Supporting End-User Debugging: What Do Users Want to Know?

Cory Kissinger[1], Margaret Burnett[1], Simone Stumpf[1], Neeraja Subrahmaniyan[1], Laura Beckwith[1], Sherry Yang[2], and Mary Beth Rosson[3]

[1]Oregon State University
Corvallis, Oregon, USA
ckissin@eecs.oregonstate.edu

[2]Oregon Institute of Technology
Klamath Falls, Oregon, USA
Sherry.Yang@oit.edu

[3]Pennsylvania State University
State College, Pennsylvania, USA
mrosson@ist.psu.edu

ABSTRACT

Although researchers have begun to explicitly support end-user programmers' debugging by providing information to help them find bugs, there is little research addressing the right content to communicate to these users. The specific semantic content of these debugging communications matters because, if the users are not actually *seeking* the information the system is providing, they are not likely to attend to it. This paper reports a formative empirical study that sheds light on what end users actually want to know in the course of debugging a spreadsheet, given the availability of a set of interactive visual testing and debugging features. Our results provide insights into end-user debuggers' information gaps, and further suggest opportunities to improve end-user debugging systems' support for the things end-user debuggers actually want to know.

Categories and Subject Descriptors

D.1.7 [Programming Techniques]: Visual Programming; D.2.5 [Software Engineering]: Testing and Debugging-Debugging Aids; D.2.6 [Software Engineering]: Programming Environments-Interactive environments; H.1.2 [Information Systems]: User/Machine Systems—Software psychology; H.4.1 [Information Systems Applications]: Office Automation—Spreadsheets; H.5.2 [Information Interfaces and Presentation) —User Interfaces (D.2.2, H.1.2, I.3.6).

General Terms

Design, Reliability, Human Factors

Keywords

End-user software engineering, end-user programming, end-user development, end-user debugging, online help.

1. INTRODUCTION

Research on end-user programming has, in the past, concentrated primarily on supporting end users' creation of new programs. But recently, researchers have begun to consider how to assist end users in debugging these programs (e.g., [1, 7, 14, 17, 22, 23]).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AVI '06, May 23-26, 2006, Venezia, Italy.
Copyright 2006 ACM 1-59593-353-0/06/0005...$5.00.

Support for end users in debugging tasks is often *problem-oriented*; the system tries to discover candidate bugs, communicate these to the users, and provide user interaction mechanisms to explore and correct the bugs. The communication about the bugs may be delivered through diagrams, color highlighting, or similar devices.

Because the user may not understand how to respond to such information displays, a debugging support system may also provide *feature-oriented* information that explains how to interpret or use the debugging features. Feature-oriented information is typically provided by user interface mechanisms that are tied to the feature in question, such as pop-up tool tips, linked help pages, video demonstration snippets, and so on. In this paper, we refer to the collection of support communications, both problem-oriented and feature-oriented, as the system's *explanations*.

Some existing debugging explanation techniques for end-user programmers have been empirically linked to debugging success. However, many of these empirical results are so focused on the success of particular features, they do not provide much general guidance to future designers of end-user debugging support, such as what needs to be explained, when, and in what context.

However, a few studies do provide some general guidance for end-user debugging explanations. Natural Programming studies for event-based Alice programs [9] revealed that 68% of the questions asked by the participants (HCI students with varying amounts of programming background) during debugging in that language were "why did" or "why didn't" questions [14]. The Surprise-Explain-Reward strategy [25] has been studied in the spreadsheet paradigm; the work on this strategy provides general guidance regarding interruption styles for communicating about end-user debugging situations [21] as well as effective reward communications in these situations [22]. Finally, because end users may not have experience with debugging support tools, they may be forced to learn about these features as they work, suggesting that studies of what online learners want to know may be helpful (e.g., [2, 20]).

This paper builds upon these previous works to help fill a critical gap in what is known about end-user debugging support: the *semantic content* of what should be explained to *end users* to support *debugging*.

Determining the semantic content needed by an end user when debugging might seem straightforward. For instance, a system could simply describe all visible features and feedback; this is a common approach to information system design. In our research prototype designed to support end-user debugging, explanations such as these have indeed been created for all visible features. We have put significant research into the semantic content of the explanations, refining them on the basis of both theory (the model of Attention Investment [6] and Minimalist Learning Theory [8]) and empirical work. Despite these efforts, it appears that the explanations are not answering what users want to know. For example, one user in a recent study commented as follows [4]:

> Interviewer: "Weren't the tool tips helpful?"
>
> S3: "Yeah, they were good but sometimes I didn't find the answer that I wanted…I needed more answers than were present."

Herein lies the problem. Little is known about what information end-user debuggers such as S3 actually want to know. This paper sets out to provide this kind of information.

In this paper, we analyze *information gap instances*—utterances expressing an absence of information—expressed by end-user programmers working on spreadsheet debugging tasks. Participants interacted with a spreadsheet environment that contained visual features providing problem-oriented debugging information (e.g., visual colorings of cells that need testing), but that provided no feature-oriented explanations to help users make profitable use of them. Within this debugging context, we investigated the following research question:

When debugging, what do end-user programmers want to know?

2. EXPERIMENT

The experiment procedure was a think-aloud using pairs of participants. The goal was to allow the participants at least a possibility of succeeding at their spreadsheet debugging, so that they would stay motivated, but without including explanations that might bias the content of the participants' information gaps.

To achieve this balance, we removed all feature-oriented information about how the debugging features worked, as we have already mentioned. We administered a tutorial to give just enough instruction to our participants to be able to perform basic functions in the particular environment (a research spreadsheet system). Finally, when the participants began their tasks, we removed the only remaining source of support from the room, the researcher himself. The participants were recorded (video and audio) and their screen state was continuously captured along with all instant messenger dialogue (explained below). Figure 1 shows what the researcher observed remotely.

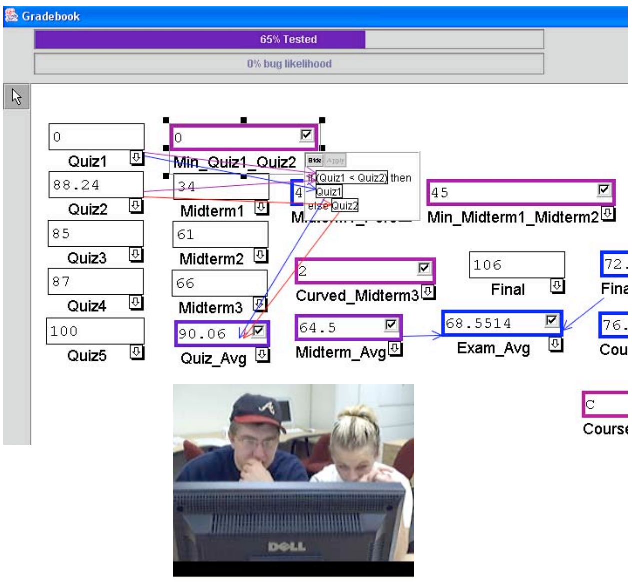

Figure 1: Experiment data capture example.

With so little information, participants could have become "stuck," at which time their think-aloud verbalizations would cease to be useful. To avert this situation, we provided mechanisms for the participants to obtain information. Although they had both received the same training, the most accessible information to a participant was his or her partner. This encouraged them to keep talking to each other, which turned out to be the primary way they worked through their information gaps.

A slightly less accessible, but potentially more valuable, source was an instant messenger dialogue between the pair and the researcher, with which the participants could ask questions. Since the researcher was out of the room, the questioner had to include relevant context information, avoiding simple "What's that?" questions. The cost of waiting for the researcher to answer this sort of question (typically 10 seconds), made discussion between the pairs less costly than using the instant messenger, in terms of time and effort. Researcher responses were restricted to the set of feature-oriented explanations that had been removed from the environment for purposes of the experiment. The researcher could also send a hint if the participants expressed confusion about a particular feature and refused to move on. Pairs typically received one such hint. The participants also had three "wild cards," which could be used as a last resort, to bring the researcher back into the room to provide a hint on how to make progress. (The participants rarely used the wild cards and only occasionally used the instant messenger.)

2.1 Participants

We chose the pair think-aloud protocol because it is particularly well suited to eliciting participants' verbalizations of problem-solving thoughts. This set-up also creates a different social context than for individuals working alone, but since collaborative debugging among spreadsheet users is extremely common [18], it does not introduce validity concerns. Because we wanted participants to feel comfortable talking together, we recruited only *pairs* of participants. This mechanism ensured that each pair already knew each other.

Eleven of the fourteen participants were business majors. The other three were in education, industrial engineering, and nutrition, none of whom were paired with each other. None of the participants had programming experience beyond a first level programming course. Gender was distributed equally, with two male-male, two female-female, and three male-female pairs.

2.2 Environment

The debugging features that were present in this experiment were a subset 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" or "X out" values that are correct or incorrect, respectively [7]. In WYSIWYT, untested cells have red borders. Whenever users notice a correct value, they can place a checkmark (√) in the decision box at the corner of the cell they observe to be correct. As a cell becomes more tested, the cell's border becomes more blue. (Figure 1 includes many cells partially or fully tested.)

Instead of noticing that a cell's value is correct, the user might notice that the value is incorrect. In this case, instead of checking off the value, the user can X-out the value. X-marks trigger fault likelihood calculations, which cause the interior of cells suspected of containing faults to be colored in shades along a yellow-orange continuum.

In addition, arrows that allow users to see the dataflow relationships between cells also reflect WYSIWYT "testedness" status at a finer level of detail. The optional dataflow arrows are colored to reflect testedness of specific relationships between cells and subexpressions. In Figure 1, the participant has turned on the arrows for the `Min_Quiz1_Quiz2` and the `Exam_Avg` cells.

2.3 Tutorial

The goals of the tutorial were to familiarize the participants with the think-aloud procedure, explain pair-programming guidelines, and to familiarize participants with the environment enough to proceed with their debugging task.

The tutorial began with a think-aloud practice where the pair verbalized a task that they had recently worked on together, namely finding their way to the experiment. The researcher also provided basic pair-programming guidelines. Specifically, the participants were told they would switch between two roles: the *driver*, controlling the mouse and keyboard, and the *reviewer*, who contributes actively to problem solving. In the experiment, they switched roles every ten minutes.

The brief tutorial on the environment was hands-on, with the pair working on a sample spreadsheet problem together at the same machine. Participants learned mechanics of changing input values and editing formulas, as well as mechanics of the unique actions available in the environment: namely, placing checkmarks, placing X-marks, and turning arrows on and off. For example, participants were instructed to "middle-click" on a cell to bring up the cell's arrows. However, the tutorial did not explain how to interpret the visual feedback they received as a result.

2.4 Tasks

We asked participants to test two spreadsheets, `Gradebook` and `Payroll`. We replicated the spreadsheets and the seeded faults of [5].

Participants had time limits of 20 and 40 minutes for `Gradebook` and `Payroll` respectively. These simulated the time constraints that often govern real-world computing tasks, and also prevented potential confounds, such as participants spending too much time on the first task or not enough time on the second task. The participants were given more time on the `Payroll` task because it was the more difficult of the two due to its larger size, greater length of dataflow chains, intertwined dataflow relationships, and more difficult faults. All participants performed the (easier) `Gradebook` task first to allow a gradual introduction to the environment before the more challenging `Payroll` task. The participants were instructed, "Test the … spreadsheet to see if it works correctly and correct any errors you find."

3. METHODOLOGY

The methodology we adopted consisted of four main activities: the segmentation of the data into topic-related units of utterances, the development of a coding scheme through a bottom-up organization of the units, the application of the codes to the data, and the calculation of agreement measures to evaluate the stability and robustness of the resulting coding scheme.

3.1 Segmentation of the Data

The primary data were audio recordings of participants' utterances, synchronized with video recordings of their physical behavior and screen states. To create an integrated data record, the audio recordings were transcribed and supplemented with context obtained from the video and screen data (e.g., gestures and actions). Because a single utterance alone does not allow an analysis to be sensitive to common context and thread of discussion, the transcripts were segmented into *stanzas* [12]. A stanza, typically about 8-15 lines, is a unit of utterances that occurs between shifts of topic (see Figure 2 for an example).

F. Let's change everything. [tries changing formula]
E. Yeah, but we got the right answer.
F. Did that change the answer at all?
F. Oh wait, did I change the symbol? [changes the formula]
F. Oh, now we're down to 30 percent tested.
F. I wonder if I go like that- [changes the formula back]
F. Oh no, crazy.
F. Oh, I guess now it's just this again. [checks off cells that changed]
F. I don't get how you get to 100%, it's like a test you can't pass. Every time I do this it gets lower.
E. Yeah, I don't know.
F. Well, that's confusing. (.)
Switch places.

Figure 2: A stanza in which participants E and F discuss the debugging strategy of "changing everything". Notation: [actions] denote actions taken, (.) a pause in speaking, and *italics* the researcher's instant message to the participants.

3.2 Deriving the Codes

The goal of the codes was to support analysis of participants' information gaps. We considered an information gap to have occurred when a participant asked a question, stated a tentative hypothesis, expressed surprise, made a judgment about whether an information gap was present, or provided an explanation to his or her partner (implying that the partner had an information gap). We refer to such utterances as *information gap instances*.

The research literature does not report coding schemes that are directly applicable to the information gaps experienced by end-user debuggers. Most studies of information gaps focus on users in learning or tutorial situations (e.g., [2, 20]). In contrast, we are interested in the just-in-time learning users undergo to enhance their *productivity*, i.e., to make progress in solving a problem; we will return to the relationships between our coding scheme and others' in Section 5. In trying to increase productivity, the user must balance the costs of learning a new technique—which may or may not be relevant to a task—against its potential benefits for task performance. In these circumstances, learning may be just one of a set of competing goals.

To develop a coding scheme that matched our aims, we first grouped stanzas from two of the transcripts into an affinity diagram[1], adjusting the concepts and relations as we progressed. This grouping process focused us on *types* of information gaps, as we

[1] A group decision-making technique designed to sort a large number of items into "related" groups, from the perspective of those doing the sorting.

Table 1: The coding scheme.

Code	Description	Examples
Feature/ Feedback	Question or statement expressing general lack of understanding of the meaning of a specific visual feedback or action item, but with no goal stated.	"So with the border, does purple mean it's straight-up right and blue means it's not right?"
Explanation	Explanation to help partner overcome an information gap. The explanation may be right or wrong.	"<border color> just has to do with how much you've been messing around with it."
Whoa	Exclamation of surprise or of being overwhelmed by the system's behavior.	"Whoa."
Help	Question or statement explicitly about the need for additional help.	"Help."
Self-Judgment	Question or statement containing the words "I" or "we," explicitly judging the participant or the pair's mastery of the environment or task.	"I'm not sure if we're qualified to do this problem."
Oracle/ Specification	Question or statement reasoning about a value and/or a formula.	"Divided by 10? I don't know...I guess it should be times 10."
Concept	Question about an abstract concept, as opposed to a question about a concrete feature/feedback item on the screen.	"What does 'tested' mean?"
Strategy Question	Explicitly asks about what would be a suitable process or what to do next.	"What should we do now"
How Goal	Asks how to accomplish an explicitly stated goal or desired action. (An instance of Norman's Gulf of Execution [19].)	"How do you get 100%?"
Strategy Hypothesis	Suggests a hypothesized suitable strategy or next step to their partner.	"Let's type it in, see what happens."

compared and organized information gap instances according to their semantic content such as a question about what might be a suitable strategy. As types of information gaps were identified, descriptions and example utterances for each candidate coding category were collated. The coding scheme was applied to one transcript repeatedly with different coders each time. The codes were refined to be less ambiguous after each application until we achieved acceptable agreement (above 80%, see below for calculation of agreement) across coders. The 10 codes we identified in this fashion are described in Table 1.

3.3 Application and Agreement

Two of the authors independently coded all of the transcripts. Multiple codes were allowed per stanza, as there may have been multiple information gap instances contained in a group of related utterances. The coders discussed their initial coding decisions and made changes if they agreed that a code had been inadvertently overlooked or misapplied.

A widely used rule of thumb is that 80% agreement or higher between coders indicates a reasonably robust coding scheme. Because more than one code could be placed on a stanza, the calculation of agreement for a particular stanza required comparing two sets of codes (one from each coder). The percentage of agreement for a stanza was calculated by dividing the size of the intersection by the size of the union. For example, if one rater coded a stanza {Help, Self-Judgment} and the second coded it {Strategy Hypothesis, Self-Judgment}, then the agreement for that stanza would be |{Strategy Hypothesis, Self-Judgment} ∩ {Help, Self-Judgment}| / |{Strategy Hypothesis, Self-Judgment} ∪ {Help, Self-Judgment}| = 1/3 = 33%. The average of all 425 coded stanzas resulted in 90% agreement.

4. RESULTS

Table 2 lists the frequencies of each type of information gap found in the 425 stanzas, and Figure 3 shows the distribution over time. (One pair was excluded from the time graphs, since their overall time was considerably less than that the others'.)

4.1 Questions and Explanations about Features and Feedback

A widely used approach to introducing users to a new interface is to provide information about the meaning of features: both user actions available, such as options user can select, and feedback items they might receive, such as red underlines under misspelled words. This information is often contained in tool tips and/or online help systems organized by feature.

In our study, the information gaps that are satisfied by this kind of explanation were observed as questions participants asked about what specific features mean, such as "What does the purple border mean?" (type Feature/Feedback), and as explanations of a specific feature's meaning by one participant to the other, such as "I think the purple means it's wrong" (type Explanation). (Explanations suggest an information gap because they imply that one participant thinks the other is lacking this information.)

As Figure 3 indicates, feature-oriented gaps were highest at the end of the experiment. Still, as can be seen in Table 2, the combined percentage of feature-oriented information gaps was a surprisingly low 16%. Recall that, as advanced business students, the participants had fixed bugs in spreadsheets before, seemingly leaving only orientation to the unfamiliar interface as a barrier. Yet, few of their information gaps were about the interface, despite our removal of feature-oriented support.

Practical implications: End-user debugging explanation approaches that center mainly on the meaning of the system's features and feedback—a common strategy in online explanation systems—would address only a fraction of what our participants wanted to know.

4.2 Big Information Gaps: Whoa! Help!

Table 2: Code frequencies.

Code	Count (Percent of Total)
Features/Feedback:	
Feature/Feedback (questions)	77 (10%)
Explanation	48 (6%)
Big Information Gap:	
Whoa	14 (2%)
Help	23 (3%)
Self-Judgment	67 (9%)
Oracle/Specification	316 (40%)
Strategy:	
Concept	8 (1%)
Strategy Question	39 (5%)
How Goal	20 (2%)
Strategy Hypothesis	169 (22%)

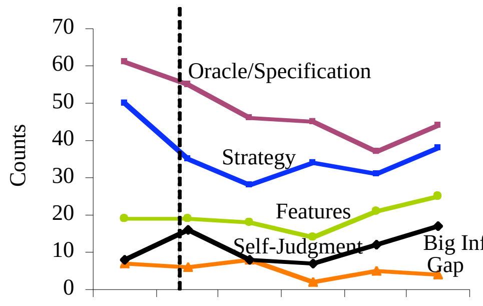

Figure 3: Code frequency within each 10-minute interval. Task 2 (Payroll) began after 20 minutes.

"Whoa!" Approximately 2% of the responses expressed surprise and confusion at feedback that had just occurred on the screen. Information gaps of type Whoa were often in response to several visible changes occurring at once, such as turning on dataflow arrows (Figure 1). Another 3% of the information gaps explicitly expressed a general need for help (coded Help), implying that there was a need for more information to even be able to verbalize a more specific question. Both of these types expressed a lack of clues about the current situation or what to do about it. These results are good reminders that sometimes when a user needs explanations, a more specific question does not readily occur to them. In our study, this happened 5% of the time.

Practical implications: A look at the timing of the Whoa and Help instances provides some guidance as to how a debugging explanation system might address this type of information gap. First, note in Figure 4 that the general requests for information (type Help) were greatest at the beginning of the first task when little was known about the environment and task, and at the 50-minute point (30 minutes into the more difficult spreadsheet). This timing suggests that end-user debuggers may need more broad-based support at the beginning of the task and in moments of particular difficulty, such as suggesting ideas to help the users (re-)connect to

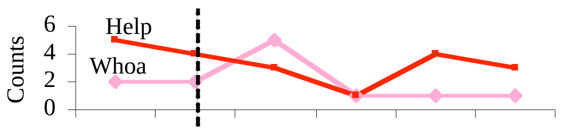

Figure 4: Frequency of Help and Whoa codes within each 10-minute interval.

features or strategies that may help them.

Second, as Figure 4 shows, type Whoa occurred mostly in the middle of the experiment. At this time participants had enough experience to form an early mental model of how the environment worked. However, the application of this model during the more difficult task pointed out a serious misconception. According to research into the psychology of curiosity [16], moments of surprise such as these are opportune times for explanations, as people curious about such surprises seek to satisfy the information gap that led to the surprise. An explanation system that kept track of the amount the user has used the features and the amount of recent feedback may be able to determine whether a generic "help!" button push is more likely to be the result of a type Help versus a type Whoa information gap. In our study, for the Whoa type of information gap, a look at the system state sometimes revealed the likely cause of confusion. In these cases, a context-sensitive explanation system might successfully respond to Whoa requests by providing assistance on the most recent feedback.

4.3 Self-Judgments: Am I smart enough to succeed at this task?

Of the participants' information gap instances, 9% were self-judgments of their own mastery of the system or of the debugging task, suggesting that self-judgment was a significant factor in their cognitive processing as they worked on the bugs. These self-judgments are instances of metacognition, in which a learner monitors the success of his or her own learning processes [11]. Metacognitive activity is well-established as an important influence on learning and understanding [24].

These judgments also provide a view of the participants' self-efficacy. Self-efficacy is a person's belief that they will succeed at accomplishing a specific task, even in the face of obstacles [3]. According to self-efficacy theory, the amount of effort put forth is impacted by a person's self-efficacy. In our own work with self-efficacy, we have seen it have a significant effect on end users' willingness to use advanced debugging features [5]. In that work, some users' self-efficacy was much lower than warranted, particularly among females. In previous studies as well as in the current one, we have also observed examples of participants overrating their own performance, saying things like "We did it right" when in fact they had not. Both overrating and underrating performance may point to failures of the system to provide accurate feedback regarding the users' debugging progress.

Practical implications: An effective explanation system that succeeds at fulfilling end-user debuggers' information gaps may also improve the accuracy of users' self-judgments. Due to the effects of self-efficacy and metacognition, this in turn may help increase debugging success simply by helping users persist in their efforts.

4.4 Oracle and Specification Questions: Is this the right value/formula?

In debugging a spreadsheet, it may not always be clear to users whether or not a *value* is correct. In software engineering, difficulty determining whether a value is right or wrong is called the "oracle problem." The oracle problem is important because its presence weakens many of the user's problem-solving devices such as the power of immediate visual feedback, user tinkering, and testing behaviors. After all, these behaviors are not helpful when the user cannot tell whether the result is right or wrong.

"How do we know if that's right or not?" This information gap instance not only shows one example of the oracle problem occurring, it also expresses a request for information about *how* to decide whether a value is right. A closely related problem that arises in debugging is whether the *formula* ("source code") correctly implements the specifications or, if the user has already determined that it does not, how to make it do so: "So the average, why is it divided by 3?"

In our study, 40% of the information gap instances fell into the Oracle/Specification category. Note that this large fraction of the total set of questions is about the task (debugging), not about the features or the system. This is consistent with Carroll and Rosson's description of the "active user" [8], who focuses much more on the task at hand than on the availability of potentially interesting user interface features.

Practical implications: Some information gap instances of this type centered on a particular cell, such as "We need some more money for this…we're missing $300." Such instances may be well served by an explanation that suggests changes to a spreadsheet to produce a desired output, such as the direction of Abraham and Erwig's goal-based debugging suggestions [1], or by an approach that explicitly supports investigation into the reason for a specific value or event, as with Ko and Myers's Whyline work [14]. Other information gap instances encompassed a larger subset of the spreadsheet, such as "Where is it getting the wrong math here?" One possible solution to this type of question might be to remind the user of narrowing-down techniques such as WYSIWYT with fault localization [7].

4.5 Strategy: What should we do?

Fully 30% of the information gap instances pertained to strategy issues. There were four codes relating to strategy: Concept, Strategy Question, How Goal, and Strategy Hypothesis. The primary type in this group at every time period was Strategy Hypothesis (Figure 5), in which participants actively hypothesized strategies, which they usually proceeded to try out. This again calls to mind the active user—one who seeks mainly information directly pertinent to their goal. Type Strategy Hypothesis alone accounted for

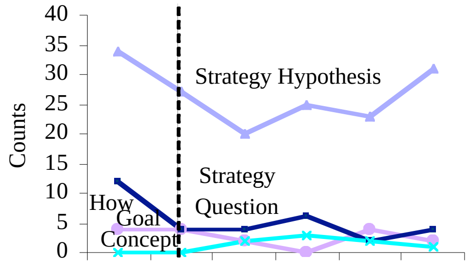

Figure 5: Frequency of strategy codes within each 10-minute interval.

22% of the information gap instances.

Practical implications: Most of the strategy information gap instances were global in nature, rather than being about a particular feature (e.g. "What should we do next?"). Due to the lack of a contextual tie, a feature-anchored explanation such as a tool tip seems a poor fit for this sort of information gap. Even so, some of the remarks, while global, still had ties to particular features. For example, "What's testing?" (Concept) could, in our setting, be answered in explanations of the testing features, and "How do we get to 100%?" (How Goal) might be answered in explanations of the testing progress indicator (top of the spreadsheet environment in Figure 1).

This group of information gaps presents a good opportunity for improvement in end-user debugging explanations. In some help systems, strategy is addressed in separate tutorials about a system's usage, but this seems an inappropriate solution given the active users our participants appear to be. The key may lie in linking feature-located and feature-centric explanations with broader explanations that tie the use of features into strategic goals. In [4] the use of broader help information in expandable tool tips, including some coverage of strategy, was proposed. This is an example of the "layered" approach to explanations recommended by [10] for use in minimalist instruction aimed at active users; given our observations of participants' active debugging style, this may be a step in the right direction.

4.6 Implications of Co-occurrences

Two code types co-occurred in the same stanza with certain other types an inordinate number of times: Self-Judgments, and Strategy Hypotheses.

A majority (64%) of the Self-Judgments occurred in stanzas also showing Oracle/Specification information gaps, as Figure 6 illustrates. Also, 47% co-occurred with Strategy Hypotheses. These were far ahead of the third most common co-occurrence, at only 19%, with Features/Feedback, not shown in the figure. (The percentages exceed 100% because more than two codes sometimes occurred in a single stanza.) This suggests that the most appropriate places for debugging explanations to attempt to improve users' ability to self-judge will be in the context of problem-oriented communications and with strategy-oriented communications. In particular, it appears that a system's feature explanations are not likely to be the right context for assisting users make more accurate self-judgments of their performance.

Furthermore, 70% of the Strategy Hypothesis instances co-occurred with Oracle/Specification instances, implying that participants' main interest in strategy was in applying it to the problem domain, as opposed to building it up with the features as

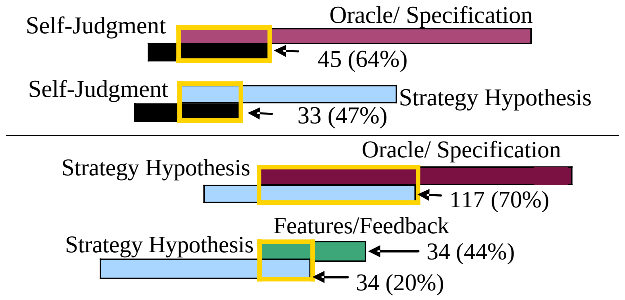

Figure 6: The top two code co-occurrences for (top) Self-Judgments and (bottom) Strategy Hypotheses.

a starting point. The second-ranked co-occurrence was with Features/Feedback at only 20% of the Strategy Hypotheses. However, the flip side of this runner-up was that 44% of the Feature/Feedback information gap instances included Strategy Hypotheses, implying that feature-centric "hooks" to strategy hints would be welcomed by users—but would not alone be enough, since they would still leave 80% of the Strategy Hypothesis gaps unfilled.

5. COMPARISON TO OTHER WORK

Researchers have developed coding schemes for users' questions and comments in settings other than end-user debugging. To help understand what might be end users' unique needs in the debugging context, we first considered and now compare our coding scheme to several others.

We searched the literature for work that coded some form of information gap. The most relevant works (Table 3) centered on questions people asked and barriers they experienced. Anthony et al. [2] analyzed the questions students directed to a simulated algebra tutor. Person and Graesser [20] summarized a number of studies examining human-human tutoring dialogs. Gordon and Gill [13] analyzed questions designed for knowledge elicitation from domain experts [13]. Ko and Myers [15] studied a usage context somewhat similar to our own—problems experienced by novices learning a programming language (Visual Basic.Net).

To compare our coding scheme with these, we studied the category description and illustrative examples for each question or comment category, to determine similarity to one or more of our codes. We used a relatively liberal decision rule in identifying overlap—if we could find multiple data instances from one of our own categories that would have been captured by a code in another scheme, we labeled it as a "match."

Despite the variation in information and task context, we identified some overlap for all of our codes except two—Help and Explanation. The lack of overlap for these codes may be partially due to our experimental set-up that reflected collaborative end-user debugging: for example, working with a partner probably encourages users to offer explanations to one another.

The codes having most overlap with other coding schemes were Feature/Feedback and How Goal. In fact, the How Goal code overlapped with multiple categories in three other schemes, pointing to users' general need for goal-specific procedural information. Recall, however, that this category was not very prevalent in our data (only 2%), suggesting that for our debugging task, the users either already knew how to perform many specific procedures or did not even know enough to explicitly state a goal.

Instead of asking about specific procedures, our users often seemed to operate at the more abstract level of goal-setting, such as asking about a suitable process to follow (Strategy Question) or making a goal-setting proposal to the partner (Strategy Hypothesis). Together these two codes accounted for 27% of our data, but we found little overlap between these codes and the other schemes. The clearest case of overlap is with Ko and Myers [15], who created a "Design" code to classify novice programmer problems that are inherent to programming and distinct from language mechanisms. These researchers' setting was similar to ours because it contained aspects likely to be unfamiliar to users while also being challenging enough that they sometimes needed help just to identify reasonable goals.

Only one of the other schemes overlapped with the Self-Judgment code (a judgment about the mastery level of "I" or "we"). The importance of reflection about one's own knowledge state (metacognition) in learning and problem solving is well-established [11]. It is not yet clear what sorts of cognitive or social settings are most likely to evoke reflection about one's capacities during problem-solving episodes. Perhaps collaborative work situations, as in our experiment, encourage self- or pair-evaluation as a sort of knowledge calibration mechanism; alternatively it may simply be that other researchers have been less attuned to metacognition and thus have made no analogous distinctions in their coding.

6. CONCLUSION

This paper has presented a pair think-aloud study aimed at capturing the information gaps arising for end users in the course of debugging spreadsheets. While further investigation is needed to determine the generality of the results to other settings, there were several implications that seem applicable to a variety of end-user debugging systems. To summarize:

- Unlike the practices in many software systems, debugging explanations for end-user programmers should not be primarily focused on how the debugging features work. In our study, feature-oriented explanations would address only a fraction of what our participants wanted to know.
- The greatest need for explanations fell in the Oracle/Specifications category: figuring out whether a value was right or wrong, whether a particular snippet of code (formula) was right or wrong, and how to fix values and formulas that were wrong. The prevalence of this category points to a need

Table 3: Overlap of our codes with others' coding schemes.

Our Codes Goal: Find out what end-user debuggers want to know	**Anthony et al. [2]** Goal: Design intelligent tutor based on student questions	**Person/Graesser [20]** Goal: Design intelligent tutor based on human-tutor dialogue	**Gordon/Gill [13]** Goal: Knowledge acquisition from experts for information systems	**Ko/Myers [15]** Goal: Describe barriers in learning a programming language
Feature/Feedback	Interface	N/A	Event, State	Understanding
Explanation	N/A	N/A	N/A	N/A
Whoa	N/A	N/A	N/A	Understanding
Help	N/A	N/A	N/A	N/A
Self-Judgment	N/A	Meta-comment	N/A	N/A
Oracle/Specification	Answer-oriented	Problem-related	N/A	N/A
Concept	Principle-oriented, Definition	N/A	Concepts	N/A
Strategy Question	N/A	N/A	N/A	Design
How Goal	Process-oriented, Interface	N/A	Goal, Goal/Action	Use, Selection, Coordination
Strategy Hypothesis	N/A	Reminding example	N/A	N/A

for more research on how to support it.

- The second most common category was Strategy. Strategy information gaps outnumbered feature-oriented information gaps by a 2:1 ratio. To date, there has been almost no research on supporting information gaps of this type.
- Debugging explanations should focus not only on local information gaps, (e.g., pertaining to one cell), but also on global information gaps (e.g., pertaining to an entire spreadsheet).
- When a generic "help" request is made, an explanation system might be able to figure out, from the system state and from the timing of the request, if it is a (re-)connect question as versus a feedback-oriented surprise.
- Debugging explanations should strive to fulfill users' needs to self-judge their progress. This category contributed a surprising 9% of the information gaps. Accurate self-judgment matters to debugging effectiveness for both its self-efficacy and its metacognitive implications.

The above results have specific implications for designers of debugging support for end-user programmers, and also identify some open research questions in this area. We hope that following up on these results will help to fill end-user programmers' critical information gaps that currently serve as barriers to the genuine effectiveness of end-user programming.

ACKNOWLEDGMENTS

We thank Andrew Ko, Curtis Cook, and John Pane for advice, ideas, and feedback. We also thank Valentina Grigoreanu and Vaishnavi Narayanan for help with the planning and administration of the experiment. This work was supported by the EUSES Consortium via NSF's ITR-0325273 and ITR-0405612.

REFERENCES

[1] Abraham, R., Erwig, M. Goal-directed debugging of spreadsheets, *IEEE Symp. Visual Langs. Human-Centric Comp.*, 2005, 37-44.

[2] Anthony, L., Corbett, A. Wagner, A., Stevens, S., Koedinger, K. Student question-asking patterns in an intelligent algebra tutor, *Conf. Intelligent Tutoring Sys.*, 2004, 455-467.

[3] Bandura, A. Self efficacy: Toward a unifying theory of behavioral change. *Psychological Review*, 1977, 191-215.

[4] Beckwith, L., Sorte, S., Burnett, M., Wiedenbeck, S., Chintakovid, T., Cook, C. Designing features for both genders in end-user software engineering environments, *IEEE Symp. Visual Langs. Human-Centric Comp.*, 2005, 153-160.

[5] Beckwith, L. Burnett, M., Wiedenbeck, S., Cook, C., Sorte, S., Hastings, M. Effectiveness of end-user debugging software features: Are there gender issues? *ACM Conf. Human Factors Comp. Sys.*, 2005, 869-878.

[6] Blackwell, A. First steps in programming: A rationale for attention investment models, *Proc. IEEE Symp. Human-Centric Comp. Langs. Envs.*, 2002, 2-10.

[7] Burnett, M., Cook, C., Rothermel, G. End-user software engineering, *Comm. ACM*, 2004, 53-58.

[8] Carroll, J., Rosson, M. Paradox of the active user, In *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, J. Carroll (Ed.), MIT Press, 1987, 80-111.

[9] Conway, M., et al. Alice: Lessons learned from building a 3D system for novices, *ACM Conf. Human Factors Comp. Sys.*, 2000, 486-493.

[10] Farkas, D. Layering as a safety net for minimalist documentation, In Carroll, J. M., (Eds.), *Minimalism Beyond the Nurnberg Funnel*, MIT Press, 1998, 247-274.

[11] Forrest-Pressly, D., MacKinnon, G., Waller, T. *Metacognition, Cognition, and Human Performance*, Academic Press, 1985.

[12] Gee, J. *An Introduction to Discourse Analysis*, Routledge, London, 1999.

[13] Gordon, S., Gill, R. Knowledge acquisition with question probes and conceptual graph structures, In T. Lauer, E. Peacock, A. Graesser (Eds.), *Questions and Information Sys.*, Lawrence Erlbaum Assc., 1992, 29-46.

[14] Ko, A., Myers, B. Designing the Whyline: A debugging interface for asking questions about program failures, *ACM Conf. Human Factors Comp. Sys.*, 2004, 151-158.

[15] Ko, A., Myers, B., Aung, H. Six learning barriers in end-user programming systems, *IEEE Symp. Vis. Lang. Human-Centric Comp.*, 2004, 199-206.

[16] Lowenstein, G. The psychology of curiosity, *Psychological Bulletin 116*, 1, 1994, 75-98.

[17] Miller, R., Myers B. Outlier finding: Focusing user attention on possible errors, *ACM User Interface Soft. Tech.*, 2001, 81-90.

[18] Nardi, B. *A Small Matter of Programming: Perspectives on End User Computing*, MIT Press, 1993.

[19] Norman, D. *The Design of Everyday Things*, New York, NY: Doubleday, 1988.

[20] Person, N., Graesser, A. Fourteen facts about human tutoring: Food for thought for ITS developers, *AIED Workshop on Tutorial Dialogue*, 2003, 335-344.

[21] Robertson, T., Lawrance, J., Burnett, M. Impact of high-intensity negotiated-style interruptions on end-user debugging, *J. Visual Langs. Comp.*, to appear 2006.

[22] Ruthruff, J., Phalgune, A., Beckwith, L., Burnett, M., Cook, C. Rewarding 'good' behavior: End-user debugging and rewards, *IEEE Symp. Visual Langs. Human-Centric Comp.*, 2004, 107-114.

[23] Wagner, E., Lieberman, H. Supporting user hypotheses in problem diagnosis on the web and elsewhere, *ACM Int. Conf. Intelligent User Interfaces*, 2004, 30-37.

[24] Weinert, F., Kluwe, R. (Eds.) *Metacognition, Motivation, and Understanding*, Lawrence Erlbaum Associates. 1987.

[25] Wilson, A., Burnett, M., Beckwith, L., Granatir, O., Casburn, L., Cook, C., Durham, M., Rothermel, G. Harnessing curiosity to increase correctness in end-user programming, *ACM Conf. Human Factors in Comp. Sys.*, 2003, 305–312.