Novice Programmer Strategies for String Formatting

Michael C. Hughes, Matthew C. Jadud

March 16, 2009

Abstract

In Java, `System.out.printf` and `String.format` consume a specialized kind of string commonly known as a format string. In our study of first-year students at the Ateneo de Manila University we discovered that format strings present a substantial challenge for novice programmers. Focusing on their first laboratory we found that 8% of all compilation errors and 100% of the exceptional, run-time behavior they encountered were due to the improper construction of format strings. In this paper, we present exemplars of their problematic interactions with the compiler and run-time environment, and recommend possible instructional interventions based on our observations.

1 Introduction

We are interested in the behavior of novice programmers while they are engaged in the act of programming. Specifically, we are focused on the process by which novices go from incomplete, but correct code, to code that completes a given homework assignment or laboratory task. By better understanding what our students do, we hope to develop interventions to improve the state of programming instruction.

In this particular study, we looked at the code written by students on a compile-by-compile basis. In our database, we captured a complete snapshot of their code after each compilation. This data is a source for many automatic analyses, and simultaneously provides a rich view into the growth of a student's code. The data we are investigating here is taken from the first laboratory in the first course of 143 students at the Ateneo de Manila. Through automated means we discovered that 8% of the syntax errors generated by these students and 100% of their run-time errors resulted from their attempt to use **format strings**.

Format strings are common in many languages. They provide a mechanism for specifying a message template at compile time that will be displayed with one or more values substituted for specially formatted placeholders in the string. In Java, the following print statement would, when executed, output the message `"pie equals 3.14"`.

```
float pie = 3.14159;
System.out.printf("pie equals %.2f", pie);
```

The symbol `%f` is a placeholder that indicates that an numeric value should be substituted. In particular, a floating-point value, and further, only two numbers should appear after the decimal point. So, even though the value of `pie` is `3.14159`, the message printed when this code is executed will be `"pie equals 3.14"`.

We have several reasons for focusing on format strings. Unfortunately, format strings show up in most introductory textbooks, and they often show up early in the text. Given that format strings are fragile at compile- and run-time, and represent another level of pattern-matching within an already complex syntax, it is unfortunate that they become the focus of so many input/output operations in introductory textbooks. Further, we say they are fragile at compile-time because they are a language unto themselves. Sadly, unlike the rest of the Java programming language, they are *unchecked* by the compiler. As a result, it is easy for a novice to write a format specifier that is incorrect, and fail to discover this fact until they compile and run their program. It is relatively easy to write a format string that passes the Java syntax checker, but is incorrectly specified. This turns a "syntax error" into a run-time exception.

It is this combination of complex issues—a language embedded within a language, prone to syntax errors that are discovered at run-time—that makes format strings interesting to study in the context of novice programming behavior. We begin by looking at related work and discuss the nature of our data collection. We then proceed to explore the behavior of several students in-depth, a process which reflects the analysis we undertook in preparing this work. Lastly, we discuss the behavior we observed as possible evidence of strategy formation on the part of novice programmers in light of recent pilot studies exploring the behavior of students engaged in debugging tasks and their ability to become "unstuck" in the face of adversity.

2 Background

Studies have indicated that students struggle a great deal with reading, writing, and understanding programs as they come out of their introductory programming course. McCracken et al. found that the majority of students could not successfully implement software that met a simple problem description[10]. Lister et al. explored students' ability to read programs and trace their execution on paper, sand found that many of their students struggled with this task[8]. This is not a new trend: Sporher and Soloway explored the formation of goals and plans in novice programmers in the mid-1980's, and it appeared then as it does now that writing a correct program to accomplish a well specified task is something that many students can not do[14, 15].

This trend was continued by graduates of the "Bootstrapping Computer Science Education Research" and "Scaffolding Computer Science Education

Research" workshops[2]. The first publications from this group of collaborators tended to yield similarly negative results—that novices struggle with many aspects of learning to program[7]. As a result, some of the graduates of these workshops have begun to focus their work on questions of what their students *can* do, rather what their students can not[1, 6, 13].

Our fundamental question is quite simple: what are students doing when they are writing their first programs? In our studies to date we have examined students in their first university-level programming course. These students are typically working on small, well-defined homework sets or laboratories, where they are given a piece of Java as a starting point. This code, typically 30-60 lines long, is syntactically and stylistically correct, and is related to both their course textbook and lectures. Sometime after they are given this starter code and a problem statement, most students find themselves in a battle with the compiler as they attempt to write code that correctly expresses their ideas.

In our observations of novices and their interactions with the compiler, we observed 130 students at the University of Kent over the course of the 2003-2004 and 2004-2005 academic years. These students were enrolled in a year-long introduction to object-oriented programming, and generated roughly 42,000 unique compilation events across 2000 programming sessions over the course of two years. At the Ateneo de Manila University during the 2006-2007 academic year, 143 students were observed in their first semester of Computing I, and generated 28,000 compilation events for study. In both cases, students were given the opportunity to opt-in to the study, and the collection of data was effectively invisible to the students; students taking part in the study experienced nothing different in their activities than those who chose not to take part. More detail about both of these populations can be found in [3] and [16], respectively.

In replicating our initial data collection, our collaborators at the Ateneo de Manila University observed behavior that reinforced many of our original findings. The distribution of syntax errors encountered by first-year students at Ateneo was nearly identical to those encountered by students at the University of Kent (Figure 1), and the speed which students made and corrected syntax errors while engaged in their programming was similar as well (Figure 2). In addition, they calculated the rate at which students dealt with repeated errors, a rate which we call the *error quotient*; this gives us a sense for how students deal with errors over time, and both studies imply that there is some correlation between the rate at which students correct syntax errors and traditional metrics for academic performance (e.g. grades).

In this study, we have focused on one aspect of novice compilation behavior. In particular, we examined how students dealt with format strings as arguments to methods like `System.out.printf` and `String.format`. We begin by presenting our methodology for exploring the students' behavior, followed by several small cases, or vignettes, exemplifying how students struggled with format strings and attempted to deal with their complexity. We use this qualitative exploration to inform some reasonable statistical explorations, and close with a discussion of possible instructional interventions to aid students struggling with these constructs.

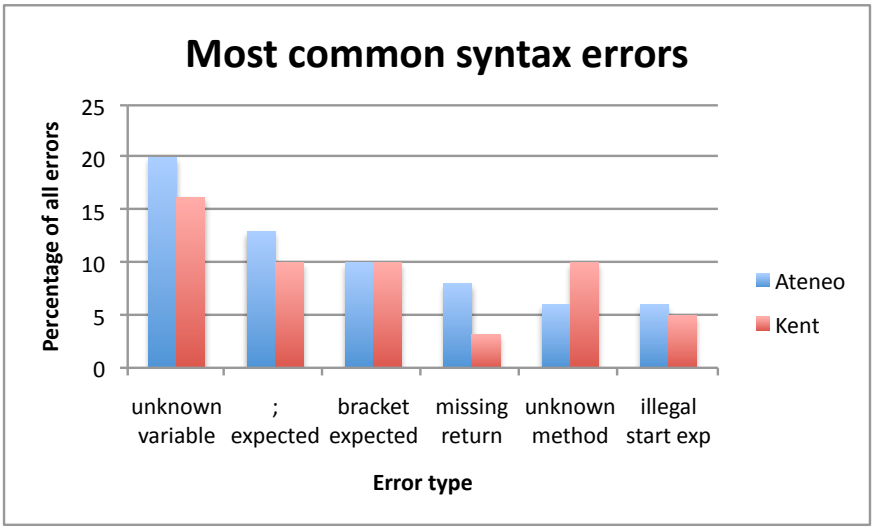

Figure 1: The most common syntax errors encountered by students at the University of Kent and the Ateneo de Manila.

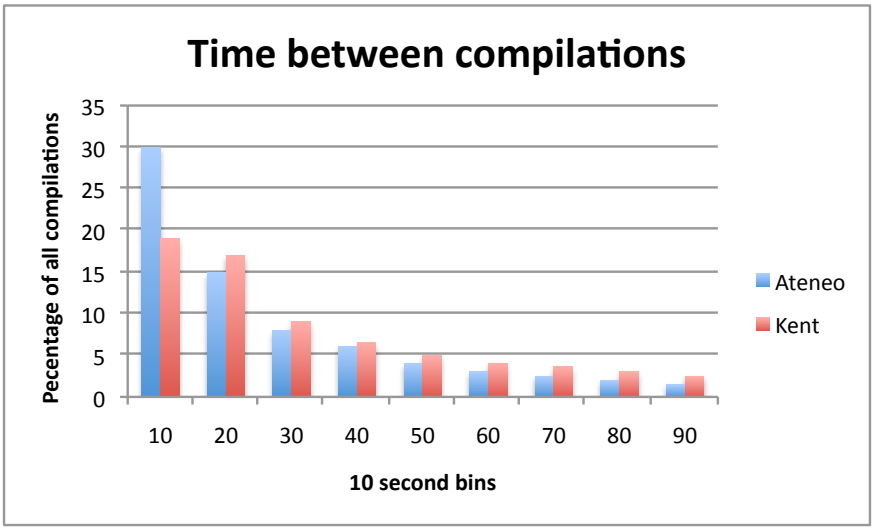

Figure 2: The time between successive compilations as observed in 270 students at the University of Kent and the Ateneo de Manila.

3 Methodology

Students struggle with errors, both at compile-time and run-time. It is not clear what roles the programming language, the development environment, and instructor play in the success of students overcoming the errors they encounter. What is clear is that we can collect rich data to begin exploring how students learn to overcome syntax and run-time errors while learning to program.

Case studies help us explore, in depth, environments where the phenomenon and its context are deeply intertwined[17]. Our goal is to better understand the behavior of novice programmers who are struggling with compilation and run-time errors in writing their first Java programs, so that we might tailor instructional interventions and tools to better support the learning process. This is because we believe that students who spend (literally) hours attempting to get their program to compile cannot possibly be learning about higher-level concepts like inheritance, encapsulation, and the role of state in control flow (to name just a few).

Our investigation begins with the instrumentation of BlueJ, so that a copy of a student's program, as well as any error messages emitted by the compiler, are captured on a remote server with every compilation. Likewise, with every class instantiation and method invocation, we capture information about the execution of their program. This yields a large, complex data set containing a sequence of snapshots of their code and run-time behavior over the course of one or more programming sessions.

A full case study of a single student would span the course of an academic year, and encompass many dozens of programming sessions. In this paper, we are not building a case regarding a single student, but instead building a case regarding the complexities of *string formatting*. Specifically, we found in our qualitative exploration of the data that beginning students struggle with writing code like:

```
System.out.printf("Cash left: %5.2f", cash);
```

In the study presented here, our exploration of programming sessions focused on the first laboratory taught at the Ateneo de Manila University in their introductory programming class during the 2006–2007 academic year. Originally reported in [16], this study observed 143 students engaged in laboratory exercises while enrolled in Computing I, which may be considered a "CS1" as typically referred to in the literature.

By looking at their first laboratory task, we could focus on the problems encountered by students programming for the first time at the University. We began by reading through many dozens of these sessions, looking for the kinds of syntactic challenges students faced. From a purely syntactic reading, we gleaned insights similar to those found in our earlier studies [4]. In our first iteration of reading and annotating the students' programs, string formatting did not present itself as being particularly problematic.

It was the inclusion of run-time errors in our data collection that provided critical insight into the students' compilation behavior. In our second reading of

individual cases, we looked not only at compile-time errors, but also the run-time errors students encountered. Run-time error analysis required the researcher to copy programs that successfully compiled into the BlueJ environment, and then execute the same sequence of object creation and method invocation steps that the student took. As a result, we discovered a number of things, the most important being the preponderance of exceptions thrown as a result of the incorrect use of format strings. We do not believe there are any exceptions thrown in our data set that come from anything other than the improper use of format strings.

4 Vignettes

Knowing that the run-time problems students encountered were restricted to format strings, we revisited students' compilation behavior to develop a more complete picture of how compilation and run-time behavior were intertwined. What we present here are three *vignettes*, or "data stories," that we hope will give the reader deeper insight into the student behaviors that we have observed as they attempt to develop strategies for dealing with the complexity of correctly authoring format strings in Java.

4.1 Adam: A Beginner's Difficulties

In this programming assignment, Adam's task is to write a program which manages the ticket sales of a typical cinema. We start our trace in the middle of his session, after he has attempted 9 compilations over 17 minutes. He is currently flushing out a new method, called `details`, for his `Ticket` object. This method's purpose is to retrieve a `String` representation of the `Ticket`, which will include the title of the movie, the price of the ticket, and the seat number assigned to the ticket holder.

His first attempt at this method looks like this:[1]

```
48  public String details()
49  {
50    System.out.println("'" +movieName +
          "' " + "(P" ticketPrice + ") -
          seat" + seatNumber);
51  }
```

When Adam compiles this version, he receives the error message

```
50: ')' expected
```

A careful reading of line 50 in his code shows that all open parentheses are indeed closed, so this is not what we call a **literal syntax error**. A literal error

[1]Lines have been broken artificially for reading in the ACM two-column style. Original sources did not break long lines.

is a syntax error whose true cause is identified in the compiler's error message. If responding to a `')' expected` error message by placing a closing paren at the correct spot within the indicated line resolves the error, then that error is literal. However, errors often turn out be non-literal, meaning that the message provided by the compiler in fact does not indicate how to solve the problem. To use Adam's code as an example, adding a `)` anywhere in line 50 will not help Adam solve the underlying syntax error. Novice programmers tend to struggle with these errors, as they require a good deal of experience and confidence to properly interpret and correct. A careful inspection of the above will reveal that this error results from improper string concatenation. Adam is missing a `+` sign to correctly combine the quoted text `"(P"` and the variable `ticketPrice`. The slot where the missing `+` should go is marked with a carat in the line below

```
50  ... + "(P"   ticketPrice + ") ...
              ^
```

The compiler is unable to recognize this problem and thus reports a `')' expected` error. Adam's immediate reaction is to avoid responding at all and instead shift his attention to a different part of the program. We see this "move on" behavior in response to difficult (usually non-literal) errors in many student sessions.

In particular, Adam decides to "move on" from the `Ticket` class and begin creating the `Cinema` class his project requires. He spends a few minutes fleshing out member variables and easily fixing a few literal syntax errors until he reaches the point at which compiling Cinema requires that the dependent class `Ticket` also compiles correctly. Forced to return to the `')' expected` error in his `details` method, Adam begins by responding literally to the error and placing a `)` at the end of the line.

```
50  System.out.println("'" +
      movieName + "' " + "(P"
      ticketPrice + ") - seat" + seatNumber));
```

Compiling this revision, Adam still receives the same error message from the compiler. This of course results from the fact that the improper concatenation occurs before Adam's edit, so the compiler will report this error first and exit.

When his attempted literal edit does not fix the error, Adam deletes the `)` he added to the end of the line. Unable to determine the source of the error, Adam again displays "move on" behavior in turning his attention to other areas of his `Ticket` class. After fixing a few errors in his constructor, he returns to the `details` method and appears to be stuck. He compiles the code repeatedly without making any changes. We observed this phenomenon of unedited repeated compilation in a number of sessions, though its intent remains unclear. Perhaps Adam needs to refresh the error on the screen, or maybe he hopes that the compiler may eventually give his a more detailed error message. In any case, Adam soon exits this cycle and attempts to radically change the `details` method to read

```
48  public String details()
49  {
50    Ticket details = new Ticket();
51
52    details.format("'" + movieName +
        "' " + "(P" ticketPrice + ") -
        seat" + seatNumber );
53
54        }
```

What prompted this drastic edit is unknown, but our best guess is that Adam had seen a `String.format` method used either in an instructor's example or in a fellow student's work. His attempt to instantiate and use a `Ticket` object within the `Ticket` class shows a rather large misconception of how objects work. This is further magnified by the fact that his `Ticket` class does not have a `format` method in his current version.

Despite these problems, the compiler still reports the same `')' expected` error, now on line 52. While we might like the compiler to say `cannot find symbol - format`, it does not, as the compiler will attempt to process arguments before method invocations on those arguments. Again, Adam "moves on" to other work on his `Cinema` and `Driver` classes. Of course, he is forced to come back when these will not compile without a clean report from the `Ticket` class which they both rely on. Returning to `details`, Adam makes the following change

```
48  public String details()
49  {
50    Ticket details = new Ticket();
51
52    System.out.println(string.format("' %s" +
        movieName + "'  (P %s" + ticketPrice + ") -
        seat" + seatNumber );
53
54  }
```

In adding format elements (such as `%s`) to his concatenation, we find that he managed to add in the previously missing `+` sign before the `ticketPrice` variable. However, Adam also forgot to add a parenthesis at the end of the line to mark the close of the `println` method. Thus, when he compiles, he still receives the error

```
52: ')' expected
```

Only now, the error is a true literal syntax error! Adding a closing paren `)` will actually make this line of code compile cleanly. Sadly though, Adam responds as if this error were still non-literal. The compiler seems to have conditioned him to respond this way.

Adam does not seem to realize what his fundamental error was or that he fixed it. He tweaks this version a few times by substituting `print` for `println` and adding one more format element - `%s` - into his concatenation. When none of these edits work, Adam finally makes the necessary literal correction and adds a closing parenthesis to the end of the line. His code now reads

```
48  public String details()
49  {
50    Ticket details = new Ticket();
51
52    System.out.print(string.format("' %s" +
        movieName + "'  (P %s" + ticketPrice + ") -
        seat %s" + seatNumber );
53
54  }
```

Adam's struggle with the `')' expected` error is finally over. His next few compilations mark a series of rapid improvements in which Adam correctly fixes semantic and syntactic errors such as the unnecessary instantiation of a `Ticket` object in the `details` method, the capitalization of `String.format`, and the need to `return` rather than print the formatted value to comply with his `details` method's declaration. His code now reads

```
48  public String details()
49  {
50
51    return (String.format("' %s" + movieName +
        "'  (P %s" + ticketPrice + ") - seat %s" +
        seatNumber ));
52
53  }
```

Compiling this version returned no syntax errors. Thus, after 44 compilations, Adam's code finally builds cleanly for the first time.

Adam immediately proceeds to run-time testing his `Ticket` class in BlueJ. After manually instantiating a basic `Ticket` object, he invokes its `details` method. BlueJ responds by reporting that a `MissingFormatArgumentException` occurred when attempting to execute line 51.

Adam responds to this error directly by comma-delimiting the format string parameters at the end of the method call.

```
48  public String details()
49  {
50    return (String.format("' + %s P %f - seat %s"
        movieName, ticketPrice, seatNumber ));
51  }
```

Compiling this change results in the return of the `')' expected` error. The cause of this is another instance in which quoted literal text and a variable are not connected or separated properly, as shown below

```
... seat %s"   movieName, ...
             ^
```

While the error is again non-literal, its underlying cause is completely different from what Adam experienced previously. Rather than a `+` concatenation symbol, a comma is needed to separate the quoted text and the variable name as two distinct arguments to the format method.

However, Adam immediately recognizes this case as another instance of his previous solution to the problem of a non-literal `')' expected` error and makes the change we see below within 27 seconds

```
48  public String details()
49  {
50    return (String.format("' + %s P %f -
        seat %s" + movieName, ticketPrice,
        seatNumber ));
51  }
```

This quick revelation is perhaps an indication that Adam is beginning to differentiate at some level between literal and non-literal syntax errors. However, it seems that rather than closely examining his code to discover the cause of an error, he may rely on selecting a response from an internal library of past experience. These "conditioned" responses may play an intermediary role in allowing a novice to gain expertise in diagnosing and resolving non-literal errors.

Adam is moving toward the correct implementation of `String.format`, which requires that all variables be passed as comma-separated method parameters in the order they will appear in the resulting string. However, his current version includes the variable `movieName` as part of the string concatenation rather than as a comma-separated parameter.

When he invokes this method in BlueJ, Adam finds that this version of `details` throws a `IllegalFormatArgumentException`. This results from Java's failed attempt to format the String `seatNumber`, which in the format string is declared to be a floating-point number by the use of `%f`. Put simply, the problem is not the order of his format string specifiers, but instead the fact that `movieName` should be passed as a parameter, and not be included in the concatenation itself. Adam attempts a number of edits to fix this problem, none of which work. His most notable attempt to eliminate his run-time error is when he splits the tasks of string concatenation and returning the string over two lines, as in

```
48  public String details()
49  {
50    String details = String.format("' + %s
```

```
        (P %5.2f) - seat %s \n" +
        movieName, ticketPrice, seatNumber );
51    return details;
52  }
```

We do not actually know why Adam did this, but perhaps he thought he could either solve the problem or better isolate the source of the run-time error if he separated the execution into two distinct tasks (formatting and returning, in this case). We find this separation of tasks fascinating, and we will return to explore this idea in depth later.

Despite his continued problems, we note that Adam does make progress toward a semantically correct program by adding the formatting specifier `%5.2f` to his literal string argument. This edit forces the displayed ticket price to be at least five characters wide and have two decimal places of precision, as the assignment requires. However, when he executes his code again, the same `IllegalFormatArgumentException` occurs. This is because Adam has not yet realized that instead of string concatenation, he needs to use a comma between the literal text and the variable `movieName`.

Adam responds to this run-time exception by collapsing the tasks back into one line. After a few more cycles of code experimentation which result in this same exception, Adam finally corrects his string concatenation completely, as shown below

```
50  return String.format("' %s (P %5.2f) -
      seat %s \n", movieName, ticketPrice,
      seatNumber );
```

Entering appropriate parameters to construct a `Ticket` object and then executing his `details` method manually in BlueJ, Adam now avoids all exceptions and instead receives the return value of the `details` method when applied to a `Ticket` for the movie Transformers

```
OUTPUT:     ' Transformers (P 120.00) - seat a1
```

Adam makes one final tweak to make sure the movie title is properly quoted, and ends his programming session.

```
OUTPUT:     'Transformers' (P 120.00) - seat a1
```

As a whole, Adam's session, which lasts 97 minutes, is distinguished by steady though extremely slow progress. He began the session with no `details` method for his `Ticket` class, and managed to end with a syntactically and semantically correct implementation. However, he likely spent far more time on the problem then either he or his instructors would have liked. This is especially notable because he never fully developed and tested his other classes (`Cinema` and `Driver`), at least in this recorded session. In particular, his `Cinema` class contained a `printSummary` method which required labeled output formatted to two decimal places of the total sales of the `Cinema`. When exiting his session, his `printSummary` method contained this statement

```
52  System.out.printf( "Total Sales: " +
      "P %4.2f" + totalsales);
```

This appears to be another problematic case of string formatting. As it stands, line 52 will likely result in a run-time exception because the variable `totalsales` is not properly passed as a separate argument. We are unsure if Adam could solve this problem easily given the struggles he endured in the past 97 minutes.

4.2 Benita: Towards Strategy Formation

Adam clearly struggled with the tasks of string concatenation and output formatting, and he was not alone. Attempting to combine multiple variable values of different types and their associated text labels into a single coherent string proved to be challenging for many of the novice programmers that we observed. In a study of 185 students tackling the `MobilePhone` programming assignment, we found that 8% of all syntax errors and 100% of all run-time errors came from lines which formatted output. Given this systematic level of difficulty, it seems that the tasks on these lines are worth examining closely for ways to reduce their difficulty. Perhaps the best way to accomplish this is to investigate the solutions to the complexity of string formatting which students naturally develop.

Consider the case of Benita, a student attempting to code a `MobilePhone` class which can track phone calls and text messages as well as add and subtract payment credits accordingly. We jump into Benita's session as he writes a method called `printSummary`, which will output a well-labeled textual summary of all the state variables of his `MobilePhone`. The required concatenation, formatting, and printing is similar to Adam's task. Benita's current code is shown below

```
61  public void printSummary()
62  {
63    System.out.printf( " Credits left: "
        "creditsLeft = P%5.2f \n",
        getLoadLeft() );
64    System.out.println( "Total call duration: "
        getTotalMinutesCalled() + " mins.");
65    System.out.println( "Rate per call: " +
        callRate);
66    System.out.println(
        "Number of text messages sent: " +
        getNumTextMessages() );
67  }
```

Upon compilation, Benita receives Adam's favorite error message

```
63: ')' expected
```

A close look at line 63 reveals, as we might fear, a non-literal error in which Benita is missing a + sign to correctly concatenate his quoted literal strings. Benita makes the following edit in response

```
63  System.out.printf(
      " Credits left: creditsLeft = P%5.2f \n",
      getLoadLeft() );
```

This fixes his problem, as he eliminated the need for concatenation in line 63 entirely by pushing all his quoted strings together. Compiling this version yields the message

```
64: ')' expected
```

We realize that his missing + mistake is not isolated, as it also occurs in the very next line of code, line 64.

```
64  System.out.println( "Total call duration: "
      getTotalMinutesCalled() + " mins.");
```

In his next edit, Benita focuses his attention not on line 64 but on the previous line, which he had already freed of syntax errors. We're not sure if he didn't read the error message closely and mistakenly thought that line wasn't fixed properly, or if he wasn't satisfied with his solution even if the compiler was. In any case, Benita made the following change, which proves to be extremely interesting

```
64  System.out.print( "Credits left: " );
65  System.out.printf( "creditsLeft = P%5.2f \n",
      getLoadLeft() );
```

With this edit, Benita has largely separated the task of string concatenation away from the task of formatting a decimal output. We note that in this version he does print a literal label for `"Credits left"` twice, but we'll see that this works itself out over time.

After this change, he moves on to fix the missing + in the `"Total call duration"` line. His new `printSummary` method now looks like

```
61  public void printSummary()
62  {
63
64    System.out.print( "Credits left: " );
65    System.out.printf( "creditsLeft = P%5.2f \n",
        getLoadLeft() );
66    System.out.println( "Total call duration: " +
        getTotalMinutesCalled() + " mins." );
67    System.out.println( "Rate per call: " +
        callRate);
```

```
68    System.out.println(
        "Number of text messages sent: " +
        getNumTextMessages() );
69  }
```

This compiles cleanly, so Benita conducts a quick run-time test. Achieving the desired formatted output for `creditsLeft`, he proceeds to adjust the print statement of his `callRate` variable in a similar fashion, since it also needs to be printed to two decimal places. Again he splits the printing of labels and output formatting over two lines.

```
67  System.out.print( "Rate per call: " );
68  System.out.printf( "callRate = P%5.2f \n",
      callRate);
```

We're not sure why Benita deliberately inserts a literal label for `callRate` twice, but perhaps it helps him track exactly what each line does. After testing this version to make sure `callRate` prints correctly, he promptly removes the repetitive labels in both `callRate` and `creditsLeft` statements. His final version of the `printSummary` method appears below

```
61  public void printSummary()
62  {
63
64    System.out.print( "Credits left: " );
65    System.out.printf( "P%5.2f \n",
        getLoadLeft() );
66    System.out.println( "Total call duration: " +
        getTotalMinutesCalled() + " mins." );
67    System.out.print( "Rate per call: " );
68    System.out.printf( "P%5.2f \n", callRate);
69    System.out.println(
        "Number of text messages sent: " +
        getNumTextMessages() );
70  }
```

With a working version of `printSummary` completed, Benita has solved every task required in the assignment. After a relatively brief struggle with concatenation and formatting, Benita has found (or perhaps stumbled upon) a solution which works nicely. By splitting the tasks of printing quoted, literal text labels of variables and printing the formatted versions of the variables themselves onto multiple lines, we believe that Benita gained a number of advantages. Splitting tasks onto multiple lines makes it easier to identify the actual cause of a syntax error, since the compiler reports an error along with the corresponding line number. When lines have less content, their broken pieces should be more easily identified. Furthermore, splitting the tasks onto multiple lines may make it easier to read and parse what each line does. We note that these advantages,

particularly the improved isolation of a problematic bit of code after an error message, could become especially useful in the event of a non-literal error.

Perhaps equally notable along with Benita's splitting tasks strategy is what he does after achieving a working program. After thoroughly testing his code, Benita proceeds to add an in-code comment to his `MobilePhone` class which explains the execution flow of his `main` method. Commenting code is a programming practice which is often under-used or entirely avoided by novices. This, taken together with Benita's easy handling of errors surrounding string formatting suggest many things, but perhaps most obviously that this may not be Benita's first time programming in Java.

4.3 Cosme: Strategic, Almost

It would seem, looking at Benita's `MobilePhone` session, that splitting the tasks of printing and formatting across multiple lines is a boon to novice programmers. But before we become too enamored with this strategy, we turn out attention to a student session in which this approach did not provide much help. In coding his `MobilePhone`, Cosme attempted to write code similar to Benita's, but with less success.

```
38  public void printSummary()
39  {
40    System.out.print( "Credits left:" );
41    System.out.printf( "getLoadLeft = P % 5.2f",
        getLoadLeft() );
42    System.out.print( "Total call duration:" );
43    System.out.printf(
        "getTotalMinutesCalled = P % 5.2f",
        getTotalMinutesCalled() );
44    System.out.println( "Rate per call:" +
        callRate );
45
46  }
```

The code looks nice at first glance and compiles cleanly. A quick run-time test, however, is stopped short by an `IllegalFormatConversionException`. Looking closely at Cosme's work, we find the source of the problem lies in line 43.

```
43  System.out.printf(
      "getTotalMinutesCalled = P % 5.2f",
      getTotalMinutesCalled() );
```

Cosme is attempting to format an integer, the return value of `getTotalMinutesCalled`, as a real number with two decimal places, as specified by `% 5.2f`. Java refuses to automatically convert between these types, and thus throws an exception.

In response, Cosme makes a few cursory edits to the literal strings involved, such as adding line breaks and removing the space to make `% 5.2f` become `%5.2f`. When these edits do not overcome Java's refusal to convert between numerical types, Cosme removes the formatting entirely from (now) line 42. This removal is semantically correct, as it is both not required and entirely unnecessary to format the output string of this integer. Cosme is left with

```
38  public void printSummary()
39  {
40    System.out.print( "Credits left:" );
41    System.out.printf( "P %5.2f \n",
        getLoadLeft() );
42    System.out.println( "Total call duration:" +
        getTotalMinutesCalled() );
43    System.out.println( "Rate per call:" +
        callRate );
44
45  }
```

After giving up on formatting `totalMinutesCalled`, Cosme ends his session. We note that he was likely distracted rather than assisted by the splitting tasks technique and ultimately failed to apply formatted output to the correct variable. We don't know if he ever got around to formatting the output of `callRate` as required. And while his output strings are split similarly to Benita's, it may not have been as intentional on Cosme's part as it was so clearly when Benita divided the tasks of printing and formatting output in the `MobilePhone` class.

5 Quantitative Analysis

In our exploration of the data, we have discovered that many novice programmers struggle with the task of printing out a well-labeled and formatted summary of a numerical variable. Over 8% of all observed syntax errors and 100% of the run-time exceptions observed in the first laboratory assignment can be traced to this task. Furthermore, Adam's vignette provides considerable evidence that such errors can limit a student's performance so severely that they may put in hours of work only to give up before their assigned task is complete. However, Benita's experience in splitting the tasks of concatenation and formatting onto multiple lines looks to be an effective strategy for helping novice programmers avoid Adam's predicament. Of course, Cosme's example proves that the splitting tasks strategy is by no means universally effective.

The strategy of splitting printing and formatting across multiple lines naturally occurs in our data set. It appears, at least for Benita, that this conflated fewer errors and aided him in rapidly correcting syntactic and run-time problems. In seeing that Cosme did not have the same experience (despite his code being similarly constructed), we wonder: does this strategy occur elsewhere in

our population, and does it help reduce the incidence of compile- and run-time errors when it comes to string formatting?

Our quantitative analysis centers around two distinct sub-populations in the data set: those who divide the tasks of label printing and formatting, and those who do not. For convenience, we'll refer to the first group as the "strategy" group and the second as the "non-strategy" group.

Given these two distinct groups, a few questions arise: First, does the strategy group incur fewer formatting errors than the non-strategy group? Next, do they encounter fewer exceptions during run-time testing than their non-strategy counterparts? Finally, does the strategy group avoid syntax errors in general more effectively than their counterparts? The first two questions directly relate to errors with concatenation, formatting, and printing and seeks to analyze whether the strategy achieves its apparent specific purpose in avoiding these errors. The last question seeks to determine whether students in the strategy group might be more successful in dealing with errors (in general) than their non-strategy counterparts.

To answer these questions, we decided to conduct a statistical study of all the available student coding sessions which tackled the `MobilePhone` programming assignment. We chose to restrict our study to this particular assignment for many reasons. First, holding the assignment consistent allows much more reliable comparison of the performance of two different students. Next, the `MobilePhone` in particular was a rather small-scale assignment with a highly standardized desired result. This allowed us to automate the task of separating strategic and non-strategic sessions rather easily. Finally, the `MobilePhone` assignment offered a comparatively high number of available sessions (over 20 strategic sessions and more than 200 sessions total).

5.1 Identifying Uses of Strategy

To isolate all student coding sessions within the data set which split the tasks of concatenation and formatting onto multiple lines, we applied the following criteria to the source code of each student session in the data set.

First, we searched for a session whose code contained both `"System.out.print"` and `"Credits left"`. This first criteria makes sure the a given coding session attempts to print out a textual label for the variable `creditsLeft`. This labeling is one requirement of the `MobilePhone` programming assignment, so it was consistently present in all student sessions. Since the assignment also requires that `creditsLeft` be formatted to two decimal places, this line of code could possibly benefit from the application of strategy.

Second, we made sure that sessions matching the first criteria had a subsequent line of code containing `%` without a coinciding attempt to print out the textual label `"Credits left"`. This second criteria ensures that the student attempted to format textual output on a subsequent line following the literal labeling of the variable in question. Together with the first it provides a reasonable method for automated identification of potential strategic coding sessions within the data.

We found in practice that these two criteria when filtered through the data resulted in 24 student coding sessions which potentially split the tasks of concatenation and formatting onto multiple lines. We then examined these sessions individually to ensure that concatenation and formatting were actually divided across multiple lines of code. We eliminated those which clearly referred to different variables in the labeling and formatting lines. We also struck out one search result in which the potential implementation of strategy was clearly a commented portion of code. The final group of strategy-implementing student sessions had 21 members. The remaining subset of the data designated as "non-strategy" held 185 members.

5.2 Variables of Interest

To properly answer our questions related to the relative performance of the strategic and non-strategic groups, we examined the sessions within each group over three parameters, one for each question. These parameters are formatting error rate r_f, exception rate r_x, and general error rate r_e. We define them as follows

$$r_f = \frac{\texttt{compilations with formatting errors}}{\texttt{compilations with syntax errors}}$$

$$r_x = \frac{\texttt{exceptions}}{\texttt{total invocations}} \tag{1}$$

$$r_e = \frac{\texttt{compilations with syntax errors}}{\texttt{total compilations}}$$

We expect that the error rate r_f will indicate whether strategy and non-strategy sessions as a whole differ in the probability that their attempts at concatenation, formatting, and printing will incur a syntax error. Similarly, exception rate r_x will show whether the groups differ in their likelihood of finding difficulty with formatting when attempting to execute their programs. Finally, we expect that the error rate r_e will indicate whether the strategy group in general receives syntax errors less frequently than their non-strategic counterparts.

5.3 Descriptive Statistics

We have chosen three dimensions along which we can compare the strategy and non-strategy groups in their ability to avoid formatting errors, run-time exceptions, and syntax errors in general. After acquiring relevant statistical data on each parameter of interest, we then subject each parameter to descriptive interpretation to determine whether strategy and non-strategy groups differ significantly in their compilation and run-time behavior.

5.3.1 Formatting Error Rate r_f

Our first analysis considers the parameter of formatting error rate, which measures the frequency at which a given session's syntax errors were related to string formatting. Descriptive statistics were calculated for 21 strategic and 185 non-strategic sessions for the formatting error rate parameter r_f. For the strategic population, the observed mean is $\bar{x_s} = .12$ and observed standard deviation $s_s = .25$. For the non-strategic population,the observed mean is $\bar{x_n} = .08$ and observed standard deviation $s_n = .16$. The difference between these two means is significantly less than the standard deviations we observed in both cases. Given this level of similarity, we can reasonably conclude that there is no substantial difference between the two populations. Both strategy and non-strategy groups appear to have equal prevalence of formatting errors within their overall collection of syntax errors according to our analysis.

5.3.2 Exception Rate r_x

Next, we consider the exception rate r_x of each session in the strategic and non-strategic groups. Examining this parameter should indicate whether the strategic groups avoid run-time exception errors better than the non-strategic group.

Exception rate values were calculated for 19 strategic and 158 non-strategic sessions. These counts are slightly lower than the counts for the other parameters because not all observed sessions contained invocations. Since calculating the exception rate requires that at least one invocation occur in the student session, we chose not to consider sessions which consisted entirely of compilations in this portion of our analysis.

Calculating descriptive statistics for this parameter, we found that the strategic population has an observed mean exception rate of $\bar{x_s} = .08$ and observed standard deviation $s_s = .19$. For the non-strategic population,the observed mean is $\bar{x_n} = .04$ and observed standard deviation $s_n = .09$. Although the strategic mean appears to be almost twice as large as the non-strategic mean, we note that both lie well within one standard deviation of each other. Given this level of similarity, it does not appear that a significant difference exists between these two parameters. Our brief statistical analysis concludes that both strategy and non-strategy groups have the same likelihood of hitting a run-time error when executing their programs.

5.3.3 Error Rate r_e

As a final step of our statistical analysis, we consider the error rate r_e of both strategy and non-strategy groups and attempt to determine whether there exists a significant difference in their respective abilities to avoid syntax errors in general. After calculating error rate for the 21 strategic and 185 non-strategic student sessions, we obtained descriptive statistics for the error rate parameter.

For the strategic population, the observed mean is $\bar{x_s} = .45$ and observed standard deviation $s_s = .29$. For the non-strategic population,the observed

mean is $\bar{x_n} = .55$ and observed standard deviation $s_n = .24$. Both means and standard deviations appear reasonably well-matched. The difference between the two means is overwhelmed by the size of the observed standard deviations. Lacking strong evidence to claim this difference is significant, we find that strategy and non-strategy groups are in practice indistinguishable in their general syntax error rate.

6 Discussion

Statistically, it does not appear that we can naively detect the use of coherent strategy in our subjects regarding the splitting of format strings over multiple statements in their code. Our aggregate statistics and our qualitative work, however, continue to point to the fact that format strings are problematic for novice programmers. We wish to examine our subjects' struggles through the lens of three recent studies, all of which may provide insights both into the observed behavior as well as future directions for study.

6.1 The quality of error messages

In their SIGCSE 2008 paper, Nienaltowski, Pedroni, and Meyer explore whether the level of detail and presentation of error messages contribute significantly to the success of novice programmers learning Java and Pascal[12]. Their tentative findings indicate that more detailed messages do not seem to be any more helpful to students when dealing with errors in their program than when the compiler displays a short message and highlights the line containing the error. However, given the small number of subjects across the two institutions participating in the study, the authors were unable to assure the significance of their findings in the majority of their hypotheses—a larger number of subjects and better experimental controls may later reverse this finding.

Our observations of novice students at University of Kent and Ateneo de Manila generally confirm Nienaltowski et al.'s notion that compiler error messages are not very helpful to novice programmers. This is best seen in the struggle novices have with identifying literal and non-literal errors, as our walk through of Adam's session illustrated. Adam's struggle to identify and correct the source of many of his syntax errors is not odd—from our data, we can see that it is consistent with the behavior of the vast majority of his classmates. Coupled with the preliminary findings of Nienaltowski et al., we begin to see get a sense for why syntax errors persist across compilations, and how much complexity we are asking our beginners to tackle in the first few weeks of learning to program. Given that many students spend the majority of their time interacting with code that does not compile (a frustrating experience for anyone, regardless of their level of experience), it would not be surprising if further work by Nienaltowski et al. demonstrates that the length and level of detail of syntax error messages does not significantly help beginners in their quest to write correct code.

6.2 Error response speed

In the same work, Neinaltowski et al. report that students with more programming experience tend to answer questions about syntax errors more quickly, regardless of whether or not the response corrected the error. They did not observe coding behavior directly to obtain this result. Rather, students were asked to observe an syntactically incorrect code fragment along with the corresponding error message and identify the correct solution from a set of possibilities. Based on timing of this questionnaire, the authors suggest that the more code students write, the more likely they are to encounter an error similar to those seen before and should therefore respond to the error more quickly.

We did not examine timing data based on different levels of student experience, so we cannot comment on the correlation between experience and response time. However, we did directly measure the interval between receiving the error message and the student's next compilation action. This data provides insight into the actual amount of time novice students spend attempting to resolve syntax errors. In comparing the behavior of students both at the University of Kent and Ateneo de Manila (Figure 2), we see that our students are recompiling their code quickly. The majority of the observed compilations take place in less than 30 seconds after the prior compilation.

For example, in Section 4.1 we looked closely at the sequence of errors that arose as Adam worked through his code. We can simplify this vignette by simply flagging each compilation as either ending in a syntax error (`'x'`) or as being syntactically correct (`'X'`), and his interactions with the compiled code as either ending in a normal exit from a method invocation (`'-'`) or a run-time exception (`'!'`). Adam's vignette, using this coding, looks like this:

```
12345678901234567890123456789012345678901234567890
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxX--!xX-!--X-!XXX-!xX-!X-!X--X--
```

The first 46 compilations in Adam's efforts ended in a syntax error. In terms of time spent between compilations (throwing away only one outlier at 11 minutes), the majority of these compilation intervals took less than a minute—which does not represent a large amount of time for a student to consider an error, make an edit, and compile their code again (Figure 3). It may be worth investigating whether increased programming experience reduces the time spent actually revising and recompiling code, rather than answering a questionnaire.

6.3 From errors to debugging

Reflecting our findings through yet another pilot study, we consider the results also published by Murphy et al. at SIGCSE 2008. In "Debugging: The Good, the Bad, and the Quirky — a Qualitative Analysis of Novices' Strategies," we are presented with the results from the observations of 21 students engaged in the process of debugging a program with one or more semantic errors[11]. These 21 subjects were spread across seven institutions, and the six authors do not go

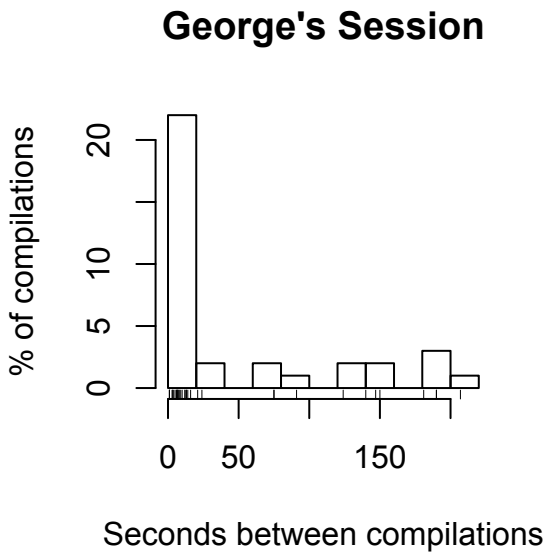

Figure 3: Distribution of time spent between compilations in George's session.

into any great detail regarding how they distilled their findings. That said, we take their gestalt findings as a preliminary result, and consider it in the light of our own readings of novice behavior as captured by our on-line protocols.

In identifying "good" behaviors, Murphy et al. focused on 12 productive strategies that they felt the students consistently exhibited. In their pool of 21 subjects, a majority of them used tracing extensively. Some did this mentally, both silently and think-aloud. External aids (like paper) were "rarely" used, and nine students altered code to isolate a problem. "Alterations" include commenting or uncommenting code and moving blocks of code around. We find the notion of altering code to isolate errors fascinating, as it lies at the heart of our observations regarding students and their attempt to deal with run-time errors surrounding format strings. From our observations, it appears that roughly 15% of our subjects broke large format strings into multiple print statements. Our suspicion is that this was intentional—a strategy used by these students to help get around or otherwise "figure out" the problems they were observing in the compilation and execution of their programs.

Murphy et al. also report that some students exhibit "tinkering", or making unproductive, seemingly arbitrary changes to their code. This includes commenting and uncommenting "suspicious" lines of code or pasting in code from other sources in an attempt to solve their problem. We observe tinkering in many of the compilation histories we read in our own study, such as Adam's inexplicable addition of the construction of a Ticket object inside the Ticket

class' `details` method.

Considering tinkering alongside other observed behavior, Murphy et al. remark that "although students read the code, it was unclear how hard they tried to understand it." We would normally question this value-laden conclusion, but between our own data and the initial findings of Nienaltowski et al., we are inclined to agree—students like Adam will press on even in the face of confusion or a lack of understanding.

6.4 Toward strategies for "unsticking"

Both Nienaltowski's and Murphy's initial investigations indicate that some students have begun to develop methods for overcoming problems while learning to program. The feedback from the environment is apparently problematic in this process for many students, but despite poor tools, some find ways of successfully working around them. However, as reported in Murphy et al.'s work, as well as in our own observations, the number of students who consistently demonstrate this skill early in their programming career are relatively few and far between.

McCartney, Eckerdal, Moström, Sanders, and Zander interviewed 14 students at six institutions who were in their final year of a degree in Computer Science (or its local equivalent)[9]. These interviews focused on strategies that these students had developed over the course of their studies for getting through challenging programming exercises. From these interviews, they identified 35 distinct strategies which they distilled into 12 more abstract categories. Among these concrete strategies are "break into parts," "use incremental development," "get help," and "be persistent/don't stop."

McCartney et al.'s work gives us some small insight into the strategies that we have observed in our own data through the words of a small group of final-year students. A few of our first-year students are clearly breaking their problematic string formatting code down into smaller pieces, although it does not appear to set them apart (on our crude measures) from their peers.

We can also say that amongst all of our subjects, it is likely that we are observing both students who seek help and those who "don't stop," pushing doggedly through a difficult error despite apparent lack of understanding. Seeking help might be inferred from a string of syntactic errors, a long pause between compilations, and a correct solution (to their problem-of-the-moment) appearing. However, our data indicates that students are more likely to attempt various kinds of work-around strategies rather than seek help. This can include repeated moving between different parts of the program, as we observed early in Adam's session, or rewriting problematic code fragments from scratch rather than debugging, as Murphy et al report. Unfortunately, unproductive tinkering is also a common alternative to seeking help.

7 Future work

From our own inquiries and these three pilot studies we see that there is a great deal of work to do regarding the understanding of novice programming behavior. Our work uncovers what we believe to be a significant problem regarding the confusion that stems from literal and non-literal compilation errors. This may shed light on the behaviors we have presented and our colleagues observed—that students will "move on" or otherwise try and avoid a problem, often with poor results. Lastly, we believe we may be seeing a kind of "conditioned" behavior amongst our students, where the compiler consistently, negatively reinforces programming behaviors that we know do not lead to learning or success. We discuss each of these in slightly more detail below.

7.1 Literal vs. Non-Literal Errors

The distinction between literal and non-literal syntax errors is difficult for beginning programmers. Adam's struggle with string concatenation and parameter delimitation within his string formatting task was never helped by any of the compiler's error messages, which almost always read `')' expected`. Furthermore, when an actual parenthesis was expected, Adam responded as if the error were still non-literal.

Lying is a common part of human relationships—we lie to present positive pictures of ourselves, to minimize conflict—but it also damages relationships and ultimately leads to mistrust[5]. Of course, a non-literal error message is not, strictly speaking, a *lie*, as the compiler correctly fails according to the grammar of the language. But it is often the case that responding literally to a syntax error will not fix the error; in the case of `';' expected`, as many as 50% have been observed to be non-literal.[2] For many novices, the idea that the computer may report an error message which does not point the user in the correct direction may run contradictory to their conceptions of computers as all-knowing machines. Future work will continue our exploration of non-literal errors in novice programming experience and further this work in developing and analyzing strategies by which students avoid or correctly diagnose these dissembling errors.

7.2 "Moving On" From Errors

A common response of novice students toward a syntax error alert during compilation is to simply "move on" from the highlighted problem and work on another portion of the source code. This behavior was observed in many of the sessions examined, including the story of Adam, which is detailed in full in Section 4.1. At first glance, we might be inclined to view this negatively, assuming that students are frustrated with a particular error in their code. However, "moving on" may also allow students to revisit an error later on with fresh eyes and be

[2]Based on a reading of roughly 900 `';' expected` errors from the University of Kent data set.

more likely to spot the problem. We are interested in using methods that provide more access to the student and their thoughts—interruption protocols and exit interviews—to get a better sense for why students leave problematic errors behind. Our hope is that this understanding will lead to better instructional interventions for aiding students in their task.

7.3 Conditioned Error Response

One of the most interesting and unexpected outcomes of the vignettes prepared was the apparently conditioned behavioral responses to compiler error messages observed in Adam's story. We do not believe we are over-reaching when we say that this appears to be a classic case of behavioral conditioning. This model may prove useful in helping instructors understand how a novice student unfamiliar with the error diagnosis and repair process will respond to compiler error messages. Our first step will be to revisit our data in search of other examples of these apparently "conditioned" responses, so as to determine how common (or rare) this phenomenon might be.

8 Conclusion

We have conducted a qualitative exploration that begins to explore the interaction of compile-time and run-time behavior of students engaged in writing their first programs in a university classroom setting. By reading through their sessions, compile-by-compile and invocation-by-invocation, we have constructed three representative vignettes that ground the reader in the complexity of the data as well as provide a deep understanding of what students do when they are struggling to write code that achieves a particular task. This initial reading implied that some students may have formed strategies for dealing with some of the complexities of format strings, and our statistical analysis explores conjecture; that analysis suggests that, in the large, the phenomenon does not occur widely, and it does not seem to differentiate one group substantially from the majority.

9 Acknowledgements

At the University of Kent, we would like to acknowledge Poul Henriksen, Michael Kölling, David Barnes, Ian Utting, and Sally Fincher for their support, and the students who participated during the 2003–2004 and 2004-2005 offerings of CO320. At the Ateneo de Manila University, the authors thank Christine Amarra, Andrei Coronel, Jose Alfredo de Vera, Sheryl Lim, Ramon Francisco Mejia, Jessica Sugay, Dr. John Paul Vergara and the technical and secretarial staff of the Ateneo de Manila's Department of Information Systems and Computer Science for their assistance with this project. We thank the Ateneo de Manila's CS 21 A students, school year 2006–2007, for their participation. Finally, we thank the Department of Science and Technology's Philippine Council

for Advanced Science and Technology Research and Development for making this study possible by providing the grant entitled "Modeling Novice Programmer Behaviors Through the Analysis of Logged Online Protocols."

References

[1] Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, Kate Sanders, and Beth Simon. Commonsense computing: using student sorting abilities to improve instruction. In *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pages 276–280, New York, NY, USA, 2007. ACM.

[2] Sally Fincher and Josh Tenenberg. Using theory to inform capacity-building: Bootstrapping communities of practice in computer science education research. *Journal of Engineering Education*, 95(4):265–278, October 2006.

[3] Matthew C. Jadud. *An exploration of novice compilation behavior in BlueJ.* PhD thesis, University of Kent, May 2006.

[4] Matthew C. Jadud. Methods and tools for exploring novice compilation behaviour. In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, pages 73–84, New York, NY, USA, 2006. ACM.

[5] Lene Arnett Jensen, Jeffrey Jensen Arnett, S. Shirley Feldman, and Elizabeth Cauffman. The right to do wrong: Lying to parents among adolescents and emerging adults. *Journal of Youth and Adolescence*, 33, 2004.

[6] Gary Lewandowski, Dennis J. Bouvier, Robert McCartney, Kate Sanders, and Beth Simon. Commonsense computing (episode 3): concurrency and concert tickets. In *ICER '07: Proceedings of the third international workshop on Computing education research*, pages 133–144, New York, NY, USA, 2007. ACM.

[7] Gary Lewandowski, Alicia Gutschow, Robert McCartney, Kate Sanders, and Dermot Shinners-Kennedy. What novice programmers don't know. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 1–12, New York, NY, USA, 2005. ACM.

[8] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 119–150, New York, NY, USA, 2004. ACM.

[9] Robert McCartney, Anna Eckerdal, Jan Erik Mostrom, Kate Sanders, and Carol Zander. Successful students' strategies for getting unstuck. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 156–160, New York, NY, USA, 2007. ACM.

[10] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *SIGCSE Bull.*, 33(4):125–180, 2001.

[11] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 163–167, New York, NY, USA, 2008. ACM.

[12] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: what can help novices? In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*, pages 168–172, New York, NY, USA, 2008. ACM.

[13] Beth Simon, Tzu-Yi Chen, Gary Lewandowski, Robert McCartney, and Kate Sanders. Commonsense computing: what students know before we teach (episode 1: sorting). In *ICER '06: Proceedings of the 2006 international workshop on Computing education research*, pages 29–40, New York, NY, USA, 2006. ACM.

[14] James C. Spohrer, Elliot Soloway, and Edgar Pope. Where the bugs are. In *CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 47–53, New York, NY, USA, 1985. ACM.

[15] James G. Spohrer and Elliot Soloway. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 230–251, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[16] Emily S. Tabanao, Ma. Mercedes T. Rodrigo, and Matthew C. Jadud. Identifying at-risk novice java programmers through the analysis of online protocols. In *Proceedings of the 8th Philippine Computing Science Congress*, Quezon City, Philippines, 2008. Computing Society of the Philippines.

[17] Robert K. Yin. *Applications of Case Study Research Second Edition (Applied Social Research Methods Series Volume 34)*. Sage Publications, Inc, December 2002.