where the prey (the information itself) seems likely to be. Information foraging theory has primarily been used to understand web browsing, but also recently has been applied to understanding and predicting professional developers' code navigation during debugging [Lawrance et al. 2008].

Figure 20 reveals two interesting points about information foraging. First, although the information foraging part of sensemaking consists of only two steps, those two steps alone accounted for half to two-thirds of all four participants' time!

Their use of foraging was to gather information about how spreadsheet cells and formulas were working and how they interrelated. Everyone began this way. For example, Participant UF used the "Evaluate Formula" tool (Table 12) early in the session to look through numerous formulas and figure out how they worked.

An example illustrating promotion from Shoebox to Evidence in the foraging subloop was Participant SM's identification of cell G12's "strange"ness. A second example was Participant UF's identification of F12 and cells like it as being of interest and decided to gather new information to pursue them.

SM: [pauses] "Interesting. I think that G12 is a strange one. None of these students had special considerations or anything, right?"

UF: "If true, that tells the percentage. And if F12 is... Oh I bet those mean what the actual percentage is... [referring to symbols she was having difficulties figuring out] I'm going to look at the trace buttons to figure out where everything is coming from."

The second point that can be seen in Figure 20 is the remarkable similarity among three of the participants' (SF, UF, and UM) allocation of time. Also note how much their sensemaking style was dominated by the foraging subloop. In contrast, Participant SM's style was somewhat more evenly distributed across sensemaking steps. Note that both styles were associated with successful debugging in that both were styles of participants who were quite successful.

6.7.2. Sensemaking and Systematic Information Processing

The Selectivity Hypothesis [Meyers-Levy 1989] offers an explanation for the differences between the two types of sensemaking loop traversals used successfully. Recall that the Selectivity Hypothesis predicts that females will gather information comprehensively: getting a comprehensive understanding before proceeding with detailed follow-up. The Selectivity Hypothesis also predicts that the males will be more selective in the information they gather, such as tending to follow up on a salient cue right away. Meyers-Levy terms this style “heuristic processing,” but because that style is characterized as being selective, we will refer to it as “selective processing.” Note that neither style is implied to be better than the other; rather, the distinction is that the former is less selective than the latter as to which information to process.

Meyers-Levy proposed the Selectivity Hypothesis to describe people when working systematically. The Merriam-Webster Dictionary defines a *systematic* approach as “a methodical procedure or plan marked by thoroughness and regularity.” The concept of a plan is closely related to the concept of *strategy*, defined by Merriam-Webster Dictionary as a “plan devised or employed toward a goal.” Both successful participants indeed demonstrated strategic thoroughness and regularity, but the unsuccessful participants did not, and therefore fall mostly outside the scope of the Selectivity Hypothesis, as the next few paragraphs explain.

As Figure 21 helps to show, Participant SF spent most of her time in the foraging subloop viewing all cells in context, gathering Shoebox data (green, lowest row) and organizing it into Evidence (yellow, next row up). She appeared to place newly collected information into the context of the overall spreadsheet and of her other data gathered, as evidenced both by the regularity of occurrence and the length of time she spent in the Schema step (orange, third row from the bottom). Also the content of her utterances during these moments expressed her views of the role of each part of the evidence:

SF: “And, what else do we have? <looks at the description handout> So we checked all the bottom rows. And... <looks at screen> We checked to make sure [the area of grades] was hardcoded up at the top, to remain consistent, not going off formulas. So, and... Let’s look in the section for class averages. Class summary.”

She seemed to have a threshold of “enough” information before moving beyond the Schema step to act upon it, as evidenced by the fact that she fixed bugs mostly in a batch, after having collected and processed much of the available information first. The only bug she fixed immediately upon finding it was an obvious fix, and did not interrupt her information gathering for long. Her approach worked well for her: recall from Table 13 that she correctly found nine bugs (more than any other participant), fixed six of them correctly, and had no incorrect fixes.

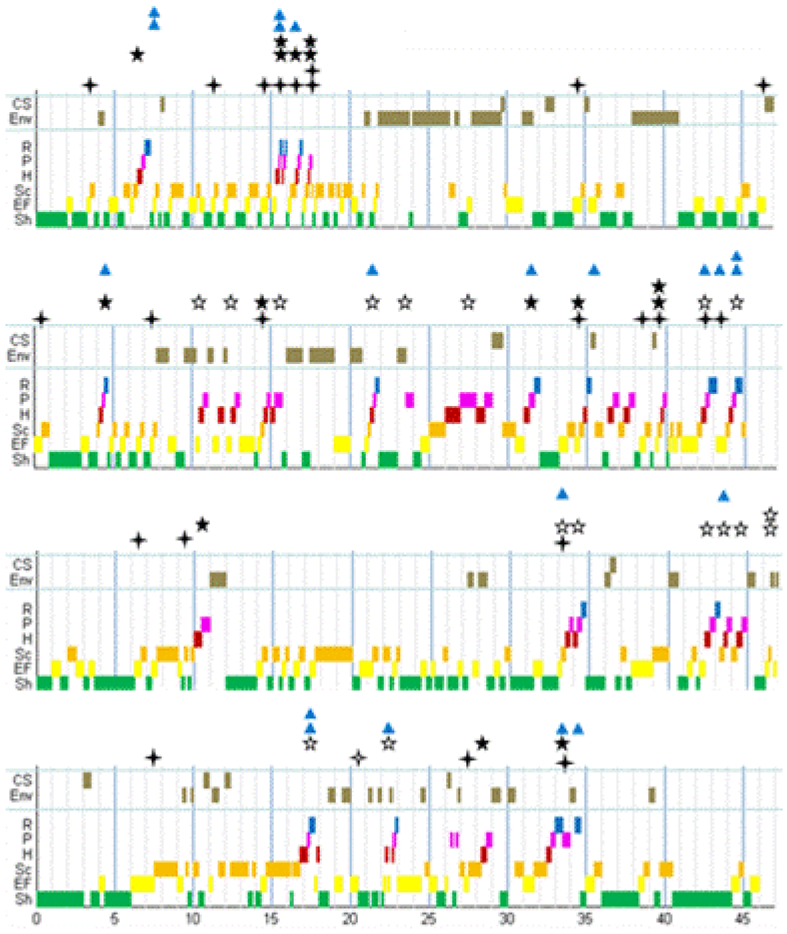


Figure 21. Full-sized view of the Sensemaking steps (y-axis) performed by four participants (from top to bottom: SF, SM, UF, UM) over time (x-axis). The top of each graph also shows the number of correct and incorrect finds and fixes, as well as the reevaluated fixes. ✦=correct (filled) or incorrect (hollow) finds; ★=correct (filled) or incorrect (hollow) fixes; ▲=reevaluates. UM has more correct finds and fixes in this figure than in Table 13, because UM introduced two bugs along the way that he later found and fixed. (The table refers only to the 10 seeded bugs.) The other three participants did not introduce bugs.

However, the comprehensive process also held disadvantages for Participant SF. Her method in the case of uncertainty was to gather more information. For example, when she thought one of the formulas looked odd (the second correct find, marked with a cross, in Figure 21's SF graph, at minute 12) but all of the cells within that region seemed equally incorrect, she did not pursue the fix right away, but rather continued with comprehensive information gathering. A disadvantage manifested itself when she did not mark the formula in any way for follow-up, and ultimately neglected to return to it. In addition to forgetting about that bug she found but never fixed, another disadvantage of comprehensive processing for Participant SF was that she did not abandon her comprehensive approach when it ceased to help her make progress. Instead, during the second half of the task, she spent most of her time following Excel's Error Checking feedback about where bugs might lie. She stayed with her comprehensive traversal through all 202 of Excel's "green triangle" warnings, even after spending over 10 minutes in this loop with only one bug find resulting. She did not attempt to fix this bug either, appearing to again rely on her memory of where the bug was, and ultimately did not follow up on a fix for it either. Instead, she opted to keep going comprehensively for the remaining 12 minutes of the task, during which time she found one more bug just before the time limit was reached.

In contrast to Participant SF, Participant SM was selective as to which information he gathered. He foraged only until he found a new bug to work on, at which point he narrowed his interest to trying to fix that bug, by moving up from the foraging subloop to the sensemaking subloop to Presentation and Reevaluation. For example, Participant SM found the same bug Participant SF found at minute 12; it was the second bug both of them found. But unlike Participant SF, he followed up on this bug right away. This happened to be the most difficult of the ten bugs to fix, but he continued to pursue it, spending a lot of time iterating on the Schema, Hypothesis, Presentation, and Reevaluation steps. He found this bug in minute 8 and fixed it in minute 32; during this time, as Figure 21 shows, Participant SM iterated through the sensemaking loop to reach the Presentation step nine times! In contrast, Participant SF, who gathered much more information up front, never iterated to the top of the sensemaking loop more than once for any bug fix. Participant SM's process worked well for him: he found eight bugs successfully and fixed six of them (including the bug with which participants had the most difficulty).

However, the selective processing style also held disadvantages for Participant SM. He missed much of the information that had enabled Participant SF to spot and fix several bugs early. Participant SF fixed six bugs during the first half of the task, compared to only two by Participant SM. Furthermore, comprehensive processing might have provided useful information in solving the difficult bug upon which he spent so much time, in addition to potentially helping to find and fix some of the other bugs more quickly.

We have pointed out how consistent the above details for the successful participants were with the Selectivity Hypothesis. This consistency was triangulated against other aspects of the data in multiple ways. First, the amount of time spent in each subloop (Figure 20, previous section) helps to confirm Participant SF's comprehensive style and Participant SM's selective style. Second, Participant SF's batch of several Presentation instances (bug fix attempts) together versus Participant SM's incremental timing of each Presentation instance (Figure 21) also helps to confirm Participant SF's comprehensive style and Participant SM's selective style. Third, consider the number of transitions between steps. Figure 22 traces participants' sensemaking paths through the sensemaking model. Notice Participant SF's heavy emphasis on traversals between Shoebox to Schema, forming a cleanly separated "module" with only one transition between the "middle" of that subloop and bug follow-ups in the upper subloop. In contrast, Participant SM's most common transition was from Hypothesis to Presentation: his style showed a fairly uniform amount of activity on each upward transition progression in his pursuit of each bug, from Shoebox to Presentation and Reevaluation.

On the other hand, Participants UF and UM were mostly not systematic. One of these participants (UF) expressed plans but did not follow them; the other (UM) did not express plans at all. Further, neither showed signs of regularity or thoroughness. Instead, their approach seems better described as a sequence of counterproductive self-interruptions [Jin and Dabbish 2009].

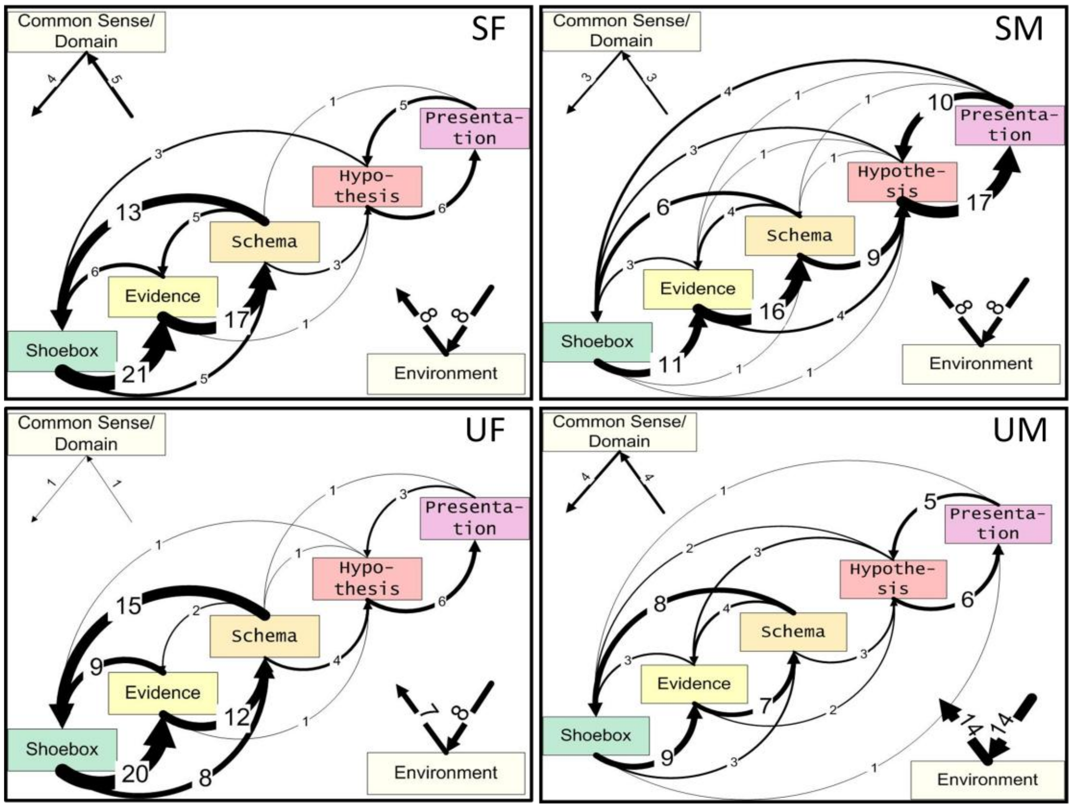


Figure 22. Frequency of transitions between sensemaking model steps by participant. Notice participant SF (top left) and participant UF (bottom left) transition mostly between different steps of the information foraging loop. Participant SM (top right) climbs up the sensemaking ladder in a mostly ordered manner, while participant UM takes a mixed approach and transitions into the Environment Loop more than any other participant.

We illustrate this first with Participant UF. Like Participant SF, Participant UF at first followed the layout of the specifications comprehensively (exhibiting regularity), but unlike Participant SF, she abandoned the comprehensive approach at her first bug find (minute 7). She focused on this bug (selective processing) for only three minutes, then found a second bug and chose to switch to that one instead (which she fixed immediately). This switch was productive in an immediate sense, but cost a loss of context regarding the first bug [Jin and Dabbish 2009]. Jin and Dabbish point out that triggered self-interruptions' disadvantages include difficulty refocusing on the first task's context, and likelihood of causing later self-interruptions. Indeed, at this point, about 10 minutes in, Participant UF's systematicness ended. For the rest of her session, her behavior was neither regular nor thorough: there ceased to be evidence of any "big picture" awareness, the focus of her verbalizations and formula reading shifted dramatically without closure on any one section or bug before moving on to the next, and her actions (cells selected for reading or editing) tended to be unrelated to the plans she did verbalize. Her behavior lacked the coverage to be considered comprehensive. Nor was it a systematic selective approach; for example, it was very different from Participant SM's selective but thorough focus on one bug, in which he always persevered with his most recent bug find until he had fixed it.

For example, although Participant UF occasionally expressed intent to follow up on one bug, she often immediately followed such verbalizations with actions *un*related to her expressed intent. For example, she expressed a plan at minute 18: "Okay, I'm gonna focus on the 'letter grades' <the first bug she found> because it seems like there is some inconsistency in how they are being calculated. Uhh, I'm not going to worry about the 'lab grades' because they all completed all of the labs. I'm going to ignore the 'total points' because it seems like those are all correct." This verbalization was, however, not followed by pursuing 'letter grades'; instead, she spent six minutes on 'total points' (which she had said she planned to ignore), and then seven minutes on 'GPA'. Following this, she briefly once again returned to the 'letter grade' formula, but for less than a minute, after which she moved to a new bug she then noticed in the 'waived' formula, never returning again to 'letter grade.' None of her activities after minute 12 led to any successful bug fixes.

Participant UM's approach was similar to Participant UF's but was even more ad hoc. Unlike Participant UF, who verbalized plans that she did not follow up on, Participant UM did not verbalize any plans at all. Many of his focus switches from one cell to the next were less than one second apart, far too little time to actually read a formula or warning message associated with that cell. "There was a, like, little green arrow thing next to D22. As I was looking down the list, and I just clicked on it. And then I just clicked on the error checking and..." This is in sharp contrast to the way Participant SF used the same Error Checking (green triangles) tool. When she used this tool, her verbalizations described use of the tool in the context of the whole spreadsheet, stating that she wanted to look at all of the inconsistent formula warnings in the spreadsheet (systematic comprehensive processing).

When Participant UM used the same tool, his ad hoc behavior and quick attention switches suggest that the tool was instead primarily a self-interruption trigger.

Considering sensemaking from the standpoint of systematicness thus yields three classes of insights. First, for the two participants whose behavior was systematic, our data supports the Selectivity Hypothesis, with the female choosing a comprehensive information processing style and the male following a selective information processing style, just as the Selectivity Hypothesis predicts. Second, we observed several advantages and disadvantages with each systematic style. Third, the lack of systematicness of the other two participants helps us to understand why they ran into trouble and where. A “why” insight comes from the details revealing their numerous self-interruptions with attendant loss of context, and “where” insights are revealed by the graphs in Figure 20-22, which make clear that both unsuccessful participants spent a lot of time trying to build a Schema and also switched quickly in and out of the Environment Loop. These are two of the trouble spots we describe in more detail in the next section.

6.8. Results: Sensemaking Trouble Spots

6.8.1. The Environment and Common Sense / Domain Loops

Recall that our sensemaking model has three loops: the main Bug Fixing Loop, the Environment Loop, and the Common Sense / Domain Loop. When participants exited the main Bug Fixing Loop to go into one of the other two loops, it was usually to go to the Environment Loop. Departures to the Common Sense / Domain Loop were few in number, and tended to be short in duration, but it is not surprising that our participants did not spend much time trying to make sense of the domain, since grade calculation is familiar to students.

Self-interruptions to switch to the Environment Loop arose in two situations. The first was when participants were having difficulties with some construct in the software (formula syntax, features, etc.). Participant UM had many instances of this situation, transitioning in and out of the Environment Loop almost twice as often as the other three participants (Figure 22). For example, while using the Evaluate Formula feature to understand a lengthy formula, he said:

UM: "And then, when I click this 'Evaluate Formula' button, it says if Z12 is greater… I forget what the name of the symbol… The greater than or equal to symbol. [Clicks Evaluate a couple of times] False. [Clicks Evaluate some more.] He gets an F. [Shakes head.] That doesn't make any sense."

The other three participants also spent time trying to understand features' meanings and operators or functions they could use. For example:

UF: "I'm just trying to figure out again how to do an 'AND' statement. [Tries it and gets an error message.] Yeah, I figured that would happen."

SM: "Um, how would I assign a number to W? [Pauses.] Let me do a look-up formula. [Searches Help for VLOOKUP.]"

The second situation in which participants switched to the Environment Loop arose when they wanted the environment's suggestions on what to do next. An example of this was Participant SF's reliance on Excel's green triangles to lead her through suspicious cells in the hopes of finding more bugs. She spent about twice as many minutes in the Environment Loop as any of the other participants (SF: 10.8 min, SM: 5.8 min, UF: 3.7 min, UM: 4.2 min).

SF: "So the rest of them are correct. [Double checking that they're correct.] And, what else do we have? [Decides to follow up on Excel's green Error Checking triangles] <details omitted> Trying to figure out why… Hm… <details omitted> Why is that one [formula] inconsistent from the one next to it?"

Participant UM also tried to follow Excel's Error Checking feedback about what to do next, although with less success.

The graphs of the successful versus unsuccessful participants show a marked difference in their excursions into the Environment Loop (Figure 21). The two successful participants both tended to remain in the Environment Loop for longer periods at a time than the other two participants, cycling through it until they reached the goals that had sent them into the Environment Loop. The unsuccessful participants, on the other hand, tended to spend only short times in the Environment Loop, usually returning to the Bug Fixing Loop without a satisfactory answer. In these outcomes, participants' short times in the Environment Loop were simply interruptions, and did not deliver benefits to their debugging efforts.

6.8.2. Sticking Points Moving Up the Sensemaking Steps

As Figure 21 shows, instances in which participants made progress—with new bugs found or fixes at least attempted—were almost all marked by (1) *rapid* transitions between (2) *adjacent* steps (3) *upward* in the sensemaking model. The rapidity of the transitions during successful periods is apparent in Figure 21: the periods culminating in bugs found or fixes attempted were characterized by numerous tiny chunks of time spent in one step before moving to another sensemaking step. A close look at the figure shows not only the rapidity of transitions, but also that the transitions during these periods of progress were almost entirely between adjacent sensemaking steps, and were almost entirely upward.

Deviations from this pattern were usually signs of trouble. A case in point was the unsuccessful participants' propensity to get "stuck" on the Schema step. We were alerted to this possibility by the odd looking bumps in their graphs at the Schema step in Figure 20. In fact, as Figure 23 shows, during the first half of the task, the two unsuccessful participants were stuck at the Schema step for minutes at a time. There were no bug finds during these long stretches and, upon exiting the Schema step, the participants almost always went all the way back to the Shoebox (which can be seen in Figure 21), rather than progressing upward to

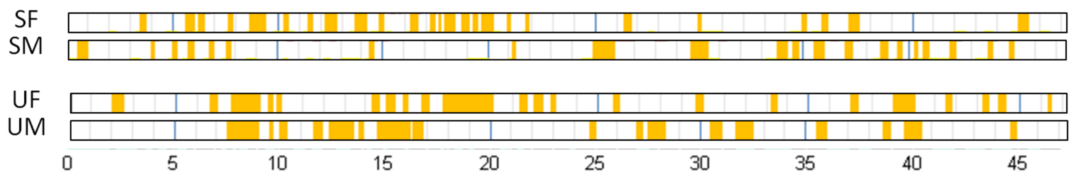


Figure 23. Excerpts from Figure 21 to bring out times spent in the Schema step. Note that the successful participants (top two) switched in and out of the Schema much more quickly than their unsuccessful counterparts (bottom two), who had a tendency to get stuck on that step.

Hypothesis or down to the adjacent Evidence File to reconsider the usefulness of data previously identified as being pertinent. Thus, the Schema step, in which participants synthesized the information they had decided was relevant evidence, was clearly a trouble spot for the two unsuccessful participants.

Transitions between some sensemaking steps may be detectable by tools. For example, attempts to fix bugs are, by definition, edits to cells that have formulas in them. Similarly, periods characterized by displaying several different formulas may correspond to the Shoebox step, and periods characterized by reviewing formulas already viewed before may correspond to the Evidence step. If these kinds of detection can be automatically done, tools may be able to use this information to discern when a user is stuck at the Schema step and having trouble making debugging progress. This knowledge might then be used to focus on-line help or assistance tools that users access during these periods.

6.8.3. Moving Down the Sensemaking Steps

Although upward transitions tended to move incrementally, downward transitions were less predictable. In fact, participants had more than twice as many “step skips” in their downward transitions as they did in upward transitions. When moving in a downward direction, they most often fell all the way down to the Shoebox stage, as Figure 22 shows. A possible interpretation of these fallbacks to the beginning is that it may have seemed easier for participants to make progress based on newly collected data than to sort out which of the earlier steps led to a correct or incorrect conclusion.

Only one step was less subject to the “back to square one” phenomenon: the Presentation step. Recall that Reevaluate was a transition (edge) from the Presentation step down to the Hypothesis step, resulting in either the validation or rejection of the hypothesis. For all four participants, this step was the *only* step in which returning to the previous step dominated over going back to the beginning.

What happened from the Hypothesis step on, is still a mystery. The Sensemaking model might suggest that participants would then search for support for their hypothesis in the Schema step, perhaps judging which assumptions made at that step were correct and incorrect, determining whether to move up or down from there. However, the transition from Hypothesis back to the Schema was taken only *once* by Participant SM and Participant UF, and *never* by Participant SF and Participant UM. Thus, it appears that the participants neither incrementally changed nor revisited their Schema after the Hypothesis step.

6.9. Discussion

The previous section's analysis makes clear the contrast in sensemaking traversal patterns between trouble spots versus instances of forward progress. This contrast suggests opportunities on how tools might detect the user's sensemaking step, which could lead to tools whose support is targeted at exactly that sensemaking stage.

For example, user accesses of help mechanisms were a sign of detours to the Environment Loop, and quick abandonment of a feature for which the user had just sought help would suggest that the detour was an unproductive one. This implies a need for the tool to explain the subject matter a different way if the user returns to the feature later. Other examples that could be detected by tools were long periods of formula inspections, which were usually in the Shoebox stage, and periods of follow-ups such as tracing formulas back through dataflow, which were often signs of the Evidence stage.

If tools like these were able to detect the user's current sensemaking step and compare it to the last few sensemaking steps, the tool might then be able to tailor its behavior depending on whether the user was progressing up the sensemaking loop versus systematically moving down versus falling down precipitously. For example, recall that often, the participants' downward transition patterns skipped many steps, thereby losing portions of sense already made, as with Participant SF who forgot about bugs she had already located and therefore never attempted to fix. Tools that could help users record and track evidence and hypotheses already gathered might be possible in a very low cost way, enabling users to systematically revisit otherwise forgotten or erroneously rejected assumptions. Just as a Sudoku player might recognize the usefulness of keeping track of penciled-in assumptions about which values are still viable for a square, and crossing them out one at a time, for end-user programmers the externalization of assumptions might help them notice important patterns and see interrelationships they may not have detected when keeping everything in the head. Two tool examples that allow tracking of one kind of assumptions in spreadsheet debugging are value range assertions for Forms/3 [Burnett et al. 2003] and Excel's data validation feature. Perhaps future, lighter weight tools are possible that allow tracking of other assumptions the user has made but might want to revisit.

One thing the model revealed was the dominance of information foraging. This aspect of sensemaking occupied half to two-thirds of participants' time; yet, foraging in end-user debugging has not yet been discussed in the literature on end-user programming practices. There is, however, recent research about professional programmers' debugging that has proposed tool possibilities based on information foraging theory: for example, constructs such as scent could be used to analyze the efficacy of environments, tools, and source code [Lawrance et al. 2008]. Our results also suggest the need for tools to explicitly support information foraging by end-user debuggers, in this case in spreadsheets, where theory constructs such as scent could be applied to spreadsheet formulas, layout, and structure.

Further, the model revealed that the two information processing styles proposed by others' research appeared to correspond to successful foraging, namely the comprehensive and selective styles. However, while both comprehensive and selective processing were successful styles, both also had disadvantages. For example, Participant SF lost track of bugs she had found while focusing on comprehensive processing, and Participant SM's bug finding and fixing seemed hampered by a lack of information. The lack of systematicness and its toll on the other two participants was far more obvious. Finally, although most of the participants attempted to traverse the spreadsheet systematically at least during some periods, they all missed some cells. These examples suggest that tools should facilitate systematic traversals of the spreadsheet, and further should do so in a way that is conducive to either the comprehensive or to useful selective styles of information processing, such as depth first.

As with all empirical studies, our study's threats to validity need to be taken into account in assessing our results. An internal validity threat in our study is that the specific spreadsheet or the specific bugs seeded could affect the participants' sensemaking process as they debugged. To reduce this threat, we harvested bugs that had been created as side effects of other work by experienced spreadsheet users working on the same real-world gradebook spreadsheet used in our study. A lack of understanding of Excel 2003 could also have influenced results, which we attempted to mitigate by requiring participants to be familiar with Excel 2003, (in fact, selecting the ten most experienced volunteers who responded to our recruitment notice), and giving a tutorial on certain features to ensure specific skill levels. Our study contains a construct validity threat because think-aloud data is known to be a highly imperfect representation of what humans actually think. We attempted to mitigate this threat by also collecting behavior data (actions), which were used for triangulating with spoken data. Regarding external validity, a think-aloud study is by definition an artificial situation that does not occur in the real world. Further, our participants may not be representative of the larger population of end-user spreadsheet users. In addition, lab experiments entail artificial constraints; in our case these included a tutorial before the task, a short time (45 minutes) to complete the debugging task, and presence of a computer-based video-recorder, any of which could have influenced participants' actions. These threats can be fully addressed only through future studies using different spreadsheets, bugs, and participants, and we hope future researchers will be interested in making use of our model for exactly this purpose.

6.10. Conclusions and Future Work

This work represents the first application of sensemaking research to end-user debugging. To gain a sensemaking perspective on end users' debugging, we began by deriving a sensemaking model for end-user debugging. The model owes its roots to the Pirolli/Card sensemaking model for intelligence analysis, and from this start, we derived the new model for end-user debugging from our participants' data. The data revealed not just one major sensemaking loop, but three intertwined major loops, which we termed the Bug

Fixing Loop, the Environment Loop, and the Common Sense / Domain Loop. We then used the model and its three major loops to shed light on end users' debugging approaches and problems that arose in the sensemaking central to debugging.

One contribution of this work has been (1) a new model of sensemaking by end-user debuggers, which consists of three connected sensemaking loops: one for reasoning about the "program" itself, one for reasoning about the programming environment, and one for reasoning about the domain. This model then enabled empirical contributions revealing (2) the dominance of information foraging in our end users' debugging efforts, (3) two successful strategies for systematic sensemaking and their consistency with gender difference literature in information processing styles, (4) a detailed account of transitions among sensemaking steps and among the three sensemaking loops in our model, (5) the sensemaking sequences that were tied to successful outcomes versus those that identified trouble, and (6) specific sensemaking trouble spots and consequent information losses. These findings suggest a number of opportunities for researchers and tool designers to consider how to support end users' sensemaking as they debug.

Regarding our own future work, we plan to do exactly that—experiment with tool ideas to better support end-user debuggers' sensemaking. We plan an intertwined set of further empirical work and tool experimentation. The purpose of these efforts will be to investigate how an end-user programming environment can better support end-user debugging by following opportunities and fulfilling needs the sensemaking model has helped reveal. The empirical component will not only inform our efforts, but also will help us ultimately to understand *whether* our new tools work and *how* tools might best support their sensemaking efforts of female and male end-user programmers. We believe that the fresh perspective the sensemaking model provides on the difficult task of debugging may hold the key to paving the way to a new generation of sensemaking-oriented debugging tools, and we hope other researchers will join us in these investigations.

PART 3 – STRATEGY-BASED DEBUGGING TOOLS:
Design, Implementation, and Evaluation

<u>Overview of Part III of the Dissertation:</u>

Chapter 7: What detailed implications for design arise from a comprehensive understanding of strategy usage in Excel?

This study presents a comprehensive overview of end-user debugging strategy items in Excel (at four levels and in two contexts) in preparation for the design of a new strategy-based end-user debugging tool: StratCel. In particular, we reanalyzed the data collected for the Sensemaking Study in order to answer two new research questions: (1) If a debugging tool were to support end-user programmers' specific debugging strategy needs, what should it take into account and how? (2) How could a design address those requirements?

This study's findings include:

- The importance of performing a comprehensive overview of the environment to be improved, since each level led to a different type of implication for design.
- Our analysis of only four participants' data led to 21 new implications for the design of end-user debugging tools and six implications for future research were also derived from this work.
- Based on triangulating evidence, we believe that both the analysis methods employed here and the implications for design generalize across debugging environments.

The resulting implications for design can be addressed together in the design of a strategy-centric end-user debugging tool.

Chapter 8: Are the strategy-centric implications for design effective in improving end-user programmers' debugging success?

Would addressing strategy-centric implications for design be powerful enough to improve debugging success? We report this iterative design, implementation, and evaluation process of StratCel, a strategy-centric debugging tool for Excel, in Chapter 8.

Employing a strategy-based approach to the design of end-user debugging tools led to the following main contributions:

- A novel empirically-based end-user debugging tool, StratCel, created to support end-user programmers' specific debugging strategy needs.
- A positive impact on end-user debugging success: (1) twice as many bugs found by participants using StratCel compared to Excel alone, (2) four times as many bugs fixed, (3) in a fraction of the time, (4) including two bugs which both the researchers and Control group had overlooked, and (5) a closing gap in success based on individual differences.
- Participants' promising comments about StratCel's usability and its applicability to their personal projects and experiences.
- Design guidelines, based on instantiated and validated empirically-based implications for design.

Lastly, we argued for the generalizability of this approach and list several opportunities for future research.

Design Implications for End-User Debugging Tools: A Strategy-Based View

Valentina Grigoreanu, Margaret Burnett, and George Robertson

Journal Paper under Review

Submitted August 19, 2009

44 pages

7. The EXCEL STRATEGY ITEMS STUDY: Implications for StratCel's Design

7.1. Abstract

End-user programmers' code (e.g., accountants' spreadsheet formulas) is fraught with errors. To help mitigate this problem, end-user software engineering research is becoming established. However, most of this work has focused on feature usage, rather than debugging strategies. If a debugging tool were to support end-user programmers' specific *debugging strategy* needs, what should it take into account and how? To consider the design of such tools, this work contributes a *comprehensive overview* of end-user debugging strategies at four strategy levels. An example empirical study in Microsoft Excel demonstrates that this view of debugging provides useful insights, and we argue that many of these insights generalize to other environments. Our results include end-user debugging *tactics* and the effective and ineffective *moves* employed to achieve them, ten end-user debugging *stratagems* applied to a new environment, and how these stratagems were used within three contexts: by *strategy* used, by sensemaking step, and by debugging phase. These findings coalesce into a comprehensive overview of end-user debugging strategies and detailed implications for the design of strategy-based end-user debugging tools.

7.2. Introduction and Related Work

7.2.1. Problem Addressed and Terminology

In the United States today, there are tens of millions more end-user programmers than there are professional developers [Scaffidi et al. 2005]. *End-user programmers* were described by Nardi as people who (as opposed to *professional developers*) do not code as an end in itself, but as a means to more quickly accomplish their own tasks or hobbies [Nardi 1993]. In fact, end-user programmers often do not have professional computer science training; examples include accountants using spreadsheet formulas to keep track of a company's budget in Excel and designers building interactive web applications in Flash.

End-user programmers' code is widely known to be rife with errors [Panko 1998; Butler 2000; Boehm and Basili 2001; EuSpRIG 2006; Panko and Orday 2005], costing companies and governments millions of dollars. For example, a Nevada city budget spreadsheet posted to the city's website and distributed to city council members falsely showed a $5 million dollar deficit in the water and sewer fund [Nevada Daily Mail 2006]. This discrepancy delayed voting on the city's yearly budget. Upon closer examination, the finance director found several bugs in the budget spreadsheet. This is just one of many such news stories [EuSpRIG 2006].

Much of the work on creating tools to help end-user programmers eliminate such bugs has been about feature usage and mapping techniques from professional software engineering to this population. However, one area which we have recently begun conducting research into is end-user programmers' specific debugging strategy needs. While in our earlier work we used the term "strategy" to refer to any "reasoned plan of action for achieving a goal," in this paper we differentiate between nuances of strategy items at different levels. To do so, we employ Bates' terminology for four strategy levels for online searching [Bates 1990].

Bates argues that search systems should be designed to make a good strategy easy to employ, by taking into account search behaviors which promote the strategic goals of searching for information [Bates 1990]. This advice also holds in designing debugging tools for end-user programming environments. Bates' four levels of strategy are moves, tactics, stratagems, and strategies. A *move* is at the lowest level of abstraction: "an identifiable thought or action" [Bates 1990]. A *tactic* is "one or a handful of moves made." A *stratagem* is "a larger, more complex, set of thoughts and/or actions." Finally, a *strategy* is "a plan which may contain moves, tactics, and/or stratagems for *the entire search*," or in our case, the entire debugging session. To refer to all four of these levels, we employ the phrase *strategy levels*, and to refer to an item at any of the four levels, we employ the term *strategy item*.

7.2.2. Background and Related Work

In this section, we briefly summarize related work to situate our current paper within the existing literature. However, we defer detailed discussion of related work until the results sections.

Three areas of research are particularly relevant to our current focus: work on end-user programmers in general, on professional programmers' debugging strategies, and on gender differences in problem-solving strategies. Research on end-user programmers is becoming established, ranging from automatic error detection in spreadsheets to a testing methodology for end users and to the design of web applications (e.g., [Abraham and Erwig 2007; Ayalew and Mittermeir 2003; Beckwith et al. 2006; Burnett et al. 2003; Burnett et al. 2004; Ko and Myers 2004; Myers et al. 2006; Nardi 1993; Rode and Rosson 2003; Rosson et al. 2007]). Our work also has strong ties to research on novice and expert professional developers' debugging practices (e.g., [Jeffries 1982; Kelleher and Pausch 2005; Nanja and Cook 1987; Pane and Myers 1996]), and especially on professional developers' debugging strategies (e.g., [Katz and Anderson 1988; Romero et al. 2007]). Finally, gender differences have been observed in the use of problem-solving strategies ranging from mathematics to financial decision making, and neuroeconomics, among others (e.g., [Bandura 1986; Byrnes et al. 1999); Carr and Jessup 1997; Gallagher and De Lisi 1994; Gunzelmann and Anderson 2008; Jay 2005; Lawton and Kallai 2002; Meyers-Levy 1989; Powell and Ansic 1997]). And they have even been observed in end-user programmers' feature usage (e.g., [Beckwith et al. 2005; Busch 1995; Ioannidou et al. 2008; O'Donnell and Johnson 2001]). Thus, in the current study, we made sure to involve an equal number of male and female participants, and to pull from the literature on end-user programming and on debugging strategies in discussing the generalizability of our findings.

Our own previous work on end-user debugging strategies has provided the background for the current study. In the upcoming few paragraphs, we use Bates' terminology to describe that work (recall that each of those papers employs the more general term "strategy").

For our first formative work in the direction of strategies, we mined existing end-user debugging moves data, using the sequential pattern matching data mining method [Grigoreanu et al. 2006]. This first study revealed frequencies of types of tactics employed by users in the Forms/3 research spreadsheet environment. The frequencies of tactic usage were almost identical for the *un*successful females and the successful males. However, for this first study, we had no in-the-head data from the users. This analysis was later extended in [Fern et al. 2009] to automatically approximate some end-user debugging stratagems based on sequences of moves alone, without verbal data from the users.

Our first study to take participants' verbal assessments of their strategy items into account revealed eight end-user debugging stratagems, and gender differences in either the preference or effectiveness of seven of them [Subrahmaniyan et al. 2008]. Furthermore, females' stratagems were least well-supported in end-user debugging environments.

The three studies mentioned above used an academic spreadsheet prototype (Forms/3) and academic study participants. This approach has clear advantages: the features presented in those studies can be carefully controlled, participants are fairly easy to come by, and there are few limitations on how the software can be modified for future studies. However, a tradeoff of such studies is that they have low external validity; the results pertain only to that very specific academic setting. We therefore applied this set of stratagems to qualitatively code IT professionals stratagems during script debugging in the commercial environment Windows PowerShell [Grigoreanu et al. 2009]. Use of a wildcard code revealed two new stratagems, and several of the stratagems' definitions were modified to generalize across end-user debugging environments. This work resulted in a set of ten stratagems (listed in Table 19), which we employed as our code set in analyzing the data from the study reported in this paper.

Testing was one of the male effective stratagems, which was especially well-supported by the Forms/3 environment. Would it be enough to encourage females to use *testing*? We added two tools to Forms/3 to answer this question: one for reducing the risk of employing the new testing features and another to explain how to employ the *testing* stratagem [Grigoreanu et al. 2008]. While this approach increased females' use of the *testing* stratagem and had a positive effect on their confidence, it did not lead to an increase in either bugs found or fixed. Encouraging users to employ one particular effective stratagem might not be the right approach. Instead, we believe end-user debugging tools should encourage the effective usage of all ten stratagems.

Moves and tactics can be observed, and stratagems can be extracted from the user with open-ended questions and think-aloud methods, as in the previous studies. However, the overall strategy the user is employing (if it even exists) is often unconscious, may change as progress is made toward the ultimate goal, is hard to observe, and is even harder to measure scientifically. To measure strategies, we turned to research on sensemaking, which provided us with a model of the entire problem-solving process.

Sensemaking is a term used to describe how people make sense of the information around them, and how to represent and encode that knowledge, so as to answer task-specific questions [Furnas and Russell 2005; Russell et al. 1993]. Three seminal sensemaking models include Dervin's sense-making triangle [Dervin et al. 2003], Russell et al.'s learning loop in measuring the cost structure of sensemaking [Russell et al. 1993], and Pirolli and Card's Sensemaking Model for Analysts [Pirolli and Card 2005]. Since the 1970's, many other sensemaking studies have been conducted in several domains (e.g., [Shrager and Klahr 1986; Furnas and Russell 2005; Leedom 2001; Klein et al. 2006]). Stefik et al. provide a comprehensive overview of the existing sensemaking models [Stefik et al. 2002]. In one of our earlier papers, borne out of the empirical study which this paper is also based on, we derived a Sensemaking Model for End-User Programmers [Grigoreanu et al. 2009] based on the Sensemaking Model for Intelligence Analysts [Pirolli and Card 2005]. Using this model, we observed two end-user debugging strategies: comprehensive and selective debugging [Grigoreanu et al. 2009].

7.2.3. Research Contributions and Overview of this Paper

The contribution of this paper is the next step in this line of research: a *comprehensive overview* of end-user programmers' *debugging strategies*. We analyzed the data collected for [Grigoreanu et al. 2009] at the other three levels of strategy: moves, tactics, and stratagems. We believe that this approach was fruitful in providing implications for both the improvement of existing features and the design of novel tools to help end-user programmers create more correct code. In particular, our research goals were to:

Examine a set of strategy data at four different levels of abstraction (moves, tactics, stratagems, and strategies),

and

Investigate the use of strategy items in the context of the purpose for which they are employed: sensemaking step and debugging phase (bug finding, bug fixing, and evaluating a fix).

This paper presents three main contributions. (1) In addressing the two goals above, it is the first to provide a comprehensive overview of strategy item usage during end-user debugging. We analyzed our participants' data at four strategy levels and in the context of two purposes. (2) Analysis at each level and for each purpose resulted in a different type of implication for the improvement of one of the most popular end-user programming environments: Excel. These implications ranged from iterative improvements to existing tools to the design of novel tools for Excel. The range of implications highlights the importance of investigating multiple strategy levels and contexts in the design of tools. (3) We discuss the generalizability of these findings to other programming environments.

We begin in Section 7.3 by presenting the study methodology and analytical methods. Next, we report our results on the comprehensive view of end-user debugging strategy items at four levels and in two contexts in Sections 7.4-7.6. This is also where we integrate the resulting implications for design, reveal opportunities for future research, and discuss the generalizability of our findings. Finally, we reiterate our conclusions in Section 7.7.

7.3. Study Methodology

The data we analyzed for this study came from an earlier experiment which examined how traces of Sensemaking model traversals can be used to better understand end-user debugging *strategies* [Grigoreanu et al. 2009]. In this second round of analysis, we complete those findings by also examining the other three strategy levels (*moves, tactics*, and *stratagems*), in the context of both individual sensemaking steps and also by debugging phase. While we have reiterated the main parts of our study methodology here, the reader should refer to [Grigoreanu et al. 2009)] for a more complete overview.

7.3.1. Participants

Our eight participants (four men and four women) were undergraduate and graduate students at Oregon State University who had experience with Excel spreadsheet formulas. They received a $20 gratuity for their participation. None of the participants were computer science or electrical engineering majors, and they did not have any programming experience beyond the requirements of their majors (for example, a Business class on Business Application Development). In this study, we examined the four most and least successful participants' strategy items in much detail. We refer to these as Participant SM (successful male), SF (successful female), UM (unsuccessful male), and UF (unsuccessful female).

7.3.2. Procedure, Task, and Materials

We used a think-aloud methodology [Ericsson and Simon 1984] to capture end-user programmers' in-the-head strategic activities data. After signing the required paperwork, each participant answered interview questions about their spreadsheet background, got a brief tutorial about the Excel auditing features, and practiced thinking aloud on a short warm-up task.

The participants' actual task is shown in Figure 24; the figure shows the paper handout we provided them to describe how the spreadsheet was supposed to work. This spreadsheet was obtained from the EUSES Spreadsheet Corpus of real-world spreadsheets [Fisher II and Rothermel 2005]. It was complex, containing two worksheets and 288 formula cells. Each participant had 45 minutes to "make sure the spreadsheet is correct and, if [you]

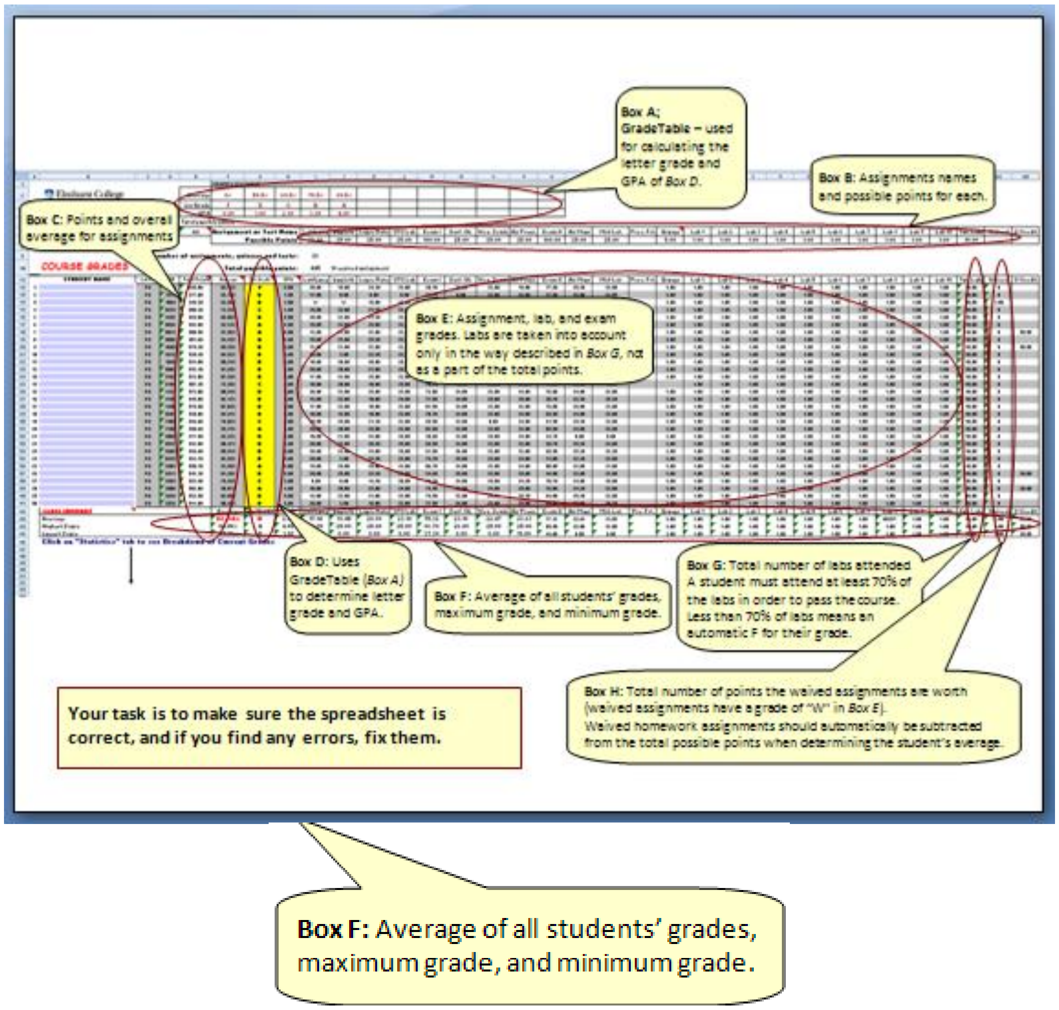

Box A; GradeTable – used for calculating the letter grade and GPA of *Box D*.
Box B: Assignments names and possible points for each.
Box C: Points and overall average for assignments
COURSE GRADES
Box E: Assignment, lab, and exam grades. Labs are taken into account only in the way described in *Box G*, not as a part of the total points.
Box D: Uses GradeTable (*Box A*) to determine letter grade and GPA.
Box F: Average of all students' grades, maximum grade, and minimum grade.
Box G: Total number of labs attended A student must attend at least 70% of the labs in order to pass the course. Less than 70% of labs means an automatic F for their grade.
Box H: Total number of points the waived assignments are worth (waived assignments have a grade of "W" in *Box E*). Waived homework assignments should automatically be subtracted from the total possible points when determining the student's average.
Your task is to make sure the spreadsheet is correct, and if you find any errors, fix them.
Box F: Average of all students' grades, maximum grade, and minimum grade.

Figure 24. (Top) A thumbnail of the description handout of the grade-book spreadsheet. (Bottom) Blowup of description Box F.

find any errors, fix them." We recorded participants' voice, facial expressions, and computer screen as they debugged, and also saved their final spreadsheet.

We seeded the spreadsheet with a total of ten varied real bugs harvested from the spreadsheets of [Beckwith et al. 2007]'s participants (Seattle-area adult spreadsheet users). These bugs were therefore realistic in terms of the kinds of bugs real end users create: six were inconsistent with other formulas in the same row or column (e.g., a missing student in a class total), three had been propagated by their original authors over the entire row or column (e.g., using the ">" operator instead of ">="), and one was not part of any group of similar formulas but was nevertheless wrong (it counted lab attendance as a part of the total points when it should not have). The number of bugs found and fixed by each of the most and least successful participants are listed in Table 16.

7.3.3. Spreadsheet Environment and Tutorial

The environment for the study was Microsoft Excel 2003. To make sure the participants were familiar with some of Excel's debugging features, we gave them a hands-on pre-session tutorial. The tutorial was only a tour of Excel's auditing features (arrows, tooltips, evaluate formula, error checking, and the watch window), and did not include any strategy hints. The participants were given time to practice using these features during the mock practice task and were allowed to use any Excel features they wanted during the actual task.

Table 16. Number of correct and incorrect bug finds and fixes by our most and least successful male and female participants. (The attempts are counted here. For example, the Successful Male made eight incorrect attempts to fix one bug.)

Participant	Bug Finds		Bug Fixes		Evaluations of Fixes
	Correct	Incorrect	Correct	Incorrect	
SF	9	0	6	0	5
SM	8	0	6	8	8
UF	3	0	1	7	2
UM	1	1	0	1	5

7.3.4. Analysis Methodology

Transcripts of the participants' think-aloud verbalizations were coded qualitatively in three passes: debugging phase, sensemaking step, and stratagem employed. One researcher first labeled the phase changes of all eight participants ("bug found", "bug fixed", and "evaluating fix"), since these were objective. The most and least successful female and male participants were chosen based on these codes. Two researchers then coded these four participants' transcripts with the additional two sets of codes.

For the strategy codes, we used the ten generalized End-User Debugging Strategy codes from [Grigoreanu et al. 2009], which are listed in Table 19. For the End-User Debugging Sensemaking codes, we used the code set derived in [Grigoreanu et al. 2009]. In that prior work, we employed eight of the nine sensemaking steps as codes plus two main loops: *external data sources* (this is the one we did not code for, since it referred to all of the information available to the users), *shoebox* (data that a participant deemed relevant enough to "touch" in the spreadsheet or study environment), *evidence file* (extracted from the shoebox, data that attracted a participant's interest enough for follow-up), *schema* (a structure or pattern a participant noticed as to how cells or information related), *hypothesis* (a tentative idea about how to fix a particular bug based on the participant's schema), *presentation* (the work product), *reevaluate* (making sure that a formula change was in fact correct after changing a formula), the tangential *Environment Loop* (a sensemaking loop for figuring out the environment), and the tangential *Common Sense* and *Domain Loops* (a sensemaking loop for answering questions about common-sense and domain questions). For more details about how we arrived at these codes, please refer to [Grigoreanu et al. 2009].

For the current study, we coded as follows: while referring back to both the participants' verbalizations and their screen captures for context, the coders first independently coded 20 minutes of the most successful female's transcript, and achieving 93% inter-rater reliability (calculated using the Jaccard index) on the Debugging Stratagems codes. Recall also that we had achieved an 84% agreement on the Sensemaking codes [Grigoreanu et al. 2009]. The two researchers then split up the remaining videos and coded them independently, since the level of agreement reached showed that the set of codes was complete and unambiguous for this data set.

7.4. Low-Level Strategy Items: Moves and Tactics

We first consider moves and tactics together in the design of end-user debugging tools. Recall that a *move* is the smallest unit of a strategic activity. In this work, we consider it simply using an environment feature. A *tactic* is the use of one or more moves with the purpose of more quickly and accurately finding or fixing a bug. Most commercial environments contain hundreds of features, so we will not list them here. Instead, we present some selected tactics employed by the participants and the moves used to implement them.

7.4.1. Some Well-Supported Tactics

Many tactics were well-supported in Excel; the participants' moves to implement those tactics were effective. In this section, when we use the term "effective" to describe moves, we mean that they were performed using features designed specifically for that tactic, rather than finding click-intensive workarounds (which we call "ineffective" because they cost users an unnecessarily high number of actions). Whenever we mention ineffective moves, we contrast them with effective ways of performing the same tactic.

Here are two examples of tactics which allowed us to identify moves that participants were able to use to good effect: "viewing dependencies" and "judging the correctness of an error checking warning."

Spreadsheets employ a dataflow-oriented execution model: each used cell has an output value and other cells can use that output value to produce further output values. It is therefore not surprising that Excel has several features to help users *view cell dependencies*. These include "Trace Precedents", "Trace Dependents", and the color-coded outlines of formula precedents attained by double-clicking a formula cell. All four participants used these features to effectively find cells' precedents and dependents. For example, Participant

SM traced the precedents of a cell which contained an error. The error had been propagated from a precedent, which the red arrows (usually blue) he brought up took him to. All four participants also often relied on the precedent arrows to help them parse complicated formulas.

Judging the correctness of an error checking warning is an important tactic to employ when using Excel's Error Checking functionality to find and fix bugs. Participant SF demonstrated how to do this in three distinct ways. First, she tried to understand why a cell was getting an "inconsistent formula" warning by scrolling up and down through the formulas that were supposed to be consistent using the arrow keys. The formulas looked consistent. Second, she traced the precedents for the cell with the warning and the precedents for cells it should have been consistent with; the dependencies were also consistent. By this point, she had decided that she should not follow the warning, and that the formula was indeed consistent, but was puzzled by the reason behind the warning. In response, the system was transparent enough to allow her to derive yet another correct answer. Participant SF noticed the "Copy Formula from Left" action button, which made her realize why the Excel inconsistency warning was flawed in this case where a row of consistent formulas intersected with a column of consistent formulas: "It says 'Copy from Left' instead of 'Copy from Above', so those formulas aren't inconsistent at all."

7.4.2. Improving Support for Tactics

While many moves were effectively employed to implement a tactic, sometimes, the tactics were good but the moves were not. Table 17 reveals several successful tactics employed by our participants, but which were implemented using an ineffective combination of moves. Software users have been reported in the past to often employ ineffective moves in performing computer tasks [Bhavnani et al. 2001].

Table 17. This is a sample of good tactics employed by our participants (SF, SM, UF, and UM), which were accomplished through a series of ineffective moves (thereby revealing usability improvements for Excel).

Tactic	Moves
1. Finding Formula Cells: Examining cells to determine which contained formulas.	[SF] Selected a range of cells and then tabbed through it to make sure that none of the selected cells contained formulas. [SF/SM/UM/UF] Using the arrow keys, and unbounded area, to do the same thing.
2. Finding Inconsistent Formulas: Figuring out which formulas do not match the pattern of the other formulas in the same area of the spreadsheet.	[SF/SM/UM/UF] Used the arrow keys to traverse what should have been consistent row or column cells, looking for inconsistencies by examining the changes in the formula bar. [SM/SF/UF] Brought up precedents arrows on two or more cells at a time to check that the formulas are consistent. [SF] Wanted to only verify cells with “Inconsistent Formula” warnings. It took her until the very end of the task, with Error Checking running in the foreground, to discover the Options button. At which point she immediately unchecked everything except for “cells containing formulas that result in an error” and “formulas inconsistent with other formulas in the region.”
3. Viewing Dependencies: Viewing a cell’s precedents or dependents to understand what feeds into the cell’s content and how it is used further.	[SF/UF/UM] Clicked on a cell to view its dependents. When the precedents or the dependents were on a different worksheet, the arrows pointed from or to a table icon, which none of the participants understood. In all of these situations, the participants would have navigated to the second worksheet. [SF/UM] Instead, only Participants SF and UM did so… And not until minute 32. Neither found the bug nested there. [SM/UF] Never switched to the second worksheet. [SM] An incorrect bug fix resulted in a circular reference to the first 23 rows of the spreadsheet, only denoted by a blue line below the 23rd row, since the other three borders were invisible. Participant SM stated, “Interesting line there.”

Table 18. (Continued)

4. Reusing Working Code: Following the example of already written code to fix a bug.	[SF/SM] Fixed inconsistent formulas by copying the correct formula over the inconsistent ones. [SM] Fixed the toughest bug by finding other formulas which were used in the spreadsheet, looking up further examples of how to use them, and trying to apply them in that context. [UF] Looked for examples in help that were kind of similar to what she wanted to do, but did not know what to search for.
5. Figuring Out What's Next: Looking to the environment for indications of what to try next.	[SF] Clicked on several buttons of the formula auditing bar to find something new to help her find and fix remaining bugs. She spent almost the entire time looking for inconsistent formulas, ignoring other types of bugs. And, she did not hover over the comments embedded in the spreadsheet until she thought she had run out of things to look at. [UF] Looked for examples in help to continue trying to figure out how to fix a bug she did not know how to fix.

The list of tactics and moves presented in this section is certainly not exhaustive, but it does provide useful low-level implications for design. How could Tactic 1, *finding formula cells*, be better supported in Excel? (See Table 17 for the ineffective moves employed by our participants.) There are better ways of finding formula cells in Excel. For example, the sequence of moves "Find & Select → Formulas" selects all formula cells in the worksheet, while "Find & Select → Constants" selects the remaining used cells. A different set of moves the participants could have performed would have been to toggle the "Show Formulas" option on, which would have switched the spreadsheet view to one which shows formulas in the cells as opposed to their calculated results. Both of these approaches have the same disadvantages: the features are hidden, and simply selecting the formula cells does not make them jump out.

Implication for Design 1: *Finding formula cells (or other code to be debugged) is an important tactic in the debugging process, which all four participants employed. Tools should provide an easily accessible button to highlight all applicable code in an obvious way.*

Researchers have started creating some tools to highlight formulas by surrounding formula cells with a special border. For example, some tools overlay a border around areas with consistent formulas to help visualize broken areas (e.g. [Sajianemi 2000]) and others color the border of formula cells for the purpose of reflecting the testing coverage achieved for that cell (e.g., [Burnett et al. 2003]). Our evaluation of moves and tactics in Excel supports such efforts, and even provides an advantage of broken areas with borders over triangle warnings: despite both being valid ways of visualizing inconsistencies, the former also makes obvious where the formulas are.

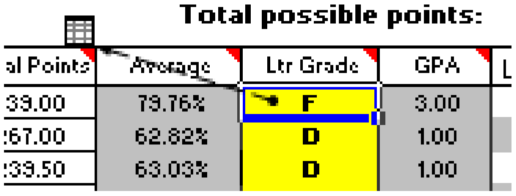
This takes us to Tactic 2, *finding inconsistent formulas*. Investigating the moves employed to accomplish this tactic by all four participants revealed some shortcomings of the error checking tool. The tactic of looking for inconsistent formulas would have been fruitful: six of the ten bugs were inconsistency bugs, and five of those were revealed by Excel. Unfortunately, our participants employed ineffective moves for accomplishing this tactic.

As Table 17 shows, one set of moves used by all four participants was to hit the arrow keys while watching the formula bar, looking for changes in the formula. One of several much more efficient set of moves to perform in Excel would have been to, while running Error Checking in the foreground, click the “Options…” button and uncheck everything except “formulas inconsistent with other formulas in the region.” Three of our four participants did not look for these more effective options and the fourth (Participant SF) only did so after clicking through almost all of the 202 error checking warnings provided by Excel by default on this spreadsheet. When she finally discovered how to ignore all but the “inconsistent formula” warnings, Participant SF contently exclaimed, “Oooh. You can pick the errors if you want to. And it only shows you the ones that are [inconsistent]… Oooooh!” This high false-positive rate might also have been the reason for how little participants employed the error checking tool. Of the 202 warnings, 197 (or 98% of them) were false-positives! Of the eight inconsistent formula warnings, however, more than half (five) were truly bugs in the formulas; a much better false-positive rate.

Implication for Design 2: *Too much feedback from the system about what might be (or lead to) an error may be as unaccommodating as having none. While automatic error checking algorithms are improving and detecting more and more types of suspicious code, default settings should rank the likelihood of each type of warning to reveal a bug, and only show the most likely warnings by default. Formatting and other warnings which are less likely to cause a bug should be made accessible by the user optionally.*

As described earlier, *viewing cell dependencies* was a well-supported tactic in Excel. However, as with the error checking tool, a closer examination of how dependency arrows were used for the purpose of implementing Tactic 3 also revealed possible improvements to the tool. As Table 17 describes, trouble arose when dependencies were between-worksheets. Showing inter-worksheet relationships is also a shortcoming of many research systems. In Excel, when a user brings up arrows for a cell which has precedents/dependents in a different worksheet, the cell points to a small table icon (see Figure 25). Three of the four participants encountered this icon, but none understood what it meant. Participant SF first hovered over it and then simply ignored the icon, stating "There's a little box, but I don't know what that means." Participant UF took the arrows down immediately. Participant UM, also first hovered over the table, perhaps expecting a tooltip. Next, he double-clicked the icon which only selected the cell behind it, followed by exclaiming, "Oh, oh!"

Furthermore, only half of the participants moved on to the second worksheet, late in the task (Participants SF and UM both first started examining formulas in the second



Total possible points:

al Points	Average	Ltr Grade	GPA	L
39.00	79.76%	F	3.00	
:67.00	62.82%	D	1.00	
:39.50	63.03%	D	1.00	

Figure 25. In Excel, when a cell's dependents are on a different worksheet, a table icon is displayed.

worksheet at minute 32, and neither found the bug in it). Thus, while arrows were very popular with all four of these participants, this detailed examination of the moves used for a particular tactic led to several observations for how viewing cell dependencies could be improved.

Implication for Design 3: *While intra-worksheet dataflow relationships are typically well-supported in spreadsheet environments, visualizations of inter-worksheet are not. In addition to being understandable, these tools should also allow users to easily navigate to those other worksheets.*

Several studies have found that end-user programmers *reuse working code* created by themselves and others (e.g., [Dorn et al. 2007; Bogart et al. 2008]). Three of our four participants attempted to employ code reuse to help them fix errors. Three examples of this include copying a consistent formula over an inconsistent one (Participants SF and SM), searching Excel Help for useful formulas to use in fixing a bug (Participants SM and UF), and looking for other formulas in the open spreadsheets which can be applied in the current situation (Participant SM). Just as Participant SM did here, inspecting code in other scripts also helped a successful female participant correctly fix a bug she had previously fixed incorrectly in the Windows PowerShell scripting environment [Grigoreanu et al. 2009], and this pattern has also been observed with professional developers [Ye and Fischer 2002].

Implication for Design 4: *Code reuse for the purpose of fixing bugs needs to be improved. In particular, searching for related code that is not in spatial proximity (as is the case with an inconsistent formula) is especially hard. One set of moves which has been used successfully to fix a bug is to recognize related code while skimming other files. End-user programming environments should facilitate this process by listing related tokens (such as, the types of formulas used in this spreadsheet, or the formulas used in any spreadsheet in a particular directory).*

Finally, the two female participants especially turned to the environment to *figure out what they can do next* (Tactic 5) to debug the spreadsheet, after they thought they had done everything they could. Examples of this tactic involved clicking features in the Audit

Toolbar and searching for Help. Unfortunately, this exploration time always came very late in the debugging process, when they were no longer very useful. For example, the first time Participant SF read a comment in one of the column titles with a description of what that formula was supposed to do was at minute 34! This was only after she believed she had run out of things to do, even though it would have been useful for her to do so while judging the accuracy of that formula (which actually contained an error she never found).

This tactic is highly related to search tools of the kind recently found to be needed by professional developers for finding appropriate knowledge, resources, and skills [Aranda and Venolia 2009; Venolia 2006]. This need to view related debugging information in the current context was also important to IT Professionals using PowerShell: for example, while inspecting code, the participants wanted to easily access incremental testing information in context (such as by hovering over a variable after running the code to a breakpoint) and, while examining error messages which resulted from running the script, participants wanted a shortcut to code related to those messages [Grigoreanu et al. 2009]. Another approach would be for the environment to generate a list of to-do items and to provide all of the relevant information by each item, solving both the problem of deciding what to do next and seeing what information can be checked next within that context.

Implications for Design 5-6: *The environment should recommend items to check. It should also provide relevant subtasks (or information which could help the user debug a particular part of the code) to the user on-the-fly in that context. Currently, the information is scattered about in menus and submenus, on paper, in emails, and in comments, among other places, and important details and resources relevant debugging are easy to overlook.*

This is only a small sample of the tactics employed by end-user programmers during debugging. Our formative work on end-user debugging strategies employed sequential pattern mining algorithms to look for common short (one to four) sequences of events performed by users, and we found 107 such frequent patterns [Grigoreanu et al. 2006]. Many of these common sequential patterns of events might reveal new debugging tactics. A more comprehensive set of end-user debugging tactics, like Bates' search tactics [Bates 1990], has yet to be compiled.

The findings presented in this subsection have implications for usability research. Even this small sample of seven tactics (two well-supported and five poorly-supported) helped us uncover several Excel usability strong points and problems, leading to detailed implications for the design of debugging tools. Furthermore, the resulting implications are likely to generalize to most other debugging environments, since most include at least one of the types of features addressed here (dataflow dependencies, error checking capabilities, areas of related code, multiple worksheets or files, reusable code, comments and other cell-specific information, etc.). Thus, examining the effective tactic a set of features was used toward proved to be a promising way to quickly improve even one of the oldest and most popular end-user programming environments.

7.5. High-Level Strategy Items: Stratagems and Strategies

A *stratagem* is a complex set of thoughts and/or actions, while a *strategy* is a plan for the entire task which may contain stratagems [Bates 1990]. In this subsection, we explore the debugging stratagems used by end-user programmers, the purposes for which the stratagems are used, and two strategies identified based on the participants' ways of traversing the sensemaking model.

7.5.1. Generalizing the Stratagems across Three Environments

Table 19 presents a set of ten stratagems [Grigoreanu et al. 2009] and their ties to gender and success. These stratagems previously generalized across two very different end-user programming environments and populations: students using the spreadsheet prototype Forms/3 and IT professionals using the commercial scripting environment Windows PowerShell [Grigoreanu et al. 2009]. Could we apply this same code set to a third environment?

Table 19. Past empirical work on end users' debugging stratagems with ties to gender. A * means statistically significant gender differences have been reported. A + means in-depth qualitative observations of the strategic activity.

Stratagems [Grigoreanu et al. 2006; Subrahmaniyan et al. 2008; Grigoreanu et al. 2009]		
Spreadsheet and Scripting	Dataflow	Following data dependencies. * Males preferred dataflow. * Males successful with dataflow. * Dataflow tied to males' success in finding bugs and evaluating their fix. +
	Testing	Trying out different values to evaluate the resulting values. * Males successful with testing. * Testing tied to males' success in finding, fixing bugs, and evaluating their fix. +
	Code Inspection	Examining code to determine its correctness. * Females successful with code inspection. * Code inspection tied to females' success in finding and fixing bugs. +
	Specification Checking	Comparing the description of what the program should do with the code. * Females successful with specification checking. *
	Feedback Following	Using system-generated feedback to guide debugging efforts. *
	To-do Listing	Indicating explicitly the suspiciousness of code (or lack of suspiciousness) as a way to keep track of which code needs further follow-up. * Females preferred to-do listing. *
	Fixing Code	Explicitly described strategy in terms of editing code to fix them. * Females unsuccessful with the "fixing code" strategy. *
	Spatial	Following the layout of the program in a spatial order. *
Scripting	Control Flow	Following the sequence in which instructions are executed. +
	Help	Accessing Help resources such as the software's built-in help, internet searches, or another person. +
	Proceed as in Prior Experience	Recognizing a situation (correctly or not) experienced before, and using that prior experience as a blueprint of next steps to take. +
Strategies [Grigoreanu et al. 2009; Grigoreanu et al. 2009]		
Scripting	Females and Males	Suggestive evidence that females' and males' success with a strategy differed with different debugging stages (finding a bug, fixing a bug, or evaluating a fix). +
Spreadsheet	Female	Comprehensive processing of the spreadsheet: traversing a spreadsheet systematically in order to look for bugs, only fixing the ones which do not derail her from her processing. +
	Male	Selective processing of the spreadsheet: systematically following up on the latest bit of relevant information collected, once a bug is found he stuck with it either until he fixed it or until a new bug was found. +

As the very high inter-rater reliability (93%) presented in Section 7.3.4 showed, the code set from the PowerShell study was both complete and unambiguous for this new data set. Despite having a wildcard “Other” code during the content analysis phase, we never had to use it: the code set was saturated. While the list of stratagems is likely to still be incomplete, this generalization of the codes across three environments (Microsoft Excel, Windows PowerShell, and Forms/3) and populations lends credence to their good coverage and unambiguous definitions.

Figure 26 shows the total minutes spent by the four participants using each stratagem. Notice that while the code set was saturated, two of the stratagems were absent: *control flow* and *to-do listing*. It is not surprising that *control flow* was absent as a stratagem, because there is no explicit control flow in spreadsheets; rather, execution sequence is derived by the system from dataflow dependencies. However, it is interesting to consider *control flow* one level up from the standpoint of what it is often used for, namely automating repetition. Participant SM wanted an effective tactic for repetition; his view was that he wanted to employ a “*for* loop” to fix one of the bugs, had it been available. Furthermore, expert users sometimes resort to additionally applying languages such as VBA [Visual Basic for Applications 2009] and its simplified version for Excel spBasic [Spreadsheet Basic (spBasic) 2007] as a control flow engine for spreadsheets. However, users who do not use such additional tools often have to resort to writing tediously lengthy formulas. For example, since Participant SM found no way of doing a *for* loop in Excel, he wrote a lengthy formula which considered each cell of a row individually. Peyton Jones et al.’s proposed user-defined functions that operate on matrices are one possible approach to introduce repetition automation in purely declarative environments [Peyton Jones et al. 2003].

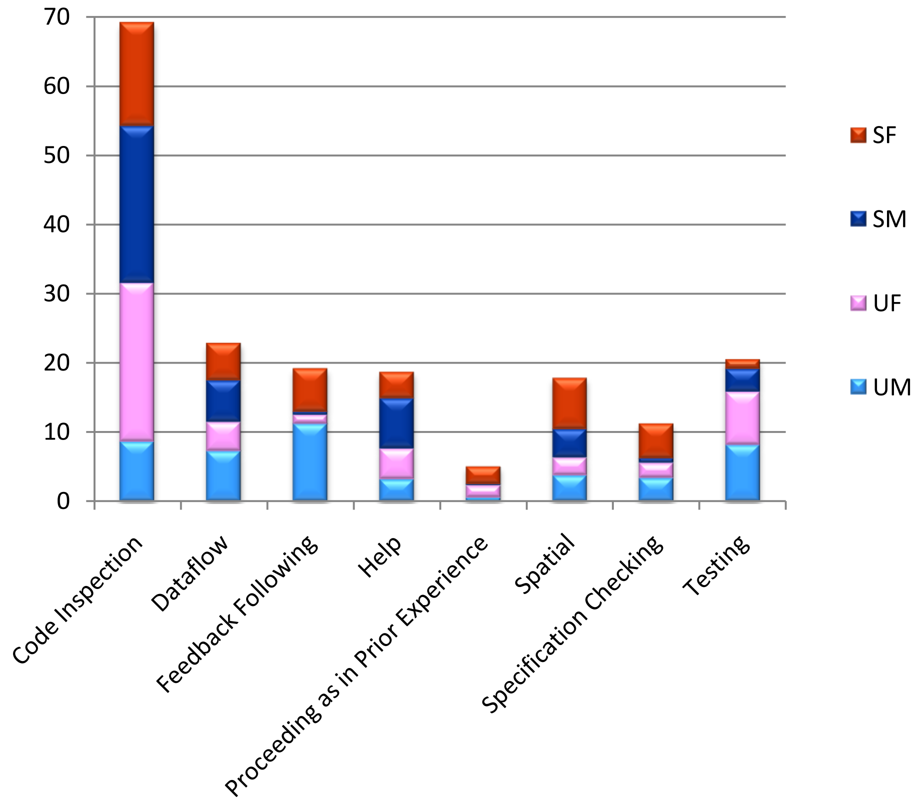


Figure 26. Total minutes the four participants allocated to each stratagem.

Implication for Design 7: *Even for declarative end-user programming environments, a feature for effectively implementing repetition is needed to reduce the tedium of creating repetitive code, without having to learn a new language and environment.*

The absence of the *to-do listing* stratagem in this study is harder to explain. This stratagem was used by participants in conjunction with the other stratagems to keep track of parts of the code that are “done” vs. “to-do” vs. “unchecked.” *To-do listing* was reported in both the Forms/3 spreadsheet environment [Subrahmaniyan et al. 2008] and in the PowerShell environment [Grigoreanu et al. 2009], and its use has also been investigated in professional developers’ work [Storey et al. 2008].

7.5.2. The Missing To-Do Listing Stratagem and Debugging Strategies

A possible reason for the lack of *to-do listing* in Excel might be the environment's lack of moves that would make the stratagem convenient. While neither the Forms/3 nor the Excel environment provides direct support for *to-do listing*, both contain features which can be repurposed for it. In Forms/3, participants repurposed a prevalent testing feature (a checkbox on each cell) to be a judgment about the formula's correctness, instead of the value's correctness as intended. In Excel, participants could have changed the background color of a cell or left a comment, for example, yet no participant did so.

There are several differences between the two spreadsheet tasks and the two environments which could account for this. First, the Excel spreadsheet was highly formatted, containing one blue column, one yellow column, four gray columns, 30 rows with alternating colors, three different font colors, 46 cells with bold fonts, five underlined fonts, many different font faces, and all borders delimiting spreadsheet regions. Thus, had formatting information been used to keep track of the to-do list (as is now even more encouraged with the new "Cell Styles" feature in Excel 2007), there would have been a big loss of formatting information in this spreadsheet. The Forms/3 spreadsheet did not contain any formatting. Thus, when the checkbox feature was used in Forms/3, there was no loss of formatting information when the border colors and/or background colors also changed. A second reason for this difference in participants' propensity to repurpose features for *to-do listing* might have been the size of the spreadsheet given. The Forms/3 spreadsheet only contained 13 formula cells to check off. In this study, our spreadsheet contained 288 formula cells, which makes the structure of the spreadsheet harder to parse in figuring out a list of items to-do.

Implication for Design 8: *Repurposing other features to support to-do listing is work-intensive for today's complex end-user programs. Instead, a tool is needed to directly support this stratagem by taking into account consistent areas of the code, while providing the flexibility of adding and removing to-do items, all without code formatting information loss.*

Just as it helped to examine moves within the context of tactics, we believe that, for the design of tools, it is also useful to observe stratagems within the context of strategies. How might the missing *to-do listing* stratagem have been useful if employed as a part of the two successful end-user debugging strategies?

Our earlier sensemaking work revealed two major strategies employed by end-user programmers during spreadsheet debugging: *comprehensive* and *selective* debugging [Grigoreanu et al. 2009] (see Table 20). These strategies and their gender ties were consistent with results on the comprehensive and selective information processing styles reported by the Selectivity Hypothesis [Meyers-Levy 1989]. While the *comprehensive* strategy held many advantages, such as finding and fixing the easier bugs early on, it also held disadvantages. Three of those disadvantages included: (1) sometimes overlooking less prominent formula cells by mistake, (2) forgetting about bugs found earlier in the task but not followed up on right away, and (3) sticking with the strategy even after it ceased to be successful for finding and fixing bugs. Support for *to-do listing* may help address all three of these downsides of the *comprehensive* strategy.

Implications for Design 9-10: *A to-do listing tool should be lightweight, automatically generating a list of to-do items directly from the code itself, to make sure none of the code is overlooked during comprehensive processing.*

Being able to mark complicated code for later follow-up might help comprehensive strategy users return to those bugs later on. This might also help comprehensive strategy users switch to other strategies when the latter has ceased to be successful.

A *to-do listing* tool might also help participants who prefer the *selective* strategy. For example, Participant SM, whose *selective* approach enabled him to fix the hardest bug, missed out on a lot of information that enabled Participant SF to spot and fix several bugs early using a *comprehensive* strategy. If end-user debugging tools could support lightweight switching among both strategies, doing so may encourage end-user programmers to make the switch when their approach becomes less productive.

Table 20. Qualitative empirical findings about the debugging strategies employed by our end-user programmers.

Scripting	Females and Males	Suggestive evidence that females' and males' success with a strategy differed with different debugging stages (finding a bug, fixing a bug, or evaluating a fix) [Grigoreanu et al. 2009].
Spreadsheet	SF	*Comprehensive* strategy: traversing a spreadsheet systematically in order to look for bugs, only fixing the ones which do not derail her from her processing [Grigoreanu et al. 2009].
	SM	*Selective* strategy: systematically following up on the latest bit of relevant information collected, once a bug is found he stuck with it either until he fixed it or until a new bug was found [Grigoreanu et al. 2009].
	SF and SM	*Consistency checking* strategy: Using all eight stratagems (especially specification checking, spatial, code inspection, and dataflow) together to make sure that bits of code which should be consistent are indeed so. This strategy was used for both *finding* and *fixing* bugs.
	SF	*Simple mapping* strategy: Combining all eight stratagems (especially feedback following, spatial, help, and code inspection) to work backward from error checking messages in *finding* bugs.
	SM	*Causal reasoning* strategy: Working backward from the code's output which seems faulty to *find* a bug (especially using testing and code inspection).
	SF and SM	*Compatibility checking* strategy: Strategy employed by our participants to *evaluate* their bug fixes (especially relying on testing, code inspection, and/or spatial).

Implication for Design 11: *To-do listing support might also help selective strategy users switch to other strategies when their approach has ceased to be successful for finding and fixing bugs. Furthermore, it will allow them to keep track of their selective progress. Thus, such a tool should be compatible with both the comprehensive and selective strategies.*

Just as with tactics and moves, our report of the two strategies might not be exhaustive. Researchers studying professional programmers' debugging have uncovered several strategies, some of which might also apply to end-user programmers. For example, Katz and Anderson classified strategies into two types: *forward reasoning* and *backward reasoning* [Katz and Anderson 1988]. Forward reasoning strategies are those where the participants' reflection starts from the code itself: *comprehension* (similar to our *comprehensive* strategy) and *hand simulation* (evaluating the code as if they were the computer). Backward reasoning strategies involved starting with a noticeable incorrect behavior of the code, and working backward to the part of the code which might have been the cause for the fault: *simple mapping* (proceeding from error message directly to the related code) and *causal reasoning* (searching backward from the program's output). We present some more strategies observed in this study in Section 7.6.1.

7.6. Stratagems in Context

Stratagems are a particularly promising strategy item to examine for the purpose of tool design: unlike with low-level *moves* and *tactics*, they are much more directly generalizable, and unlike *strategies*, they are low-level enough to lead to very specific implications. Thus, in this section, we further examine stratagem usage in two new contexts: by debugging phase (an overview of the debugging process) and by sensemaking step (a model for the overall problem-solving process).

7.6.1. Stratagems in the Context of Strategies by Debugging Phase

A stratagem was rarely used in isolation. In fact, a strategy is defined as being a series of stratagems used together. Thus, we expected examining patterns of stratagems used together to reveal additional end-user debugging strategies. Also, in earlier work, we observed that PowerShell scripters' success with a stratagem appeared to depend on the debugging phase at which it was being used [Grigoreanu et al. 2009]. Therefore, in this paper, we differentiate between strategies used for the different debugging phases (bug finding, bug fixing, and evaluating fixes).

How did our participants employ stratagems together successfully in the context of strategies at each phase? To answer this question, we narrowed our focus to only the successful sensemaking loop traversals (i.e., those ending in a correct find or fix) from the successful participants' transcripts. We then broke those successful traversals down further by debugging phase: a bug find, a bug fix, or the evaluation of a fix.

To find bugs, Participant SF primarily employed two systematic patterns, or strategies. For the first half of her task, she relied heavily on a pattern containing four intertwined stratagems: *specification checking* to understand what a part of the spreadsheet was supposed to do, *spatial* to examine areas containing similar cells, *code inspection* to run though the formulas in that spatial area and make sure the formulas are consistent, and *dataflow* to see how those cells depended on other cells in the spreadsheet. The following excerpt from her session exemplifies this strategy, which we called the *consistency checking* strategy:

SF: "It uses Box A [refers to an outlined region in the Specification handout] to determine the letter grade and GPA <note: specification checking>. So we can look at the GPA in column H, and we go down to see if there are any major differences between the formulas <note: spatial employment of code inspection>. And checking, their dependents, which would be the average, the highest, and the lowest scores… And precedents, which would be looking things up from the average and the GPA table <note: dataflow>."

In addition to heavily relying on these four stratagems in tandem, she also had seven instances of the other stratagems: *prior experience* in knowing that cells pertaining to a spatial region should be checked for consistency, *feedback following* in looking at an error checking warning, *help* in reading the warning associated with a suspicious cell, and *testing* in making judgments about values' correctness. Thus, she relied most heavily on four of the stratagems, but used all eight stratagems at some point during bug finding.

But Participant SF relied on a different pattern of stratagems in tandem during the second half of the session for bug finding: a core set of four stratagems plus a variable fifth stratagem. For the core set, she used *feedback following* and *spatial* to systematically walk through cells only paying attention to those which expressed the formula inconsistently with others in the area, *help* to understand what made those cells suspicious, and *code inspection* and *spatial* to judge whether or not the green triangle feedback should be ignored. We will call this strategy *simple mapping* since it is identical to Katz and Anderson's strategy for finding bugs based on the software's error messages [Katz and Anderson 1988]. This strategy also included a variable fifth stratagem from the set {*specification checking*, *dataflow*, *prior experience*, *testing*}. Thus, she ultimately used all eight stratagems in her second bug finding approach as well. The following quote comes from a part where *dataflow* was the variable stratagem in her pattern:

> *SF: "So, I ... that one's inconsistent. <note: feedback following and spatial> [Reads the warning] 'The formula in this cell differs from the formula in this area of the spreadsheet. Click on help with this error.' [The help is slow to open up] It sits there with a nice box. [Closes help box] That doesn't work. So renew, making sure they are all just unprotected formulas. 'Omits adjacent cells.' Which one omits? E6. [Inspects formula which also highlights precedents for which cells it should include] It doesn't need to include any adjacent cells. <note: code inspection and dataflow> 'Numbers stored as text,' 'unprotected formulas.' Moving all the way through, looking for something that says 'inconsistent formula.' <note: continues with feedback following and spatial>"*

Turning to the bug fixing phase, Participant SF fixed all six bugs during the first half of the task, using the *consistency checking* strategy.

Finally, in evaluating her fixes, she looked at cells' output values (*testing*) and/or formulas (*code inspection*), often comparing them to others in the area (*spatial*). We called this the *compatibility checking* strategy since it involves bringing in as much related information as possible and making sure that there are no discrepancies in their compatibility. For example, after using *spatial*, *code inspection*, and *dataflow* to find and fix a bug, Participant wanted to see a change in the output value, and she did:

> *SF: "The average changes. <note: Testing>"*

Participant SM found half the bugs based on those cells' output values not matching what the formula should give (i.e., using *testing* and *code inspection*), and half while trying to fix a different bug (through *spatial*, *code inspection*, and *dataflow*). His use of the latter pattern is identical to the female's *consistency checking* strategy. And, the former pattern is once again identical to one of Katz and Anderson's professional developers' debugging strategies: the *causal reasoning* strategy [Katz and Anderson 1988]. And, and here is an example of the former strategy:

SM: "Looking here right away I see a couple what look like problems. <note: testing> [Noticed a faulty output value of "F" in G12. He pauses and stares at G12's formula.] <note: code inspection> I'm trying to see here…I'm looking at the letter grade and average and it looks like the first one isn't associated correctly."

Participant SM fixed bugs using the *selective* strategy, sticking with the latest bug find until it was either fixed or until he found a new bug to follow up on. When the bugs he was trying to fix were inconsistency bugs, he also continued to use *spatial*, *code inspection*, and *dataflow* to fix them). For the other bugs, during successful bug fixing traversals, he relied most heavily on a combination of *code inspection*, *dataflow*, and *help*, with a bit of *specification checking* and *spatial*. Finally, he used the same stratagems as Participant SF for reevaluating bug fixes. For example, here is what he said and did as he fixed the hardest bug,

SM: "In the grading it says students must attend 70% of the labs in order to pass the course. <note: specification checking> [Asks the researcher a question:] Are, uh, is it possible to be waived from the labs? <note: help> [The researcher tells him to do the best he can with what he has been given.] Ok, I will not go with that, then. [Looks at formula he was editing] [Pauses and thinks] <note: code inspection> I'm supposed to assign a number to that, each one that is waived. <note: dataflow> So I know how many to look for. I've got to do that. Just got to find a way to make that true. [goes back to editing formula for AG 21] Ok, it mostly works for waive now. Lets test if there is a waive or not. [Looking at values] I tested all of them so… <note: testing>"

The quote above is a reminder that the *help* stratagem was not always about asking the software for help on features. Sometimes, *help* involved asking Google or other people. Also, when the participant received answers, they often led to assumptions on which participants based further actions: here, Participant SM assumed that labs could not be waived and fixed the bug with that assumption in mind. All of the end-user debugging strategies we have observed these successful participants employ are reiterated in Table 20.

Implications for Design 12-15: *This subsection has several new end-user debugging strategies for finding bugs (consistency checking, simple mapping, causal reasoning), fixing bugs (consistency checking), and evaluating those fixes (compatibility checking) to be supported by tools.*

End-user debugging tools should remain flexible enough to also support both of the strategies we identified earlier: comprehensive and selective debugging.

Some stratagems played a central role in each strategy, as pointed out in this section, and those groups of stratagems can be used as a starting point for the tool's design.

However, the other stratagems should also be supported at least in a secondary way in the context of each strategy this section revealed.

7.6.2. Stratagems in the Context of Sensemaking

Another approach to derive strategy-based implications for design is to look at strategy items in the context of sensemaking about a spreadsheet's correctness. This second approach allows us to look at the more immediate purpose for which to use particular stratagems.

Three Outlier Stratagems in the Context of Sensemaking. Recall from Figure 26 that, of the stratagems used to make sense about a spreadsheet's correctness, most were used for about 20 minutes total by the four participants together. However, three stratagems were different: code inspection, which was used extensively, and proceeding as in prior experience and specification checking, both of which were used very lightly.

In aggregate, our participants used *code inspection* about 70 minutes in total, more than three times as much as any other stratagem. Further, as Figure 27 shows, participants used *code inspection* for all six of the main loop's sensemaking steps as well as in both of the other two loops. *Code inspection* was popular in our previous studies as well (e.g., questionnaire answers revealed that *code inspection* and *testing* were used by 33% more participants than other stratagems in [Subrahmaniyan et al. 2008]), but this study's time-on-stratagem data shows that the extent of its popularity amounted to outright dominance.

There are at least three possible explanations. One of those is simply that *code inspection* is relatively ubiquitous because it complements other stratagems. For example, using *dataflow* order participants are likely to inspect formulas as they move from one cell to another. Another possibility is that the phenomenon may be particular to Excel. Excel is not set up for *code inspection*: users can see only one formula at a time unless they switch entirely out of the output value view. Perhaps the lack of support for *code inspection* in Excel

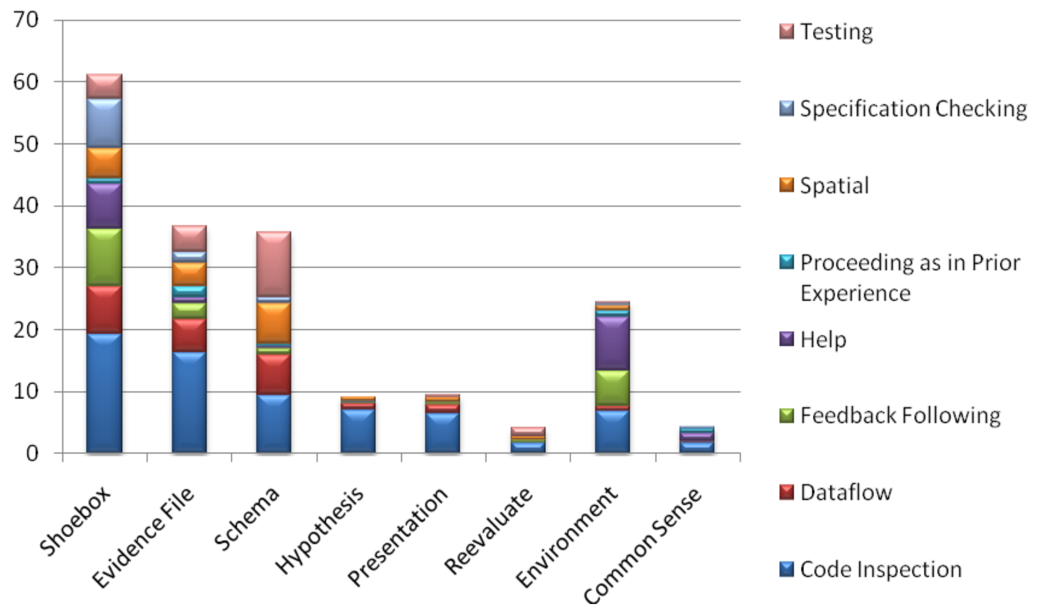


Figure 27. Total minutes spent using each stratagem at each step of the sensemaking model by the four participants.

simply required our participants to spend more time in it than they would have in other environments in order to make any progress. A third possible explanation is that *code inspection* may be the stratagem with which the participants were most familiar, and therefore tended to rely on the most. This would be consistent with a previous study, in which females reported that they relied on the familiar feature of formula editing because they thought other features would take too long to learn [Beckwith et al. 2005]. The findings lead to three opportunities for future research, implications 1-3 in Table 21.

Turning to the lightly used stratagems, the lack of *proceeding as in prior experience* (only about five minutes in total) may have been simply a matter of participants failing to verbalize. Having recognized how to proceed from prior experience, participants may have quickly moved ahead to execute their changes. This strategy was mostly verbalized during early sensemaking steps (*shoebox* and *evidence file*), and in making sense of the *environment*. The participants' use of *prior experience* while making sense of the *environment* is reminiscent of the needs for "closeness of mapping" (how closely the notation corresponds to the problem world) and "consistency" (after part of the notation has been learnt, this refers to how much of the rest can be guessed) in the design of

Table 21. Open questions for future research implied by Section 7.6.2's findings.

Implications for Future Research 1-3: - Why is code inspection such a dominant stratagem? - How does usage of code inspection differ by sensemaking step? - Should designers create one code inspection tool to support all sensemaking steps, or specialized tools for the different steps?
Implications for Future Research 4-5: - How can prior experience be supported during the early steps of sensemaking: in skimming information and in deciding which information to follow up on? - What are the ties between prior experience and the reuse tactic mentioned in Section 7.4.2.?
Implications for Future Research 6: - Would it be useful to support all eight stratagems as a part of the later sensemaking steps of hypothesis, presentation, and reevaluation?

environments: two of the cognitive dimensions of notation [Green and Petre 1994]. This led to several more opportunities for future research (see Table 21, implications 4-5).

Like *prior experience*, participants did little *specification checking*, only about ten minutes in total. This could be because, in order to do this, participants would have had to switch back and forth between the paper handout specification and the spreadsheet. Also, perhaps due to the specifications being on paper, *specification checking* was used mostly during the shoebox sensemaking step for debugging (see Figure 27).

Implication for Design 16: *End-user programming environments need to support specification checking in a lightweight way, especially for the shoebox "data collection" step. Specifications are one type of relevant information which can be tied to task items as described in Section 7.4.2.*

Stratagems during Foraging versus Sensemaking Steps. It is useful to note that the list of stratagems and their definitions, culminating in those presented here from [Grigoreanu et al. 2009], covered all of the sensemaking steps in this study (see Figure 27). The early steps of the Sensemaking model, which include *shoebox* and *evidence file* are a part of a sensemaking subloop called the "foraging subloop," while *schema*, *hypothesis, presentation,* and *reevaluation* happen during the "sensemaking subloop" [Pirolli and Card 2005]. The figure shows that, as participants progressed through the foraging subloop and the sensemaking subloop, the variety of stratagems used at each step also decreased: *shoebox* (8), *evidence file* (8), *schema* (8), *hypothesis* (6), *presentation* (6), *reevaluate* (4).

Implications for Design 17-18: *Information foraging tools for end-user debugging environments should support all debugging stratagems.*

Tools to help users organize their findings should also support all stratagems.

The dominance of *code inspection* during the *hypothesis* and *presentation* steps is surprising (see Figure 27); code inspection was used more than twice as much as all the other stratagems put together during those steps. It is unlikely that *code inspection* is the only reliable stratagem for hypothesizing about a bug fix and actually making the fix. One stratagem that can be used in Excel is to follow the environment's feedback about how to fix a bug: *feedback following*. For example, when a formula is inconsistent with the others in that column, or if it otherwise raises a red flag, Excel adds a small green Error Checking triangle to the top-left corner of that suspicious cell. It then gives the participant the option to fix the bug by clicking on the "copy from above" button, for example, to make it consistent with the rest of the column.

Recall that six of the ten bugs were inconsistency bugs, five of which Excel found. It is therefore surprising that *feedback following* only accounted for about 2% of the participants' *presentation* step, and less than 1% of their *hypothesis* step. While one possibility is that *feedback following* was used so little because it was efficient (it only took a button click, after all), so the total time could have been short despite its frequent usage, this was not the case. The participants had very low counts of *feedback following* during *hypothesis* and *presentation*. In fact, only Participant UM had any such instances at all, and his count was one for each. Thus, we believe the real reason for which *feedback following* was scarcely used, especially for those later sensemaking steps, was because of the high number of false-positives the error checking tool brought up by default. This leads to Implication for Future Research 6 in Table 21.

Stratagems used successfully during Schema and other Sensemaking Steps. While we have argued that all stratagems should be supported for the *shoebox, evidence file,* and *schema* sensemaking steps, an understanding of which stratagems the successful participants employed at each step can give tool designers an idea of which stratagems to make the focus of each tool. We first provide an example of how tool designers can use these findings to support the *schema* step, and then also list the most successful stratagems for the other steps.

The *schema* step was particularly troublesome for our two unsuccessful participants. After spending from half to two-thirds of their time gathering information, these unsuccessful participants got stuck in the *schema* step for lengthy periods of time, trying to organize it [Grigoreanu et al. 2009]. Successful participants, however, had faster transitions in and out of this sensemaking step. Thus, it appears that strategy support for the *schema* step may be especially useful in helping increase users' debugging success.

Implication for Design 19: *Supporting the schema step is particularly important, since unsuccessful participants tended to get stuck on creating a structure for the information they collected.*

How did the successful participants successfully create a structure by organizing the information they collected? To answer this question, we narrowed in on the stratagems used during a successful debugging session, as defined by leading to a correct bug find or fix. The following four stratagems each accounted for at least 10% of successful schematizing: *code inspection* (38% of *schema* time), *spatial* (29%), *testing* (14%), and *dataflow* (10%). While these four stratagems were most common during the Schema creation sensemaking step, the remaining stratagems were also used slightly, and therefore should be supported in a secondary way.

Implications for Design 20-21: *Based on these findings, a tool for supporting the schema sensemaking step should primarily focus on revealing the ties between the code items (e.g., spreadsheet's spatial layout and their underlying cell dataflow dependencies), as well as displaying both the code itself (code inspection) and output values (testing) in the context of that structure.*

If possible, the tool should also display the information revealed by the others stratagems (e.g., specifications) in the context of the main four stratagems.

Once again, while these findings come from examining a particular population in a specific environment, they are general enough to be applicable to any debugging environment. For example, Littman found that professional programmers also use similar stratagems for systematically comprehending programs: they try to understand the dependencies (*dataflow* / *control flow*) of multiple modules (*code inspection* / *spatial*) to make sense of them (create a *schema*) [Littman et al. 1986]. The one main schematizing stratagem Littman does not mention is *testing*. However, based on the popularity of hovering over variables to see their values during *control flow* (break point) debugging for both professional programmers and IT professionals [Grigoreanu et al. 2009], we believe output values should also be visible alongside the executed code in order to help users quickly organize the information they have collected.

Another clue to the importance of supporting these stratagems for the purpose of *schema* creation has been the research direction of tools for visualizing hidden spreadsheet structures. These have focused particularly on visualizations for spreadsheets' *spatial* and/or *dataflow* structures, including tools which highlight spreadsheet areas that seem to create an entity for visualizing broken areas (e.g. [Sajianemi 2000]), tools for visualizing and/or animating dataflow dependencies (e.g., [Igarashi et al. 1998]), and even visualizations which show the dependencies in 3D layers [Shiozawa et al. 1999]. None of these tools support all four (let alone all ten) of the stratagems in conjunction and, unfortunately, these studies also do not report empirical data on users' success in comprehending the spreadsheet or in debugging it. We thus do not know how successful these tools are at helping users create a *schema*. However, our findings reinforce the need for such tools.

The *schema* step, while it seemed to be the most problematic for our unsuccessful participants, is of course not the only sensemaking step which needs to be supported through strategy-based tools. For all the rest of the sensemaking steps (e.g., how to successfully find new information, generate correct hypotheses about bug fixes, and evaluate a fix), Table 22 shows the stratagems employed successfully by the two successful participants. As before, our threshold for inclusion in the table was that the stratagem needed to be used at least 10% of the time spent in a sensemaking step.

Table 22. Participants SF's and SM's most common stratagems for each sensemaking step. The amount of minutes spent with each stratagem is in parentheses. Bold: Appeared in both participants' top 10%. Italics: This stratagem was also used by the other participant at this step, though for less than 10% of the time. Plaintext: The other participant did not use this stratagem at all at this step.

Shoebox	SF: **Code Inspection** (5.9), Specification Checking (3.6), **Dataflow** (2.0), Feedback Following (1.6) SM: **Dataflow** (3.4), *Help* (2.4), **Code Inspection** (2.2)
Evidence File	SF: **Code Inspection** (2.7), **Dataflow** (2.1), **Spatial** (1.5) SM: **Code Inspection** (6.0), **Dataflow** (1.2) , **Spatial** (1.0), Help (1.0)
Schema	SF: **Code Inspection** (2.9), **Spatial** (3.1) SM: **Code Inspection** (2.8), *Testing* (1.5), **Spatial** (1.3), *Dataflow* (1.1)
Hypothesis	SF: **Code Inspection** (0.6), *Spatial* (0.1), *Dataflow* (0.1) SM: **Code Inspection** (4.4)
Presentation	SF: **Code Inspection** (0.3), *Spatial* (0.3), *Dataflow* (0.1) SM: **Code Inspection** (4.5)
Reevaluate	SF: **Testing** (0.4), **Spatial** (0.2), **Code Inspection** (0.1) SM: **Code Inspection** (0.9), **Testing** (0.7), **Spatial** (0.2)
Environment	SF: *Feedback Following* (3.5), **Help** (3.2), **Code Inspection** (1.8) SM: **Help** (3.0), **Code Inspection** (1.8)
Common Sense	SF: Prior Experience (0.8), **Code Inspection** (0.7), Dataflow (0.2) SM: Help (0.5), Specification Checking (0.3), **Code Inspection** (0.3)

7.7. Conclusions, Discussion, and Future Work

This paper is the first to present a comprehensive strategy-based approach for analyzing the usability of end-user programming environments. In particular, we analyzed empirical strategy data at four levels and in two contexts.

Overall, our work revealed the importance of this approach: each level led to a different type of implication for design. Analysis of the data at the lower levels (moves and tactics) led to somewhat more incremental changes to existing features, whereas examining the higher levels (stratagems and strategies) led to implications for the design of tools which do not yet exist but should, and also revealed several new opportunities for future research. The two contexts (debugging phase and sensemaking step) helped us understand the purpose for which a strategy item was employed. These two dimensions were also important to examine since a stratagem might currently be useful in one context (e.g., *feedback following* for *bug finding*) but not another (e.g., *feedback following* for *bug fixing*).

This comprehensive understanding of strategy items led to 21 new implications for the design of end-user debugging tools and to six implications for future research. While these implications can be addressed individually, we believe that they should all be taken into account in the design of any one tool, since many of them are complementary.

For example, let us consider how all the implications can be taken into account in the design of a to-do listing tool for end-user debugging environments. Implications for design 5-6 and 8-11 would be central to the design of a tool to directly support this stratagem, since they are all about the missing *to-do listing* stratagem in this study. These implications reveal the importance of a tool which automatically generates a list of consolidated (by some measure of similarity, such as consistent formulas) to-do items. The elementary functionality of this tool would be to keep track of which items are unchecked vs. checked vs. marked for later follow-up, without losing the original cell formatting. In the context of each item, additional information should be provided either as editable data or as subtasks for that particular task item. The tool should not force a particular order in visiting unchecked items, since it should be flexible enough to support both *comprehensive* and *selective* debugging strategies.

The remaining implications, while more secondary to the design of this particular tool, provide additional insights into it. The tool could superimpose the status of to-do items onto the spreadsheet, highlighting areas of similar formulas and their status (implication 1). Certain items which have a high likelihood of containing an error, such as an inconsistent formula, could be highlighted in the list (implication 2). An item could be any piece of code executable by the environment (implication 7), and related information about each item (implications 15-18) should be displayed, especially spatial and dataflow dependencies to help users organize the data they collect (implications 19-21). Furthermore, the tool should facilitate understanding of inter-worksheet dataflow relationship ties and help navigating them (implication 3). It should be flexible enough to not render useless any of the successful overall strategies the user might want to employ (implications 12-14). *Code inspection* for the purpose of viewing related code to help fix bugs could be taken into account by displaying "related formulas" (for example, ranked by physical proximity or syntactic similarity) within the context of each task item (implication 4).

This is an example of how all 21 implications for design can be supported in the creation of a to-do listing tool. However, if the user were planning on building a better *code inspection* tool or *feedback following* tool instead, a different facet of these same implications could be considered in *its* design.

Any empirical study also has limitations. Two such limitations result from having picked a particular environment, population, and task. First, the choice of spreadsheet used and inserted bugs could have led to lower internal validity, since they might have affected the users' strategies. To lower this risk, we harvested bugs created by real end-user programmers working on this same spreadsheet. Second, our participants might not be representative of the overall end-user programming population, since we only analyzed four participants' data in detail and since the study was conducted in only one end-user programming environment, Excel. To help address this external validity problem, we have discussed the ties between our results and others' work from different environments in our results sections. Thanks to that triangulating evidence, we believe that both the analysis methods employed here and the implications for design generalize across debugging environments.

Finally, this comprehensive exploration of strategy usage revealed several opportunities for further research. First, we will build a tool which addresses some of the implications for design listed here. The tool's evaluation will allow us to measure the effectiveness of these implications in helping end-user programmers debug spreadsheets. Other implications for further research involve a better understanding of the very popular *code inspection* stratagem in different contexts, of the ties between the *prior experience* stratagem and reuse, and of how to more directly support *hypothesis creation*, *presentation*, and *reevaluation* in end-user programming environments.

This strategy-based approach to usability has revealed many improvements to even one of the oldest and most popular end-user programming environments: Microsoft Excel. For future studies, applying this same approach and a different experiment setup in studying the usability of other end-user programming environments (e.g., a field study with Windows PowerShell) may reveal new implications for the design of strategy-based end-user debugging tools.

A Strategy-Centric Approach to the Design of End-User Debugging Tools

Valentina Grigoreanu, Margaret Burnett, and George Robertson

CHI'10 (Atlanta, GA, USA: April 10 - 15, 2010)

Association for Computing Machinery, Inc., New York, NY.

Issue manuscript will appear in the *Proceedings of the ACM Conference on Human Factors in Computing Systems* (this is a pre-print).

8.

The STRATCEL TOOL STUDY: A Strategy-Based Tool for Excel

8.1. Abstract

End-user programmers' code is notoriously buggy. This problem is amplified by the increasing complexity of end users' programs. To help end users catch errors early and reliably, we employ a novel approach for the design of end-user debugging tools: a focus on supporting end users' effective *debugging strategies*. This paper has two core contributions. We first demonstrate the potential of a strategy-centric approach to tool design by presenting *StratCel*, a strategy-based tool for Excel. Second, we show the benefits of this design approach: participants using StratCel found *twice* as many bugs as participants using standard Excel, they fixed *four* times as many bugs, and all this in only a small fraction of the time. Furthermore, this strategy-based approach helped the participants who needed it the most: boosting novices' debugging performance near experienced participants' improved levels. Finally, we reveal several opportunities for future research about strategy-based debugging tools.

8.2. Introduction and Related Work

End-user programmers are people who program, not as an end in itself, but as a means to more quickly accomplish their tasks or hobbies [Nardi 1993]. For example, an accountant creating a budget spreadsheet would fit this description. Many studies have found end-user programmers' code to be rife with errors (e.g., [Panko and Orday 2005]) and the negative consequences of these errors have been reflected in numerous news stories, many of which are recounted at the EuSpRIG site [EuSpRIG 2006]. One recent example that received media attention came following Lehman Brothers' collapse. Barclays Capital agreed to purchase some of Lehman's assets but, due to a spreadsheet error resulting from hidden cells, the company purchased assets for millions of dollars more than they had intended [Hayes 2008]. A few weeks later, Barclays filed a motion in court asking for relief due to the mistake.

The impact of end-user programming errors like the Lehman-Barclays example is amplified by the quickly increasing complexity of end-user programs and by the large number of end-user programmers. The complexity of corporations' spreadsheets doubles in both size and formula content every three years [Whittaker 1999]. In addition, there are tens of millions more end-user programmers than there are professional programmers [Scaffidi et al. 2005].

In response to this problem, end-user software engineering research has begun to emerge in the spreadsheet realm and in many other areas. Debatably, the first step in this direction was taken by Backus' team when it designed Fortran in 1954 [Backus 1998]. Other examples include teaching kids to create programs (e.g., [Cypher and Smith 1995; Kelleher et al. 2007]), programming for and over the web (e.g., [Kandogan et al. 2005; Rosson et al. 2007]), uncovering learning barriers [Ko et al. 2004], and even programming household appliances [Rode et al. 2004].

Of particular relevance to this paper are research spreadsheet debugging tools. The hidden structure of spreadsheets is an end-user debugging pain point [Nardi and Miller 1990] and tools such as Davis' overlaid arrows [Davis 1996], Shiozawa et al.'s dependencies in 3D [Shiozawa et al. 1999], and Igarashi et al.'s animated dataflow visualizations [Igarashi et al. 1998] have sought to address it. Tools which visualize broken areas (e.g., [Sajianemi 2000]) also aim to make the spreadsheet structure more transparent. Some debugging tools improve the automatic detection of errors (e.g., Abraham and Erwig's UCheck system [Abraham and Erwig 2007]). Others empower the user to systematically test their spreadsheets using the What You See Is What You Test (WYSIWYT) testing methodology [Burnett et al. 2004].

However, we believe that a critical stone has been left unturned in the design of spreadsheet debugging tools: how tools can be designed to directly support end-user programmers' existing *debugging strategies* (users' plans of action for accomplishing a task). Building upon a recent comprehensive overview of Excel users' debugging strategies [Grigoreanu et al. 2009], this approach led to the following main contributions:

- A novel empirically-based end-user debugging tool, StratCel, created to support end-user programmers' specific debugging strategy needs.
- A positive impact on end-user debugging success: (1) twice as many bugs found by participants using StratCel compared to Excel alone, (2) four times as many bugs fixed, (3) in a fraction of the time, (4) including two bugs which both the researchers and Control group had overlooked, and (5) a closing gap in success based on individual differences.
- Participants' promising comments about StratCel's usability and its applicability to their personal projects and experiences.
- Design guidelines, based on instantiated and validated empirically-based implications for design.
- Lastly, we argue for the generalizability of this approach and list several opportunities for future research.

8.3. StratCel's Empirically-Based Design

In this section, we address the question of whether a strategy-centric approach in the design of end-user debugging tools is practical and, if so, how it can be achieved. Toward this end, we report our experience building StratCel: an add-in for the popular end-user programming environment Microsoft Excel.

In the first subsection, we provide a quick overview of the iterative approach and methods we employed in StratCel's design. In the latter subsections, we then list several candidate design guidelines from a study which reveals a comprehensive overview of Excel users' debugging strategies [Grigoreanu et al. 2009]. We also detail how we employed these candidate guidelines in our design of StratCel to see which would prove effective: we later evaluate these.

Each time we refer to a candidate design implication *from that earlier study*, we format it as follows:

Candidate 0: This is an example implication from [Grigoreanu et al. 2009].

The implications for design revealed by the earlier study fell under three categories (hence the three subsections), based on the level of strategy from which they came: (1) a *strategy* is the user's approach for the entire task, which (2) one or more *stratagems* can be used in combination to achieve, and which are in turn made up of (3) clusters of low-level *moves* with a purpose (i.e., *tactics*) [Bates 1990]. For the remainder of this paper, we will use these more specific definitions of the four strategy levels.

8.4. Iterative Approach

As Schön points out, prototyping activities are important to any tool-building endeavor, since they encourage reflection on the tool's design [Schon 1983]. We first defined the tool's scope using empirical work about end-user debugging strategies, a scenario, a storyboard, and sample real users from our target population. The sample users were real participants in a previous spreadsheet study [Grigoreanu et al. 2009]. For example, the most successful female was in her twenties and had worked as an auditor for the past two years, building large and complex spreadsheets to check clients' paperwork (e.g., bank statements and personal records). As a Business major, she also used spreadsheets in her classes and her personal life, and had programmed in VB.NET for one class. Continuing with an iterative approach, we cycled dozens of times through design, implementation, testing, integration, maintenance, and usability evaluation. To guide our iterations, we continued with the earlier methods and also added walkthroughs with a paper prototype, walkthroughs of the tool itself, and sandbox pilot sessions.

8.5. The Design Impact of Strategies and To-Do Listing

Implications for design based on the overall *strategies* can help us frame the functionality of the debugging tool as a whole, because strategies are followed by the user throughout the entire task.

Candidate 1: Supporting both comprehensive (getting an overall understanding of the spreadsheet by visiting cells in a systematic order) and selective (following up on the most relevant clues as they come along) debugging strategies by:

- Helping comprehensive users keep track of cells they want to return to later on.

- Highlighting which cells selective users have looked at versus those they might have skipped.

In other words, support for the *to-do listing* stratagem (or "a user's explicit indication of the suspiciousness of code, or lack thereof" [Grigoreanu et al. 2009]) may help reduce the cognitive load of both comprehensive and selective users by helping them keep track of items they need to look at in the future. Table 23 summarizes empirical findings from seven studies encouraging support for *to-do listing*. Note that, since both of these strategies needed to be supported, StratCel does not impose an order in which to proceed through to-do items or their related information.

Candidate 2: Provide explicit support for to-do listing.

Candidate 3: Automatically generate list of items to check.

To address these implications for design, the core functionality of StratCel involves automatically generating a list of to-do items and providing actions related to managing a task list, such as setting the item's status and priority (see Figure 28). Each item in the list is a range of consistent formulas automatically consolidated into one item. Using the tool, the user can change the status of each to-do item. Item status can be: (1) unchecked, meaning that the user has not yet made a decision about whether that item was completed, (2)

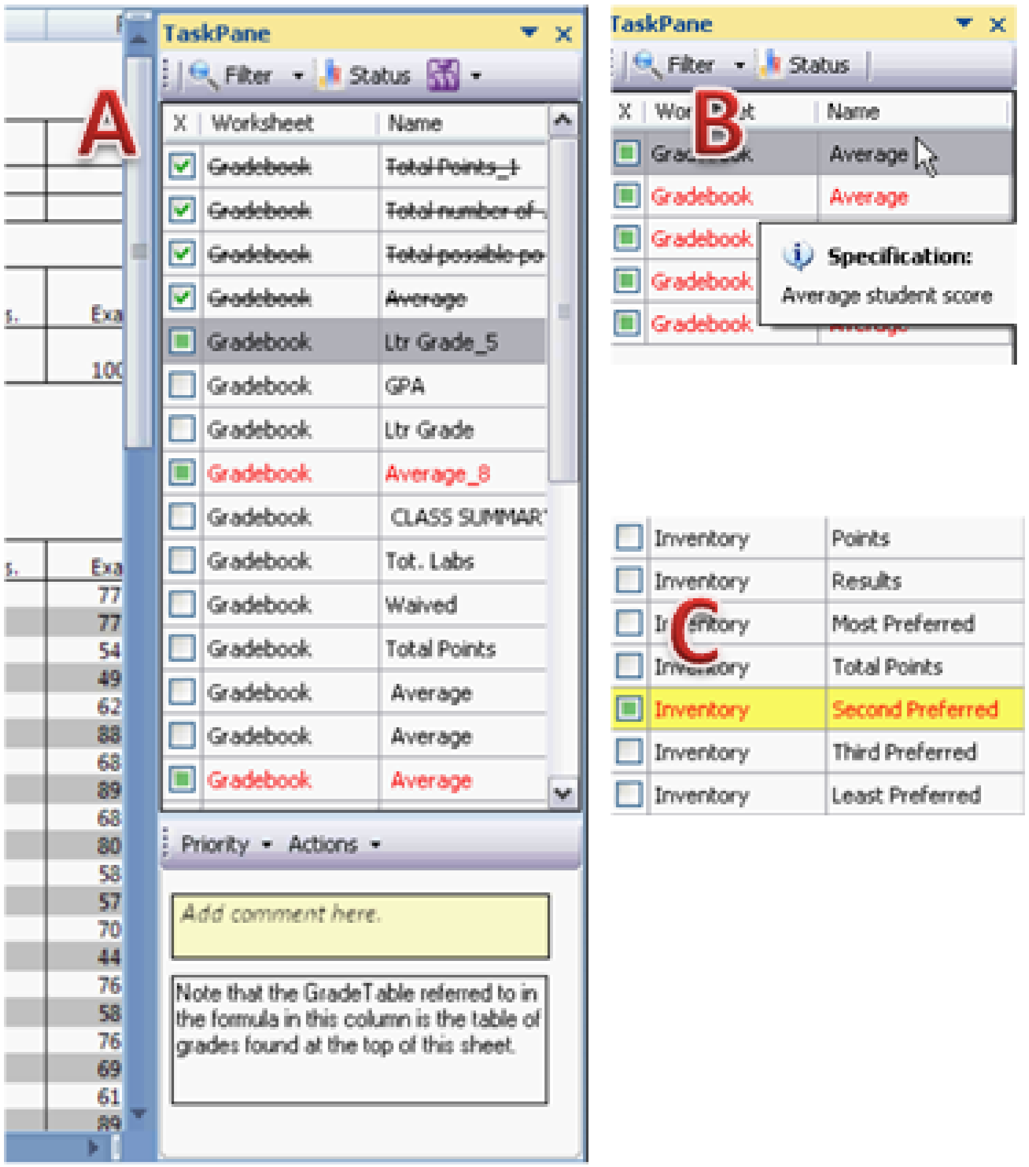
Table 23. Summary of empirical findings about the need to support to-do listing in debugging environments. See Figure 34 in the Appendix for a full version of this table.

Finding	Evidence
To-do listing is an end-user debugging stratagem.	Used breakpoints, open-close files, paper [Grigoreanu et al. 2009] and "*…checks and X's to show me what I'd already checked*" [Subrahmaniyan et al. 2008].
To-do listing is poorly supported in debugging tools.	PowerShell, Forms/3, and Excel: No explicit support for *to-do listing* [Subrahmaniyan et al. 2008; Grigoreanu et al. 2009; Grigoreanu et al. 2009].
Requests for to-do listing support transcend individual differences.	Males and females using Forms/3 [Subrahmaniyan et al. 2008], PowerShell [Grigoreanu et al. 2009], and even integrated development environments want *to-do listing* support [Storey et al. 2008].
Danger: Relying on existing features to be repurposed.	Misuse of the features can lead to incorrect feedback from tools [Phalgune et al. 2005], a loss of formatting information, or simply be ineffective. Perhaps why no participants from [Grigoreanu et al. 2009] employed it in Excel.
Benefit: Shows promise in increasing debugging success.	Often used in conjunction with *code inspection,* a female success stratagem [Subrahmaniyan et al. 2008; Grigoreanu et al. 2009]. May remind comprehensive Participant SF about cells she found suspicious and selective Participant SM about cells he had skipped over [Grigoreanu et al. 2009].

checked, meaning that the user has verified that item and decided s/he does not need to return to it, and (3) to-do, meaning that the user would like to return to that item later on.

This explicit support for *to-do listing* helps guard against users having to use costly workarounds which change the spreadsheet's existing formatting. While the "automatic generation" implication seems to suggest that users would have less flexibility in creating their own to-do lists, storyboards and expert walkthroughs with the prototype backed the need for this implication.

Candidate 4: Provide relevant information in the context of each to-do item.



X	Worksheet	Name
✓	~~Gradebook~~	~~Total Points_1~~
✓	~~Gradebook~~	~~Total number of~~
✓	~~Gradebook~~	~~Total possible po~~
✓	~~Gradebook~~	~~Average~~
🟩	Gradebook	Ltr Grade_5
☐	Gradebook	GPA
☐	Gradebook	Ltr Grade
🟩	Gradebook	Average_8
☐	Gradebook	CLASS SUMMAR
☐	Gradebook	Tot. Labs
☐	Gradebook	Waived
☐	Gradebook	Total Points
☐	Gradebook	Average
☐	Gradebook	Average
🟩	Gradebook	Average

Add comment here.

Note that the GradeTable referred to in the formula in this column is the table of grades found at the top of this sheet.

X	Worksheet	Name
🟩	Gradebook	Average
🟩	Gradebook	Average
🟩	Gradebook	
🟩	Gradebook	
🟩	Gradebook	

Specification:
Average student score

☐	Inventory	Points
☐	Inventory	Results
☐	Inventory	Most Preferred
☐	Inventory	Total Points
🟩	Inventory	Second Preferred
☐	Inventory	Third Preferred
☐	Inventory	Least Preferred

Figure 28. (a) The to-do list task pane is automatically populated with consolidated items and their properties (e.g., worksheet name, a default item name). The user can mark each item as "done" (e.g., Total Points_1), "unchecked" (e.g., GPA), or "to-do" (e.g., Average_8). Other basic to-do list management capabilities include adding a comment, (b) filtering on a status, and (c) assigning priorities to items (the darker the yellow, the higher the priority).

StratCel also automatically reports information about each item to help the user identify it, including: the worksheet name, an automatically generated name (from headers), a description pulled from cell comments, the item's priority, and the item's spreadsheet address. Following walkthroughs and sandbox pilots, we decided that the priority could be encoded in a color instead of having its own field in the list (see Figure 28c).

One important implication followed by other end-user debugging tools has been to directly overlay or tie hidden information about the structure of the spreadsheet to the spreadsheet itself (e.g., [Sajianemi 2000]). Therefore, in StratCel, we synchronized cell selection and to-do item selection: selecting an item in the list also highlights the cells to which that item refers, and vice-versa.

8.6. The Design Impact of Stratagems

While *strategies* cover the entire task from start to finish, the debugging tool has multiple smaller components which further help make sure the task is accomplished accurately and quickly. For example, let us say that the first to-do item is about cell A1. Subtasks for checking off that particular item may include: examining the formula to make sure it matches the specification, testing different conditions and making sure the output is right for them, getting help when stuck, etc. These smaller components which allow users to act upon a unit of the to-do list are based on implications for design about end-user debugging *stratagems* (e.g., *code inspection, specification checking*, *testing*, and *help* are stratagems referred to in the previous sentence).

Candidate 5: Providing information about the nine remaining stratagems in the context of each to-do item.

Researchers have so far observed ten end-user debugging stratagems: code inspection, control flow, dataflow, error checking, help, prior experience, spatial, specification checking, testing, and to-do listing.

We have already addressed how to-do listing can be explicitly supported in order to facilitate the use of the comprehensive and selective debugging strategies. And control flow is the only stratagem which StratCel does not support. Even though Excel's language is declarative, there is poor support for implementing repetition. How StratCel can better support this remains to be determined.

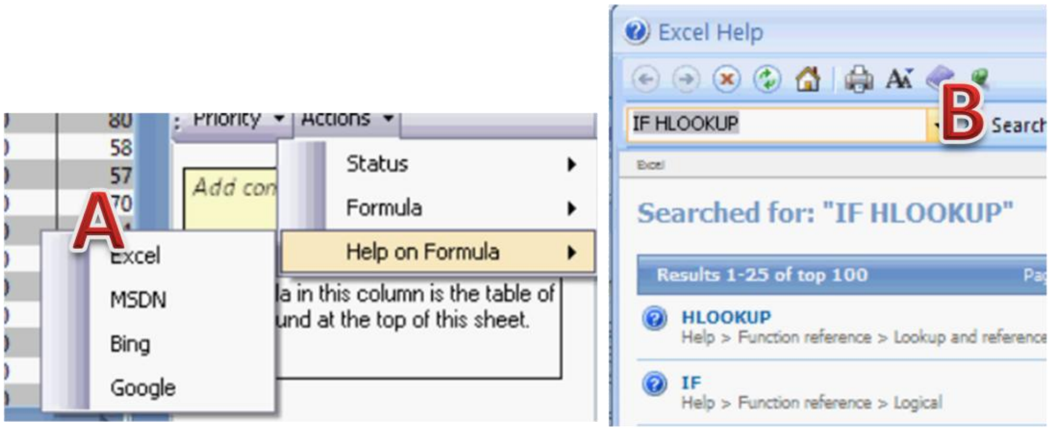
The remaining eight stratagems are all supported in the context of each to-do item: each provides additional information about the item. For example, selecting an item in the to-do list also selects it in the spreadsheet. This displays a representative formula in the formula bar (code inspection) and highlights its value(s) in the spreadsheet (testing). Also related to formulas is the following:

Candidate 6: An easy way of accessing formulas related to the current code may help users fix more bugs through reuse.

To access more information related to the content of a formula, StratCel provides a "Help on Formula" feature to search several databases for information related to it (the *help* stratagem). Figure 29 shows the search result when looking up a formula containing both the 'IF' and 'HLOOKUP' functions in the Excel documentation (and three other information sources are also available). Another type of search which could be added to this list in the future is a search of Excel documents in a user-defined directory. This helps the user access the collective *prior experience*. Keeping track of done versus to-do items might help organize *prior experience,* while Excel's recently used formulas feature may highlight relevant formulas.

Candidate 7: Perfect viewing spatial and dataflow relationships to help users organize collected data.

Four stratagems remain to be addressed: *dataflow, error checking, spatial*, and *specification checking*. A directed graph shows the *dataflow* dependencies between task items (see Figure 30). The graph can also be used to navigate the items; hovering over the items in the graph selects the related item in the task list as well as the related cell(s) in the spreadsheet. Since consistent formulas are highlighted as the graph is navigated, this also



la in this column is the table of
und at the top of this sheet.

Figure 29. (a) "Help on Formula" gives the user several options (Excel Help, MSDN, Bing, and Google) in which to search key terms from the formula. (b) For example, if the formula looks like this: =IF(E17<>"",HLOOKUP(E17,GradeTable,2),"") **, then selecting Excel looks up "IF LOOKUP" in Excel's documentation. The same thing would happen with the other search engines.**

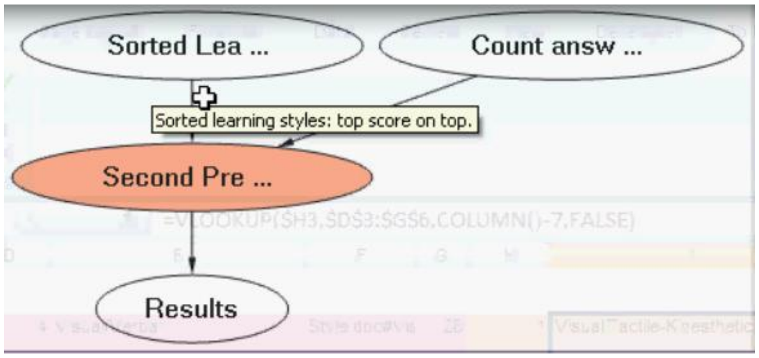


Figure 30. Dependencies between to-do items (recall that their names are automatically generated and can be modified) are displayed in a directed graph: StratCel visualizes both within and between worksheet transitions.

reveals the *dataflow* dependencies between *spatial* areas of consistent formulas. *Spatial* relationships can also be deduced from status borders: users can bring up borders around to-do items by clicking on a button in the ribbon. Areas of unchecked formulas are blue. To-do items are red. And items marked as "checked" have a green border. (There is an option for changing the colors to assist colorblind users.)

This way, inconsistent cells brought to the user's attention by the *feedback following* support (cells which have red borders originally), and also from the cells that get highlighted when a task item is selected.

Finally, item *specifications* are automatically generated from comments in the spreadsheet and can also be modified by the user. They are displayed in the white box at the bottom of the task pane (see Figure 28a) and also in tooltips when hovering over items in the list (see Figure 28b) or in the navigation graph (see Figure 30).

8.7. The Design Impact of Tactics and Moves

Finally, implications for design based on observed *tactics* and *moves* are the lowest-level observations. As such, they are most applicable to fine-tuning the features implemented based on the *stratagem* implications.

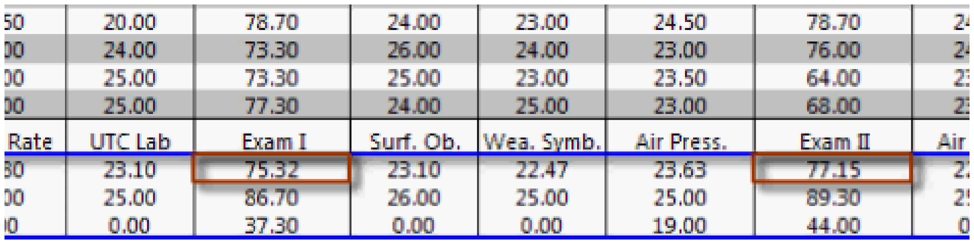
For example, we mentioned a *dataflow* graph for navigating the spreadsheet. The tactic of navigating dependencies in Excel led to the following implication:

Candidate 8: Include inter-worksheet relationships.

Due to this implication for design, StratCel's dependency graph feature displays both inter-worksheet relationships between to-do items as well as intra-worksheet relationships. Hovering over the nodes in the different worksheets allows the user to navigate between those worksheets.

Candidate 9: Allow users to easily identify areas of the spreadsheet on which to focus their attention (e.g., formulas, buggy formulas, unchecked formulas).

To address this implication in StratCel, users can superimpose the to-do status of task items onto the spreadsheet. While we originally used a shaded circle in each cell to display the to-do status of that cell, walkthroughs revealed that this was overwhelming when the status of many cells was displayed. We therefore switched to only coloring the outside borders of spreadsheet areas with a particular status. For example, Figure 31 depicts an area of the spreadsheet with many unchecked formulas (blue borders) and two cells with to-do status (red borders).



50	20.00	78.70	24.00	23.00	24.50	78.70	24
00	24.00	73.30	26.00	24.00	23.00	76.00	24
00	25.00	73.30	25.00	23.00	23.50	64.00	2
00	25.00	77.30	24.00	25.00	23.00	68.00	2
Rate	UTC Lab	Exam I	Surf. Ob.	Wea. Symb.	Air Press.	Exam II	Air
30	23.10	75.32	23.10	22.47	23.63	77.15	2
00	25.00	86.70	26.00	25.00	25.00	89.30	2
0	0.00	37.30	0.00	0.00	19.00	44.00	0

Figure 31. Users can bring up borders around to-do items by clicking on a button in the ribbon. Areas of unchecked formulas are blue. To-do items are red. And items marked as "checked" have a green border. (There is an option for changing the colors to assist colorblind users.)

Candidate 10: Too much feedback about where possible errors may lie is overwhelming, so only the most likely cells to contain errors should be highlighted by default.

StratCel currently automatically highlights inconsistent formulas by setting them as to-do items (see red items in Figure 28 and Figure 31), since those have a high likelihood of being incorrect. However, other Excel error checking warnings are ignored to reduce the false-positive rate of bugs found; sometimes, too much feedback is as bad as none at all. This lends support to the *feedback following* stratagem (following the environment's feedback about where an error may be [Grigoreanu et al. 2009]).

8.8. Evaluation

To gauge the success of employing a strategy-centric approach in the design of debugging tools, we conducted a preliminary evaluation of StratCel. In so doing, we wondered whether a strategy-centric approach to the design of debugging tools would lead to an increase in debugging success, whether StratCel was intuitive to use, and what design guidelines we could pass on to designers.

8.8.1. Experimental Setup

Procedure and Tutorial. We employed the same procedure as [Grigoreanu et al. 2009]. Participants first received a short (about 20 minutes) hands-on tutorial about

Microsoft Excel's auditing tools and StratCel's functionality on a practice spreadsheet task. The StratCel functionality presented included selecting to-do items from the list, viewing information related to the item, marking the item as "done", "to-do", or "unchecked", and adding user-defined to-do items.

Task. The task was also the same as in [Grigoreanu et al. 2009]: "Make sure the grade-book spreadsheet is correct and if you find any bugs fix them." The grade-book spreadsheet contains 1718 cells, 288 of which were formula cells, and two worksheets: one for the students' individual grades and one for summary statistics for the class. The spreadsheet is also highly formatted, containing one blue column, one yellow column, four gray columns, 30 rows with alternating colors, three different font colors, 46 cells with bold fonts, five underlined fonts, many different font faces, and all borders delimiting spreadsheet regions.

This grade-book spreadsheet is real-world. It was selected from the EUSES Spreadsheet Corpus of real-world spreadsheets [Fisher II and Rothermel 2005], originating from a college. In addition, it has been used successfully in other studies (e.g., [Beckwith et al. 2007; Grigoreanu et al. 2009).

While we originally thought the spreadsheet had ten nested bugs harvested from real users, as was reported in [Grigoreanu et al. 2009] and also based on our own experience, there were in fact 12 bugs in the spreadsheet (see the Results section for how our participants used StratCel to find two bugs which had previously been overlooked). These bugs were unintentionally introduced by the professor and by spreadsheet users from [Beckwith et al. 2007] when they attempted to add new features to this spreadsheet. There were: six inconsistency bugs (e.g., omitting some students' grades in calculating the class average for an assignment), three propagated logic errors (e.g., using the ">" operator instead of ">="), and *three* (instead of the expected one) logic bugs on individual cells (e.g., counted lab attendance as a part of the total points). The participants had a total of 45 minutes to find and fix these bugs.

Unlike in [Grigoreanu et al. 2009], where participants were provided a handout description of what different areas of the spreadsheet were meant to do, we incorporated the descriptions directly into the StratCel tool's white "specification" field (see bottom of Figure 28a).

Participants. In this pilot study of StratCel, we used five participants of varied backgrounds and spreadsheet experience. One male and one female were self-described novices, one male was a self-described intermediate, and two females were self-described experts. Our participants were members of two Seattle area clubs: the females came from a knitting circle and the males from an archery club. None of them had seen the new tool before the study. This was the group who had the StratCel Excel add-in available to them, and we will call them the "Treatment participants".

We compared their success to the eight participants from [Grigoreanu et al. 2009]. There, three males and three females were self-described spreadsheet experts and one male and one female described themselves as intermediates (no novices). We will call these participants the "Control participants".

There was no significant difference in any background variable between the Control and Treatment groups: age (Control median: 25, Treatment median: 25), major (Control: 6 non-CS science, 2 non-science; Treatment: 3 non-CS science, 2 non-science), and computer science experience (Control median: 0.5 classes, Treatment median: 0 classes). All thirteen participants had at one point edited spreadsheet formulas for work, school, or personal reasons.

However, two of the Treatment participants (one male and one female) did have less spreadsheet experience than was accepted in the Control group; they were self-described novices. We brought these two participants in for two reasons. First, we wanted to see how they would do in comparison to the experts from the other group. Second, we wanted to see how they would do against the experts in their own group.

Analysis Methodology. Since our data were not normally distributed, we employed the Wilcoxon rank-sum test with continuity correction in analyzing our quantitative data. This is non-parametric alternative to the t-test.

We also report qualitative observations about the participants' actions and verbalizations. These analyses helped both triangulate our quantitative findings and further explain the reasons behind the statistical differences.

8.8.2. Improving Debugging Success

The Treatment participants performed better by every success measure: the number of bugs found, the number of bugs fixed, the time to each bug find and bug fix, the reduced impact of individual differences, and participants' verbalized satisfaction with StratCel. To further help designers build better end-user debugging tools, we also highlight those empirically-based *guideline candidates* which had the biggest impact on our participants' success by listing them as *design guidelines* in this subsection.

Number of Bugs Found. In general, participants who had StratCel available to them were better at finding bugs. They found more bugs, including two previously unnoticed bugs, faster, and with less variability resulting from individual differences.

Specifically, Treatment group participants found significantly more bugs (Rank-sum test: Z=-2.639, p=0.0042) than the Control group participants. Figure 32 shows the distribution of bugs found by Control participants (*M:* 4.50, *SD:* 2.70) and Treatment participants (*M:* 9.00, *SD:* 0.89). This difference is striking: only one of the participants from the Control group found nine bugs, whereas all of the Treatment participants found at least nine bugs. (Both Treatment novices performed at least as well as the Control experts.)

Qualitative observations of *how* participants used StratCel revealed several reasons for this sharp increase in bug finding success. The first was Candidate 10: *Too much feedback about where errors may lurk is as bad as no feedback at all.* Since StratCel set inconsistent formulas as to-do items by default, all five participants found those six bugs. For example, to

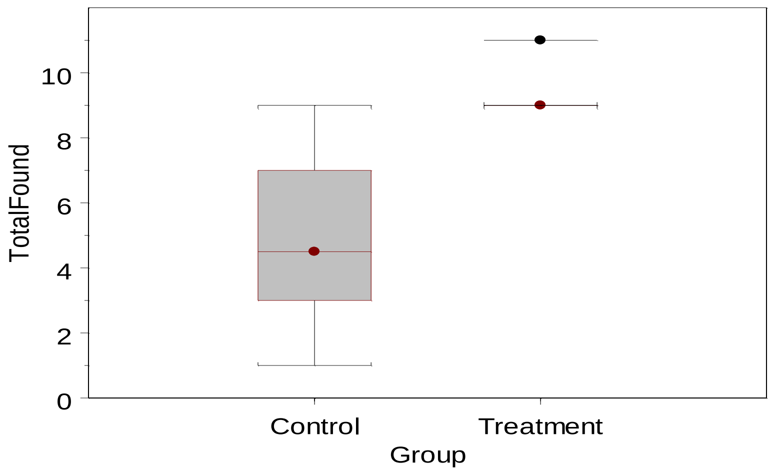


Figure 32. Participants in the Treatment group (right) found significantly more bugs than the Control group participants (left).

do this, the intermediate male participant immediately filtered the task list to only show items automatically set as to-do: inconsistent formulas. Figure 28b shows his list right after filtering.

The novice Treatment male and an experienced Treatment female employed our response to Candidate 9 to find the inconsistent formulas: *Easily find areas of the spreadsheet on which to focus their attention*. He brought the status borders up immediately at the start of the task to view the items which were automatically given to-do status (i.e., a red border).

The remaining two female participants (one novice and one expert) used a different method: they both walked through the list one item at a time, starting at the top, and only took on inconsistency items once they reached them in the to-do list. One mentioned she was also able to tell where inconsistencies laid based on the address of each to-do item being shown. For example, if an item covered the range from “A1:A3, A5” that is what showed up in the “address column” of that to-do item. This allowed her to quickly notice A4 was missing, which therefore must have been an inconsistent formula:

"This was really helpful because it has a way to say these are all your formulas… These are the ones you need to go look at. And I like this part [the address field] which shows me where I can find all of the formulas, so I can see them. For example, on this one, I could see there was a gap for E16 and I could go back and look specifically at that cell, because I expect it to be the same, and see what's going on."

Overwhelmed by the number of false-positive bug warnings (Excel green triangles in cell corners), most of the Control group participants were unable to find these inconsistencies. Our Treatment participants, however, found inconsistencies in the spreadsheet much more easily (all five participants found and fixed all six inconsistency errors) and in a variety of ways. Thus, we would like to reiterate three of the empirically-based candidate guidelines mentioned earlier but, this time, as validated design guidelines for end-user debugging tools:

Design Guideline 1: *With automatic error detection tools, it is critical to value quality (low number of false-positives) over quantity (detecting more possible types of errors). Only cells containing likely errors should be highlighted by default.*

Design Guideline 2: *As most tools currently already do, important information about cells (e.g., to-do status) should be overlaid onto the spreadsheet to give the user a quick overview of the to-do status of both individual cells and of the overall spreadsheet.*

Design Guideline 3: *Some users prefer to get a comprehensive understanding of the spreadsheet before fixing bugs (e.g., the novice female), whereas others will start by trying to fix apparent bugs right away (e.g., the intermediate male). Since both approaches have advantages and disadvantages, both should be supported.*

All participants found at least nine bugs. Other than the six inconsistency bugs, there were four other bugs which the researchers had inserted [Grigoreanu et al. 2009] and two more which were not observed by either the researchers or the Control participants, but which were found and fixed by the users in this study! These unnoticed bugs, while fairly easy to fix once spotted, were well-hidden: one individual cell was in the upper-right corner of the spreadsheet, and the second was hidden in the middle of the second worksheet.

These two previously evasive bugs were the crowning glory of the usefulness of StratCel in bug finding: some hidden bugs can evade the eyes of many experts and novices alike. However, the to-do list enabled participants to give an equal amount of attention to each item: even items in the top-left corner of the first worksheet and cells in the middle of the second worksheet.

Design Guideline 4: *Strategy-based tools should provide explicit support for to-do listing.*

Design Guideline 5: *To improve debugging of end-user programs, it helps to automatically generate a list of items to check so that all areas of the code are given equal attention.*

Number of Bugs Fixed. Just as with the number of bugs found, Treatment participants also fixed significantly more bugs (Rank-sum test: Z=-2.8905, p=0.0019) than the Control group participants. Figure 33 shows the distribution of bugs fixed by Control participants (*M:* 2.00, *SD:* 2.3299) and Treatment participants (*M:* 8.00, *SD:* 1.3038). Thus, while Treatment participants found twice as many bugs on average than Control participants, the difference in bugs fixed is even more striking: Treatment participants fixed *four times* more errors on average! (This time, the male and female Treatment novices performed better than even the most successful Control participant.)

What caused the striking difference in the number of bugs fixed? A major contributor was that Treatment participants had found more bugs, therefore also having the opportunity to fix more. Furthermore, the six inconsistency bugs were trivial fixes once the users had found them. Had the Treatment group participants only fixed the inconsistencies, they would have already fixed three times more bugs than the Control participants on average.

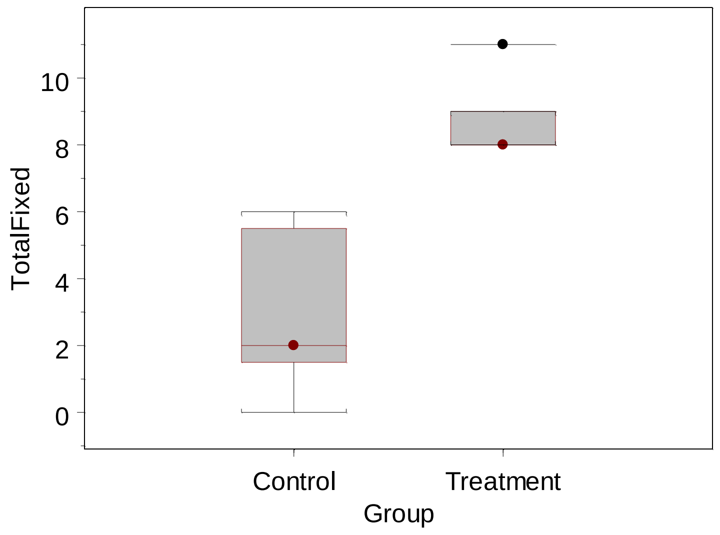


Figure 33. Treatment participants (right) fixed significantly more bugs than Control participants (left).

The two to five additional bug fixes varied by participant, but the methods by which they were fixed always involved the additional information given in the context of an item. For example, the intermediate male used Excel's "Recently Used" function library to find a formula used in a different spreadsheet (the tutorial spreadsheet) which could have been used to fix one of the most complicated bugs in the spreadsheet. All of the participants employed the descriptions provided for each item. These helped them fix two bugs consistently: two bugs on individual cells which were easy to overlook without StratCel pointing them out, but straightforward to fix once there (two cells incorrectly took into account labs as a part of the total grade): none of the Control participants found or fixed either of those bugs, and the researchers only knew about one of the two. Each of the features available in StratCel was used by at least one participant, backing the importance of showing related information in the context of each to-do item.

Design Guideline 6: *Information about the remaining stratagems should be provided in the context of each to-do item to provide more information on which to base a bug fix.*

Design Guideline 7: *Viewing formulas related to an item (e.g., the consistent formulas in an inconsistency case, recently used formulas, or formulas used in files in a certain directory) might be particularly useful for improving debugging success.*

Time to Each Bug Find and Fix. Spreadsheet debugging is often a time-sensitive activity, whether a trained accountant does it [Powell et al. 2007] or a young clerk as was the case in the Lehman-Barclays mix-up. Thus, another important measure of debugging success in addition to the number of bugs found and fixed is how long it took participants to find and fix those bugs.

On average, Treatment participants found and fixed each bug consistently faster than the Control participants. The Wilcoxon rank-sum test allows us to measure statistical difference in bugs found and fixed based on order, without worrying about missing data such as those of participants who never found or fixed a bug.

The advantage of Treatment participants was clear from the very beginning of the task. Treatment participants found the first bug significantly faster (Rank-sum test: Z=2.62, p=0.0044) and fixed it significantly faster (Rank-sum test: Z=2.8663, p=0.0021) than the Control participants. Treatment participants also found and fixed all of the remaining bugs significantly faster than Control participants (up to the tenth bug found, after which there was not enough data to prove significance, with only one Treatment participant finding and fixing eleven bugs total).

Thus, when time is short, StratCel users should be able to more quickly pinpoint errors and their solutions from the very start and keep that advantage throughout the task. It also appears that the more complex the spreadsheet is, the more useful StratCel will become, though this remains to be tested in future studies.

Closing Gaps Based on Experience and Gender. Another surprising discovery was that the Treatment participants performed very similar to one another, despite their individual differences. In previous studies on end-user debugging, both gender (e.g., [Beckwith et al. 2005]) and experience (e.g., [Grigoreanu et al. 2009]) have impacted end-user debugging success.

Also, recall that even the novices from the Treatment group performed at least as well as the most experienced and successful Control participants. When comparing Treatment novices to Treatment experts, there was little variation between the Treatment particiants, despite their very different backgrounds: the SD was twice as great for the Control group than the Treatment group. Treatment novices did not do much worse than Treatment intermediates and experts. In particular, for the Control group, bugs found ranged from 1-9 and bugs fixed from 0-6. In the Treatment group, bugs found ranged from 9-11 and bugs fixed from 8-11. Since there is a much less pronounced difference between the less experienced and the more experienced participants in the Treatment group, it appears that StratCel helps everyone, and especially less experienced users. The following quote comes from the novice Treatment female:

"I feel like it would be extra useful for someone like me who, well, I can use Excel and I can figure it out, but, like, I'm definitely not an expert at Excel. […] I think the only problems I had were with the Excel functions I hadn't learned. This is like a really good way of helping me keep track of what I've done and not get lost."

In terms of gender, comparing the median number of bugs found (CF: 4.5, TF: 9.0, CM: 5.0, TM: 9.0) and fixed (CF: 3.5, TF: 9, CM: 2.5, TM: 8.5) by females and males in the Control and Treatment groups, we noticed that there were few gender differences between them. Even so, Treatment participants were a little closer to each other than Control participants in terms of success: meaning that StratCel helped both males and females.

Overall Experience: StratCel's Usability. While we did not ask our participants for feedback beyond their verbalizations during the task, the participants were nevertheless anxious to give it.

Several comments revealed possible iterative improvements to the tool. For example, participants had a feature available to add to-do items to the automatically generated list. The most successful Treatment female used it as a way to add two comments for the next person who will look at the spreadsheet: one about how little she trusts the spreadsheet and a second about a change she would have liked to have made to one of the formulas in the future. The most successful male also added a custom to-do item, but he did so by mistake. Their feature request was to add the functionality of *removing* items from the to-do list.

Another improvement requested by the two experienced females was the capability to sort the to-do list by clicking on the field headers. One of the potentially most critical problems with the to-do functionality is that it is too easy to check off items as done, to never be returned to again. One of the experienced females put it this way:

"The only thing that I was thinking about is that it's really easy to say 'Oh, I've looked at this.' and just check it off. And I don't know if there could be a way to make sure that that's what they meant. […] So, I actually had something… Where I went through, and I think I'm on one line but I'm actually on another when I check off the task being done. But I think that's just... A user has to be smart enough to know not to do that. There's only just so much that you can help a user avoid."

One possibility for making sure that the user really meant to check something off would be to list each of the "stratagem tool components" (e.g., the specification) as individual subtasks for each task. This way, users would have to check off several subtasks in order to achieve an overall "check" for the item. Further research is needed to what the best method is for doing this.

Overall, however, the participants' unrequested comments were very positive, and most immediately thought of ways to apply StratCel to their own day-to-day tasks. Here are selected few of the quotes:

"So, can I use your tool? You should sell this and make a million dollars!"

“I think this would be useful for my complex accounting spreadsheets. If you would like to share the tool, I would love to try it on those.”

"Looking at [StratCel], I was thinking I have to have a way of tracking my [knitting] patterns. So things that… Ok. I have a pattern and I have steps I have to go through. And I need a way to track them.”

"And this is straight-forward and makes a lot of sense. When you look at it, you know what it is. There are lots of tools, where you can tell that people said, 'well… there's just a workaround and you can just do it this way'. But this one, it just seemed very straightforward and it builds on everything from Excel.”

8.9. Conclusions and Future Work

In this paper, we have shown that a *strategy-based approach* alone can be effectively applied in the design of debugging and troubleshooting tools to improve the correctness of end-user programmers’ code.

As a part of this effort, we instantiated our approach in StratCel: a new strategy-based add-in for one of today’s most widely used end-user programming environments, Excel. StratCel addresses implications for design at four strategy levels.

Our results showed that tools can be built to support a comprehensive understanding of strategies directly. We employed implications derived from higher strategy levels (stratagems and strategies) to frame the functionality of the tool as a whole, while implications based on lower levels of strategy (moves and tactics) helped us fine-tune individual features. For example, support for the *to-do listing* stratagem provided a way to reduce end-user programmers’ cognitive load, by helping *comprehensive* participants better keep track of to-do items to revisit and by helping *selective* participants see which formulas they had skipped. The remaining nine stratagems defined the core activities which needed to be supported within the context of each to-do list item (e.g., specifications, help about the formula as a whole, etc.) in our instantiation. Finally, the implications from the lower strategy levels (moves and tactics) helped us fine-tune the features supporting each stratagem: for example, making sure that the *dataflow* dependencies showed inter-worksheet relationships and facilitated navigating between items on different worksheets.

Even for an environment as mature as Excel, the addition of a strategy-based tool did improve end-user programmers' debugging success using many measures:

- Participants who had StratCel available to them found twice as many bugs, fixed four times as many bugs, and in only a fraction of the time.
- While StratCel helped everyone, it was particularly helpful to less experienced users. StratCel also helped males and females equally.
- Participants found StratCel intuitive to use and immediately thought of ways in which the tool applied to their day-to-day work.

This approach to end-user debugging tool building has raised many questions, opening the door to opportunities for future research.

- The current instantiation of StratCel centers on the *to-do listing* stratagem, supporting the other stratagems within the context of each to-do item. A future goal might be to create a new tool which centers around one of the other stratagems (say *code inspection* or *testing*) and which supports all other nine stratagems within the context of either a formula or of an output value, in those two cases respectively. Would the addition of another strategy-centered tool improve users' success even further?
- Even within its current instantiation of the implications for design, each of StratCel's components can be improved with further research. For example, StratCel currently only highlights inconsistency errors, but both Excel and other tools provide many other automatically generated warnings. An ordered list of the available automatic spreadsheet error detection algorithms and their false-positive rates, would be required to further improve the *error checking* component, in order to know which algorithms to turn on by default.

- Finally, related empirical work has drawn parallels across programming populations and environments: from spreadsheets, to scripting environments, and integrated development environments (recall Table 23). Can StratCel's core functionality be transferred to one of these other environments? If so, will it also lead to increased debugging success there? Do these concepts change when users are not able to manipulate the code directly and have to work at a higher level of abstraction (e.g., when troubleshooting a printer failure)?

In summary, we have shown that a strategy-based approach to building debugging tools is both achievable and beneficial. Powerful but disconnected features may be the approach of the past, and be replaced by features which work together to support users' effective debugging and troubleshooting strategies.

9. Conclusions and Future Work

This dissertation addressed a gap in today's research on end-user software engineering and end-user debugging by taking into account end users' debugging *strategies*. To address this gap, we presented eight empirical studies (Data Mining Study, Feature Usage Study, Forms/3 Stratagems Study, PowerShell Stratagems Study, Designers Grounded Theory Study, Sensemaking Study, and Excel Strategy Items Study, StratCel Tool Study), as well as StratCel itself (a new strategy-based end-user debugging tool for Excel).

The eight empirical studies reported in this dissertation investigate the problem of *how to design end-user debugging tools such that they support end users' debugging strategies*. The Data Mining Study revealed the importance of paying attention to both genders' strategies (Chapter 2). Data mining generated three hypotheses. In this dissertation, we tested only one of those hypotheses: *the debugging strategy items that help with males' success are not the right ones for females' success*. However, the work also revealed two other hypotheses that remain to be tested (see Table 24).

Table 24. Implications for future research resulting from our studies in Chapters 2-5.

Chapter	Implication for Future Research
2	Hypothesis: Unsuccessful males overdo use of arrows – unlike successful males, successful females, or unsuccessful females.
2	Hypothesis: Females with lower self-efficacy are likely to struggle longer to use a strategy that is not working well, before moving on to another strategy.
3	RQ: What do users learn from strategy explanations and how they applied this knowledge to their debugging task?
3	RQs: How can explanations best guide males' and females' toward effective end-user debugging strategy items? How can they best support all four strategy levels?
4	RQ: How can tools better support code inspection and specification checking?
4	Hypotheses: Males' and females' success with a stratagem differs by debugging stage (finding a bug, fixing a bug, or evaluating a fix). Code inspection is tied to females' success in fixing bugs and tool support for it should promote greater debugging success for females. Dataflow is tied to males' success in fixing bugs and evaluating fixes. Tool support for dataflow should promote greater debugging success for males. Testing is tied to males' success at correctly finding and fixing bugs, thus environments that support incremental testing should increase males' debugging success.
5	Goal: Further develop the set of stratagems by generalizing it across more environments, more populations, and using several methodologies. E.g.: What are *designers'* debugging strategy items? Do other end-user programmers also use code cleanup, switching levels of granularity, and usability testing during debugging?
5	RQs: Why do male and female designers tend to speak about their needs at two different levels of the design (males at the code level and females at the interface level)? What implications does this have for the design of tools to support their debugging needs?
5	Goals: Design a versioning system which 1-keeps track of alternate latest design versions, 2-shows the hierarchical ties between design components, and 3-keeps track of design decision as a part of the versioning system. Design a multi-view software development environment based on team members' role to provide all software professionals a common artifact to work on.

The Feature Usage Study (Chapter 3) further motivated this work revealing that just improving feature usage alone, without taking features' roles in strategy into account, may not increase success (as was the case in our study. Despite higher feature usage and improved confidence, Treatment females were still not performing better. Based on the combination of the studies presented in this dissertation, we now know that this was likely because the features were a mismatch to females' stratagems: females were using more testing and dataflow features, but as the Forms/3 Stratagems Study [Subrahmaniyan et al. 2008] revealed, those were not the stratagems which tied to more bugs being found and fixed by females. The other research questions opened by the explanations feature in the Forms/3 Feature Usage Study are listed in Table 24.

The data collected from an open-ended question during the Forms/3 Stratagems Study revealed eight debugging stratagems employed by end-user programmers [Subrahmaniyan et al. 2008]. That set was further expanded in the PowerShell Stratagems Study to generalize across both spreadsheet (Forms/3 and Microsoft Excel) and scripting (Windows PowerShell) environments (Chapters 4 and 7). They also generalized across end-user programming experiences, ranging from undergraduate students to IT professionals with more than a decade of experience (Chapters 4, 5, and 7). Thus, one of our work's main contributions is our final set of ten debugging stratagems derived using both statistical and qualitative methods: code inspection, control flow, dataflow, feedback following, help, proceeding as in prior experience, spatial, specification checking, testing, and to-do listing.

While the stratagems listed above are the ones we employed for the remainder of the dissertation, this list is likely not yet complete. For example, there was evidence of our stratagems' applicability to professional designers' debugging of interactive application designs (Chapter 5). However, that work also uncovered some potential additions to the stratagem list, including: cleaning up code, changing the level of abstraction, and usability testing. Thus, some stratagems may be more obvious in certain environments and with particular populations. Another factor which influenced which stratagems we noticed was the experimental setup. For example, we only observed *proceeding as in prior experience* during the think-aloud sessions; this was not a stratagem users recalled employing in the

Forms/3 Stratagems Study, which did not include a think-aloud component. See Table 24 for implications for future research resulting from Chapter 5.

Males and females statistically preferred and were effective with different stratagems, and females' stratagems (code inspection, to-do listing, and specification checking) were least supported by environments like Forms/3 and PowerShell (Chapter 4). We observed these differences both statistically (in *what* stratagems were preferred by males and females) and qualitatively (in *how* stratagems were used by males and females to find bugs, fix bugs, and evaluate fixes).

The earlier studies (e.g., Chapter 4) were in-depth explorations of stratagems. However, this work also revealed the importance of observing strategy at four levels: moves, tactics, stratagems, and strategies (Chapters 6 and 7). Each level of strategy led to a different type of implication for design, ranging from iterative low-level changes to the design of completely new tools. Employing a Sensemaking approach revealed two end-user debugging strategies, *comprehensive* and *selective*, and the advantages and disadvantages of each (Chapter 6). We named them as such because they aligned with the comprehensive and selective information processing styles. The Sensemaking approach also revealed the pervasiveness of information foraging during debugging, especially for the non-selective participants. See Table 25 for implications for future research resulting from this study.

Finally, an examination of participants' behaviors and verbalizations during Excel debugging revealed detailed implications for the design of strategy-based end-user debugging tools (Chapter 7). This study also highlighted several opportunities for future research (see Table 25).

Table 25. Implications for future research resulting from our studies in Chapters 6 and 7.

Chapter	Implication for Future Research
6	RQs: How could tools detect the user's current sensemaking step in order to tailor its behavior depending on whether the user is or is not progressing systematically? In particular, can incremental moves down the sensemaking model be supported by keeping track of various kinds of assumptions?
6	RQ: How can information foraging in spreadsheets be better supported, taking into account theory constructs such as "scent"?
6	RQ: What other *strategies* do end-user programmers employ?
7	Goal: Improving code reuse for the purpose of fixing bugs by allowing users to search for related code that is not in spatial proximity (e.g., skimming other files in the directory or online with similar functions, descriptions, or formulas).
7	Goal: Providing the ability effectively implement repetition in declarative environments (such as Excel), without the user having to learn a new language and environment.
7	RQs: Why is code inspection such a dominant stratagem? How does its usage differ by sensemaking step? Should designers create one code inspection tool to support all sensemaking steps, or specialized tools for the different steps?
7	RQs: How can prior experience be supported during the early steps of sensemaking: in first skimming information and in deciding which information to follow-up on? What are the ties between prior experience and designers' reuse tactic?
7	RQ: Would it be useful to support all eight stratagems as a part of the later sensemaking steps of hypothesis, presentation, and reevaluation?

The implications for design from these studies materialized in the StratCel end-user debugging add-in for Excel, which provides explicit support for end-user debugging strategies, stratagems, tactics, and moves (Chapter 8). StratCel makes two contributions: (1) it demonstrates an approach for designing tools around strategy items, and (2) it demonstrates the benefits to debugging success that such a tool can provide. In particular, participants using the tool found *twice* as many bugs as participants using standard Excel, they fixed *four* times as many bugs, and achieved these accomplishments in only a small fraction of the time. Furthermore, this strategy-based approach helped both male and female participants, and especially helped the participants who needed it the most: boosting novices' debugging performance near experienced participants' improved levels. Furthermore, this approach allowed us to validate the implications for design derived from the Excel Strategy Items Study, which led to seven guidelines for the design of end-user debugging tools. Table 26 gives implications for future research resulting from the StratCel Tool Study.

In summary, this work has empirically shown the need for taking end users' debugging strategies into account when designing tools to support them, has presented an approach for designing a debugging tool from a strategy perspective, and has empirically demonstrated significant advantages of the strategy-centric approach over traditional feature-centric approaches to the design of tools that aim to support end-user debugging. In so doing, this strategy-centric approach has been shown to help everyone: males and females, novices and experts.

Table 26. Implications for future research resulting from the StratCel Tool Study (Chapter 8).

Goal: The current instantiation of StratCel centers on the to-do listing stratagem, supporting the other stratagems within the context of each to-do item. Imagine nine other tools, each centering around one of the other stratagems.	
	RQs: Is it possible to do this for all stratagems? What could those tools' designs look like? RQs: Does strategy support have additive properties or is on tool enough? For example, would the addition of a tool which centers around code inspection improve users' success even further?
Goal: Staying with a to-do listing centric approach, StratCel's support for each of the stratagems can be improved with further research.	
	Example: StratCel currently only highlights inconsistency errors, but both Excel and other tools provide many other automatically generated warnings. An ordered list of the available automatic spreadsheet error detection algorithms and their false-positive rates, would be required to further improve the error checking component, in order to know which algorithms to turn on by default.
Goal: We expect this technology to be transferable to other contexts, since related empirical work has drawn parallels across programming populations and environments: from spreadsheets, to scripting environments, and integrated development environments (recall Table 23).	
	RQs: Can StratCel's core functionality be transferred to one of these other environments? If so, will it also lead to increased debugging success there? RQ: Do these concepts change when users are not able to manipulate the code directly and have to work at a higher level of abstraction (e.g., when troubleshooting a printer failure)?

Bibliography

Abraham, R., and M. Erwig. "UCheck: A spreadsheet unit checker for end users." *Journal of Visual Languages and Computing 18(1)*, 2007: 71-95.

Agrawal, R., and R. Srikant. "Mining sequential patterns." *Eleventh International Conference on Data Engineering.* 1995. 3-14.

Allwood, C. "Error detection processes in statistical problem solving." *Cognitive Science 8(4)*, 1984: 413-437.

Ambler, A., and J. Leopold. "Public programming in a web world." *Symposium on Visual Languages.* Nova Scotia, Canada: IEEE, 1998. pp. 100.

Aranda, J., and G. Venolia. "The secret life of bugs: Going past the errors and omissions in software repositories." *International Conference on Software Engineering.* IEEE, 2009. 298-308.

Ayalew, Y., and R. Mittermeir. "Spreadsheet debugging." *European Spreadsheet Risks Interest Group.* 2003. 13 pages.

Backus, J. "The history of Fortran I, II, and III." *IEEE Annals of the History of Computing 20(4)*, 1998: 68-78.

Bandura, A. "Self-efficacy: Toward a unifying theory of behavioral change." *Psychological Review 8(2)*, 1977: 191-215.

—. *Social Foundations of Thought and Action.* Englewood Cliffs, NJ: Prentice Hall, 1986.

Basili, V., and R. Selby. "Comparing the effectiveness of software testing strategies." *Tansactions on Software Engineering 13(12)*, 1987: 1278-1296.

Bates, M. "Where should the person stop and the information search interface start?" *Information processing and Management 26(5)*, 1990: 575-591.

Beckwith, L., D. Inman, K. Rector, and M. Burnett. "On to the real world: Gender and self-efficacy in Excel." *IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2007. 119-126.

Beckwith, L., et al. "Tinkering and gender in end-user programmers' debugging." *ACM Conference on Human Factors in Computing Systems.* ACM, 2006. 231-240.

Beckwith, L., M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, and M. Hastings. "Effectiveness of end-user debugging software features: Are there gender issues?" *Conference on Human-Computer Interaction.* ACM, 2005. 869-878.

Beckwith, Laura, and Margaret Burnett. "Gender: An important factor in end-user programming environments?" *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2004. 107-114.

Benford, S., G. Giannachi, B. Koleva, and T. Rodden. "From Interaction to Trajectories: Designing Coherent Journeys through User Experiences." *Proceedings of CHI 2009.* ACM Press, 2009. 709-718.

Berners-Lee, T. *Hypertext markup language.* Geneva, Switzerland: CERN, 1993.

Bhavnani, K., F. Reif, and B. John. "Beyond command knowledge: Identifying and teaching strategic knowledge for using complex computer applications." *Proceedings of the Conference on Human Factors in Computing Systems.* ACM Press, 2001. 229-236.

Blackwell, A. "First steps in programming: a rationale for attention investment models." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2002. 2-10.

Boehm, B., and V. Basili. "Software defect reduction top 10 list." *Computer 34(1)*, 2001: 135-137.

Boehm, B., et al. *Software cost estimation with COCOMO II.* Upper Saddle River, NJ: Prentice Hall, 2000.

Bogart, C., M. Burnett, A. Cypher, and C. Scaffidi. "End-user programming in the wild: A field study of CoScripter scripts." *Proceedings of the Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2008. 39-46.

Brewer, J., and A. Bassoli. "Reflections of gender: Designing ubiquitous computing technologies." *Gender and Interaction: Real and Virtual Women in a Male World, Workshop at AVI.* 2006. 9-12.

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

Burnett, M., C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. "End-user software engineering with assertions in the spreadsheet paradigm." *International Conference on Software Engineering.* ACM, 2003. 93-103.

Burnett, M., S. Chekka, and R. Panday. "FAR: An End user Language to Support Cottage E-Services." *Symposia on Human-Centric Computing Languages and Environments.* Stresa, Italy: IEEE, 2001. 195-202.

Busch, T. "Gender differences in self-efficacy and attitudes toward computers." *Journal of Educational Computing Research 12(2)*, 1995: 147-158.

Butler, R. "Is this spreadsheet a tax evader? How H.M. Customs & Excise test spreadsheet applications." *Hawaii International Conference on System Sciences.* IEEE, 2000. 6 pages.

Byrnes, J., C. Miller, and D. Schafer. "Gender differences in risk taking: A meta-analysis." *Psychological Bulletin 125*, 1999: 367-383.

Carr, M., and D. Jessup. "Gender differences in first-grade mathematics strategy use: Social and metacognitive influences." *Journal of Educational Psychology 89(2)*, 1997: 318-328.

Carroll, J. *Minimalism Beyond "The Nurnberg Funnel".* Cambridge, MA: MIT Press, 1998.

Cervone, G., and R. Michalski. "Modeling user behavior by integrating AQ learning with a database: Initial results." *Intelligent Information Systems.* 2002. 43-56.

Chandrasekaran, B. "Design problem solving: A task analysis." *Artificial Intelligence Magazine 11.* 1990. 59-71.

Charmaz, K. "Grounded theory: Objectivist and constructivist methods." In *Handbook of qualitative research*, by N. Denzin and Y. Lincoln, 509-535. Thousand Oaks, CA: Sage, 2000.

Choo, C. "Working with knowledge: How information professionals help organizations manage what they know." *Library Management 21(8)*, 2000: 395-403.

Chung, J., and K. Tang. "Inherent gender differences as an explanation of the effect of instructor gender on account students' performance." *7th Annual Teaching Learning Forum.* 1998. 72-79.

Compeau, D., and C. Higgins. "Application of social cognitive theory to training for computer skills." *Information Systems Research 6(2)*, 1995: 118-143.

Cross, N. "Expertise in design: An overview." *Design Studies 25(5)*, 2004: 427-441.

Cypher, A., and D. Smith. "KidSim: End-user programming of simulations." *Conference on Human Factors in Computing Systems.* ACM, 1995. 27-34.

Cypher, A., et al. *Watch what I do: programming by demonstration.* Cambridge, MA: MIT Press, 1993.

Czerwinski, M., D. Tan, and G. Robertson. "Women take a wider view." *Conference on Human Factors in Computing Systems.* ACM, 2002. 195-202.

Danis, C., W. Kellogg, T. Lau, J. Stylos, M. Dredze, and N. Kushmerick. "Managers' email: Beyond tasks and to-dos." *Conference on Human Factors in Computing Systems.* ACM, 2005. 1324-1327.

Darley, W., and R. Smith. "Gender differences in information processing strategies: An empirical test of the selectivity model in advertising response." *Journal of Advertising 24*, 1995: 41-56.

Davis, J. "Tools for spreadsheet auditing." *International Journal of Human-Computer Studies 45*, 1996: 429-442.

De Angeli, A., and S. Brahnam. "Sex stereotypes and conversational agents." *Proceedings of Gender and Interaction, Real and Virtual Women in a Male World Workshop.* Venice, 2006.

Dervin, B. "A theoretic perspective and research approach for generating research helpful to communication practice." *Public Relations Research and Education 1(1)*, 1984: 30-45.

Dervin, B., L. Foreman-Wernet, and E. Launterbach. *Sense-making methodology reader: Selected writings of Brenda Dervin.* Cresskill, NJ: Hampton Press, Inc., 2003.

Designer. http://dictionary.reference.com/browse/designer (accessed December 4, 2008).

Dorn, B., A. Tew, and M. Guzdial. "Introductory computing construct use in an end-user programming community." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2007. 27-32.

Elkan, C. "Magical thinking in data mining: Lessons from CoILChallenge 2000." *International Conference on Knowledge Discovery in Data Mining.* ACM, 2001. 426-431.

El-Ramly, M., E. Stroulia, and P. Sorenson. "From run-time behavior to usage scenarios: An interaction-pattern mining approach." *International Conference on Knowledge Discovery and Data Mining.* ACM, 2002. 315-323.

Ericsson, K., and H. Simon. *Protocol analysis: Verbal reports as data.* Cambridge, MA: MIT Press, 1984.

EuSpRIG. *Spreadsheet mistakes news stories.* April 12, 2006. http://www.eusprig.org/stories.htm (accessed June 20, 2009).

Fern, X., C. Komireddy, and M. Burnett. "Mining interpretable human strategies: A case study." *International Conference on Data Mining.* IEEE, 2007. 475-480.

Fern, X., C. Komireddy, V. Grigoreanu, and M. Burnett. "Mining problem-solving strategies from HCI data." *ACM Transactions on Computer-Human Interaction*, 2009: (to appear).

Finucane, M., P. Slovic, C.-K. Merz, J. Flynn, and T. Satterfield. "Gender, race and perceived risk: the white male effect." *Health, Risk and Society 2(2)*, 2000: 159-172.

Fisher II, M., and G. Rothermel. "The EUSES Spreadsheet Corpus: a shared resource for supporting experimentation with spreadsheet dependability mechanism." *Workshop n End-User Software Engineering.* ACM, 2005. 47-51.

Furnas, G., and D. Russell. "Making sense of sensemaking." *Conference on Human Factors in Computing Systems.* ACM, 2005. 2115-2116.

Gallagher, A., and R. De Lisi. "Gender differences in scholastic aptitude-test mathematics problem-solving among high-ability students." *Journal of Educational Psychology 86(2)*, 1994: 204-211.

Gallagher, A., R. De Lisi, P. Holst, A. McGillicuddy-De Lisi, M. Morely, and C. Cahalan. "Gender differences in advanced mathematical problem solving." *Journal of Experimental Child Psychology 75(3)*, 2000: 165-190.

Glaser, 1978. *Theoretical sensitivity.* Mill Valley, CA: Sociology Press, 1978.

Glaser, B., and A. Strauss. *The discovery of grounded theory: Strategies for qualitative research.* Chicago, IL: Aldine, 1967.

Gorriz, C., and C. Medina. "Engaging girls with computers through software games." *Communications of the ACM*, 2000: 42-49.

Green, T., and M. Petre. "Usability analysis of visual programming environments: A `cognitive dimensions' framework." *Journal of Educational Psychology 86(2)*, 1994: 204-211.

Greenberg, S. "Toolkits and interface creativity." *Multimedia Tools and Applications 32(2)*, 2007: 139-159.

Grigoreanu, V. *Complex patterns in gender HCI: A data mining study of factors leading to end-user debugging success for females and males.* Master's Thesis, 2007.

Grigoreanu, V., et al. "Can feature usage design reduce the gender gap in end-user software development environronts?" *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2008. 149-156.

—. "Gender differences in end-user debugging, revisited: What the miners found." *IEEE Symposium on Visual Languages and Human Centric Computing.* IEEE, 2006. 19-26.

Grigoreanu, V., J. Brundage, E. Bahna, M. Burnett, P. ElRif, and J. Snover. "Males' and females' script debugging strategies." *International Symposium on End-User Development.* Springer Berlin / Heidelberg, 2009. 205-224.

Grigoreanu, V., M. Burnett, and G. Robertson. "A strategy-centric approach to the design of end-user debugging tools." *Conference on Human Factors in Computing Systems (CHI'10).* ACM, 2010. 10 pages.

Grigoreanu, V., M. Burnett, and G. Robertson. *Design implications for end-user debugging tools: A strategy-based view.* 44 pages: Technical Report: http://hdl.handle.net/1957/12443, 2009 (Under Review).

Grigoreanu, V., M. Burnett, S. Wiedenbeck, J. Cao, and K. Rector. *Females' and males' end-user debugging strategies: A sensemaking perspective.* Corvallis, OR: http://ir.library.oregonstate.edu/jspui/handle/1957/12074, 2009.

Grigoreanu, V., R. Fernandez, K. Inkpen, and G. Robertson. "What designers want: needs of interactive application designers." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2009. 139-146.

Gunzelmann, G., and J. Anderson. *An ACT-R model of the evolution of strategy use and problem and difficulty.* http://repository.cmu.edu/psychology/53: Carnegie Mellon university, 2008.

Halpern, D. *Sex differences in cognitive abilities.* Mahwah, NJ: Lawrence Erlbaum Associates, Inc., 2000.

Hartzel, K. "How self-efficacy and gender issues affect software adoption and use." *Communications of the ACM*, 2003: 167-171.

Hatonen, K., M. Klemettinen, P. Ronkainen, and H. Toivonen. "Knowledge discovery from telecommunication network alarm data bases." *12th International Conference on Data Engineering.* 1996. 115-122.

Hayes, F. *Rules for users.* October 20, 2008. http://www.pcworld.com/businesscenter/article/152509/rules_for_users.html (accessed September 01, 2009).

Heger, N., A. Cypher, and D. Smith. "Cocoa at the Visual Programming Challenge ." *Journal of Visual Languages and Human-Centric Computing*, 1998: 151-169.

Horne, M. "Early gender differences." *Conference of the International Group for the Psychology of Mathematics Education.* 2004. 65-72.

Igarashi, T., J. Mackinlay, Chiang B., and P. Zellweger. "Fluid visualization of spreadsheet structures." *Symposium on Visual Languages.* IEEE, 1998. 118-125.

Ioannidou, A., A. Repenning, and D. Webb. "Using scalable game design to promote 3D fluency: Assessing the AgentCubes incremental 3D end-user development framework." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2008. 47-54.

Jay, T. "Gender differences in knowledge transfer in school mathematics." *British Congress of Mathematics Education.* 2005. 81-88.

Jeffries, R. "Comparison of debugging behavior of novice and expert programmers." *AERA Annual Meeting.* Pittsburgh, PA: Department of Psychology, Carnegie Mellon University, 1982.

Jin, J., and L. Dabbish. "Self-interruption on the computer: A typology of discretionary task interleaving." *Conference on Human Factors in Computing Systems.* ACM, 2009. 1799-1808.

Jones, M., et al. "Tool time: Gender and students' use of tools, control, and authority." *Journal of Research in Science Teaching 37(8)*, 2000: 760-783.

Kandogan, E., E. Haber, R. Barrett, A. Cypher, P. Maglio, and H. Zhao. "A1: end-user programming for web-based system administration." *Symposium on User Interface Software and Technology.* ACM Press, 2005. 211-220.

Katz, I., and J. Anderson. "Debugging: An analysis of bug-location strategies." *Human Computer Interaction 3*, 1988: 351-399.

Kelleher, C., and R. Pausch. "Lowering the barriers to programming: A survey of programming environments and languages for novice programmers." *Computing Surveys.* ACM, 2005.

Kelleher, C., R. Pausch, and S. Kiesler. "Storytelling Alice motivates middle school girls to learn computer programming." *ACM Conference on Human-Computer Interaction.* ACM, 2007. 1455-1464.

Kissinger, C., et al. "Supporting end-user debugging: What do users want to know?" *Working Converence on Advanced Visual Interfaces.* ACM, 2006. 135-142.

Klein, G., B. Moon, and R. Hoffman. "Making sense of sensemaking 1: Alternative Perspectives." *IEEE Intelligent Systems 21(4)*, 2006: 70-73.

Ko, A., and B. Myers. "Designing the Whyline: A debugging interface for asking questions about program failures." *International Conference on Human-Computer Systems.* ACM, 2004. 151-158.

Ko, A., B. Myers, and H. Aung. "Six Learning Barriers in End-User Programming Systems." *Symposium for Visual Languages and Human-Centric Computing.* IEEE, 2004. 199-206.

Ko, A., R. DeLine, and G. Venolia. "Information needs in collocated software development teams." *International Conference on Software Engineering.* ACM, 2007. 344-353.

Lawrance, J., R. Bellamy, M. Burnett, and K. Rector. "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks." *Conference on Human Factors in Computing Systems.* ACM, 2008. 1323-1332.

Lawton, C., and J. Kallai. "Gender differences in wayfinding strategies and anxiety about wafinding: A cross-cultural comparison." *Behavioral Science 47(9-10)*, 2002: 389-440.

Leedom, D. *Final report: Sensemaking symposium.* http://www.dodccrp.org/events/2001_sensemaking_symposium/docs/FinalReport/Sensemaking_Final_Report.htm: Command and Control Research Program, Office of the Assistant Secretary of Defense for command, Control, Communications and Intelligence, 2001.

Littman, D., J. Pinto, S. Letovsky, and E. Soloway. "Mental models and software maintenance." *ESP* (ESP), 1986: 80-98.

Lorigo, L., B. Pan, H. Hembrooke, T. Joachims, L. Granka, and G. Gay. "The influence of task and gender on search and evaluation behavior using Google." *Information processing and Management.* 2006. 1123-1131.

Mannila, H., H. Toivonen, and A. Verkamo. "Discovery of frequent episodes in event sequences." *Data Mining and Knowledge Discovery.* 1997. 259-289.

Martocchio, J., and J. Webster. "Effects of feedback and playfulness on performance in microcomputer software training." *Personnel psychology 45*, 1992: 553-578.

Meyers-Levy, J. "Gender differences in information processing: A selectivity interpretation." In *Cognitive and Affective Responses to Advertising*, by P. Cafferata and A. Tybout, 219-260. Lexington, MA: Lexington Books, 1989.

Miller, G. "The magical number seven, plus or minus two: Some limits on our capacity for processing information." *The Psychological Review 63*, 1956: 81-97.

Mobasher, B., R. Cooley, and J. Srivastava. "Automatic personalization based on web usage mining." *Communications of the ACM.* ACM, 2000. 142-151.

Myers, B., A. Ko, and M. Burnett. "Invited research overview: End-user programming." *Extended Abtracts of the Conference on Human Factors in Computing Systems.* ACM, 2006. 75-80.

Myers, B., S. Park, Y. Nakano, G. Mueller, and A. Ko. "How designers design and program interactive behaviors." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2008. 177-184.

Nanja, N., and C. Cook. "An analysis of the on-line debugging process." *Empirical Studies of Programmers: Second Workshop.* NJ: Ablex Publishing Corporation, 1987. 172-184.

Nardi, B. *A small matter of programming: Perspectives on end-user computing.* Cambridge, MA: MIT Press, 1993.

Nardi, B., and J. Miller. "The spreadsheet interface: a basis for end user computing." *International Conference on Human-Computer Interaction.* North-Holland Pubishing Co., 1990. 977-983.

Naumer, C., K. Fisher, and B. Dervin. "Sense-Making: A methodological perspective." *Conference on Human Factors in Computer Systems.* ACM, 2008. Workshop paper.

Nevada Daily Mail. *Budget discrepancies attributed to computer error.* 2006. http://www.nevadadailymail.com/story/1135458.html (accessed July 2009).

Newman, M., and J. Landay. "Sitemaps, storyboards, and specifications: a sketch of Web site design practice." *Conference on Designing Interactive Systems: Process, Practices, Methods, and Techniques.* New York, NY: ACM Press, 2000. 263-274.

Newman, M., J. Lin, J. Hong, and J. Landay. "DENIM: An informal web site design tool inspired by observations of practice." *Human-Computer Interaction*, 2003: 263-274.

O'Donnell, E., and E. Johnson. "The effects of auditor gender and task complexity on information processing efficiency." *International Journal of Auditing 5*, 2001: 91-105.

Pane, J., and B. Myers. *Usability issues in the design of novice programming systems.* TR CMU-CS-96-132: Carnegie Mellon University, School of Computer Science, 1996.

Panko, R. "What we know about spreadsheet errors." *Journal of End User Computing 10(2)*, 1998: 15-21.

Panko, R., and N. Orday. "Sarbanes-Oxley: What about all the spreadsheets? ." *European Spreadsheet Risks Interest Group.* 2005. 45 pages.

Pennington, N. "Stimulus structures and mental representations in expert comprehension of computer programs." *Cognitive Psychology 19(3)*, 1987: 295-341.

Perkowitz, M., and O. Etzioni. "Adaptive web sites: Automatically synthesizing web pages." *15th National Conference on Artificial Intelligence.* 1998. 727-732.

Peyton Jones, S., A. Blackwell, and M. Burnett. "A user-centred approach to functions in Excel." *International Conference on Functional Programming.* ACM, 2003. 165-176.

Phalgune, A., C. Kissinger, M. Burnett, C. Cook, L. Beckwith, and J. Ruthruff. "Garbage in, garbage out? An empirical look at oracle mistakes by end-user programmers." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2005. 45-52.

Pirolli, P., and S. Card. "Information foraging." *Psychology Review 106(4).* 1999. 643-675.

—. "The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis." *International Conference on Intelligence Analysis.* 2005.

Pokorny, R. *Budget discrepancies attributed to computer error.* January 6, 2006. http://www.nevadadailymail.com/story/1135458.html (accessed June 20, 2009).

Powell, M., and D. Ansic. "Gender differences in risk behaviour in financial decision-making: An experimental analysis." *Journal of Economic Psychology 18(6)*, 1997: 605-628.

Powell, S., K. Baker, and B. Lawson. "An Auditing protocol for Spreadsheet Models." January 2007. http://mba.tuck.dartmouth.edu/spreadsheet/product_pubs.html (accessed August 28, 2007).

Power, D. *DSSResources.com.* August 30, 2004. http://dssresources.com/history/sshistory.html, version 3.6 (accessed September 1, 2007).

Prabhakararao, S., et al. "Strategies and behaviors of end-user programmers with interactive fault localization." *Symposium on Human-Centric Computing Languages and Environments.* IEEE, 2003. 15-22.

Rigby, P., D. German, and M. Storey. "Open Source Software Peer Review Practices: A Case Study of the Apache Server." *International Conference on Software Engineering.* ACM, 2008. 541-550.

Rode, J., and M. Rosson. "Programming at runtime: Requirements and paradigms for nonprogrammer web application development." *Symposium on Human-Centric Languages and Environments.* IEEE, 2003. 23-30.

Rode, J., E. Toye, and A. Blackwell. "The fuzzy felt ethnography - understanding the programming patterns of domestic appliances." *Personal and Ubiquitous Computing 8*, 2004: 161-176.

Rode, J., Y. Bhardwaj, M. Prez-Quinones, M. Rosson, and J. Howarth. "As easy as "Click": End-user web engineering." *International Conference on Web Engineering.* Berlin, Germany: Springer-Verlag, 2005. 263-274.

Romero, P., B. du Boulay, R. Cox, R. Lutz, and S. Bryant. "Debugging strategies and tactics in a multi-representations software environment." *International Journal on Human-Computer Studies*, 2007: 992-1009.

Rosson, M., H. Sinha, M. Bhattacharya, and D. Xhao. "Design planning in end-user web development." *IEEE Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2007. 189-196.

Rothermel, G., M. Burnett, L. Li, C. DuPuis, and A. Sheretov. "A methodology for testing spreadsheets." *Transactions on Software Engineering and Methodology 10(1)*, 2001: 110-147.

Rowe, M. *Teaching science as continuous inquiry: A basic.* New York, NY: McGraw-Hill, 1978.

Russell, D., M. Stefik, P. Pirolli, and S. Card. "The cost structure of sensemaking." *Conference on Human Factors in Computing Systems.* ACM, 1993. 269-276.

Ruthruff, J., A. Phalgune, L. Beckwith, M. Burnett, and C. Cook. "Rewarding good behavior: End-user debugging and rewards." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2004. 115-122.

Ruthruff, J., M. Burnett, and G. Rothermel. "An empirical study of fault localization for end-user programmers." *International Conference on Software Engineering.* ACM, 2005. 352-361.

Sajianemi, J. "Modeling spreadsheet audit: A rigorous approach to automatic visualization." *Journal on Visual Languages and Computing 11(1)*, 2000: 49-82.

Sandstrom, N., J. Kaufman, and S. Huettel. "Differential cue use by males and females in a virtual environment navigation task." *Cognitive Brain Research 6*, 1998: 351-360.

Scaffidi, C., M. Shaw, and B. Myers. "Estimating the number of end users and end-user programmers." *Symposium on Visual Languages and Human-Centric Computing.* IEEE Computer Society, 2005. 207-214.

Schon, D. *The reflective practitioner: How professionals think in action.* New York: Basic Books, 1983.

Seno, M., and G. Karypis. "SLPMiner: An algorithm for finding frequent sequential patterns using length decreasing support constraint." *International Conference on Data Mining.* 2002. 418-425.

Shiozawa, H., K. Okada, and Y. Matsushita. "3D interactive visualization for inter-cell dependencies of spreadsheets." *Symposium on Information Visualization.* IEEE, 1999. 79.

Shrager, J., and D. Klahr. "Instructionless learning about a complex device: the paradigm and observations." *International Journal of Man-Machine Studies 25*, 1986: 153-189.

Simon, H. "The structure of ill-structured problems." *Artificial Intelligence 4*, 1973: 181-202.

Smith, G., and G. Browne. "Conceptual foundations of design problem solving." *Transactions on Systems, Man and Cybernetics 23(5)*, 1993: 1209-1219.

Spreadsheet Basic (spBasic). 2007. http://spbasic.com/ (accessed August 15, 2009).

Stefik, M., et al. *The knowledge sharing challenge: The sensemaking white paper.* http://www2.parc.com/istl/groups/hdi/papers/sensemaking-whitepaper.pdf: Xerox PARC, 2002.

Storey, M., J. Ryall, R Bull, D. Myers, and J. Singer. "TODO or to bug: Exploring how task annotations play a role in the work practices of software developers." *International conference on Software Engineering.* ACM, 2008. 251-260.

Strauss, A., and J. Corbin. *Basics of qualitative research: Techniques and procedures for developing grounded theory.* Thousand oaks, CA: SAGE Publications, 1998.

Subrahmaniyan, N., et al. "Explaining debugging strategies to end-user programmers." *Symposium on Visual Languages and Human-Centric Computing.* IEEE, 2007. 127-134.

—. "Testing vs. code inspection vs. what else? Male and female end users' debugging strategies. ." *Conference on Human Factors in Computing Systems.* ACM, 2008. 617-626.

Sutherland, I. "Sketchpad: A Man-Machine Graphical Communication System." *Proceedings of the SHARE Design Automation Workshop.* 1964. 6.329-6.346.

Torkzadeh, G., and X. Koufteros. "Factorial validity of a computer self-efficacy scale and the impact of computer training." *Educational and Psychological Measurement 54(3)*, 1994: 813-821.

Turkle, S. "Computational reticence: Why women fear the intimate machine." In *Technology and Women's Voices*, by C. Kramerae, 41-61. 1988.

Venolia, G. *Textual allusions to artifacts in software-related repositories.* MSR-TR-2006-73, 2006.

Visual Basic for Applications. 2009. http://msdn.microsoft.com/en-us/isv/bb190538.aspx (accessed August 15, 2009).

Wagner, E., and H. Lieberman. "Supporting user hypotheses in problem diagnosis on the web and elsewhere." *International Conference on Intelligent User Interfaces.* ACM Press, 2004. 30-37.

Weaver, W., and C. Shannon. *The mathematical theory of communication.* Urbana, Illinois: University of Illinois Press, 1949.

Weiser, M. "Programmers use slices when debugging." *Communications of the ACM 25(7)*, 1982: 446-452.

Whitaker, S., R. Cox, and S. Gribble. "Configuration debugging as search: Finding the needle in the haystack." *Symposium on Operating System Design and Implementation.* 2004. 77-90.

Whittaker, D. "Spreadsheet errors and techniques for finding them." *Management Account 77(9)*, 1999: 50-51.

Wilcox, E., J. Atwood, M. Burnett, J. Cadiz, and C. Cook. "Does continuous visual feedback aid debugging in direct-manipulation programming systems?" *Conference on Human Factors in Computing Systems.* ACM, 1997. 258-265.

Wilson, A., et al. "Harnessing curiosity to increase correctness in end-user programming." *Conference on Human Factors in Computer Systems.* ACM, 2003. 305-312.

Windows PowerShell. http://en.wikipedia.org/wiki/Powershell (accessed August 20, 2008).

Wing, J. "Computational thinking." *Communications of the ACM 39(3)*, 2006: 33-35.

Wolber, D., Y. Su, and Y. Chiang. "Designing dynamic web pages and persistence in the WYSIWYG interface." *International Conference on Intelligent User Interaces.* 2002. 228-229.

Ye, Y., and G. Fischer. "Supporting reuse by delivering task-relevant and personalized information." *Proceedings of the International Conference on Software Engineering.* ACM, 2002. 513-523.

Yuan, C., et al. "Automated known problem diagnosis with event traces." European Conference on Computer Systems. ACM, 2006. 375-388.

APPENDIX

Figure 34. To-do listing details

Finding	Evidence
To-do listing is an end-user debugging stratagem.	**Forms/3** participants repurposed a checkbox feature to keep track of items done and to-do: *"…find errors using checks and X's to show me what I'd already checked"* (Ch. 5). **PowerShell** participants employed breakpoints, opening/closing files, and used pen and paper to do the same (Ch. 6).
To-do listing is poorly supported by today's end-user programming environments.	None of the PowerShell, Forms/3, and **Excel** environments provides explicit support for to-do listing (Ch. 5, Ch. 6, and Ch. 8). Excel, to-do listing was the only applicable stratagem which was not used (Ch. 8).
Support for to-do listing is requested by both end-user and professional developers alike, male and female.	For **end-user programmers**: In 2006 interviews by the spreadsheet analysis company RedRover Software, all (male) interviewed auditors suggested the equivalent of to-do tracking of code inspection (personal communication Aug. 29, 2007 with Matthew Johnen, CEO of RedRover Software) (Ch. 5). Forms/3 participants and IT professionals repurposed features to keep track of items to-do, done, and unchecked (Ch. 5 and Ch. 6). [Storey et al. 2008] provide an overview of the important role task annotations (such as "TODO" comments) play in the work practices of **professional software developers** in managing their tasks, and also of the poor support for to-do listing even in integrated development environments (Ch. 8).
Relying on existing features to be repurposed is dangerous.	While users currently resort to repurposing features such as comments and testing checkmarks for to-do listing, this is not enough. Repurposing features can lead to loss of formatting information and can be ineffective for today's complex real-world spreadsheets. This might be why none of the eight participants observed debugging in Excel employed the to-do listing stratagem (Ch. 8).
To-do listing shows promise for increasing end-user programmers' debugging effectiveness.	Code inspection was a female success stratagem in the Forms/3 (Ch. 5) and PowerShell Stratagem (Ch. 6) Studies, can help participants systematize code inspection. In the Sensemaking Study, Participant SF was a comprehensive processor (Ch. 7), and might have benefited from the ability to systematically keep track of done, to-do, and unchecked code. To-do listing might have helped nudge her into attempting to fix the harder bugs she forgot about when her comprehensive approach stopped being productive. Participant SM might have also benefitted from to-do listing by better being able to keep track of the cells he selected to check and those he did not.
While the former are all about *why* to support to-do listing, the Excel Strategy Items Study also helped reveal *how* to do so.	The tool should be designed specifically for to-do listing (e.g., no loss of formatting) and: highlight formula cells, filter out some of the false-positive feedback, work for multiple worksheets, display related information (especially by supporting all ten stratagems) in context, generate a list of recommended items to check, allow users to change an item's status to either "done" or "to-do" and back, support both comprehensive and selective strategies, and supporting creation of a schema in the ways identified in Ch. 8.