Garbage In, Garbage Out?
An Empirical Look at Oracle Mistakes by End-User Programmers

Amit Phalgune*, Cory Kissinger*, Margaret Burnett*,
Curtis Cook*, Laura Beckwith*, and Joseph R. Ruthruff†

*Oregon State University
Corvallis, Oregon 97331
{phalgune, ckissin, burnett, cook, beckwith}@cs.orst.edu

†University of Nebraska-Lincoln
Lincoln, Nebraska 68588
{ruthruff}@cse.unl.edu

Abstract

End-user programmers, because they are human, make mistakes. However, past research has not considered how visual end-user debugging devices could be designed to ameliorate the effects of mistakes. This paper empirically examines oracle mistakes*—mistakes users make about which values are right and which are wrong—to reveal differences in how different types of oracle mistakes impact the quality of visual feedback about bugs. We then consider the implications of these empirical results for designers of end-user software engineering environments.*

1. Introduction

Errors occur in every domain of human action, and thus it should be no surprise that they arise in end-user programming. For example, evidence abounds of errors in the spreadsheet domain, where errors are frequent and in some cases very costly (e.g., [10, 14, 15]). Panko's work [14] points out the substantial error rates in spreadsheets and possible reasons for these rates, such as the lack of debugging tools available in spreadsheet systems and overconfidence on the part of the spreadsheet users.

In the early decades of computing, a common saying was "garbage in, garbage out." That is, mistakes in communicating with a computer were aberrations, and if users provided bad data (garbage in), then they should expect the software to produce incorrect answers (garbage out).

Of course, with the joint advents of interactive systems and HCI as a subarea of computer science, it was realized that people do make mistakes in communicating with computers, and features began to appear to help prevent mistakes (such as menus instead of typed-in commands) and to allow people to detect and recover from them (such as immediate feedback and undo facilities).

Still, below this surface level, the philosophy of "garbage in, garbage out" remains: if the user's mistake somehow gets in unnoticed, then surely he or she still should expect "garbage out." The unfortunate consequence is that it then seems reasonable for developers to assume that software needs to work correctly only when no mistaken data has worked its way into the system.

In this paper, we consider whether this is a reasonable assumption in software whose purpose is to help end-user programmers reason about and debug their programs. In the problem solving end-user programmers must perform to find and remove errors from their programs, it is not always straightforward for users to make correct judgments about how well different parts of their program are working, and thus some mistakes are *inevitable*. We consider the prevalence and effects of these mistakes in order to determine whether end-user software engineering environments offering testing and debugging support to end-user programmers must be designed with this inevitability in mind.

The type of testing and debugging mistakes upon which this paper focuses are *oracle mistakes* [23], a term meaning a falsely positive or falsely negative judgment as to whether an output computed by the program is correct. The end-user programming environment prototype in which we consider this type of mistakes is a spreadsheet environment that includes a visual testing and fault localization device we have developed known as WYSIWYT (What You See Is What You Test) [16, 18]. In this paper, we explore how different subsets of oracle mistakes impact the effectiveness of interactive, visual testing and debugging support for end-user programmers, and what might be done to ameliorate these impacts.

2. End-User Debugging and Mistakes

2.1 End-User Debugging

There has been recent research focusing on assisting end-user programmers in debugging. Virtually all of this work communicates with the user largely in the form of visual devices. Woodstein [22] is a software agent that visually assists users in debugging e-commerce errors. Ko and Myers present the Whyline [13], an "interrogative debugging" device for the event-based programming environment Alice. There has also been a variety of work supporting program comprehension and debugging by end users in the spreadsheet paradigm. For example, Igarashi

et al. [11] present devices to aid spreadsheet users in dataflow visualization and editing tasks. S2 [21] provides a visual auditing feature in Excel 7.0: similar groups of cells are recognized and shaded based upon formula similarity, and are then connected with arrows to show dataflow. This technique builds upon the Arrow Tool, a dataflow visualization device proposed by Davis [8]. Ayalew and Mittermeir [3] present a method of fault tracing based on "interval testing" and slicing, which is similar to our own work on assertions to help users automatically guard against faults [6]. There is also recent work to automatically detect certain kinds of errors, such as errors in spreadsheet units [1] and types [2].

2.2 WYSIWYT with Visual Fault Localization

We have been working on a vision of end-user software engineering [7] that we have prototyped in the spreadsheet paradigm because it is so widespread. Our vision of end-user software engineering involves holistic support of the facets of software development in which end users engage, tied together through incremental visual devices. These visual devices are necessarily low in cost to maintain the immediate responsiveness expected by spreadsheet users, and are immediately updated as end users add to, modify, test, and debug their programs.

WYSIWYT with visual fault localization is part of this vision. An example from our research prototype, Forms/3 [5], is shown in Figure 1. In the course of developing a spreadsheet, users can communicate a judgment that a cell's value is correct with a checkmark (√), or that a cell's value is incorrect with an X-mark (X), as shown in the figure. Checkmarks contribute to the "testedness" of the cells according to an adequacy criterion detailed in [16], and a cell's testedness is reflected in border colors along a red-to-blue continuum (light gray to black). Dataflow arrows reflecting dependencies, whose color reflects the testedness of specific relationships among cells and subexpressions, are available on demand. The system combines the user's checkmarks and X-marks with the dependencies in the cells' formulas to estimate likelihoods of the *fault* (erroneous formula) being located in various cells. It colors these cells' interiors in light-to-dark amber (gray) to reflect these likelihoods [18].

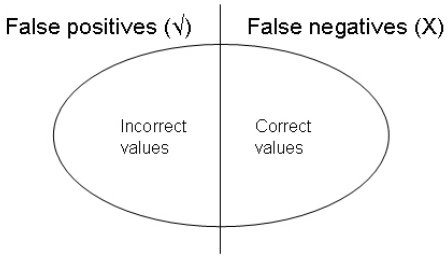

Figure 2: Oracle mistakes consist of false positives and false negatives.

Given these communication devices, a *false positive* is a checkmark on a value that is incorrect, and a *false negative* is an X-mark on a value that is correct (Figure 2).

2.3 Mistakes in Interactive Testing and Debugging Environments

Like most environments for end-user programmers, the spreadsheet paradigm is modeless and interactive: users incrementally experiment with their software and see how the results work out after each change; an example of this in spreadsheets is the automatic recalculation feature. This means that testing and debugging support in these types of environments must accommodate continuous interaction with the end user.

Figure 1 shows an example from our research prototype. In the figure, suppose there is a fault in cell Midterm1_Perc, causing incorrect values in several downstream cells. Unfortunately, the user has made an oracle mistake by checking off (√) Min_Midterm1_Midterm2's value (a false positive). Because of this mistake, Midterm1_Perc is only mildly implicated as the culprit by the visual feedback, as the figure shows.

The research literature provides little guidance regarding how the developers of visual devices for interactive testing and debugging should handle oracle mistakes and their impacts on such devices' accuracy of feedback. In

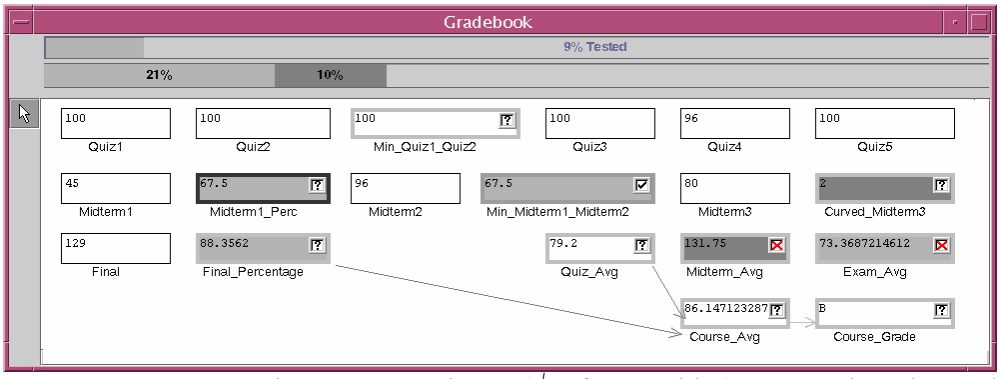

Figure 1: Gradebook spreadsheet with an oracle mistake (√, a false positive) on cell Min_Midterm1_Midterm2. The correct mark (X) would have resulted in a more accurate prediction of the faulty cell's (Midterm1_Perc) fault likelihood, with a resultingly darker coloring than the relatively light coloring it has here.

fact, the presence of *any* mistakes runs contrary to a common assumption in traditional testing and debugging research—that all information is accurate and reliable—meaning that much of the prior software visualization research may be inherently unsuited for the interactive environments that end-user programmers utilize.

There is, however, some information about mistakes in end-user debugging upon which we can build. Galletta et al. found that presenting users with formulas and values while debugging spreadsheets did not help them find errors, but did diminish the number of false positives users committed [9]. In the course of other investigations we have done on end-user debugging, we have reported the presence of oracle mistakes observed [17, 19, 20], which gives a little preliminary information about their prevalence. Ko and Myers [12] use the phrase "determining a failure" to describe all errors in perceiving and understanding output, which has a significant overlap with oracle mistakes. In their observational study, problems of the "determining failure" type made up *28 of the 29 breakdowns* that occurred during debugging, strongly implying that this type of problem is one of the most important factors when debugging goes astray. Combined, these works suggest that designers of interactive fault localization devices may not be able to ignore the possibility of oracle mistakes. This paper investigates this possibility.

3. Experiment

To gain insight into the importance of oracle mistakes on end-user testing and debugging we consider the following research questions:

RQ1: How often do oracle mistakes occur?
RQ2: Do oracle mistakes impact effectiveness of a fault localization device?
RQ3: Are oracle mistakes tied to end users' understanding of the debugging device?
RQ4: Do oracle mistakes impact end users' ability to debug?
RQ5: Do "smart" oracle mistakes impact effectiveness of the fault localization device differently?

3.1 Design

This investigation consists of both an observational study and a planned experiment.

A collection of naturally occurring oracle mistakes was available to us in the electronic transcripts from a previous experiment [4]. We refer to this collection of transcripts, which contains the oracle mistakes the participants actually made, as *Version Original*. For the observational study, we simply analyze effects of an observed independent variable (number of oracle mistakes) on observed dependent variables (discussed later).

For the planned experiment, the design was a within-subjects design, in which the treatment variable being manipulated—i.e., the independent variable—starts with the same collection of observed oracle mistakes. We then manipulated these observed oracle mistakes, in ways we describe later in this section, to generate new simulated versions with which we can compare Version Original.

3.2 What the Original Participants Did

The previous experiment [4] produced Version Original. After completing a background questionnaire, the 51 participants were given a tutorial that taught the use of the WYSIWYT checkbox for checking off correct values and associated feedback, but did not teach debugging or testing strategy content. We did not even teach the use of fault localization; rather, participants were introduced to the mechanics of placing X-marks and given time to explore any resulting feedback that they found interesting.

The participants were then provided with two spreadsheets to debug. The use of two spreadsheets reduced the chances of the results being due to any one spreadsheet's particular characteristics. The experiment was counterbalanced with respect to task order so as to distribute learning effects evenly. We collected electronic transcripts of every action taken by the participants and the system's resulting feedback.

The two spreadsheets were Gradebook (Figure 1) and Payroll (shown in [19]). To make the spreadsheets representative of real end-user spreadsheets, Gradebook was derived from an Excel spreadsheet of an (end-user) instructor, which we ported into an equivalent Forms/3 spreadsheet. (To accommodate Forms/3 features, a minor change was made to two minimization operators.) Payroll was a more complicated spreadsheet designed by two Forms/3 researchers using a payroll description from a real company.

These spreadsheets were seeded with five faults created by real end users. Gradebook was seeded with three of these users' mechanical faults, one logical fault, and one omission fault, and Payroll with two mechanical faults, two logical faults, and one omission fault. Under Panko's classification [14] mechanical faults include simple typographical errors or wrong cell references. Logical faults are mistakes in reasoning and are more difficult than mechanical faults. An omission fault is information that has never been entered into a cell formula, and is the most difficult to detect [14].

The participants were supported by a fault localization algorithm known as Test Count [18]. Let NumFailingTests (NFT) be the number of failed tests in which a cell c has participated, and NumSuccessfulTests (NST) be the number of successful tests in which c has participated. With Test Count, if cell c has no failed tests, the fault likelihood of c is "None". Otherwise, the fault likelihood of cell c is computed as:

Fault likelihood(c) = max (1, 2*NFT - NST)

3.3 Current Experiment's Procedures

Using Version Original as a base, we manipulated the numbers and types of oracle mistakes to generate three more versions of the data: Version FalsePositivesOnly, Version FalseNegativesOnly, and Version Ideal.

3.3.1 Three Generated Versions

In *Version FalsePositivesOnly,* we replaced each false negative mistake (in which the user placed an X-mark on a value that was in fact correct) with a non-mistaken judgment, namely a checkmark. We corrected each false negative mistake one at a time, in isolation from the others. Specifically, we ran a simulation on the original data to collect the fault localization feedback after each action, and as soon as a false negative mistake was detected, we replaced the erroneous X-mark with its correct counterpart (a checkmark), reported the new fault localization feedback that the system provided, and then restored the original mistake before proceeding on with the simulation. This procedure enabled us to isolate the difference in the visual feedback provided by the system whenever a false negative mistake was corrected. It also prevented this "what if" version from straying too far from what the original participants actually saw by preventing a cascading effect from the accumulation of changes.

Similarly, in *Version FalseNegativesOnly*, we replaced each false positive mistake (in which the user checked off a value that was in fact incorrect) with a non-mistaken judgment, namely an X-mark, using the same safeguard against accumulated effects as for Version FalsePositivesOnly.

Version Ideal was intended to reflect the best feedback each participant could have had if he or she had made no mistakes at all. It corrected both types of oracle mistakes. For this version, there was no reason to guard against cascading effects; it simply reflected the ideal feedback that could be achieved by making no oracle mistakes when judging (marking) whatever cells each participant judged.

3.3.2 "Smart" Mistakes

By definition, testing is validating output *values*, and the role of an oracle is to accurately judge whether the output values are correct [23]. However, during several earlier think-aloud studies we have noticed some users placing marks to communicate judgments about the correctness (or lack thereof) of the *formulas* (i.e., source code). For example, even when a value was incorrect, some users checked off the cell because the formula was correct. Although this unintended usage may not seem a mistake from a user's perspective, it is still a (serious) mistake from the perspective of a testing system's reliability, because these systems reason according to the above definitions. But, it is a different sort of mistake, one that has a possible rationale behind it. Thus, we term this type of mistake as a *smart mistake*. The remaining mistakes are termed *no-rationale mistakes*. See Figure 3.

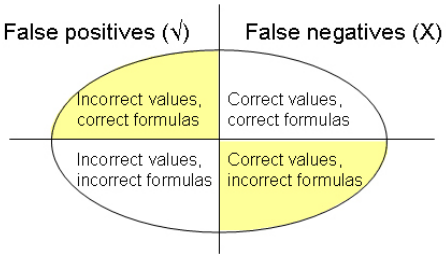

Figure 3: The shaded sectors show the smart mistakes.

In investigating the impacts of these two, mutually exclusive, subsets of mistakes, we created a variation of the above Version Ideal that isolated the smart mistakes (correcting only the no-rationale mistakes). This version is termed *Version Smart*.

3.4 Dependent Variables and Measures

For the portion of our investigation conducted via the observational study, the observed dependent variables were the participants' actual bugs fixed and their bugs introduced. We chose to focus on these variables because oracle mistakes seem likely to affect debugging success.

We also required a measure of the fault localization technique's effectiveness. Since an important goal of this experiment was to study the impact of oracle mistakes on the visual feedback of our fault localization device, as in our previous work [17], we defined the fault localization technique's effectiveness as the technique's ability to correctly and visually differentiate the correct cells in the spreadsheet from those that actually contain faults. Let *FaultyCells(AvgFL)* be the average fault likelihood of colored faulty cells. Let *CorrectCells(AvgFL)* be the average fault likelihood of colored correct cells. The formula to calculate visual effectiveness (VE) according to this measure is then:

VE = FaultyCells(AvgFL) – CorrectCells(AvgFL)

4. Results

4.1 RQ1: Prevalence of Oracle Mistakes

As Table 1 illustrates, 17.1% and 22.5% of the judgments made in Gradebook and Payroll, respectively, were mistaken. This frequency is even worse than that observed in previous work [17, 20], which ranged from 5% to a bit over 20%. In fact, in this study, only two out of the 51 participants managed to *not* make oracle mistakes.

4.2 RQ2: Impact on Visual Effectiveness

To measure the impact of oracle mistakes on the effectiveness of fault localization feedback, we compared the visual effectiveness of the feedback the user actually saw to the visual effectiveness they might have seen if each single oracle mistake had instead been a correct judgment.

Table 1: Frequency of oracle mistakes for each task (as observed in Version Original).

	Number of marks (judgments)	Number of oracle mistakes	%	Number of false positives	Number of false negatives	Mean mistakes per user	Median
Gradebook	899	154	17.1	144	10	3.02	2
Payroll	1,696	381	22.5	354	27	7.27	4

(Alternatively, we could have compared with a version in which each oracle mistake was simply removed, i.e., no decision was made. However, at the point an oracle mistake was made, the user was ready to make a decision, so fixing the erroneous decision seemed a closer scenario to reality than simulating the user making no decision at that point.) Since we wanted to stay as close as possible to what the users really saw, Versions FalsePositivesOnly and FalseNegativesOnly were the right versions for this question, but Version FalsePositivesOnly had too few changes to benefit from statistical analysis.

Thus, we compared Version Original with the version with most of the mistakes fixed (i.e., Version FalseNegativesOnly), using the following (null) hypothesis as a statistical vehicle:

H2-1: There will be no difference between the visual effectiveness of the feedback produced in Version Original and that in Version FalseNegativesOnly.

For both Gradebook and Payroll (Table 2), there was a significant difference in the average visual effectiveness scores between the two versions (paired t-test: df = 50, Gradebook: t=10.57, p<.001; Payroll: t=2.19, p=.03). Thus, we reject H2-1.

Discussion: Obviously, the use of a device, any device, in an incorrect manner will negatively impact that device's effectiveness. Still, this result combined with the findings from RQ1 establishes a critical point: designing an interactive fault localization device under the assumption that oracle mistakes can be ignored is not reasonable. The investigation of RQ1 shows they occurred with great frequency and the investigation of RQ2 shows they did significant damage.

4.3 RQ3: Relationship to Understanding

Suppose a lack of understanding of the fault localization device is causing the mistakes. If that is the case, we might work on increasing users' understanding of the device to reduce the number of oracle mistakes and mitigating the type of damage observed in RQ2. To consider whether this would be a profitable direction, we considered the following hypothesis:

Table 2: Mean / median of Version FalseNegativesOnly feedback and Version Original (p < .05 is shaded).

	VE with most mistakes corrected	VE Original
Gradebook	.73 / .71	.01 / .00
Payroll	.25 / .12	.01 / .00

H3-1: There will be no relationship between users' oracle mistakes and their understanding of the debugging device.

Understanding was measured via post-test questionnaire scores, with a maximum score possible of 10. Regression analysis on the observed data in Version Original (Figure 4), showed no significant relationship between the users' understanding of the debugging device and the number of oracle mistakes made (linear regression: $F(1,49) = .104$, $\beta = .204$, $R^2 = .002$, $p = .75$).

Discussion: The users' understanding of the device does not seem implicated. The results show that participants with a better understanding of the device did not make fewer mistakes.

4.4 RQ4: Impact on Debugging

Section 4.2 considered the *system's* ability to produce good feedback in the presence of oracle mistakes by making comparisons with a generated version of the data. We now turn to solely observed data to consider the *user's* ability to succeed at debugging in the presence of oracle mistakes.

H4-1: The number of oracle mistakes users made will have no relationship to the number of bugs they fixed or introduced.

Results of the regression analyses of the participants' number of oracle mistakes compared to their ability to fix the bugs we seeded, and to avoid introducing new bugs, is shown in Table 3. The regression coefficient is the slope of the least squares fitting of number of mistakes against the progress measures.

We found a significant relationship between oracle mistakes and both bugs fixed and introduced for the Gradebook task (linear regression: data shown in Table 3). The Payroll task, however, did not show any significant relationship between the number of oracle mistakes a user made and the user's debugging. The relationships are illustrated in Figure 5. Thus, for the Gradebook task, we

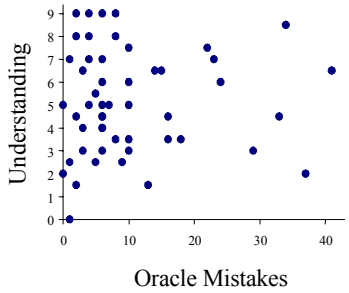

Figure 4: Most participants made 1 to 10 oracle mistakes, regardless of their understanding scores.

reject H4-1. (Even so, note that the low R^2 values do not indicate a good fit despite the level of significance.)

Discussion: We were surprised that on the Payroll task, the harder of the two, oracle mistakes had no relationship to a user's debugging success. However, a closer analysis of the characteristics of the oracle mistakes themselves may explain this, which we consider next.

4.5 RQ5: Impact of Smart Mistakes

Consider the implications of making a "smart mistake". For example, suppose the user placed an X-mark on a cell with a correct value but an incorrect formula. Most fault localization algorithms are based on evidence of "guilt" (judgments that values are bad), and this is the case with our fault localization algorithm as well. As a result, this cell will be colored "more faulty" (darker in our prototype) because the user has just implicated it. Thus, although the user has mistakenly communicated that the value is wrong, a desirable side effect occurred: the cell, which is indeed faulty, has just gotten darker. For this particular cell then, its visualized fault likelihood is *better* than if the user had not made the oracle mistake!

However, looking to this cell's backward slice (the cells contributing to this cell's value), there is likely to be a detrimental effect on visual effectiveness, because all of these cells will be wrongly assumed to be contributing to an incorrect value (which they did not), and as therefore being potentially faulty.

So, these "smart" mistakes are helpful in some respects and harmful in others. To consider just how helpful or harmful they are, we compared the effects of smart mistakes to the ideal:

H5-1: There will be no difference in visual effectiveness between Version Smart and Version Ideal.

As Table 4 shows, there were only a few smart mistakes in the Gradebook task, but over half of the oracle mistakes in Payroll were smart mistakes. (This is interesting, and could be tied to the difference in relative difficulty between the two.)

Table 3: Regression analyses of number of oracle mistakes vs. bugs fixed and bugs introduced.

Progress Measure	F(1,49)	β	R^2	p-value
Gradebook:				
Fixed	4.34	-.179	.081	.043
Introduced	8.02	.195	.141	.007
Payroll:				
Fixed	< .001	< .001	< .001	.986
Introduced	.69	.021	.014	.411

Table 4: Number of smart mistakes made compared to total number of mistakes.

	Number of smart mistakes	Number of oracle mistakes	%
Gradebook	10	154	6.5
Payroll	213	381	55.9

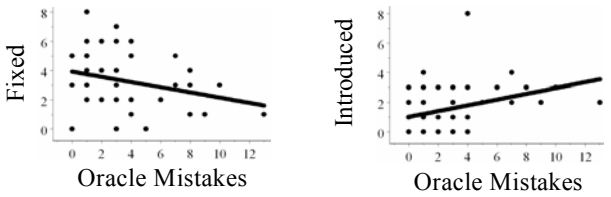

Figure 5: In Gradebook, oracle mistakes were inversely proportional to the number of bugs fixed and directly proportional to the number of bugs introduced.

Because Gradebook had so few smart mistakes, the effects of smart mistakes on its visual effectiveness scores could not be analyzed statistically, but we analyzed Payroll as follows. The feedback in Version Smart (the version from which the no-rationale mistakes had been corrected) was compared to that of Version Ideal. The difference in the average visual effectiveness scores was not significant in magnitude, due in part to the fact that there were also a number of correct marks present. However, the average visual effectiveness score for each participant under Version Smart was higher a startling number of times (Figure 6). Analysis of the counts showed that Version Smart indeed had significantly better feedback than Version Ideal (Fishers Exact Test: p = .03).

Discussion: Our results above showed that, for this particular spreadsheet, smart mistakes were better than perfection! These results may, of course, be due to the particular relationships present in that spreadsheet. (For example, for a spreadsheet with longer dataflow chains the detrimental effect on the backward slice could outweigh the positive effect on the cell that was directly marked.) But despite the fact that in some cases the global negatives could win out, the fact remains that this type of mistake was extremely common in Payroll, and has a strongly positive impact on the cell being marked.

RQ5's result may explain the differences that were seen between Gradebook and Payroll in RQ4. Many of the mistakes in Payroll were smart mistakes, and RQ5 implies that these mistakes would not have deleterious effects on debugging.

Finally, recall that the results of RQ2 showed that, overall, oracle mistakes had a negative impact on visual effectiveness. Since the smart mistakes included in RQ2's

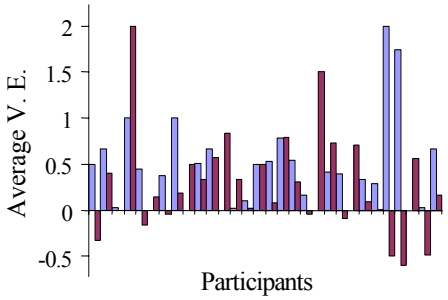

Figure 6: Average visual effectiveness for each participant under Version Smart (light bars) and Version Ideal (dark bars).

results were actually helping the visual effectiveness of the feedback, the implication is that the no-rationale mistakes had a very strong negative effect—strong enough to significantly negate the positive effects of the correct marks and smart mistakes together.

5. Implications for Designers of End-User Environments

Given these results, how should designers of end-user environments proceed? At least two possible strategies present themselves: (1) find ways to lessen the impact of no-rationale mistakes, and (2) find ways to strengthen the positive impacts of smart mistakes.

As it happens, the Test Count fault localization algorithm already tempers the negative impact of no-rationale mistakes. The way it does so is through a "robustness" property [20]. Suppose there is a cell c marked with an X-mark. The robustness property requires that all cells in the backward slice of cell c receive at least some visual fault coloring, no matter how many positive tests have also been run. This feature guarantees that any faulty cell contributing to a failure the user observed will be one of the cells highlighted. Since false positives were by far the most common type of oracle mistake, without this robustness feature, the detrimental impact of mistakes would have been even worse than it was in our study.

Another fortunate attribute of Test Count is that it gives double the weight to negative judgments as it does to positive judgments (Section 3.2). This is fortunate because most of the mistakes were false positives, not false negatives, as has also been true in our previous experiments. Hence, correct judgments are getting more weight than incorrect ones. This may well account for the mostly positive (above zero) visual effectiveness scores observed throughout our results.

The surprising results of RQ5 suggest an opportunity for improved robustness. We realized that any negative effects of smart mistakes must be solely the global effects, since locally a smart mistake is actually beneficial. The algorithm used in this paper, Test Count, gives equal weight to local and global impacts of the marks made. Thus, we wondered if reducing the impact a mark can have on cells in its backward slice (or equivalently, increasing the local impact) would improve visual effectiveness. We decided to evaluate this idea empirically.

Recall from Section 3.2 that NumFailingTests (NFT) is the number of failed tests in which a cell c participated, and NumSuccessfulTests (NST) is the number of successful tests in which c participated. For an increased effect on local impacts (an algorithm variant we will term "Test Count Local"), we partitioned NFT into NFT_L (number of failed tests locally) and NFT_G (number of failed tests globally). Similarly, we partitioned NST into NST_L and NST_G. To double the impact of local decisions, the fault localization formula for Test Count Local is thus defined as:

Fault likelihood(c) = max(1, (4*NFT_L + 2*NFT_G) - (2*NST_L + NFT_G))

We then compared the visual effectiveness of the feedback the participants actually received (Version Original) to that which they would have received with the Test Count Local algorithm. A paired t-test (used in the same manner as in RQ2 on visual effectiveness scores) revealed that Test Count Local produced feedback whose visual effectiveness was significantly better than the feedback produced by Test Count Original (paired t-test: Gradebook: df=23, t=4.73, p<.001; Payroll: df=31, t=2.99, p=.005). See Figure 7 and Table 5. This is especially revealing since Gradebook had few smart mistakes whereas Payroll had many. Thus, Test Count Local was significantly better overall at ameliorating the effects of oracle mistakes than Test Count Original.

These results do not appear to be specific to our Test Count algorithm. In a previous investigation of fault localization algorithms [17], one algorithm, Nearest Consumer, outperformed the other two (one of which was Test Count), both with and without the presence of oracle mistakes. We were unable to explain this, since Nearest Consumer actually uses less information and is less precise than Test Count. However, it emphasizes local im-

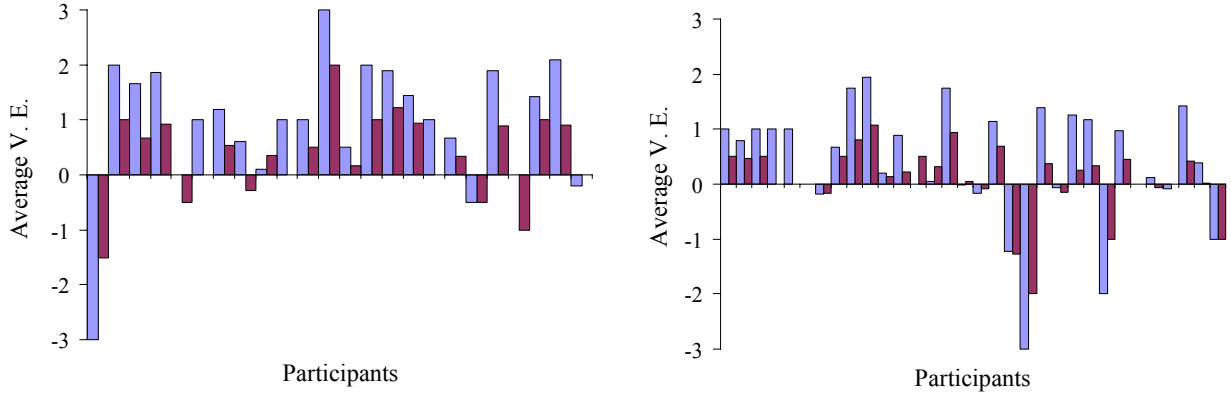

Figure 7: Average visual effectiveness for each user with Test Count Local (light bars) versus Test Count Original (dark bars). Left: Gradebook. Right: Payroll.

Table 5: Mean / median of visual effectiveness with Test Count Local and Test Count Original.

	VE Test Count Local	VE Test Count Original
Gradebook	.94 / 1.00	.36 / .35
Payroll	.38 / .39	.09 / .13

pacts in a manner similar to Test Count Local, which now seems likely to have played an important role in that algorithm's increased effectiveness and robustness.

In fact, these findings may well be applicable beyond the spreadsheet paradigm. For example, any dataflow-oriented debugging devices, such as Woodstein [22] and the Whyline [13], are potential beneficiaries of the findings and recommendations in this paper.

6. Conclusion

In this paper, we have investigated the impact of different types of oracle mistakes on the quality of visual feedback that can be achieved by end-user fault localization devices based on testing. Our investigation revealed that, contrary to traditional assumptions, *it is not reasonable to ignore oracle mistakes in designing interactive fault localization devices.*

Following up on this result, our findings also led to the following recommendations for handling oracle mistakes:

- For robustness given smart mistakes, a system should weight local effects more heavily than global effects.
- For robustness against false positives, a cell or variable implicated in a fault should always retain some degree of implication, no matter how many positive tests in which it participates.
- Because there are more false positives than false negatives, a system should trust (and weight) positive judgments less than negative judgments.

Although further empirical study is needed for generality, the implications of our empirical findings to date suggest that following these recommendations can improve the effectiveness of interactive fault localization devices for end-user programmers.

Acknowledgments

This work was supported in part by the EUSES Consortium via NSF grant ITR-0325273.

References

[1] R. Abraham and M. Erwig, "Header and unit inference for spreadsheets through spatial analyses", *Proc. IEEE Symp. Visual Langs. Human-Centric Computing*, 2004, 165-172.

[2] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi, "A type system for statically detecting spreadsheet errors", *Proc. IEEE Conf. Auto. Soft. Eng.*, 2003.

[3] Y. Ayalew and R. Mittermeir, "Spreadsheet debugging", *Proc. European Spreadsheet Risks Interest Group*, 2003.

[4] L. Beckwith, M. Burnett, S. Wiedenbeck, C. Cook, S. Sorte, M. Hastings, "Effectiveness of End-User Debugging Software Features: Are There Gender Issues?", *ACM Conf. Human Factors in Computing Systems,* 2005, 869-878.

[5] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S.Yang, "Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm", *J. Functional Programming*, 11, 2, 2001, 155- 206

[6] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace, "End-user software engineering with assertions in the spreadsheet paradigm", *Proc. Int. Conf. Soft. Eng.*, 2003, 93-103.

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

[8] J. S. Davis, "Tools for spreadsheet auditing", *Int. J. Human-Computer Studies,* 45, 1996, 429-442.

[9] D. F. Galletta, K. S. Hartzel, S. E. Johnson, J. L. Joseph, and S. Rustagi "Spreadsheet presentation and error detection: An experimental study", *J. Management Info. Systems*, 13, 3, 1997, 45-63.

[10] D. Hilzenrath, "Finding errors a plus, Fannie says; Mortgage giant tries to soften effect of $1 billion in mistakes", *The Washington Post,* 2003.

[11] T. Igarashi, J. D. Mackinlay, B. W. Chang, and P. T. Zellweger, "Fluid visualization of spreadsheet structures", *Proc. IEEE Symp. Visual Langs.*, 1998, 118-125.

[12] A. J. Ko and B. A. Myers, "Development and evaluation of a model of programming errors", *Proc. IEEE Symp.Visual Langs. Human-Centric Computing Langs.*, 2003, 7-14.

[13] A. J. Ko and B. A. Myers, "Designing the Whyline: A debugging interface for asking questions about program failures", *Proc. ACM Conf. Human Factors Computing Systems*, 2004, 151-158.

[14] R. Panko, "What we know about spreadsheet errors", *J. End User Computing,* 1998.

[15] G. Robertson, "Officials red-faced by $24M gaffe: Error in contract bid hits bottom line of TransAlta Corp." *Ottawa Citizen,* 2003.

[16] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov, "A methodology for testing spreadsheets", *ACM Trans. Soft. Eng. Meth.* 10, 1, 2001, 110-147.

[17] J. Ruthruff, M. Burnett, and G. Rothermel, "An empirical study of fault localization for end-user programmers", *Proc. Int. Conf. Soft. Eng.* 2005, 352-361

[18] J. Ruthruff, E. Creswick, M. Burnett, C. Cook, S. Prabhakararao, M. Fisher II, and M. Main, "End-user software visualizations for fault localization", *Proc. ACM Symp. Soft. Visualization,* 2003, 123-132.

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

[20] J. Ruthruff, S. Prabhakararao, J. Reichwein, C. Cook, E. Creswick, and M. Burnett, "Interactive, visual fault localization support for end-user programmers", *J. Visual Langs. Computing,* 16, 1-2, 2005, 3-40.

[21] J. Sajanieme, "Modeling spreadsheet audit: A rigorous approach to automatic visualization", *J. Visual Langs. Computing,* 11, 1, 2000, 49-82.

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

[23] E. Weyuker, "On testing non-testable programs", *The Computer Journal,* 25, 4, 1982, 465-470.