Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems

Andrew Jensen Ko
Human-Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
ajko@cmu.edu

Bob Uttl
Department of Psychology
College of Liberal Arts
Oregon State University
Corvallis, OR 97331 USA
bob.uttl@alfalab.com

Abstract

This study examines the effect of individual differences on the program comprehension strategies of users working with an unfamiliar programming system. Participants of varying programming expertise were given a battery of psychological tests, a brief introduction to a statistical programming environment, and a 20-minute debugging task. Our data show that while there were distinct comprehension strategies that were related to programming experience, no strategy had an advantage in debugging performance. Rather, individuals with stronger domain knowledge for specific bugs tended to succeed.

1. Introduction

To make successful modifications to a program, not only must programmers have an adequate understanding of a program, but they also must adequately understand the programming system (comprised of an environment and language). However, programmers frequently must gain understanding of a program and programming system at the same time. For example, students in introductory programming courses are often given modification tasks and must learn a new language and environment while understanding an existing system. Software engineers often face the same challenge, except with more complex tools and larger programs. This is a difficult process, in which programmers rely heavily on the support of more experienced individuals [1] as they acquire new skills and domain-specific knowledge.

While there are many program comprehension theories and findings that can predict comprehension strategies programmers may use in unfamiliar programming systems, the majority of this work only addresses unfamiliarity with a *program*, and not unfamiliarity with a *programming system*. How does the effect of individual differences on comprehension strategies and debugging change in this situation? In particular:

- What comprehension strategies do programmers use in unfamiliar programming system?
- What individual differences predict these comprehension strategies?
- How do these individual differences and comprehension strategies affect debugging?

The participants in this study, which had widely varying programming expertise, used three distinct strategies, some of which map to strategies in previous studies. While the comprehension strategies were related to programming experience, no particular strategy resulted in the best debugging performance. Rather, individuals with stronger domain knowledge for specific bugs and positive attitudes towards computers tended to succeed.

2. Related Work

A number of theoretical analyses have suggested models of program comprehension, and these models maker certain predictions about the strategies programmers using an unfamiliar programming system might use. For example, von Mayrhauser and Vans' model [11] suggests that programmers build a mental representation of a system's control and data flow from the bottom up as code, documentation, and domain-knowledge are acquired. In this model, unfamiliarity with an environment and language may make a programmer use documentation and domain-knowledge more than normal, since programmers may not be able to view and understanding code in familiar ways.

Gilmore's model of debugging [6] depicts a flexible, reconstructive process in which programmers compare perceived design decisions of a system with the actual performance of the system. In this model, the comprehension process draws from domain knowledge and expertise until mental models of the systems' goals and the program code are reconciled. Under this model, it would seem that unfamiliarity with a programming system could impact a programmer's ability to explore and comprehend code, placing a performance bottleneck on comprehension, but not changing their strategy.

Many empirical studies relate comprehension strategies to individual differences, suggesting ways in which unfamiliarity with a programming system could influence comprehension strategies. For example, Davies [3] argues that working memory and domain knowledge interact to determine comprehension strategies, suggesting that unfamiliar environments may make programmer's working memory capacity and domain-knowledge more influential than in a familiar environment. Also, in Koenemann and Robertson's [10] study, participants' choice of an as-needed comprehension strategy seemed influenced by their frustration with the unfamiliar code viewer in the experiment. This suggests that unfamiliarity with a programming system may make programmers resort to strategies they would not otherwise use.

Many findings on how expert programmers differ from novices may not apply to programmers using unfamiliar environments. For example, Nanja and Cook [12] and Jeffries [8] observed that experts, but not novices, used a strategy of reading a program in the order in which it would be executed. If experts are unfamiliar with a programming system, they may revert to the novice strategy or another strategy altogether in order to read a program. Also, Teasley [14] found that variable naming style can affect the functional program comprehension of novices, but not of experts; this effect may not appear if experts are unfamiliar with the language. Wiedenbeck and colleagues [16] suggest that experts' memory for program representations is better because they recognize the procedural nature of a program and can more efficiently utilize their working memory. Given Davie's finding [4] that experts rely greatly on external displays for comprehension, this advantage may not appear if experts are inexperienced with an unfamiliar programming system's external representation of code.

Clearly, individual differences such as expertise and domain-knowledge can affect program comprehension strategies and performance. The question investigated in this study is how these findings change when programmers are unfamiliar with a programming system.

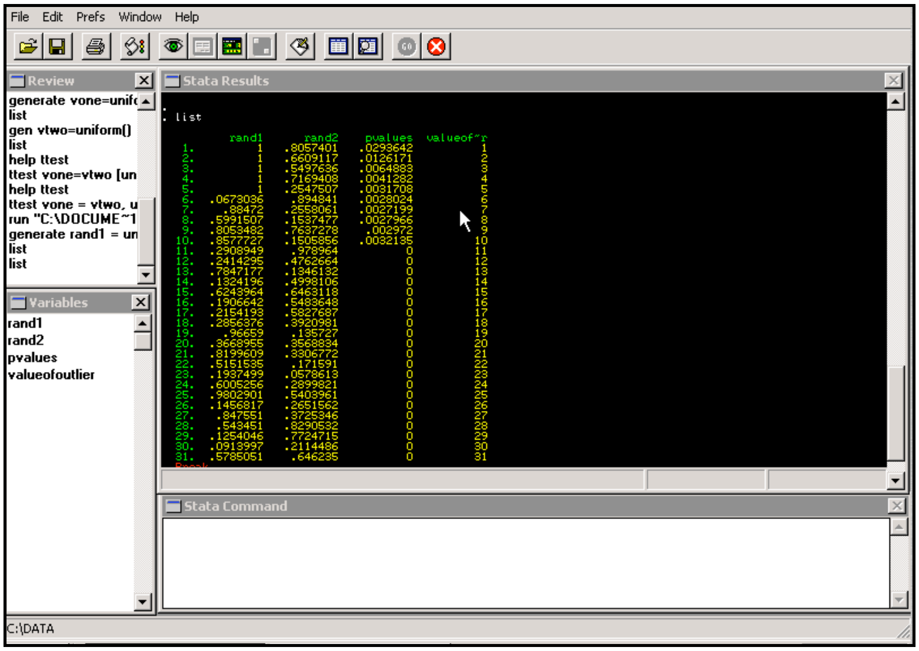
3. Method

An exploratory experiment was designed in which participants comprehended and debugged a simple program in an unfamiliar programming system. Participants were exposed to the same experimental condition, so the only variation was in participants' backgrounds, experience, and comprehension strategies.

3.1 The Unfamiliar Environment

The unfamiliar programming system participants used was Intercooled Stata 7.0, a commercially available programmable statistical package (www.stata.com). A typical view of Stata can be seen in Figure 1. Stata provides a single window interface which contains a *variables* window, which shows the variables in the current data set being used, a *review* window that shows a history of user-generated and computer-generated commands, a *results* window, that shows a history of textual output generated by Stata, and a *command* window in which users enter textual commands. When graphs are generated, Stata displays the graph in a resizable, non-interactive window. Stata also provides a spreadsheet-like data editor with limited functionality.

Creating a data set in Stata consists of setting a system variable that specifies the number of objects in a data set to the desired size (for example, "`set obs 1000`" makes a data set of 1000 objects). To create variables within this data set, the "`generate`" command is used; for example, the command "`generate height = 5`"



```
generate vone=unifc
list
gen vtwo=uniform()
list
help ttest
ttest vone=vtwo [un
help ttest
ttest vone = vtwo, u
run "C:\DOCUME~1
generate rand1 = un
list
list
```

```
rand1
rand2
pvalues
valueofoutlier
```

```
.
. list

            rand1       rand2     pvalues   valueof~r
  1.            1    .8057401    .0293642           1
  2.            1    .6609117    .0126171           2
  3.            1    .5497636    .0064883           3
  4.            1    .7169408    .0041282           4
  5.            1    .2547507    .0031708           5
  6.     .0673036     .894841    .0028024           6
  7.       .88472    .2558061    .0027199           7
  8.     .5991507    .1537477    .0027966           8
  9.     .8053482    .7637278     .002972           9
 10.     .8577727    .1505856    .0032135          10
 11.     .2908949     .978964           0          11
 12.     .2414295    .4762664           0          12
 13.     .7847177    .1346132           0          13
 14.     .1324196    .4998106           0          14
 15.     .6243964    .6463118           0          15
 16.     .1906642    .5483648           0          16
 17.     .2154193    .5827687           0          17
 18.     .2856376    .3920981           0          18
 19.       .96659     .135727           0          19
 20.     .3668955    .3568834           0          20
 21.     .8199609    .3306772           0          21
 22.     .5151535     .171591           0          22
 23.     .1937499    .0578613           0          23
 24.     .6005256    .2899821           0          24
 25.     .9802901    .5403961           0          25
 26.     .1456817    .2651562           0          26
 27.      .847551    .3725346           0          27
 28.      .543451    .8290532           0          28
 29.     .1254046    .7724715           0          29
 30.     .0913997    .2114486           0          30
 31.     .5785051     .646235           0          31
Break
```

Figure 1. A typical view of Stata. Commands are entered in the window at the bottom, results are displayed in the large window with the black background, and a history of commands is displayed in the window on the upper-left labeled "Review." Figure printed with permission from Stata Corporation, College Station, TX.

would create a variable named *height*, which has the value 5 for all 1000 objects. Stata provides many functions that can be used in conjunction with the generate command, such as in "`generate height = uniform()`" which creates a variable named height in which each observation is a random number between 0 and 1.

Stata provides an online help system with descriptions and examples of command use. The online help can be reached through the commands "`help`" or "`search`", in which case help is displayed in the *results* window. Thhe same help is available through the help menu in the main Stata window, but the help text is instead displayed in a graphical, resizable window.

3.2 Testing Instruments

A number of tests were used to assess participants' individual differences.

The *Vocab27* test measured verbal intelligence, a correlate of general IQ, and consisted of 27 multiple-choice questions from Ekstrom [5]. For each item, participants saw a target word and chose the word most closely related in meaning out of the four alternatives. No time limit was placed on the test. Cronbach's alpha, a widely used measure of a psychological test's reliability, was .75. In general, any reliability below .5 in this paper was considered unreliable, and thus a poor measure.

The *problem-solving* test obtained a measure of general problem solving ability. Fifty items were selected from various intelligence tests. In the first section, the participant was asked to determine the next number in a series of numbers. For example, the sequence "1, 2, 4, 8, __" would be provided and the participant would fill in the next number in the sequence. In the second section, participants were given a word pair with an unidentified relationship and were asked to choose two words from a list that had the same relationship. The other three sections tested synonyms, antonyms, and visual sequences in similar ways. Participants were given 20 minutes to complete the test. Reliability was .78.

The *statistics* test assessed domain knowledge (since Stata is a statistical programming environment, domain knowledge was statistics). The test consisted of 10 multiple-choice questions (varying from three to five choices) and tested material from introductory statistics and hypothesis testing courses. Participants were given 10 minutes to complete the test. As participants scored near the floor, no reliability measure is reported.

A background questionnaire was administered to gather basic information about participants' age, major, gender, native language, as well as self-reported experience with statistics software, mathematics, computers, and programming. Attitudes towards these areas and the experiment were also measured. Attitudes and prior experience were measured using a seven-point Likert scale, ranging from Strongly Disagree (1) to Neutral (4) to Strongly Agree (7). Experience and attitude reliabilities were all above .60.

3.3 Participants

To allow for the possibility that expert programmers might use the same strategies as less experienced programmers, sampling was blind to programming expertise. Participants were recruited from undergraduate computer science, psychology, and statistics courses. All participants had at least one introductory statistics course and experience with hypothesis testing. Participants were offered extra credit in their class and entry into a raffle for a $100 prize in exchange for their participation. The original sample consisted of 86 participants, but those who did not understand the experimental materials because of poor English skills were removed from the sample, leaving 75 participants. Scores and responses on the tests of individual differences are listed in Table 1 by academic major and gender.

Two-sample independent t-tests were performed by gender, major, and native language to test for differences in self-reported ability and experience, and performance on the *Vocab27, problem solving*, and *statistics* tests. Participants whose native language was not English performed significantly lower than native English speakers on *Vocab27* (p < .001); females had significantly less self-reported experience and ability with mathematics (p < .05), programming (p < .001), and software (p < .05), but these results are likely confounded by the low proportion of female computer science majors. Computer science majors reported greater ability and experience with mathematics (p < .01) and programming (p < .01).

Measure	**Major**								
	Psychology			*Computer Science*			*Other*		
Gender	*M*	*F*	*All*	*M*	*F*	*All*	*M*	*F*	*All*
Count	8	12	20	21	3	24	12	19	31
Age	20.6	22.4	21.7	25.4	26.7	25.6	24.3	26.1	25.4
	(1.3)	(3.6)	(3.0)	(8.2)	(7.4)	(8.0)	(3.9)	(6.7)	(5.7)
Vocab27	.59	.57	.58	.62	.68	.63	.60	.58	.59
	(.09)	(.10)	(.10)	(.12)	(.21)	(.13)	(.15)	(.11)	(.12)
Problem	.57	.51	.53	.54	.68	.55	.48	.52	.50
Solving Test	(.09)	(.10)	(.10)	(.15)	(.20)	(.16)	(.16)	(.18)	(.17)
Statistics	.25	.33	.30	.36	.27	.35	.34	.34	.34
Test	(.15)	(.21)	(.18)	(.16)	(.15)	(.16)	(.11)	(.17)	(.15)
Mathematics	3.91	2.96	3.34	5.55	5.25	5.51	4.96	4.71	4.81
Experience	(1.52)	(1.43)	(1.51)	(.99)	(.66)	(.95)	(2.28)	(1.39)	(1.76)
Stats Soft	1.25	1.93	1.66	1.08	2.25	1.23	2.15	1.26	1.60
Experience	(1.34)	(1.08)	(1.21)	(.99)	(.75)	(1.02)	(.98)	(1.33)	(1.27)
Programming	.58	.31	.42	4.24	3.39	4.13	2.19	.96	1.44
Experience	(.85)	(.91)	(.88)	(1.16)	(.92)	(1.15)	(1.89)	(1.09)	(1.55)
Computer	5.44	5.50	5.48	6.12	5.83	6.08	6.04	5.66	5.80
Experience	(1.45)	(1.11)	(1.09)	(.71)	(1.26)	(.76)	(.72)	(1.21)	(1.05)
Experiment	4.72	4.52	4.60	5.36	4.14	5.24	4.88	4.83	4.85
Attitudes	(.92)	(.95)	(.92)	(.97)	(.63)	(.97)	(.66)	(1.12)	(.96)
Computer	4.28	3.88	4.04	6.45	6.17	6.42	5.48	4.89	5.11
Attitudes	(1.45)	(1.19)	(1.28)	(.46)	(.63)	(.48)	(.96)	(1.07)	(1.05)
Statistics	2.81	2.77	2.78	3.43	3.83	3.48	3.98	3.32	3.57
Attitudes	(1.18)	(1.58)	(1.40)	(.93)	(.88)	(.91)	(.88)	(1.31)	(1.19)
Mathematics	3.63	3.67	3.65	5.36	5.25	5.34	5.10	4.43	4.69
Attitudes	(1.74)	(1.84)	(1.75)	(1.21)	(1.15)	(1.18)	(1.63)	(1.51)	(1.56)

Table 1. Means (and standard deviations) of individual differences by major and gender.

3.4 Two Tasks

Participants were given two problems in the experiment. The first asked participants to create a data set of 1000 objects, create two variables with random numbers between zero and one, perform a t-test to compare the means of the two lists of data, and then report the p-value returned by the t-test to the experimenter. The task was intended to give participants practice with the environment and language. Participants were given 30 minutes for the problem. If they did not complete the problem or completed it incorrectly, they were given a standard solution so that all participants would have the same understanding for the next problem.

The second problem was intended to elicit participants' program comprehension strategies, and is the focus of this paper. The problem consisted of a short sequence of Stata commands in a text-file (see Figure 2) and a problem description, which served as the program specification. The purpose of the program was to create a graph that visualized the influence of increasingly large outliers on the p-value from a t-test. The program created a data set of 1000 objects with two variables with random values between zero and one, and then looped ten times, each time changing five of the values in the first data set to increasingly large outliers from values 1 to 10. As these outliers increased from 1 to 10, the means of the two lists of data became increasingly different, and thus the p-value from the t-test between these two lists became smaller. The graph listed the outlier values from 1 to 10 on the x-axis, and the p-value generated by the t-test for each value of the outliers on the y-axis.

The resulting graph was supposed to be a smooth curve with ten data points, as seen on the left in Figure 3. However, four bugs were inserted into program that the participants received (highlighted in Figure 2), which changed the graph significantly, as seen on the right in Figure 3. Each of the bugs had a specific rationale:

- The bug starting on the line "`for num 10/1: replace rand…`" (hereon referred to as the *for loop range* bug) changed the order of execution to count values from 10 to 1 instead of 1 to 10. Since this did not affect the data or the graph, this was a test of participants' syntax understanding.

```
* This script will generate two sets of 1000 observations of uniformly ;
* distributed numbers, and place an outlier in the first five observations ;
* of the first set. The goal is to graph the p-values of unpaired ;
* two-sample t-tests as this outlier changes from 1 to 10. Here is ;
* the basic order of execution: ;
*  - clear all variables from the system ;
*  - set the number of observations to 1000 ;
*  - generate two variables with random numbers from 0 to 1 ;
*  - generate a variable to store all of the p-values ;
*  - loop from 1 to 10 and ;
*     - change the outliers in the first set to the current value ;
*     - execute the t-test between the two sets ;
*     - put the resulting p-value into the p-value variable in the ;
*       current row ;
*  - generate a variable that contains the values from 1 to 10 ;
*  - graph the p-values against the values 1 to 10 ;

* Tell the system that every command ends with a semi-colon ;
#delimit ;

* Clear the system before we start! ;
clear;

* First we set the number of observations ;
set obs 1000;

* Generate two variables for which all observations are a random ;
* number from 0 to 1 ;
generate rand1 = uniform();
generate rand2 = uniform();

* Next, we need to generate a variable to store the p-values in. ;
* We put 0's in every observation for now. ;
generate pvalues = 0 in 1/100;

* Then we need to loop through outlier values from 1 to 10,;
* at intervals of 1. First, we update the first five observations ;
* of variable rand1. Then, we run an unpaired t-test between the ;
* two variables. Finally, we place the p-value in the pvalues ;
* variable, in the current row. ;
for num 10/1: replace rand1 = X in 1/5 \ ttest rand1 = rand2, unpaired \
replace pvalues = r(p) in X;

* Generate a variable that contains 1 through 10 ;
generate valueofoutlier = _n in 1/100;

* Graph the results, pvalues versus valueofoutlier ;
graph valueofoutlier pvalues, title(p-values of a t-test as the magnitude of
outliers increase) connect(l);
```

Figure 2. The source code provided to participants for problem 2. The four bugs are highlighted in grey.

- The bugs in the line "`generate pvalues = 0 in 1/100;`" and "`generate valueofoutlier = _n in 1/100`" (hereon refered to as the *pvalues range* and *valueofoutlier range* bugs) were supposed to be "1/10" instead of "1/100" because only ten p-values were being calculated. This bug required an

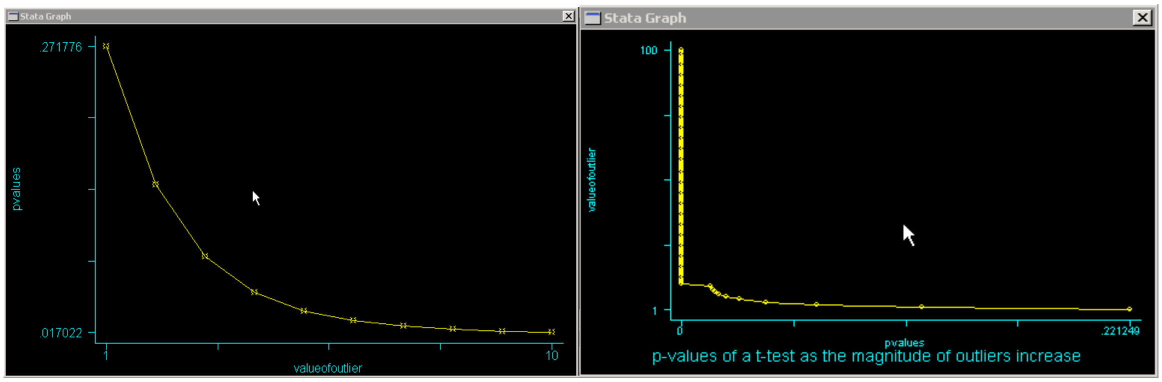


Figure 3. The intended output of the Stata program (left) and the actual output with bugs (right).

understanding of the syntax in “1/100” which refers to the integers 1 through 100. The two commands were different in that the *valueofoutlier* bug contained a comment that explicitly conflicted with the code, whereas the *pvalue* bug did not. The *valueofoutlier* command also used a system variable that referred to current row being altered.

- The last bug was in the last line of the file, which graphed the p-values against the numbers 1 to 10. The buggy command inverted the axes (hereon referred to as the *inverted axes* bug). To detect this bug, participants had to inspect the graph output and understand the semantics of the command.

Participants were given 20 minutes to change the program to produce the correct graph, which was shown in their problem description.

3.5 Experiment Procedure

Participants worked individually in 2-hour sessions. Following any questions the participant had about the experiment, the experimenter administered the *Vocab27*, *problem solving*, and *statistics* tests. After an optional break, participants were given the background questionnaire. The experimenter then began a 10-minute tutorial on how to write Stata commands, create a data set, create variables, list data in a data set, and search for help in Stata. The intention of the tutorial was to allow participants to get accustomed to the basic features the environment and language provided, while avoiding suggesting strategies for learning the environment and language. Participants were allowed to ask questions during the tutorial only about topics in the tutorial.

Next, the experimenter told the participant that they would be working on two problems within Stata and that there would be three rules regarding the problem solving sessions: (1) they were not allowed to ask the experimenter questions (except about the problem description), (2) they were not allowed to use the Internet to solve their problems, and (3) they were to work until they solved the problem or time expired. Participants were also asked to think aloud while working, but were not prompted. During the problem solving sessions, the only window visible to participants was the main Stata window. Participants’ interactions were recorded with screen capturing software and videotape over the shoulder.

4. Results

First we consider variations in participants’ debugging performance, shown in Table 2. As seen in the top of Table 2, the most frequently found bug was the *valueofoutlier range* bug and the least was the *inverted axes* bug. As a group, participants fixed almost every combination four bugs. Most fixed no bugs or fixed some combination of the *valueofoutlier range, pvalues range*. Few participants fixed the *inverted axes* bug, but those who did tended to fix the other bugs.

***valueofoutlier* Range**	***pvalues* Range**	**For Loop Range**	**Inverted Axes**	**Frequency**
Overall Success on Each Bug				
47 (62.7%)	28 (37.3%)	25 (33.3%)	16 (21.3%)	
Failure	Failure	Failure	Failure	14 (18.7%)
Failure	Failure	Success	Failure	7 (9.3%)
Failure	Failure	Failure	Success	1 (1.3%)
Failure	Failure	Success	Success	0 (0.0%)
Failure	Success	Failure	Failure	4 (5.3%)
Failure	Success	Success	Failure	1 (1.3%)
Failure	Success	Failure	Success	1 (1.3%)
Failure	Success	Success	Success	0 (0.0%)
Success	Failure	Failure	Failure	10 (13.3%)
Success	Failure	Success	Failure	7 (9.3%)
Success	Failure	Failure	Success	5 (6.7%)
Success	Failure	Success	Success	3 (4.0%)
Success	Success	Failure	Failure	10 (13.3%)
Success	Success	Success	Failure	6 (8.0%)
Success	Success	Failure	Success	5 (6.7%)
Success	Success	Success	Success	1 (1.3%)

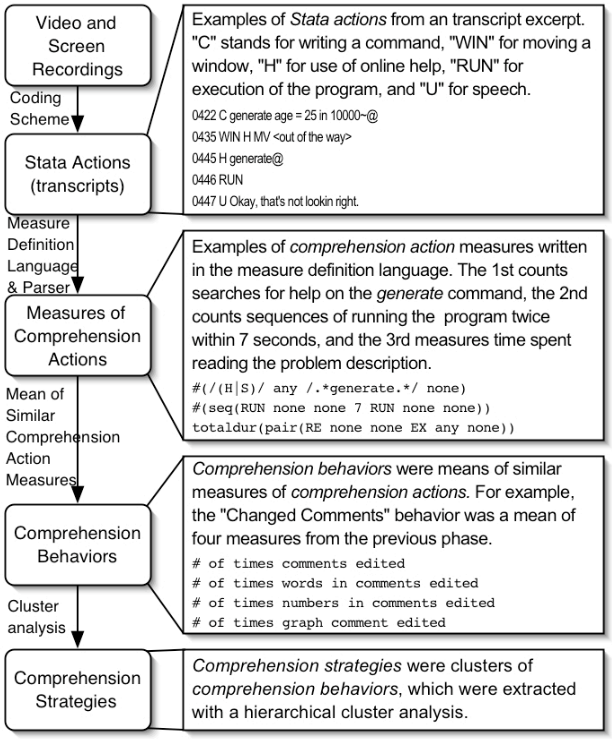
Table 2. Total success on each bug and the number of participants fixing each combination of bugs.

4.1 Individual Differences and Debugging

Table 3 shows individual differences by success at each bug. Positive attitudes towards computers and mathematics, as well as self-reported computer experience were related to success on the *valueofoutlier range* bug. Self-reported math abilities and statistics software and computer experience were related to success on the *inverted axes* bug. MANOVAs were performed on individual differences by gender, major, and the bugs, revealing some interaction effects. However, because of the proportion of female computer science students, these interactions effects were likely confounded by gender.

Measure	**Fixed?**	***Valueof-* Range**	***pvalues* Range**	**For Loop Range**	**Inverted Axes**
Vocab Test	No	.54 (0.15)	.53 (0.15)	.51 (0.13)	.53 (0.16)
	Yes	.52 (0.15)	.53 (0.16)	.56 (0.18)	.53 (0.14)
Problem-Solving Test	No	.59 (0.12)	.59 (0.11)	.59 (0.12)	.59 (0.13)
	Yes	.60 (0.12)	.61 (0.13)	.62 (0.13)	.62 (0.10)
Statistics Test	No	.34 (0.15)	.34 (0.16)	.33 (0.16)	.31 (0.15)
	Yes	.32 (0.16)	.31 (0.16)	.32 (0.16)	.40 (0.18)
Self-reported Math Abilities	No	3.91 (1.68)	4.63 (1.62)	4.42 (1.77)	4.41 (1.72)
	Yes	5.07 (1.54)	4.66 (1.81)	5.09 (1.42)	5.48 (1.23)
Self-reported Stats. Soft. Exp.	No	1.53 (1.10)	1.45 (1.19)	1.66 (1.20)	1.41 (1.22)
	Yes	1.48 (1.23)	1.58 (1.17)	1.19 (1.10)	1.83 (0.96)
Self-reported Programming Exp.	No	1.76 (1.96)	2.12 (2.04)	1.69 (1.95)	1.85 (1.90)
	Yes	2.19 (1.98)	1.87 (1.86)	2.71 (1.86)	2.69 (2.14)
Self-reported Computer Exp.	No	5.64 (1.24)	5.83 (0.97)	5.79 (1.06)	5.73 (1.07)
	Yes	5.90 (0.82)	5.77 (1.06)	5.84 (0.87)	6.09 (0.58)
Attitudes towards the experiment	No	4.75 (0.98)	4.95 (1.01)	4.86 (0.93)	4.83 (0.99)
	Yes	5.00 (0.97)	4.83 (0.92)	5.01 (1.06)	5.19 (0.87)
Attitudes towards computers	No	4.82 (1.54)	5.30 (1.21)	5.06 (1.39)	5.08 (1.37)
	Yes	5.49 (1.14)	5.15 (1.55)	5.62 (1.15)	5.83 (1.02)
Attitudes towards statistics	No	2.95 (1.25)	3.22 (1.22)	3.43 (1.25)	3.23 (1.19)
	Yes	3.56 (1.13)	3.53 (1.17)	3.15 (1.12)	3.79 (1.23)
Attitudes towards mathematics	No	4.02 (1.85)	4.57 (1.59)	4.55 (1.54)	4.43 (1.67)
	Yes	4.98 (1.37)	4.71 (1.71)	4.77 (1.81)	5.33 (1.28)

Table 3. Mean (and standard deviations) of individual differences by success on each bug. Highlighted rows are significant at □=.05.


Video and Screen Recordings
Coding Scheme
Stata Actions (transcripts)
Measure Definition Language & Parser
Measures of Comprehension Actions
Mean of Similar Comprehension Action Measures
Comprehension Behaviors
Cluster analysis
Comprehension Strategies

Examples of *Stata actions* from an transcript excerpt. "C" stands for writing a command, "WIN" for moving a window, "H" for use of online help, "RUN" for execution of the program, and "U" for speech.

```
0422 C generate age = 25 in 10000~@
0435 WIN H MV <out of the way>
0445 H generate@
0446 RUN
0447 U Okay, that's not lookin right.
```

Examples of *comprehension action* measures written in the measure definition language. The 1st counts searches for help on the *generate* command, the 2nd counts sequences of running the program twice within 7 seconds, and the 3rd measures time spent reading the problem description.

```
#(/(H|S)/ any /.*generate.*/ none)
#(seq(RUN none none 7 RUN none none))
totaldur(pair(RE none none EX any none))
```

Comprehension behaviors were means of similar measures of *comprehension actions.* For example, the "Changed Comments" behavior was a mean of four measures from the previous phase.

```
# of times comments edited
# of times words in comments edited
# of times numbers in comments edited
# of times graph comment edited
```

Comprehension strategies were clusters of *comprehension behaviors*, which were extracted with a hierarchical cluster analysis.

Figure 4. The process converting participants' actions in Stata to distinct comprehension strategies.

4.2 Assessing Comprehension Behaviors

To assess participants' comprehension strategies, we followed the process in Figure 4 to convert participants' actions in Stata to measures of comprehension behaviors (for full details, see [9]). Video and screen recordings were coded into transcripts of *Stata actions*, using a coding scheme developed prior to the study. These included actions such as typing commands, inserting and removing code, and consulting online help. The two coders had high levels of agreement on five test transcripts. The final transcripts, including over 33,000 actions, were checked for syntax and semantic errors.

A measure definition language and parser were created so that measures of *comprehension actions* could be automatically extracted from the transcripts. For example, one measure definition shown in Figure 4, `#(H|S)/ any /.*generate.*/ none`, counted the number of searches for help on the `generate` command. The parser counted the number of `H` and `S` entries in transcripts (which stood for `help` and `search` Stata commands) that had the word "generate" in them. Almost 200 measures of *comprehension actions* were defined and extracted from transcripts in similar ways. Since measures were on different scales (frequencies and durations), the measures were standardized to a normal distribution. Measures that had low reliability or highly skewed distributions were excluded, for two reasons. First, including them would have added noise to the aggregate measures later in the process; and second, many of the 200 measures measured similar actions, and so throwing poor measures out would not artificially limit the scope of the data.

The remaining 132 measures of *comprehension actions* were then grouped according to similarity of what they measured. For example, five of the 132 measures of *comprehension actions* measured ways in which participants read or modified comments. The means of the 22 groups of similar measures were calculated, representing final measure of *comprehension behaviors*. The resulting *comprehension behavior* measures are shown in Table 4 with descriptions and reliabilities.

	Comprehension Behavior	*valueofoutlier* Range	*pvalue* Range	For Loop Range	Inverted Axes	Measured Degree to which...	□
1	For Loop			>		*for* command was inspected or modified	.63
2	*pvalues* Command	>	>>			*pvalues* command was inspected or modified	.46
3	Graph Command			<	>	*graph* command was inspected or modified	.31
4	*valueofoutlier* Command	>			>	*valueofoutlier* command was inspected or modified	.59
5	Familiar Commands	<<	>>	<<	<<	familiar commands were inspected or changed	.14
6	Changed Comments			<	>>	command comments were changed	.43
7	Changed Words			>		non-numerical text was changed	.60
8	Change Range			>		ranges (such as 1/10) were changed in commands	.75
9	Guess and Check	>				solutions were derived iteratively	.67
10	Desire to Seek Info			>		online-help was consulted	.71
11	Cyclic Behavior	<		>		actions were repeated unnecessarily	.43
12	State Comprehension	>			>	state of variables, code and data was inspected	.97
13	Graph	>>	>	<	>>	graph was inspected or attended to	.76
14	Attention to Feedback		<<	<		feedback from Stata was attended to or used	.42
15	Use of Examples	<		>>	<	examples from tutorial or help were referenced or used	.23
16	Use of Specification			<		specifications were referenced or used	.55
17	Use of GUIs		<	>	<	GUIs were used to comprehend program	.44
18	Use of Commands	>				commands were used in testing and debugging the do-file	.90
19	Window Management					information from multiple windows was compared	.96
20	Frustration		<			confusion was expressed through words or behavior	.30
21	Syntax Confusion	>	>		>	commands were improperly constructed	.63
22	Verbalization			>		thoughts about problem and solutions were verbalized	.55

Table 4. Names, descriptions, and reliabilities of comprehension behavior group measures. The four bug columns relate participants successful on the bug to participants unsuccessful on the bug: << or >> means an absolute standardized score of .40 or more, < or > means .20 or more, and no symbol means less than .20.

4.3 Comprehension Behaviors and Debugging Performance

Participants' comprehension behavior measures were split by success at fixing each of the four bugs and compared relative to each other. These comparisons are shown in Table 4. The table is read as in this example: participants successful at fixing the *for loop range* bug tended to inspect or modify the *for* loop command (1st row) more than participants who did not fix the bug. Or, participants who were successful at fixing the *valueofoutlier range* bug tended to look at the graph output (13th row) more than participants who were not.

4.4 Distinct Comprehension Strategies

The assumption was that comprehension strategies are distinct patterns of *comprehension behaviors*. Thus, a hierarchical cluster analysis was performed on the comprehension behavior measures shown in Table 4 in order to derive distinct *comprehension strategies* (this is the last process in Figure 4). Ward's method [15] was used, which minimizes the sum of squares between clusters. Solution sizes of 2 to 7 clusters were generated. Since further subdivision of the 3-cluster solution only split the smallest cluster, the three-cluster solution was chosen. The three clusters are portrayed in Figure 5 with respect to the comprehension behaviors from Table 4 that define them. Measures that *do not* have "(ns)" next to their name were significantly different between clusters using ANOVA with □=.01. As shown in the Figure 5, there were many significant differences. For example, participants in cluster 1 (in white) inspected the *pvalues* and *valueofoutlier* commands less, exhibited less guess and check behavior, and sought more information. Other important differences will be discussed in the discussion.

4.5 Predictive Power of Strategies

Categorical characteristics of the participants in each cluster are shown in Table 5, in terms of cluster size and proportions of major, gender, and bug success. As seen in Table 5, Chi-squared tests revealed that clusters were related to major (p < .01) and success at the *valueofoutlier range* bug (p < .01). Post-hoc analyses revealed that cluster 1 was more successful at solving the *valueofoutlier range* bug than cluster 2 (p < .01).

Clusters were also related to continuous measures of individual difference with ANOVAs (see Table 6). There was a main affect for programming experience (p < .01); post-hoc analyses revealed cluster 3 had more programming experience than clusters 1 and 2, cluster 1 had more experience than cluster 2. There was also a main effect for attitudes towards the experiment (p < .01); post-hoc analyses showed cluster 3 had more positive attitudes towards the experiment than the other clusters. Finally, there was a main effect for the average number of bugs solved per participant, with post-hoc analyses revealing participants in cluster 2 and cluster 3 were significantly more successful than participants in cluster 1 (p < .05).

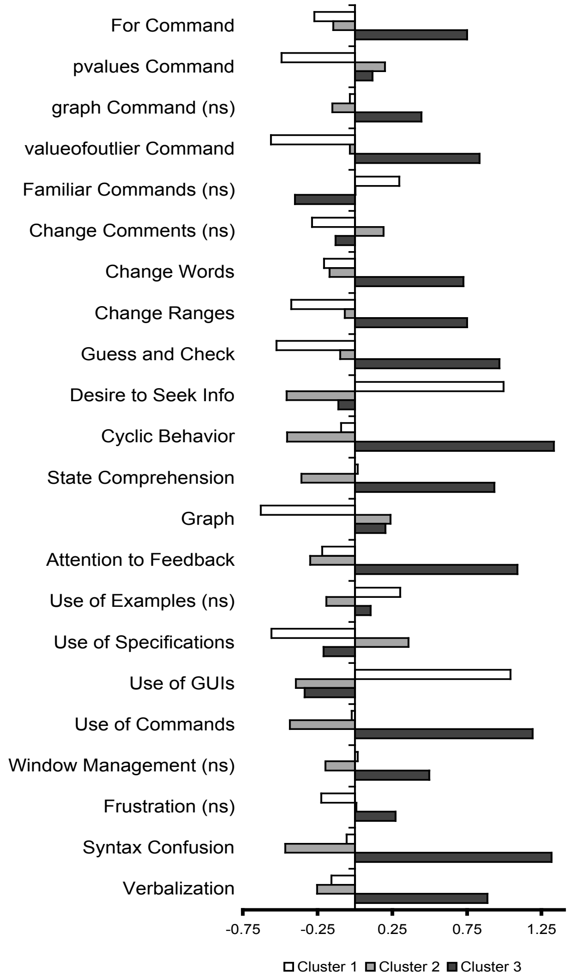


Figure 5. Measures of comprehension behavior by cluster. The figure should be read, "Cluster 3 inspected or modified the *for* command ~.75 standard deviations above average." Measures with "(ns)" next to their name were *not significant* between clusters.

Cluster	Size	Major			Gender		*Valueofoutlier* Range	*pvalue* Range	For Loop Range	Inverted Axes
		Psych	*CS*	*Other*	*Female*	*Male*	*Success*	*Success*	*Success*	*Success*
1	20	3 (15.0%)	8 (40.0%)	9 (45.0%)	10 (50.0%)	10 (50.0%)	13 (65.0%)	5 (20.0%)	7 (35.0%)	2 (10.0%)
2	40	16 (40.0%)	8 (20.0%)	16 (40.0%)	19 (47.5%)	21 (52.5%)	12 (30.0%)	18 (45.0%)	11 (27.5%)	9 (22.5%)
3	15	1 (6.7%)	8 (53.3%)	6 (40.0%)	5 (33.3%)	10 (66.6%)	3 (20.0%)	6 (40.0%)	7 (46.7%)	5 (33.3%)
Total	75	20 (26.7%)	24 (32.0%)	31 (41.3%)	34 (45.3%)	41 (54.7%)	28 (37.3%)	29 (38.7%)	25 (33.3%)	16 (21.3%)

Table 5. Composition (size and percent) of each cluster by major, gender, and success at each bug. Highlighted columns are significant at □=.01 using a Chi-Squared test.

Measure	Cluster		
	1	2	3
	Mean (Standard Deviation)		
Percent Correct			
Vocab27	.58 (0.14)	.50 (0.14)	.53 (0.18)
Problem Solving Test	.63 (0.12)	.58 (0.10)	.61 (0.15)
Statistics Test	.36 (0.14)	.31 (0.17)	.34 (0.15)
Self-reported experience (0=none, 1=least, 7=most)			
Mathematics Experience	4.93 (1.19)	4.26 (1.83)	5.27 (1.64)
Statistics Software Experience	1.35 (1.05)	1.50 (1.29)	1.70 (1.07)
Programming Experience	2.52 (1.76)	1.20 (1.62)	3.59 (2.00)
Computer Experience	6.13 (0.90)	5.65 (1.10)	5.80 (0.75)
Attitudes Towards... (1=negative, 7 = positive)			
Experiment	4.89 (0.84)	4.68 (0.98)	5.53 (0.90)
Computers	5.65 (1.05)	4.76 (1.39)	5.98 (1.05)
Statistics	3.34 (1.07)	3.39 (1.33)	3.18 (1.09)
Mathematics	4.83 (1.27)	4.40 (1.68)	4.95 (1.91)
Debugging Performance (0 = none, 4 = all)			
Average # Bugs Fixed	1.00 (0.97)	1.65 (.98)	2.00 (1.07)

Table 6. Means (and standard deviations) for continuous measures of individual differences by cluster. Highlighted rows are significant at □=.05.

5. Discussion

5.1 Comprehension Strategies

In this section, we address the first two questions posed in the introduction:

- What comprehension strategies do programmers use in an unfamiliar programming system?
- What individual differences predict these comprehension strategies?

Among all 75 participants, there seemed to be three distinct comprehension strategies. Individuals in cluster 1 tended to seek more of information and use the GUI more, but spent less time attending to the problem specification, the graph output, and each of the problematic commands in the program. Cluster 1 tended to be non-psychology majors with slightly below average programming experience. Given the group's high rate of fixing the *valueofoutlier range* bug, these individuals may have been more absorbed in understanding and fixing the *valueofoutlier* command by learning about the language and environment, rather than focusing on the whole program. Thus, this cluster seemed to be using a depth-first comprehension strategy.

Cluster 2 participants seemed to be very inactive: they sought less information, exhibited less cyclic behavior and comprehension of the environments state, typed fewer commands and used the GUI less. However, cluster 2 did inspect the problem specification graph output more than the other groups. The group tended to be psychology and other majors (but not computer scientists) and had the least programming experience. In general, it seemed cluster 2 seemed hesitant to learn and use the environment and language and instead focused on the problem description and graph output. Since the group had little programming experience, this may have been due to anxiety towards the task's programming aspect.

Cluster 3 was largely different from clusters 1 and 2 in that participants tended to inspect every command, attend more often to errors from Stata, and inspect the graph output more. Cluster 3 also seemed make many changes, use more commands, and exhibited more cyclic and guess and check behavior, while seeking an average amount of information from the online help. Participants in cluster 3 tended to be mostly computer science and other majors (not psychology majors), had much higher mathematics and programming experience, and had significantly more positive attitudes towards the experiment (and thus possibly more motivation). Since participants in cluster 3 inspected every aspect of the program, it would seem they used a more breadth-first comprehension strategy.

In general, the only individual differences that predicted comprehension strategy were major and programming experience. Thus, it seems reasonable that comprehension strategy was largely a factor of past experiences with programming and problem solving, as reflected by participants' major.

5.2 Factors Influencing Debugging Success

In this section, we address the last question posed in the introduction:

- How do these individual differences and comprehension strategies affect debugging?

Cluster 2 and 3 participants tended to have the best overall debugging performance. Thus, for this task, a breadth-first strategy *or* a focus on the problem specification seemed to be the most effective strategies. Participants in cluster 1, which solved 1.00 bugs on average and tended to use a depth-first strategy, typically only found the *valueofoutlier range* bug. This was likely because they were first drawn to the complexity of *valueofoutlier* command, and spent the rest of the session searching for information on the command's semantics.

Considering each bug's nature identifies suggests how individual differences impacted debugging performance. For example, the *for loop range* bug was embedded in the longest command in the program yet did not affect the program's output. As seen in Table 4, participants who fixed this bug spent much less time focusing on other lines of code, made many more changes to only the *for* loop command, and focused on example code from the online help more. Since over a third of each cluster fixed this bug, the complex nature of the command seemed to influence debugging success more than the comprehension strategy. This is likely because of the unfamiliarity with the language: if participants had understood the command's semantics, the complexity of the command may not have received as much attention.

The *inverted axes* bug had to be fixed by actually inspecting the graph output, which seems like it would be influenced by comprehension strategy. In fact, the more programming experience a cluster had (which influenced strategy), the higher the rate the bug was fixed. However, this was not statistically significant. Was there anything that had more influence on success on this bug? Inspecting Table 4, participants successfully fixing this bug spent much more time actually looking at the graph output by the program and inspecting the *graph* command. Looking at Table 3, we can see that participants successful on this bug had higher self-reported experience with math, statistics software, and computers. Combining these two observations suggests that participants who were successful at this bug were better able to read and understand the graph produced by the program with their domain-knowledge in statistics.

The *pvalue range* and *valueofoutlier range* bugs differed only in that the *valueofoutiler* command had a comment above it that contradicted the command's range, and it used a system variable named "_n" to refer to the current row being operated on (see Figure 2). Inspecting Table 2, we see that nearly twice as many participants found the *valueofoutlier range* bug and those that found it were more likely to fix the *pvalue range* bug. Clearly, the comment helped participants find the *valueofoutlier range* bug, and thus reveal the similar problem with the *pvalue* command. What other factors influenced success on these bugs? As discussed in the previous section, cluster 1 seemed to focus on and fix the *valueofoutlier* command, and clusters 2 and 3 were far less successful. In other words, strategy seemed to influence *how much* attention these commands received. Also, participants who fixed the *valueofoutlier range* bug tended to have more math experience and more positive attitudes towards math and computers. Possibly, participants with more math experience were better enabled to understand the mathematical nature of the system variable in the *valueofoutlier* command.

In sum, it seems that although clusters 2 and 3 were more successful than cluster 1 overall, no strategy was particularly successful. Furthermore, success on specific bugs may be more a factor of domain knowledge, rather than programming experience or comprehension strategy.

5.3 Implications

The results presented here are largely consistent with studies of program comprehension of expert programmers. For example, Corritore and Wiedenbeck [2] showed that procedural programmers used a more bottom-up, breadth-first strategy. Stata provides a more procedural-like language, and cluster 3, with the most programming expertise, also used a breadth-first strategy. Thus, for expert programmers, using an unfamiliar programming system did not seem to affect their comprehension strategy. However, this study provides further informs these findings with observations of intermediate programmers (primarily those participants in cluster 1), who tended to use a *depth*-first strategy.

Models of expert debugging, such as Gilmore's [6], accurately predict the effects of programmers using an unfamiliar environment. Unfamiliarity forced expert programmers in cluster 3 to rely more on their domain-knowledge to fix bugs rather than their knowledge of the language and environment.

Findings on novice programmers also seem to generalize to programmers using an unfamiliar environment. The findings presented here are similar to Perkins and Martin's observations of "fragile knowledge and neglected strategies" [13], in that cluster 2 participants seemed unable to apply their domain-knowledge (characterized by their general inactivity).

However, with regard to findings on the influence of comprehension strategy on debugging performance, the results presented here are largely inconsistent with studies on debugging in familiar programming systems [7, 12]. These studies suggest that experts' comprehension strategies resulted in more efficient and effective debugging performance than novices. In this study, no comprehension strategy was particularly successful (few participants fixed more than 3 bugs); and, comparing the success of strategies shows that cluster 2 and 3's strategies were comparable. Rather, *bug-specific* domain knowledge was the best predictor of debugging success, suggesting that programmers using unfamiliar programming systems utilize their domain-knowledge to aid the debugging process. It may be that as programmers become accustomed to a language and environment, comprehension strategy has a greater influence. This may be similar to Corritore and Wiedenbeck's [2] finding that over time, the impact of programming *paradigm* on comprehension strategies is much less significant.

The results of this study have many important implications. If programming system designers offered interactive tutorials of an environment and language, they may improve programmers' initial performance on debugging tasks. Programming system designers also should ensure high walk-up-and-use usability of their systems, to limit the effect that unfamiliarity with programming system may have on debugging.

These results also suggest that project leaders in industry settings should be aware that programmers unfamiliar with a program and programming system would be more effective if assigned tasks that leveraged their domain knowledge. For example, if a programmer has a lot of mathematics experience, she is more likely to succeed at debugging code that involves mathematics.

Future work will analyze the current data set for patterns in *sequences* of participants' comprehension strategies, in order to further characterize the comprehension strategies found in this study. The authors believe that analysis at this granularity may reveal details hidden by the higher-level data reported here.

5.5 Limitations

There are some obvious limitations to this study, given its exploratory nature. Seventy-five participants is a relatively small number to ensure the validity of the hierarchical clustering. Furthermore, comparing clusters by the measures that define them is dangerous, since the measures are confounded by the size of the cluster. A more formal investigation of program comprehension in unfamiliar programming systems would give participants the same task in a familiar and unfamiliar environment, and compare their strategies and performance.

The findings presented here may not hold in other domains and programming systems, particularly because the strategies found were completely dependent on measurements of participants' interactions with the Stata. For example, if a visual programming language had been used instead, there would have been no measures of the number of commands entered; a comparable visual measure may have changed the clustering results. These results may generalize to other programming systems with similar support for programming and debugging.

6. Conclusion

This paper provides empirical data about the program comprehension strategies of programmers of various expertise using an unfamiliar programming environment and language. The results of this study suggest that comprehension strategies in unfamiliar programming systems are largely determined by experience, but debugging performance depends on bug-specific domain-knowledge. We believe that this study has only begun to examine this the comprehension strategies of programmers using unfamiliar environments, and suggest future work that may better inform these findings.

7. Acknowledgements

Andrew thanks his undergraduate thesis advisors, Dr. Margaret Burnett and Dr. Bob Uttl, for their guidance for the year and a half duration of this study. Andrew also thanks his wife, who helped code over forty hours of videotape in her last trimester of pregnancy. Thanks also to Brad Myers for his help with editing. This work was supported by an Oregon State University URISC grant. Stata software was provided by Stata Corp.

8. References

[1] L. M. Berlin, "Beyond Program Understanding: A Look at Programming Expertise in Industry," at Empirical Studies of Programmers, 5th Workshop, Palo Alto, CA, 1993.

[2] C. L. Corritore and S. Wiedenbeck, "An Exploratory Study of Program Comprehension Strategies of Procedural and Object-Oriented Programmers," *Intl. J. of Human-Computer Studies*, pp. 1-23, 2001.

[3] S. P. Davies, "Models and Theories of Programming Strategy," *Intl. J. of Man-Machine Studies*, pp. 236-267, 1993.

[4] S. P. Davies, "Display-Based Problem Solving Strategies in Computer Programming," at Empirical Studies of Programmers: Sixth Workshop, Washington, D.C., 1996.

[5] R. B. Ekstrom, J. W. French, H. H. Harman, and D. Dermen, *Kit of Factor-Referenced Cognitive Tests*. Princeton, NJ: Educational Testing Service, 1976.

[6] D. J. Gilmore, "Models of Debugging," *Acta Psychologica*, pp. 151-173, 1992.

[7] L. Gugerty and G. M. Olson, "Comprehension Differences in Debugging by Skilled and Novice Programmers," in *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Washington, DC: Ablex Publishing Corporation, 1986, pp. 13-27.

[8] R. T. Jeffries, A. A. Turner, P. G. Polson, and M. E. Atwood, "The Processes Involved in Designing Software," in *Cognitive Skills and Their Acquisition*, J. R. Anderson, Ed. Hillsdale, NJ: Erlbaum, 1981.

[9] A. J. Ko, "Individual Differences in Programming, Testing, and Debugging Strategies in a Statistical End-User Programming Environment," Undergraduate Thesis, Honors College, Oregon State University, 2002

[10] J. Koenemann and S. P. Robertson, "Expert Problem Solving Strategies for Program Comprehension," at Conference on Human Factors and Computing Systems, New Orleans, Louisiana, 1991.

[11] A. v. Mayrhauser and A. M. Vans, "Program Understanding Behavior During Debugging of Large Scale Software," at Empirical Studies of Programmers, 7th Workshop, Alexandria, VA, 1997.

[12] M. Nanja and C. R. Cook, "An Analysis of the On-Line Debugging Process," in *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Shepard, and E. Soloway, Eds. Norwood, NJ: Ablex, 1987, pp. 172-184.

[13] D. N. Perkins and F. Martin, "Fragile Knowledge and Neglected Strategies in Novice Programmers," at Empirical Studies of Programmers, 1st Workshop, Washington, DC, 1986.

[14] B. E. Teasley, "The Effects of Naming Style and Expertise on Program Comprehension," *Intl. J. of Human-Computer Studies*, pp. 757-770, 1994.

[15] J. H. Ward, "Hierarchical Grouping To Optimize An Objective Function," *Journal of the American Statistics Association*, vol. 58, pp. 236-244, 1963.

[16] S. Wiedenbeck, V. Fix, and J. Scholtz, "Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical Study," *Intl. J. of Man-Machine Studies*, pp. 793-812, 1993.