In *Empirical Studies of Programmers*, Washington, D. C., October 1997 (to appear).

A BUG'S EYE VIEW OF IMMEDIATE VISUAL FEEDBACK IN DIRECT-MANIPULATION PROGRAMMING SYSTEMS

Curtis Cook, Margaret Burnett†, and Derrick Boom
Department of Computer Science
Oregon State University, Corvallis, Oregon 97331
{cook, burnett, boom}@cs.orst.edu

KEYWORDS: Direct manipulation, debugging, end-user programming, spreadsheets, visual programming languages, liveness, empirical study

ABSTRACT

Immediate visual feedback is becoming a common feature in direct-manipulation programming systems of all kinds—from demonstrational macro builders to spreadsheet packages to visual programming languages featuring direct manipulation. But does immediate visual feedback actually help in the domain of programming? We previously reported on an empirical study to determine whether the inclusion of immediate visual feedback into a direct-manipulation programming system helps with one particular task: debugging. In that study, subjects debugged programs with and without immediate visual feedback. We found that although immediate visual feedback did not significantly help with debugging in general, it did significantly help with debugging in some circumstances.

In this paper, we follow up on those results, looking at attributes of the bugs themselves to see if they help to determine the circumstances in which feedback helps with debugging. We analyze how particular bugs and collections of bugs grouped by error type related to subjects' debugging abilities with and without immediate visual feedback, which we term the "which" questions; how bugs' position on the screen related to subjects' debugging abilities with and without immediate visual feedback, termed the "where" questions; and whether the presence or absence of immediate visual feedback affected the speed and order in which bugs were corrected, termed the "when" questions. The results show that a bug's error type and screen position were together a strong predictor of whether feedback would aid in identifying and correcting it, and that these two factors also significantly influenced how feedback affected the speed and order in which the bugs were corrected.

1. INTRODUCTION

Shneiderman describes three principles of direct manipulation systems (Shneiderman, 1992; italics added for emphasis):

1. *continuous representation* of the objects of interest;
2. physical actions or presses of labeled buttons instead of complex syntax; and
3. rapid incremental reversible operations *whose effect on the object of interest is immediately visible*.

Many systems today use visual or demonstrational mechanisms with immediate visual feedback to support forms of programming based upon these principles. Examples include spreadsheets, "watch what I do" macro recording systems, and visual programming languages aimed at audiences

†This work has been supported in part by the National Science Foundation under ASC95-23629 and an NSF Young Investigator Award.

ranging from experienced programmers to novice programmers to end users. In devising these direct-manipulation programming systems, many researchers and developers have expended great effort to support the features shown in italics above. But do these features actually help in the domain of programming?

Our research has focused on one programming task—debugging—in direct-manipulation programming systems. We previously reported early results of an experiment on how immediate visual feedback affects debugging in these systems (Wilcox, Atwood, Burnett, Cadiz, & Cook, 1997). The type of immediate visual feedback that we have been studying is termed "liveness" (Tanimoto, 1990), which means immediately-updated *semantic* visual feedback. One example of such feedback is seen in the automatic recalculation feature of spreadsheets, in the immediate display of updated cell values after every formula edit.

In our study, the primary result previously reported was that liveness was not the debugging panacea that developers of direct-manipulation programming systems might like to believe it is, but it did help significantly with accuracy in some cases. (For example, as Figure 1 shows, the effects of liveness on accuracy were different in the two problems of our study.) Liveness also significantly affected several aspects of the subjects' debugging speed and debugging behavior.

The question that arises then is, exactly what are the cases in which liveness helps? Our results pointed to three factors that, when varied, seemed to impact liveness's effects on debugging accuracy, debugging speed, or both. The three factors were: the type of problem (as illustrated in Figure 1), the type of user, and the type of bug. Although the importance of these factors in debugging is documented in classical debugging literature, the new information resulting from our study suggested that they may in fact be critical in determining whether *liveness* will help debugging or hinder it.

In this paper, we concentrate on the third factor, type of bug, presenting new evidence about the impact of liveness on debugging from the point of view of attributes of the seeded bugs. We focus on three kinds of questions: "which", "where", and "when". The "which" questions consider how the relationship between liveness and subjects' debugging accuracy varied with particular bugs and with collections of bugs grouped by error type. The "where" questions consider whether the screen position of a bug had different effects on accuracy in live versus non-live circumstances. The

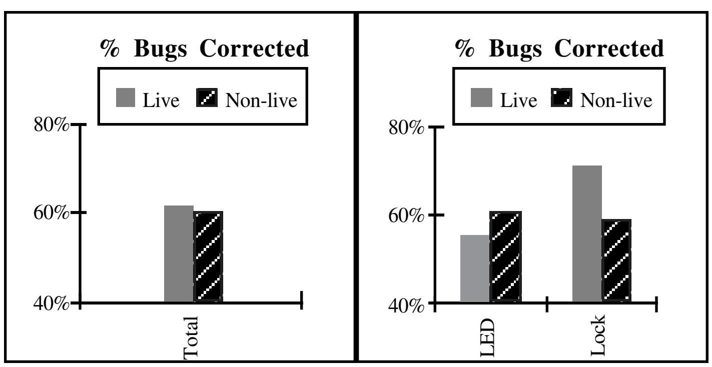

Figure 1: The negligible difference in overall accuracy in correcting bugs (left) was comprised of a variety of opposite totals. For example, note the differences between the totals for the two problems (right). The problem we call the LED problem had somewhat lower accuracy live than non-live, whereas the other problem, termed the Lock problem, had significantly higher accuracy live than non-live. (These problems will be described in detail in the next section.)

“when” questions consider how the speed and order in which the bugs were corrected varied in live versus non-live circumstances with the “which” and “where” attributes.

2. EXPERIMENT

Since this paper reports the results of new analysis of the data from the previous study (Wilcox, et al., 1997), we repeat the description of the experiment in this section.

The experiment’s goal was to learn whether liveness helped or impaired subjects’ debugging. We conducted the experiment in a lab with students seated one per workstation, using the visual programming language Forms/3 (Burnett & Ambler, 1994; Atwood, Burnett, Walpole, Wilcox, & Yang, 1996). First, the lecturer led a tutorial of Forms/3 in which each student actively participated by working with the example programs on individual workstations. Following the tutorial, the students were given two Forms/3 programs to debug, each with a 15-minute time limit: one that displays a graphical LED image of a numeric digit, and one that verifies a password mathematically in order to determine whether to unlock access to some hypothetical resource. All subjects debugged one of the programs using the normal Forms/3 system, which is live, and the other using the same system but with immediate feedback removed. Before carrying out the main experiment, we conducted a pilot study with four participants to test the experiment’s design.

The experiment was counterbalanced with respect to the liveness factor. Thus, each subject worked both problems, but half the subjects debugged the LED problem live and the Lock problem non-live, while the other half did the opposite. The assignments into these two groups were made randomly. The LED problem was always first, giving the Lock problem a learning advantage. (Since there was no assumption that the two problems were of equal difficulty, this did not affect the validity of the results.) The data collected during the experiment were post-problem and post-test questionnaires answered by the subjects, their on-line activities collected by an electronic transcripting mechanism, and their final debugged programs.

2.1. The Programming Environment

Specifying a program in Forms/3 is similar to specifying a program in a spreadsheet: the user creates cells and gives each cell a formula, and as soon as the user enters a formula, it is immediately evaluated and the result displayed. See Figure 2 for a sample Forms/3 program.

Forms/3 supports direct manipulation in several ways, among which are the ability to point at cells instead of typing a textual reference, and the ability to specify some kinds of formulas by demonstration. However, the most important attribute of Forms/3 with respect to this study is its automatic and immediate feedback about the effects of changes, i.e. its support of liveness. In languages supporting liveness, there is no compile phase, and no need to query variables in order to see values. Thus, in Forms/3, changing a cell’s formula causes its value, as well as the values of any cells dependent on the change, to be recalculated and displayed immediately and automatically.

To allow isolation of the liveness factor, we also created a non-live version of Forms/3 for this experiment, in which the automatic computation facilities were replaced by a button labeled *compile*, which would be used to submit the program to the system for execution. Each such submission took 90 seconds, simulating the time not only to compile, but also to set up and run a test as would be required in a non-live system. Upon completion, all the values would be displayed and would remain on display throughout the subject’s explorations until the subject actually made another formula change, at which point all the values were erased (since the values could no longer be determined without re-execution).

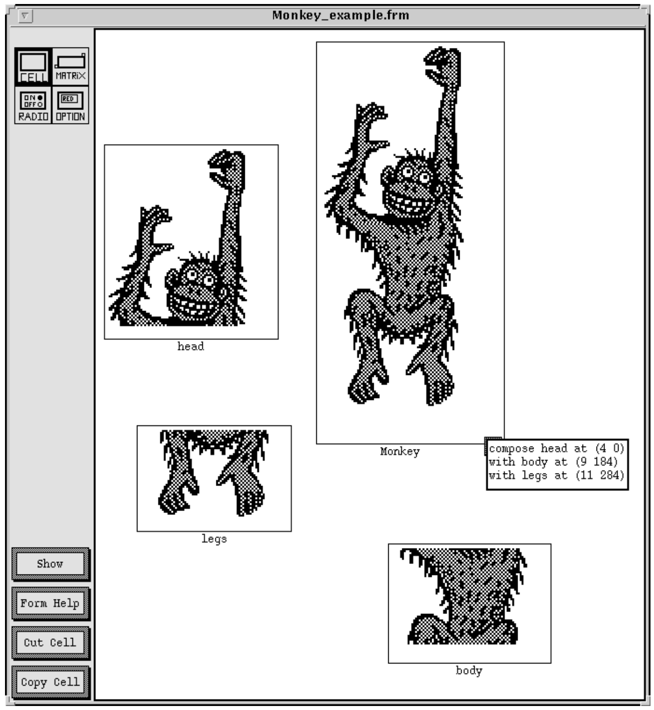

Figure 2: A Forms/3 program to display a whole monkey, given a collection of clip-art body parts. This program was used in the experiment's tutorial to demonstrate "compose" formulas like the one for cell Monkey. *This formula could be either typed or demonstrated by arranging the* head*,* body*, and* legs *cells as desired and rubberbanding the arrangement.*

2.2. The Subjects

The subjects were students enrolled in a senior level operating systems course. All subjects had experience programming in C or C++, 86% had used spreadsheets, and 55% had programmed in LISP. One student had seen Forms/3 before, but had not actually programmed in it. As Table 1 shows, there was very little background difference between the two groups. Although 12 subjects claimed professional experience, these were simply internships of one year or less for all but two subjects, one in each group. These 12 subjects' small amount of professional experience did not provide a performance advantage; there was actually a small negative correlation (-0.11) between the presence of professional experience and debugging accuracy.

	Cum. GPA	CS 411 grade	CS courses taken	Prog. langs. known	Subjects with prof. experience
	mean	counts	mean	mean	count
Group 1 (n=14): Live first	3.18	6 As 7 Bs 1 C	9.85	5.00	4
Group 2 (n=15): Live second	3.30	5 As 10 Bs	9.36	4.40	8

Table 1: Summary of the 29 subjects' backgrounds.

2.3. The Programs

To avoid limiting our study to exclusively graphically-oriented programs or exclusively mathematically-oriented programs, we chose one of each kind for the subjects to debug. The LED program was the first program the subjects debugged. This program produces graphical output similar to LED (light-emitting diode) displays on digital clocks. See Figure 3. Cell *output* is a composition of the lines needed to draw the digit in cell *input*. A useful aspect of this kind of program is that many subjects' prior experience with digital clocks helps them recognize the right answer when they see it.

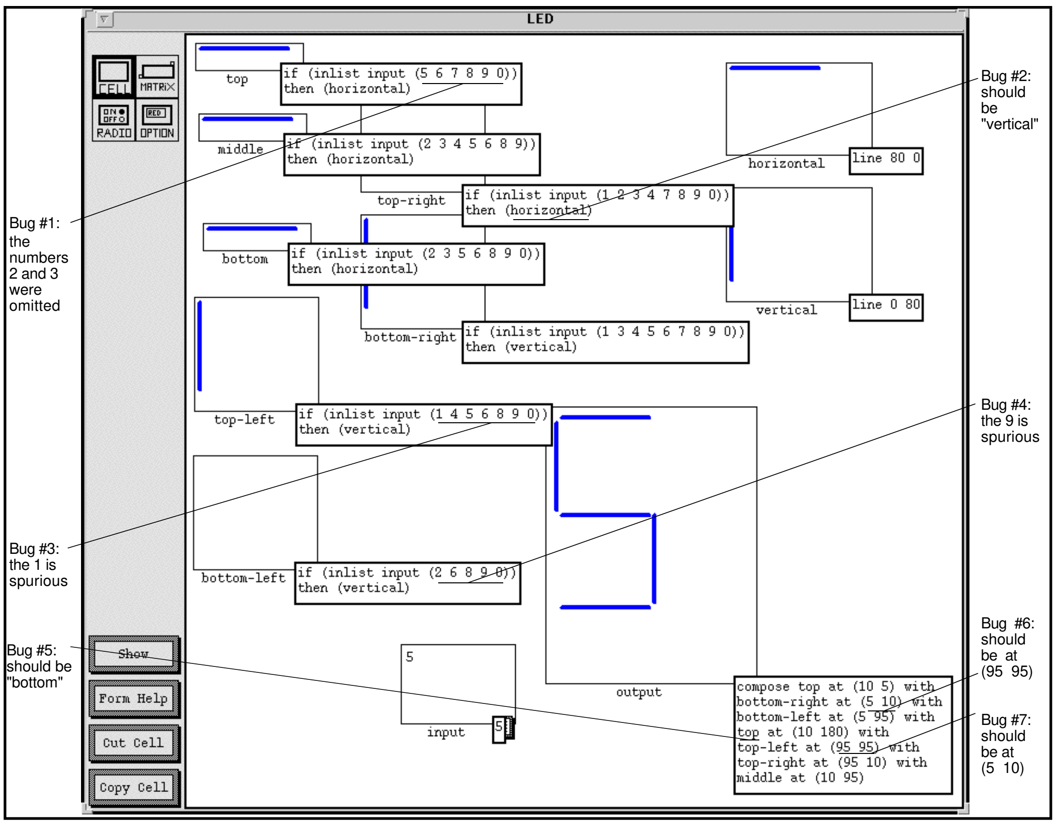

Figure 3: The LED program produces graphical output similar to LED (light-emitting diode) displays on digital clocks. Given a number (cell input*), the program draws it using line segments (cell* output*).*

The Lock program shown in Figure 4 was designed to emphasize mathematics instead of graphics. The basic premise was that the lock could only be unlocked with a two-step security mechanism. To gain access to such a lock, a person might insert an ID card containing his/her social security number and then key in two combinations. If the combinations were valid for that social security number, the lock would unlock. The Lock program used a formula similar to the quadratic formula to compute the two valid combinations from the social security number (in cells *a*, *b*, and *c*), and match them with the keyed-in input (cells *num1* and *num2*). The formula used was (-b±floor(sqrt(b^2+4ac)))/2a. Its differences from the true quadratic formula prevented complex roots and simplified the output formats. These points were explained to the subjects before the problem began.

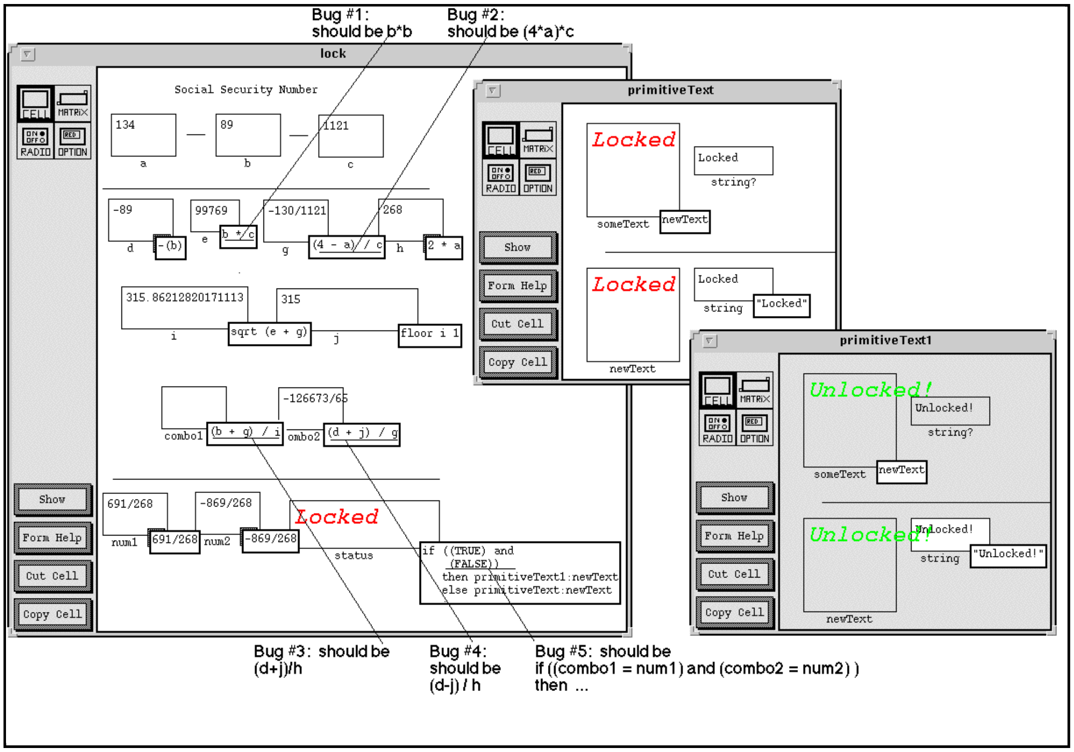

Figure 4: In the Lock problem, the subjects were told that the inputs num1 *and* num2 *were correct for the given social security number. When the bugs are removed, the values of cells* num1 *and* num2 *need to match cells* combo1 *and* combo2 *respectively to unlock the lock.*

If we had put the entire formula into a single cell, some subjects might have simply rewritten the formula rather than trying to find the bugs. (In fact, such behavior was observed during the pilot with a programming problem that we decided to replace for that reason.) To avoid this problem, we broke the formula's computation into parts using several intermediate cells, each with its own small formula.

2.4. Summary of Previously-Reported Results

Analyses of subject accuracy, subject behavior, and subject speed in identifying and correcting bugs, produced significant evidence that type of problem and type of user were important factors in determining whether liveness would help or impair debugging. To summarize the results reported in (Wilcox, et al., 1997):

Accuracy: The *overall* differences in accuracy were not significant in the live versus non-live versions. However, there were opposing totals for the LED and Lock problems (see Table 2). Subjects corrected slightly fewer bugs in the live version of the LED problem, but significantly more in the live version of the Lock problem. The effects of liveness also differed between the more-skilled and less-skilled subjects: subjects who corrected at least half the bugs in both problems did not show significant accuracy differences in the live versus non-live versions, but those who were below that level of accuracy in at least one problem performed significantly better in the live version than in the non-live version.

Behavior: There were significant differences live versus non-live in every aspect of behavior that

	LED (7 bugs)	Lock (5 bugs)	Total
Live	(14 subjects)	(15 subjects)	(29 subjects)
Identified	69.4%	94.7%	80.4%
Corrected	55.1%	70.7%	61.3%
Non-Live	(15 subjects)	(14 subjects)	(29 subjects)
Identified	69.5%	82.9%	74.9%
Corrected	60.9%	58.6%	60.0%

Table 2: Average percentages of bugs identified and corrected by subjects working live versus non-live. We use the term "identified bug" to mean one of the planted bugs for which the subject has changed a formula (but not necessarily correctly) and the term "corrected bug" to mean an identified bug that the subject succeeded at fixing. (This table is the basis for Figure 1, which was shown in the introduction to this paper.)

we measured. Subjects made significantly more changes to cells containing bugs live, spent significantly less time per change live, spent a significantly greater percent time making changes live, distributed their changes over time in a significantly more "burst"-oriented way live, and made their first change significantly earlier live.

Speed: At the 5-minute point, subjects who corrected at least half the bugs in both problems were significantly closer to being finished in the live version than in the non-live version, but this speed advantage was overtaken by the non-live version by the 10-minute point. Subjects who corrected fewer than half the bugs were slower to finish in the live version, which for these subjects, means that they "gave up" later. (This may be the reason for the accuracy advantage in the live version for this class of subjects).

3. RESULTS

The above results also contained indications that type of bug may be playing an important role in determining the circumstances in which liveness helped. In this section, we present evidence about this factor from a bug-centered perspective.

3.1. The "Which" Questions

The overall research question we address in this section is:

Which bugs were more (less) accurately identified/corrected in the live version?

To help identify patterns in the data, we categorized each bug by considering whether or not it was an incorrect or omitted reference to another cell (the same kind of error as referring to the wrong variable in a more traditional language, or grabbing the wrong object in a demonstrational language). We called this kind of bug a "reference bug", and all other bugs "non-reference bugs". An example of a reference bug is Bug 5 of the LED problem, in which cell *output* referenced cell *top* where it should have referenced cell *bottom*. An example of a non-reference bug is Bug 2 of the Lock problem, in which the error is in the choice of operators. There were 6 reference bugs and 6 non-reference bugs drawn from both problems. An important difference between these two error types is that reference bugs are non-local, meaning the bug comes from referring to the wrong entity external to this cell, whereas non-reference bugs are entirely local, coming from local errors such as incorrect or missing constants or incorrect operators within the cell.

We considered the subjects' accuracy live versus non-live for each bug separately and grouped by error type (reference/non-reference). A bug was termed "corrected" only if it was actually fixed; it was termed “identified” if the corresponding cell’s formula was edited, regardless of whether the edit successfully fixed the bug. In the case of multiple bugs in one cell, we looked at which portion of a formula was edited in order to determine which bug was identified.

Results. The clearest pattern that emerged from this analysis is that, where there were significant accuracy advantages, these advantages were associated with liveness. Tables 3 and 4 show the individual accuracy figures for each bug within each problem, and whether each was classified as a reference or non-reference bug. As Table 3 shows, no individual bug in the LED problem promoted any significant accuracy advantage live or non-live, and total performance, as predicted by Figure 1, shows a slightly lower number of bugs corrected live than non-live. However, for the Lock problem (Table 4), Bugs 2, 3, and 4 were corrected by significantly more subjects when using the live version (for each, χ^2=3.50, p=0.0614, df=1), and Bug 5 was identified by significantly more subjects when using the live version (χ^2=5.79, p=0.016, df=1). Interestingly, Bug 5 seemed to be particularly difficult to actually correct even after identifying it: although 13 of the 15 subjects working the Lock problem in the live version identified this bug, only 4 of them succeeded in their efforts to correct it. In fact, although so many more subjects identified that bug in the live version, one more subject actually corrected it in the non-live version than in the live version.

Grouping the data on each bug into error types (Table 5) did not suggest a pattern regarding which bugs were more easily corrected live. Rather, the accuracy differences live versus non-live were similar for the two error types: although subjects were slightly more successful live than non-live in identifying and correcting both reference bugs and non-reference bugs, these differences were not significant. The bottom section of the table shows that overall, the accuracy differences were small between the two error types, which seems to indicate that they were at about the same level of difficulty in this experiment.

Discussion. As these results show, some particular bugs promoted a significant accuracy advantage for liveness. In contrast, there were no significant advantages in the non-live version for any bug or for either error type (reference or non-reference). The pattern for which there was a liveness advantage did not seem to be explained by the bug’s error type, since the differences in accuracy rates live versus non-live were similar small amounts for both types. However, as the next section shows, error type did indeed play an important role when considered in combination with the bugs’ position categories.

We elected to analyze reference bugs versus non-reference bugs after the experiment was run, and for this reason, the distribution of reference and non-reference bugs was not equal between the two problems—there were more non-reference bugs than reference bugs in the LED problem, and the opposite was true in the Lock problem. Thus, problem attributes (such as problem domain) may be confounding factors, and a redesigned experiment is needed to more precisely separate error type from other attributes of the programs.

Although we did not find accuracy differences between the two error types in this experiment, they could arise in other direct-manipulation programming systems, and in fact there is a good chance that in some systems of this class, reference bugs would be more difficult than they were in our experiment. The reason is that, in some direct-manipulation programming systems, there are few, if any, names given to objects by users, and hence the static representation of programs in these systems often refers to objects by non-mnemonic system-assigned names (e.g., R1C2 or Circle-1234). This is especially common in spreadsheets and in programming by demonstration. From

	Bug 1 Non-Ref	Bug 2 Ref	Bug 3 Non-Ref	Bug 4 Non-Ref	Bug 5 Ref	Bug 6 Non-Ref	Bug 7 Non-Ref
Live (14 subjects)							
Identified	11 (79%)	11 (79%)	10 (71%)	11 (79%)	8 (57%)	10 (71%)	11 (79%)
Corrected	11 (79%)	11 (79%)	8 (57%)	7 (50%)	6 (43%)	7 (50%)	8 (57%)
Non-Live (15 subjects)							
Identified	14 (93%)	12 (80%)	10 (67%)	12 (80%)	6 (40%)	7 (47%)	8 (53%)
Corrected	14 (93%)	12 (80%)	10 (67%)	9 (60%)	4 (27%)	5 (33%)	6 (40%)

Table 3. The number of subjects who identified and corrected each bug in the LED problem. The reference bugs are labeled "Ref", and the others are non-reference bugs. 3 bugs (Bugs 5, 6, and 7) had a slightly higher correction rate in the live version, as compared to 4 bugs (Bugs 1, 2, 3, and 4) in the non-live version.

	Bug 1 Ref	Bug 2 Non-Ref	Bug 3 Ref	Bug 4 Ref	Bug 5 Ref
Live (15 subjects)					
Identified	14 (93%)	14 (93%)	15 (100%)	15 (100%)	13 (87%)
Corrected	13 (87%)	12 (80%)	12 (80%)	12 (80%)	4 (27%)
Non-Live (14 subjects)					
Identified	12 (86%)	12 (86%)	14 (100%)	12 (86%)	8 (57%)
Corrected	12 (86%)	8 (57%)	8 (57%)	8 (57%)	5 (36%)

Table 4. The number of subjects who identified and corrected each bug in the Lock problem. The reference bugs are labeled "Ref". Bugs 1, 2, 3, and 4 were all corrected more frequently in the live version, but only Bug 5 was corrected more frequently non-live. The correction rate difference was significantly larger live than non-live for Bugs 2, 3, and 4, and the identification rate was significantly larger live than non-live for Bug 5.

	Reference Bugs Identified/Corrected		Non-Reference Bugs Identified/Corrected	
Live (29 subjects)	88 possible		85 possible	
Identified	76	(86%)	67	(79%)
Corrected	58	(66%)	53	(62%)
Non-Live (29 subjects)	86 possible		89 possible	
Identified	64	(74%)	63	(71%)
Corrected	49	(57%)	52	(58%)
Total (29 subjects)	174 possible		174 possible	
Identified	140	(80%)	130	(75%)
Corrected	107	(61%)	105	(60%)

Table 5: Bug identification and correction were somewhat better live than non-live, but not significantly so. Overall, there were no significant accuracy differences between reference bugs and non-reference bugs.

work such as (Shneiderman & McKay, 1976) showing the importance of mnemonic names to debugging, we would expect this use of non-mnemonic names to increase the difficulty of debugging reference bugs. If further experimentation verifies this hypothesis, it would suggest to designers of such systems providing users extra help in tying the referenced objects together with the parts of the program that refer to them in order to ease debugging. The dataflow arrows available in some spreadsheets are an example of the kind of technique that can be used toward this end.

3.2. The "Where" Questions

The overall research question in this section is:

What effects did bug location have on which bugs were more (less) accurately identified/corrected in the live version?

To investigate this question, we divided the bugs into groups based on their location on the screen. We divided the screen horizontally into thirds: top, middle, and bottom. There were an equal number of reference, non-reference, and total bugs in each third (see Table 6).

	Top Third	Middle Third	Bottom Third
LED Problem	1 Ref: #2 1 Non-Ref: #1 2 Total	2 Non-Ref: #3, #4 2 Total	1 Ref: #5 2 Non-Ref: #6, #7 3 Total
Lock Problem	1 Ref: #1 1 Non-Ref: #2 2 Total	2 Ref: #3, #4 2 Total	1 Ref: #5 1 Total
Total	2 Ref 2 Non-Ref 4 Total	2 Ref 2 Non-Ref 4 Total	2 Ref 2 Non-Ref 4 Total

Table 6. Distribution of bugs among the screen thirds for each problem.

Results. Three interesting facts emerged. First, considering screen position alone without any additional factors, it is clear from the "Totals" section of Table 7 that the higher a bug was on the screen, the more likely subjects were to identify and correct it. These differences were significant: Fisher's Exact Test[1] showed that subjects were more likely to correct bugs in the top section than in the middle ($p=.009$), and more likely both to identify and to correct bugs in the middle than in the bottom (identified: $p=.056$; corrected: $p=.046$). In fact, this pattern was so pronounced, *none* of the subjects corrected a bug in the middle or bottom sections of the screen without also correcting at least one bug in every section higher.

Second, from the screen position statistics live versus non-live (given in the top two sections of the table), it is clear that, although a low screen position was an accuracy disadvantage in both versions, the effect of low position was offset to some extent by liveness. One piece of evidence is the fact that, in the bottom third of the screen, a significantly greater number of subjects identified bugs in the live version than in the non-live version (Fisher's Exact Test, $p=.009$). Another indication is that the accuracy differences linked to screen position were more pronounced

[1] Fisher's Exact Test (Ramsey & Schafer 1997) is an alternative to the χ^2 test that, unlike χ^2, can be used for 2x2 tests in which an expected value is less than 5.

in the non-live version: using Fisher's Exact Test on subjects' accuracy at different positions, the live version had 2 significant position-related effects whereas the non-live version had 5 significant position-related effects, most of which were more pronounced non-live than the corresponding effect in the live version. A summary of these relationships among accuracy, position, and liveness is shown in Figure 5.

Third, when the effects of liveness and position on accuracy are grouped separately for reference and non-reference bugs, the opposing totals for these two error types become visible. Considering first the commonality between the reference and non-reference bugs, as the bottom sections of Tables 8 and 9 show, accuracy rates for both the reference and non-reference bugs were quite affected by

	Bugs Identified/Corrected in the Top Third		Bugs Identified/Corrected in the Middle Third		Bugs Identified/Corrected in the Bottom Third	
Live	58 possible		58 possible		57 possible	
Identified	50	(86%)	51	(88%)	42	(74%)
Corrected	47	(81%)	39	(67%)	25	(44%)
Non-Live	58 possible		58 possible		59 possible	
Identified	50	(86%)	48	(83%)	29	(49%)
Corrected	46	(79%)	35	(60%)	20	(34%)
Total	116 possible		116 possible		116 possible	
Identified	100	(86%)	99	(85%)	71	(61%)
Corrected	93	(80%)	74	(64%)	45	(39%)

Table 7: Overall, the higher the bugs were on the screen, the more success the 29 subjects had identifying and correcting them. Screen position made a bigger difference on accuracy in the non-live version than in the live version, as Figure 5 brings out.

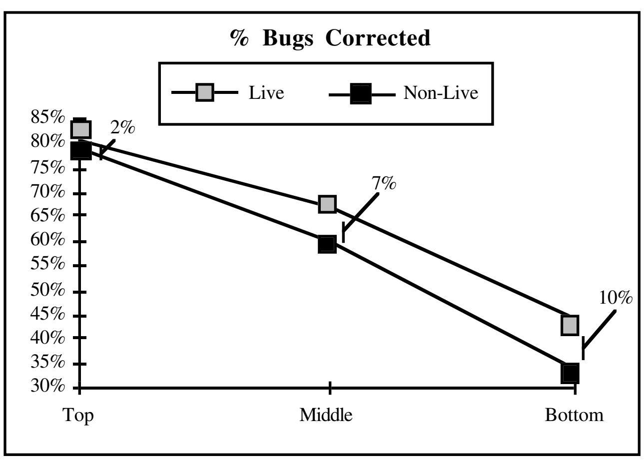

Figure 5: Summary of Table 7. The annotations point out the growing differences in bug correction rate live versus non-live as the bug placement moves down the screen. The disadvantage of a low position on the screen was more detrimental to bug correction rate in the non-live version than in the live version.

position, consistently favoring higher positions in all except one of the "Total" row entries in both tables. Fisher's Exact Test verifies these tables, showing that subjects were significantly more successful correcting at least one bug in the top than in the middle, in the middle than in the bottom, and in the top than in the bottom, both for reference bugs (p=.021, .032, .00002 respectively) and for non-reference bugs (p=.051, .091, .00006 respectively). Identification rates were affected similarly by position: significantly more subjects identified bugs in the higher sections for both types of bugs (reference: middle vs. bottom p=.012; non-reference: middle vs. bottom p=.051, top vs. bottom p=.021).

The opposing totals are in how liveness affected bugs in the three positions of the screen differently for reference bugs than for non-reference bugs. As Figure 6 points out, for the middle third of the screen, accuracy was significantly higher for reference bugs than non-reference bugs in the live version. (Fisher's Exact Test, corrected: p=.045, identified: p=.002). On the other hand, at the bottom of the screen, liveness's effects favored the non-reference bugs over the reference bugs. (The bottom section did not lend itself to straightforward statistical analysis, since the distribution of these bugs made it difficult to isolate paired data from independent data.)

Discussion. These "where" findings show that position had significant effects on subjects' abilities to debug. Moreover, there is strong evidence that the disadvantage of a lower screen position was partially offset by liveness. From the fact that reference bugs had about the same accuracy rate as non-reference bugs, one might expect the effects of position and liveness to be about the same on both error types. Yet, the "where" findings brought out a significant "which" difference, showing that for the middle section of the screen, the effects of liveness were significantly different for reference bugs than for non-reference bugs.

The "where" accuracy results presented in Figure 6 represent the numbers of bugs corrected. If we had instead graphed bugs corrected as a percent of bugs identified, a slightly different picture would have emerged. In that set of statistics, the effects of liveness at the top and middle would be similar to those shown in Figure 6 for both reference and non-reference bugs, but the bottom third for both bug types would have shown better success non-live than live. However, we do not place as much faith in that view as the one presented in Figure 6, because our classification of which bugs were identified was only an approximation, as compared to the exact data regarding which bugs were corrected. (Recall that bugs classified as "identified" were those whose formulas were edited; but it is possible that a subject identified a bug without having gotten around to editing it, or that a subject might have experimentally modified a cell's formula without having decided whether it had a bug in it.)

How do the "where" findings relate to the previously-reported overall accuracy findings of this study? Recall that the LED problem was not significantly easier live or non-live, but that the Lock problem produced significantly better accuracy results live than non-live. Also recall from the previous section that most of the bugs in the Lock problem were reference bugs and most of the bugs in the LED problem were non-reference. Although it is clear that whether the bugs were reference or non-reference bugs did not alone determine whether a bug would be easier to correct with liveness or without it, the "where" findings show that when position was added, liveness brought a significant advantage to some reference bugs (namely, those in the middle of the screen). This may explain the different effects liveness had on accuracy in the two problems.

	Bugs Identified/Corrected in the Top Third		Bugs Identified/Corrected in the Middle Third		Bugs Identified/Corrected in the Bottom Third	
Live	29 possible		30 possible		29 possible	
Identified	25	(86%)	30	(100%)	21	(72%)
Corrected	24	(83%)	24	(80%)	10	(34%)
Non-Live	29 possible		28 possible		29 possible	
Identified	24	(83%)	26	(93%)	14	(48%)
Corrected	24	(83%)	16	(57%)	9	(31%)
Total	58 possible		58 possible		58 possible	
Identified	49	(84%)	56	(97%)	35	(60%)
Corrected	48	(83%)	40	(69%)	19	(33%)

Table 8: Reference bugs.

	Bugs Identified/Corrected in the Top Third		Bugs Identified/Corrected in the Middle Third		Bugs Identified/Corrected in the Bottom Third	
Live	29 possible		28 possible		28 possible	
Identified	25	(86%)	21	(75%)	21	(75%)
Corrected	23	(79%)	15	(54%)	15	(54%)
Non-Live	29 possible		30 possible		30 possible	
Identified	26	(90%)	22	(73%)	15	(50%)
Corrected	22	(76%)	19	(63%)	11	(37%)
Total	58 possible		58 possible		58 possible	
Identified	51	(88%)	43	(74%)	36	(62%)
Corrected	45	(78%)	34	(59%)	26	(45%)

Table 9: Non-Reference bugs.

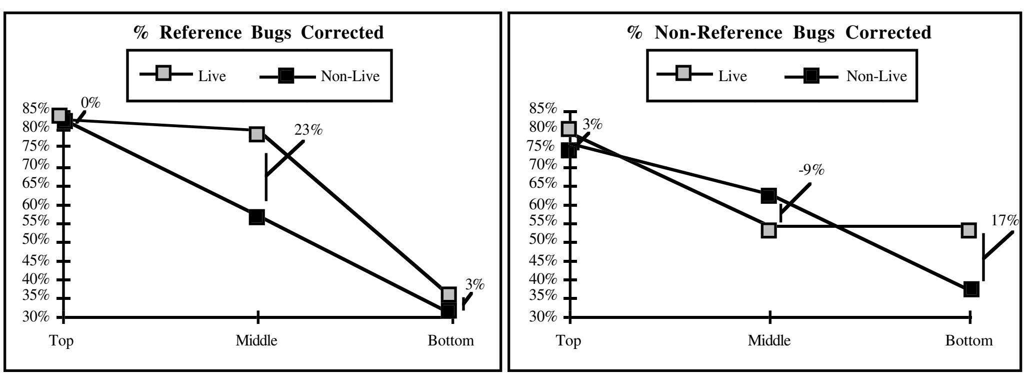

Figure 6: There were opposing results for reference bugs and non-reference bugs regarding how liveness affected correction rate for bugs in the middle and bottom of the screen.

The importance of screen position in debugging was not surprising to us for the non-live version, but it was for the live version. Even more surprising to us was the interaction between position and error type. Liveness's pronounced effect on reference bugs in the middle of the screen might be explained by the fact that those bugs are referring to the wrong cell; thus, when a cell changes, subjects' attention may be drawn to the unexpected movement from the value of a reference bug cell changing when it shouldn't. This explanation would not seem to be particular to middle-positioned cells, but it appears that top-positioned bugs were made so easy by the advantage of their position, there was no further advantage to be gained by liveness. (This advantage of the top-positioned cells seems to be tied to debugging order, which will be discussed in the next section.)

The design of this experiment was such that each program fit entirely on one screen, and thus position of the bug relative to the screen is the same thing as position of the bug relative to the program. This raises interesting follow-up questions related to which of those two notions of position would cause differences in larger programs. For example, consider the greater influence of screen position on the non-live version than on the live version. Assuming that the reason is that the non-live version's values were not visible, then it is possible that multi-screen programs in live systems would be more heavily influenced by relative position in the program than the small programs of this study, because not all the values can be visible at once (thereby approximating the non-live situation). Another follow-up question regarding larger programs is that, if a program is larger than a physical screen, requiring navigation to see the entire thing, will rearranging the program on the screen cause the bugs newly positioned at the top to be identified and fixed with the increased accuracy associated with top-of-screen bugs in the single-screen programs of our study? If so, this would suggest that direct-manipulation programming systems might benefit from an automatic "debugging coach" feature that would help users find the bugs by suggesting (or performing in some semi-automated way) movement of different sections of the program to the top of the screen.

3.3. The "When" Questions

Previous sections have reported accuracy differences in bug identification and correction live versus non-live. In this section we discuss how these differences occurred over time. The overall research question of this section then is:

What effects did liveness have on debugging order and on debugging speed?

Results. The importance of position on accuracy shown by the last section was a strong hint that subjects may have chosen a position-oriented strategy for debugging. In this section, evidence abounds that subjects' debugging order was indeed heavily influenced by screen position. Tables 10 and 11 show the order in which the bugs were corrected, with the maximums highlighted. Debugging order was influenced by position in both versions in terms of where subjects began debugging (at the top of the screen) and where they debugged last (at the bottom of the screen). For example, when subjects managed to correct at least one bug, that first correction was in the top third of the screen 22 out of 24 times in the live version and 24 out of 27 times in the non-live version. Also, in the 11 out of 16 performances in which a subject corrected at least 5 bugs, the subjects finished the debugging session at the bottom of the screen. (When fewer than 5 bugs were corrected, most subjects never got to the bottom of the screen at all.)

However, the influence of position on debugging order was much more pronounced in the non-live version than in the live version. This can be seen by the non-live version's stronger adherence to the diagonal pattern in both tables, i.e., correcting the bug in the first (top leftmost) position first, the bug in the second position second, and so on. Also, for *all* the subjects who, working non-live, corrected a bug at the bottom of the screen in the LED problem, it was the last bug that they corrected.

	Bug 1	Bug 2	Bug 3	Bug 4	Bug 6	Bug 5	Bug 7
Live							
1st	5	4	2	0	0	0	0
2nd	1	2	5	1	0	1	1
3rd	1	3	0	2	1	1	2
4th	2	1	0	1	2	1	2
5th	0	1	1	0	4	0	2
6th	2	0	0	0	0	3	1
7th	0	0	0	3	0	0	0
Non-Live							
1st	9	3	0	2	0	0	0
2nd	2	4	6	1	0	0	0
3rd	2	1	4	4	0	0	0
4th	1	4	0	2	1	1	0
5th	0	0	0	0	2	3	1
6th	0	0	0	0	2	0	3
7th	0	0	0	0	0	0	2

Table 10: Correction order frequency for each bug of the LED problem. For example, 5 subjects corrected Bug 1 first in the live version, 1 subject corrected Bug 1 second, and so on. The bugs are listed in the top-down order they appear on the screen (which is why Bug 6 is listed before Bug 5). Gray highlighting indicates the column maximums. A diagonal pattern of gray starting top left indicates strong adherence to position in the order bugs were corrected. The non-live version follows this pattern more closely than does the live version.

	Bug 1	Bug 2	Bug 3	Bug 4	Bug 5
Live					
1st	13	0	0	0	0
2nd	0	6	5	1	1
3rd	0	2	5	5	0
4th	0	4	2	5	0
5th	0	0	0	1	3
Non-Live					
1st	10	2	0	0	1
2nd	2	5	0	2	0
3rd	0	1	7	0	0
4th	0	0	0	6	1
5th	0	0	1	0	3

Table 11: Correction order frequency for each bug of the Lock problem. Gray highlighting indicates the maximum of each column. Note the strength of the adherence in the non-live version to a position-oriented debugging order.

The differences in influence of position live versus non-live in the LED problem can be seen even more clearly by partitioning debugging strategies into two position-oriented types, plus "other" for everything else: top-first, in which subjects corrected both bugs in the top third before proceeding downward, and left-first, in which subjects corrected both bugs on the left side of the screen before proceeding rightward. (Only subjects who successfully corrected at least 2 bugs on a problem could be classified using this scheme.) For the LED problem, significantly fewer subjects were position-oriented live than non-live (5 of 11 subjects were position-oriented live, versus 10 of 13 subjects position-oriented non-live; χ^2=6.27, p<0.025, df=1). However, for the Lock problem the strategies live versus non-live were similar: 11 of 13 subjects were position-oriented live and 7 of 9 were position-oriented non-live. The top-first strategy was much more popular than the left-first strategy: 29 of the position-oriented subjects used top-first, compared with only 5 using left-first.

Turning now to debugging speed, we analyzed how liveness related to speed by dividing the 15-minute debugging session into three 5-minute time periods and counting how many subjects had identified/corrected each bug in those periods. The results showed that for 7 of the bugs (4 of the 7 LED bugs and 3 of the 5 Lock bugs), when subjects successfully corrected the bug, they did so significantly faster live than non-live. In contrast to this, there were only 2 bugs in which the non-live version was significantly faster. (Both were in the LED problem). The correction time periods for each bug are shown in Tables 12 and 13.

	Bug 1	Bug 2	Bug 3	Bug 4	Bug 5	Bug 6	Bug 7	Total
Live (14 subjects)								
0-5 min	3	4	4	0	1	1	1	14
5-10 min	6	5	4	2	2	3	5	27
> 10 min	2	2	0	5	3	3	2	17
Non-Live (15 subjects)								
0-5 min	8	1	4	2	0	0	0	15
5-10 min	5	7	6	6	0	0	0	24
> 10 min	1	4	0	1	4	5	6	21

Table 12. Number of subjects correcting each LED bug during each time period. Speed was greater for some bugs live and others non-live in this problem, and the totals column shows that overall for this problem, the number of bugs corrected in each time period were about the same live versus non-live.

	Bug 1	Bug 2	Bug 3	Bug 4	Bug 5	Total
Live (15 subjects)						
0-5 min	12	7	9	9	2	39
5-10 min	1	3	1	1	1	7
> 10 min	0	2	2	2	1	7
Non-Live (14 subjects)						
0-5 min	8	3	1	0	1	13
5-10 min	2	2	3	5	2	14
> 10 min	2	3	4	3	2	14

Table 13. Number of subjects correcting each Lock bug during each time period. In this problem, several of the bugs were corrected significantly faster live, and the totals column shows that in the live version, approximately 3/4 of the bugs that would ultimately be corrected had been done in the first 5 minutes, while in the non-live version only about 1/3 of the bugs were corrected that early.

By Fisher's Exact Test, the 7 bugs in Tables 12 and 13 corrected significantly earlier live than non-live were: LED Bugs 5, 6, and 7 by 10 minutes ($p=0.09661$, $p=0.04214$, $p=0.01685$ respectively), Lock Bug 2 by 10 minutes ($p=0.04571$), and Lock Bugs 3 and 4 by 5 minutes ($p=0.003648$, $p=0.000499$) and by 10 minutes ($p=0.004571$, $p=0.09739$). The 2 bugs corrected significantly earlier non-live were LED Bug 1 by 5 minutes ($p=0.08212$) and LED Bug 4 by 10 minutes ($p=0.06786$). The totals in these tables, showing overall correction speeds for each time period, were significantly different live versus non-live for the Lock problem ($\chi^2=28.4$, $p<0.005$, $df=2$) and overall ($\chi^2=9.1$, $p<0.025$, $df=2$). Interestingly, the identification speed (not shown in these tables) was significantly faster live than non-live for both the LED and the Lock problem (LED: $\chi^2=7.7$, $p<0.025$, $df=2$; Lock: $\chi^2=26.4$, $p<0.005$, $df=2$).

How do these faster speeds live relate to the results of the previous sections, namely the interaction between the reference/non-reference factor and screen position? To analyze this question, we separated the reference bugs from the non-reference bugs and arranged them by screen position, and looked at the speed differences live versus non-live. The results are shown in Tables 14 and 15, and summarized in Figure 7. *All* of the reference bugs were corrected earlier live than non-live, 4 of the 6 significantly so (Wilcoxon rank sum: Lock 1: $p<0.005$; Lock 3: $p<0.005$; Lock 4: $p<0.005$; LED 5: $p<0.05$). Some of the non-reference bugs were corrected earlier live and some earlier non-live; the only significant difference was that LED 7 was corrected earlier live (Wilcoxon rank sum, $p<0.05$).

Discussion. From the "when" results, it is clear that position had a strong impact on subjects' debugging order, but much more so in the non-live version than in the live version. Whether the particular way subjects traversed these positions (top-down, left-right) was tied to culture at least in part, as per Western reading practices, was not studied but seems likely, since most of the subjects in this study were American.

The information about debugging order sheds new light on the screen position issue as well. The bottom third was corrected quite late for many subjects; in fact, in the non-live version of the LED problem, no subject identified or corrected a single one of the bottom bugs until 10 minutes into the experiment. Three possibilities present themselves: the bottom of the screen may really be more difficult no matter what bugs are there, or the bugs that were placed at the bottom of the screen were more difficult for some other reason, or the subjects' top-first debugging order simply means they ran out of time before they got to the bottom of the screen. Further experimentation of these possibilities is required.

Regarding debugging efficiency, the "when" data provided strong evidence that liveness significantly improved efficiency of correcting the bugs that were ultimately corrected, especially for reference bugs. At first glance, this evidence of increased debugging speed under liveness would seem to contradict the results we previously published, in which we reported that in high performances, subjects started out working significantly faster live, but by the 10-minute point their non-live performances were faster. (High performances were those in which a subject corrected at least half the bugs in a problem; however, there were only 3 subjects who corrected all the bugs in both problems.) We also reported that in low performances, subjects consistently had a higher engagement rate live, which we speculated might explain the higher accuracy rate live for subjects who had difficulties with at least one problem.

What the new bug-centered evidence presented in this section considers is speed only as it relates to bugs that were successfully corrected. Thus, the high performers' speed drop-off at the 10-minute point in the previous report could have been when those subjects began to have difficulty with the bugs they were working on, at which point they may have switched to the more engaged (slower) behavior that was present throughout for the low performers. In essence then, the previous report shows that when a bug was proving difficult to correct, subjects were more likely to stay engaged longer with liveness, and this paper shows that when a bug was actually corrected, subjects were more likely to correct it faster with liveness.

	Top		Middle		Bottom	
	Lock 1	LED 2	Lock 3	Lock 4	Lock 5	LED 5
Live	82.31 (109.31)	406.73 (210.76)	302.92 (275.95)	340.42 (270.58)	437.75 (213.48)	574.33 (188.83)
Non-Live	328.75 (230.55)	480.75 (174.56)	607.13 (280.19)	597.00 (274.12)	611.80 (379.49)	735.50 (86.90)

Table 14: Mean (sd) time in seconds at which each reference bug was corrected. All the reference bugs were corrected at an earlier mean time live than non-live.

	Top		Middle		Bottom	
	LED 1	Lock 2	LED 3	LED 4	LED 6	LED 7
Live	390.91 (219.91)	320.33 (255.93)	297.50 (137.61)	639.29 (151.05)	587.43 (231.04)	570.88 (236.50)
Non-Live	305.79 (210.88)	465.13 (286.37)	334.90 (98.51)	414.89 (193.47)	733.60 (96.54)	756.83 (61.85)

Table 15: Mean (sd) time in seconds at which each non-reference bug was corrected. Some of the non-reference bugs were corrected earlier live and some earlier non-live.

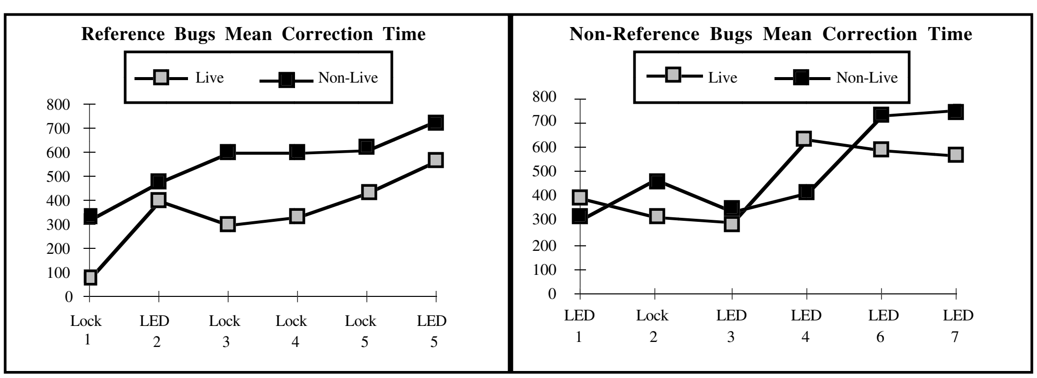

Figure 7: Summary of Tables 14 and 15: Mean time in seconds at which each bug was corrected. The reference bugs at every screen position were corrected earlier live than non-live. This was not the case for the non-reference bugs.

4. GENERAL DISCUSSION

This study has been the first to investigate the effects of liveness on debugging in direct-manipulation programming systems, and as such, provides only the beginning of the evidence needed to understand this issue. In fact, there has been very little investigation into debugging at all in direct-manipulation programming systems; the only directly pertinent work we have been able to locate is a small body of evidence about debugging in spreadsheets and in HyperCard, which shows that users spend considerable time debugging and that they have difficulty succeeding at it (Brown & Gould, 1987; Nardi & Miller, 1991; Eisenstadt, 1993). There have been several studies of other aspects of visual programming languages (Cunniff & Taylor, 1987; Green, Petre, & Bellamy, 1991; Moher, Mak, Blumenthal, & Leventhal, 1993; Pandey & Burnett, 1993; Petre, 1995; Modugno, Corbett, & Myers, 1996; Whitley, 1997), but the effects of liveness on debugging (or on any other task) in these languages were not studied.

Our evidence connecting liveness to engagement in enhancing the persistence with which subjects pursued bugs is consistent with the findings of others who have considered the importance of feedback and engagement to programmers. For example, in the context of algorithm animation systems, Lawrence, Badre, and Stasko learned that, whereas passively observing animations did not significantly increase students' understanding of algorithms, active involvement in the creation of input values to the animations did significantly increase understanding (Lawrence, Badre, & Stasko, 1994). There are also several studies showing that programmers in traditional languages perform significantly more interactions when there are shorter response times (Shneiderman, 1992), which supports both our engagement results and speed results favoring liveness.

The connection between liveness, engagement, and debugging accuracy, is also consistent with work such as Green's and Petre's research into cognitive aspects of programming languages (Green & Petre, 1996) and Gugerty's and Olson's studies of debugging behavior (Gugerty & Olson, 1986). They report that novices and experts alike tend to rely strongly on what Green and Petre term "progressive evaluation" (ability to execute bits and pieces of programs immediately, even if program development is not yet complete). According to this body of work, novices require this feature to find bugs, and experts, who can debug without it if necessary, rely on it even more heavily than novices when it is available. Although our subjects were new to Forms/3, their prior background in debugging and in spreadsheets seems to place them about halfway between the novice and expert categories, and our analysis turned up both speed and accuracy benefits from liveness for this population.

However, there are several studies of problem-solving behavior that point to feedback being unhelpful or even detrimental to problem solving on computers. One example is the work of Martin and Corl, who contributed to a long line of response time studies; see (Shneiderman, 1992) for a survey of that literature. While they, like their predecessors, found that faster response time improved productivity for simple data entry tasks, contrary to their predecessors they found that the effect declined for more complex data entry tasks and disappeared entirely for problem-solving tasks performed using a statistical package (Martin & Corl, 1986). Consistent with this are studies by Gilmore and by Svendsen, which showed that too much feedback and responsiveness can interfere with people's problem-solving ability in solving puzzles (Gilmore, 1995; Svendsen, 1991), a task with much in common with debugging. O'Hara's findings are similar; in three experiments that varied the (time) cost of performing certain operations, he found that high costs encouraged people to engage in more advance planning, resulting in more efficient solutions in terms of number of operations (O'Hara, 1994).

It should be noted that an important difference between debugging and the other kinds of problem-solving tasks that have been studied is that in debugging, the subjects must decide (analytically or experimentally) whether they are finished, and doing so takes time and is itself subject to error. In contrast to this, in such tasks as puzzle-working, statistical package tasks, and complex data entry tasks, it is usually obvious when the task has been completed. This factor could be important enough to explain why feedback was detrimental in problem-solving studies that were not about debugging, while it showed benefits in this debugging study and in the research about progressive evaluation.

Still, a possibility that is raised by these studies on problem-solving for the realm of debugging is that feedback encourages people to place too great an emphasis on ad-hoc testing, instead of choosing a more analytical approach. This question remains open. It is true that the "when" results strongly suggest that subjects used a debugging strategy for working live that was different from their position-oriented strategy non-live. However, it is not clear that adherence to position is an analytical approach, or that a position-oriented strategy would be followed in a larger program, or that the subjects' lower adherence to position in the live version meant they were using ad-hoc methods. Since the strategies used live by our subjects were more successful from a speed perspective than were the strategies used non-live, further investigation into just what these strategies were would help shed some light upon this question.

In the future, the debugging strategy that subjects follow could be influenceable by the programming environment, and this in turn could affect debugging success. That visual devices can make a difference in programmers' understanding has previously been shown in work on typographic devices for textual languages (Oman & Cook, 1990), and an opportunity for a visual environment is to use visual feedback devices to focus the user's attention on relevant areas of the program, perhaps thereby improving users' debugging accuracy or speed further. For example, in a system like Forms/3, the effects of the most recent formula change could be highlighted, which might encourage users to more consistently verify their most recent bug correction before moving on to another bug.

5. CONCLUSION

Our previous study was the first to study the effects of liveness on debugging in direct-manipulation programming systems. Its opposing totals—some favorable to liveness and some showing no benefit—pointed toward at least three factors that may determine when liveness helps with debugging. One of these factors was type of bug.

In this paper, we have presented a bug-centered analysis of data from that experiment. The key new results from this analysis were:

- Screen position was a critical factor in both the likelihood of a bug being corrected and in the order subjects followed when debugging. However, both of these aspects were less strongly influenced by position in the live version than in the non-live version.
- A bug's error type (reference versus non-reference) did not alone seem to affect whether the bug would be corrected live, non-live, or in total. However, when combined with screen position, there were significant differences in accuracy between the two error types live versus non-live.
- A bug's error type was also a predictor of how liveness would affect speed: all reference errors were corrected earlier live than non-live, many by a significant amount. In contrast to this, the non-reference errors were mixed, some being corrected earlier live and others non-live.

An important aspect of how these findings supplement the previous report is that, of the new results obtained that showed statistical significance, almost all were favorable to liveness.

6. ACKNOWLEDGMENTS

The experiment from which this paper's results ensue owes important parts of its design and administration to the work of Eric Wilcox, J. J. Cadiz, and John Atwood. Thanks are also due to Margaret Burnett's Fall '95 CS 411 students for their enthusiastic participation in the study, and to Rebecca Walpole and her Summer '95 CS 381 students for their help during the pilot. Finally, we

thank the anonymous reviewers for their very helpful comments.

7. REFERENCES

Atwood, J., Burnett, M., Walpole, R., Wilcox, E., & Yang, S. (1996). Steering Programs Via Time Travel. *1996 IEEE Symp. Visual Languages*, Boulder, CO, Sept. 3-6, 1996, 4-11.

Brown, P. & Gould, J. (1987). Experimental Study of People Creating Spreadsheets. *ACM Transactions on Office Information Systems* 5, 258-272.

Burnett, M. & Ambler, A. (1994). Interactive Visual Data Abstraction in a Declarative Visual Programming Language. *Journal of Visual Languages and Computing* 5(1), Mar. 1994, 29-60.

Cunniff, N. & Taylor, R. (1987). Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension. In G. Olson, S. Sheppard, E. Soloway (Eds.), *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex.

Eisenstadt, M. (1993). Why HyperTalk Debugging Is More Painful Than It Ought To Be. *HCI'93 Conference*, pub. as *People and Computers VIII*. Cambridge, UK: Cambridge University Press, 443-462.

Gilmore, D. (1995). Interface Design: Have We Got It Wrong? In K. Nordby, D. Gilmore, and S. Arnesen, (Eds.), *INTERACT '95*. London, England: Chapman & Hall.

Green, T. & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing* 7(2), June 1996, 131-174.

Green, T., Petre, M., & Bellamy, R. (1991). Comprehensibility of Visual and Textual Programs: A Test of Superlativism Against the 'Match-Mismatch' Conjecture. In J. Koenemann-Belliveau, T. Moher, S. Robertson (Eds.), *Empirical Studies of Programmers: Fourth Workshop*. Norwood, NJ: Ablex.

Gugerty, L. & Olson, G. (1986). Comprehension Differences in Debugging by Skilled and Novice Programmers. In E. Soloway and S. Iyengar (Eds.), *Empirical Studies of Programmers: First Workshop*. Norwood, NJ: Ablex.

Lawrence, A., Badre, A., & Stasko, J. (1994). Empirically Evaluating the Use of Animations to Teach Algorithms. *1994 IEEE Symp. Visual Languages*, St. Louis, MO, Oct. 4-7, 1994, 48-54.

Martin, G. & Corl, K. (1986). System Response Time Effects on User Productivity. *Behaviour and Information Technology* 5(1), 3-13.

Modugno, F., Corbett, A., & Myers, B. (1996). Evaluating Program Representation in a Demonstrational Visual Shell. In W. Gray and D. Boehm-Davis (Eds.), *Empirical Studies of Programmers: Sixth Workshop*. Norwood, NJ: Ablex.

Moher, T., Mak, D., Blumenthal, B., & Leventhal, L. (1993). Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In C. Cook, J. Scholtz, and J. Spohrer (Eds.), *Empirical Studies of Programmers: Fifth Workshop*. Norwood, NJ: Ablex.

Nardi, B. & Miller, J. (1991). Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development. *International Journal of Man-Machine Studies* 34, 161-194.

O'Hara, K. (1994). Cost of Operations Affects Planfulness of Problem-Solving Behaviour. *CHI '94 Conf. Companion*, Boston, MA, Apr. 24-28, 1994, 105-106.

Oman, P. & Cook, C. (1990). Typographic Style is More Than Cosmetic. *Communications of the ACM* 33, May 1990, 506-520.

Pandey, R. & Burnett, M. (1993). Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study. *1993 IEEE Symp. Visual Languages*, Bergen, Norway, Aug. 24-27, 1993, 344-351.

Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Comm. ACM* 38(6), June 1995, 33-44.

Ramsey, F. & Schafer, D. (1997). *The Statistical Sleuth: A Course in Methods of Data Analysis.* Belmont, CA: Wadsworth Publishing Company.

Shneiderman, B. (1992). *Designing the User Interface: Strategies for Effective Human-Computer*

Interaction.. Reading, MA: Addison-Wesley.

Shneiderman, B. & McKay, D. (1976). Experimental Investigation of Computer Program Debugging and Modifications. *Proceedings 6th International Congress of the International Ergonomics Association*, July 1976.

Svendsen, G. (1991). The Influence of Interface Style on Problem-Solving. *International Journal of Man-Machine Studies* 35, 379-397.

Tanimoto, S. (1990). VIVA: A Visual Language for Image Processing. *Journal of Visual Languages and Computing* 2(2), June 1990, 127-139.

Whitley, K. (1997). Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing* 8(1), February 1997, 109-142.

Wilcox, E., Atwood, J., Burnett, M., Cadiz, J., & Cook, C. (1997). Does Continuous Visual Feedback Aid Debugging in Direct-Manipulation Programming Systems? *CHI'97,* Atlanta, GA, March 22-27, 1997, 258-265.