J. EDUCATIONAL COMPUTING RESEARCH, Vol. 49(3) 293-325, 2013

AN ANALYSIS OF JAVA PROGRAMMING BEHAVIORS, AFFECT, PERCEPTIONS, AND SYNTAX ERRORS AMONG LOW-ACHIEVING, AVERAGE, AND HIGH-ACHIEVING NOVICE PROGRAMMERS*

MA. MERCEDES T. RODRIGO, THOR COLLIN S. ANDALLAZA, FRANCISCO ENRIQUE VICENTE G. CASTRO, MARC LESTER V. ARMENTA, THOMAS T. DY
Ateneo de Manila University

MATTHEW C. JADUD
Berea College

ABSTRACT

In this article we quantitatively and qualitatively analyze a sample of novice programmer compilation log data, exploring whether (or how) low-achieving, average, and high-achieving students vary in their grasp of these introductory concepts. High-achieving students self-reported having the easiest time learning the introductory programming topics. In a quantitative analysis, though, high-achieving and average students were: 1) more effective at debugging (on average, as quantified by Jadud's Error Quotient (EQ)) than low-achieving students; and 2) were least confused, as quantified using Lee's confusion metric. However, the differences in EQ and confusion between groups were not statistically significant. This implied that all groups struggled with programming to similar extents. This finding was further supported by

*This research was supported by the Loyola Schools Faculty Work Grant entitled, "A Usability Study of Four First Programming Languages" of the Ateneo de Manila University.

doi: http://dx.doi.org/10.2190/EC.49.3.b
http://baywood.com

the qualitative analysis, in which we found that no group was immune to a particular error type. All groups had difficulties finding small errors, including: differences in capitalization and missing commas; recognizing the scope and lifetime of a variable; confusing the assignment operator with the equal comparator; misusing the variants of the print statement; misusing or failing to use parameters; and implementing principles of object-oriented design such as information hiding and encapsulation. For computer science researchers and educators, the findings suggest learning pitfalls against which teachers can guard during instruction. The findings also imply the need for more reactive and anticipatory support environments, and the usefulness of large scale data collection and analysis in the study of novice programmers.

Programming is an essential skill that all students of computer science and related disciplines must learn. It is the primary means through which programmers turn abstractions into reality and, in doing so, engage in computation. To learn to program is to acquire a body of interrelated knowledge, skills, and practices. At the most basic level, a student has to learn to formulate a logical solution to a problem, express that logic in a formal computer language, and locate and remove syntactic or logical errors (bugs).

These skills are not easy to develop. A number of international studies have shown that many students' understanding of programming continues to be fragile even after one or two semesters of formal education (see Lister, Adams, Fitzgerald, Fone, Hamer, Lindholm, et al., 2004; McCracken, Almstrum, Diaz, Guzdial, Hagan, Kolikant, et al., 2001). Many students were unable to predict the outcome of a code fragment or complete a short piece of code so that it performs a specified function (Lister et al., 2004). Others had a hard time with problem abstraction, the first step in arriving at a solution to a programmatic problem (McCracken et al., 2001).

All learners experience these difficulties, and researchers have investigated whether they do so to the same extent. This study attempts to quantify and qualify the differences, if they exist, among low-achieving, average, and high-achieving students. We examine differences along four dimensions: Jadud's Error Quotient (EQ) (2006), Lee's confusion rate (2011), student self-reported perceptions of the ease or difficulty of introductory computer science topics, and a qualitative analysis of student code. Our analysis answers four questions:

1. How do the groups differ in terms of EQ?
2. How do the groups differ in terms of confusion rate?
3. How do the groups differ in their perception of the ease or difficulty in learning introductory computer science topics?
4. What errors or misconceptions is each group more prone to make?

PRIOR WORK

Since at least the 1980s and using a variety of methods, computer science educators and researchers have been trying to learn more and more about what programming students know and do not know. The study of Bonar and Soloway (1983) made use of videotaped interviews to understand how novices program. They found that a gap existed between students' "step-by-step natural language" and a programming language's syntax. Novices were aware of what they wanted to achieve but were unable to express this in a programming language.

Researchers have found examinations of paper-based outputs informative. For example, process journals (Lewandowski, 2003) provided researchers with qualitative information about students' problem-solving processes; for example, how they approached the program specifications, what bugs they encountered, what significant roadblocks they had to overcome. The journals also gave instructors the opportunity to empathize with student frustration or make helpful suggestions. The ways in which novices grouped programming concepts were revealing of what they knew and didn't know (Lewandowski, Gutschow, McCartney, Sanders, & Shinners-Kennedy, 2005). The most abstract concepts tended to be classified under "don't know" or "not sure." A study of paper-based object-oriented designs revealed that students sometimes define classes they never use, or else invoke classes they never define (Thomasson, Radcliffe, & Thomas, 2006).

Lister et al. (2004) examined the doodles (defined as any kind of marking made by the student; McCartney, Moström, Sanders, & Seppälä, 2005) that students made while tracing through paper-based code excerpts. They noted that effective novices made use of notes to supplement working memory, while less effective novices made use of working memory alone. These findings were corroborated by de Leon et al. (2008), who found that students who opted to doodle tended to be among the highest achievers in their classes, with scores of 91 to 100 out of a possible 100 in the midterm examination. On the other hand, 85% of the students who opted not to doodle had midterm scores of 90 out of 100 or lower.

Of particular interest to researchers is how students cope with errors. Since the 19th century, engineers in many fields have used the term "bug" to mean "a small error." In the 1960s, on the Univac 1107 at Case University, it was "possible to debug a moderate size program in less than an hour by gaining frequent access to the computer" (Lynch, 1967). Despite its long history, it is only recently that systematic exploration of syntactic errors made by programmers has been pursued vigorously.

In the seminal study by Perkins et al. (1986), researchers observed students learning to program in BASIC. They noted that students fell into one of three categories: Stoppers, Movers, and Extreme Movers. Movers referred to students who identify programming errors and systematically experiment with solutions until they gravitate to an appropriate solution. Stoppers are students who give up on the problem-solving process. Finally, Extreme Movers, also known as Tinkerers, experiment with solutions haphazardly and therefore struggle or fail to arrive at real solutions.

Recent research has shown that these categorizations are not absolute. Worsley and Blikstein's (2013) detailed study of student compilations showed that students use their code updates differently. Some update their code to check syntax while others do so to optimize their logic. In some cases, students alternated between making small and large changes, or tinkering and planning respectively.

What makes the examination of these details possible is the collection and analysis of online protocols (defined as the collection of all programs submitted to the compiler; Bonar, Khrlich, Soloway, & Rubin, 1982) to achieve an understanding of student problem-solving experiences. Researchers have documented the incidence of compiler errors (see Denny, Luxton-Reilly, & Tempero, 2012; Dy & Rodrigo, 2010; Jackson, Cobb, & Carver, 2005) and have found that sets of most common errors tended to be the same across different populations: missing semicolons, undefined identifiers, missing braces or brackets, and incompatible types. The implication on pedagogy was that support for these error messages in particular was needed.

Online protocols reveal how well students cope with these errors both behaviorally and emotionally. Ahmadzadeh et al. (2005) found that ineffective debuggers demonstrated knowledge of a debugging technique in one instance, but were unable to transfer that knowledge to another instance. The study of Jadud (2006; to be discussed in greater detail later) proposed EQ as a metric for quantifying student debugging ability. Lee (2011), on the other hand, examined online protocols to quantify student confusion (also to be discussed in greater detail later) and its effect on achievement (Lee, Rodrigo, Baker, Sugay, & Coronel, 2011).

Online protocols also imply what might be wrong with our computer languages and integrated development environments and how these might be improved. Despite the knowledge that small syntactic errors are common in the practice of programming, the primary tool we present to novices (a compiler) is one that strictly reacts to their errors, and provides little to no support for the learning of the practice of programming. Decker and Anderson in their 1986 paper *From Reactive to Proactive: A Continuum* present a continuum model (Figure 1) for thinking about educational programming that might serve as an initial lens for focusing future work on novice programmer behavior and, importantly, tools to support that work.

Many of the tools we give our students today are primarily reactive-defensive and reactive-responsive. In most novice programming environments, there is support for syntax highlighting and code indentation based on language syntax; we would claim that this kind of support is reactive-defensive, as it attempts to provide a learner with visual cues regarding the correctness of their program's

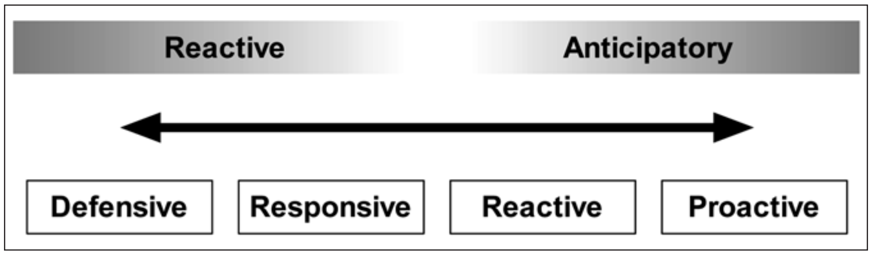


Figure 1. Decker and Anderson's (1986) continuum for thinking about educational programming.

syntax before they press the "compile" button. The compiler itself is reactive-responsive, as it is only capable of reporting errors when the student invokes it. To further compound novice programmer difficulties, some of the errors that compilers report are non-literal (Dy & Rodrigo, 2010; Hughes, Jadud, & Rodrigo, 2010), meaning they do not accurately reflect what was wrong with the code and do not guide the student toward a solution. Indeed, Traver (2010) recommended the redesign of compiler error messages so that they are clearer, more specific, and properly phrased.

A better understanding of student difficulties, errors, misconceptions, behaviors, affective states, and so on can lead to the development of anticipatory, proactive tools for teaching, learning, and assessment. Efforts in this direction date back to at least the late 1980s onwards. Syntax-directed editors such as those discussed by Goldenson (1989) and Miller et al. (1994) enforce the creation of a syntactically correct program by allowing the student to enter only syntactically correct code at any point of the editing process. During the editing process, all unfilled parts of the program are substituted with placeholders. Writing a program is thus a process of repeatedly selecting from a list of allowed fillers per placeholder, eliminating the possibility of syntax errors. In recent years, systems such as Scratch (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010) and Dann, Cooper, and Pausch (2008) follow the same philosophy. They use highly graphical, drag-and-drop or menu driven interfaces to restrict what programming constructs a student can insert at any point in a program.

In more full-featured development environments (for example, Eclipse, a common open source development environment for the Java programming language), a programmer's code is continuously recompiled and analyzed for a wide variety of errors, in much the same way that modern word processors are capable of spell checking a document continuously as it is being written. The group of Truong et al. (2004) developed a tool that could help analyze program structure without running the code to determine potential bugs, unnecessary complexity, or high maintenance areas.

A number of environments have been developed to assist students with debugging. In the early 1990s, Freund and Roberts (1996) developed THETIS, an integrated environment with improved error reporting and debugging visualization tools. More recently, the groups of Sison and Chen (2005) and Kummerfeld and Kay (2006) developed web-based reference systems to support students learning C and C++. Both systems cataloged compiler error messages and proposed at least one possible correction, based on a pre-defined bug library. Dy and Rodrigo (2010) developed a system that takes non-literal Java compiler error messages and tries to infer what might actually be wrong with the code. HelpMeOut by Hartman and colleagues (2010) is a social recommender system that supports debugging by suggesting solutions from the experiences of past programmers.

Capitalizing on student online protocols opens the door to the assessment of the programming process, not just its final outcome. The work of Lane & VanLehn (2005) and Vee et al. (2006) measured the distance between students' intermediate solutions and a known solution to a problem. Known as intention-based scoring, this approach enabled researchers to assess students' ability to produce algorithmically correct code.

Other types of support tools have also been built on top of online protocols. Retina (Murphy, Kaiser, Loveland, & Hasan, 2009) was a tool that collected online protocols in a database for post-hoc inspection. Retina had the ability to send recommendations to students when they had a high rate of errors, were spending too long on an assignment, or were making the same error multiple times. The BlueJ Browser (Rodrigo, Tabanao, Lahoz, & Jadud, 2009) had similar features. It allowed a post-hoc analysis of student submissions, generating reports such as most common errors, and time between compilations.

As mentioned earlier, the goal of this study is to quantify and qualify the differences among low-achieving, average, and high-achieving students. To do so, we make use of some of the techniques and approaches documented in prior work—EQ (Jadud, 2006), confusion rate (Lee, 2011), self-reports, and a qualitative analysis of student online protocols.

METHODS

The population under study was composed of Ateneo de Manila University Computer Science (CS) freshmen and Management Information Systems (MIS) sophomores taking CS21A Introduction to Computing, the introductory computer science course. CS21A teaches the students object-oriented programming using Java, following an objects-first approach. Every school year, the Ateneo has five to six sections of CS21A, taught by two to three faculty members. All lecture materials for these sections are uniform. The laboratory exercises, midterm exam, and final exam are departmental as well. Teachers are, however, free to give their own hands-on exams and quizzes.

The teachers introduce students to debugging strategies in several ways. The lecture components cite common errors, for example, type incompatibility or off-by-one errors. Students are taught how to use the debugging features of the IDE and are taught how to step through their code, while inspecting variable values. During labs, students can consult teachers about errors. Teachers do help them resolve syntactical errors in particular, on a one-on-one basis. Debugging is also assessed. In the midterm and final exams, students are presented with incorrect code and asked to explain what is wrong. Finally, at least one of the hands-on exams is a debugging exam; that is, students are given syntactically and logically erroneous code, and the expected output, and they are expected to correct the errors.

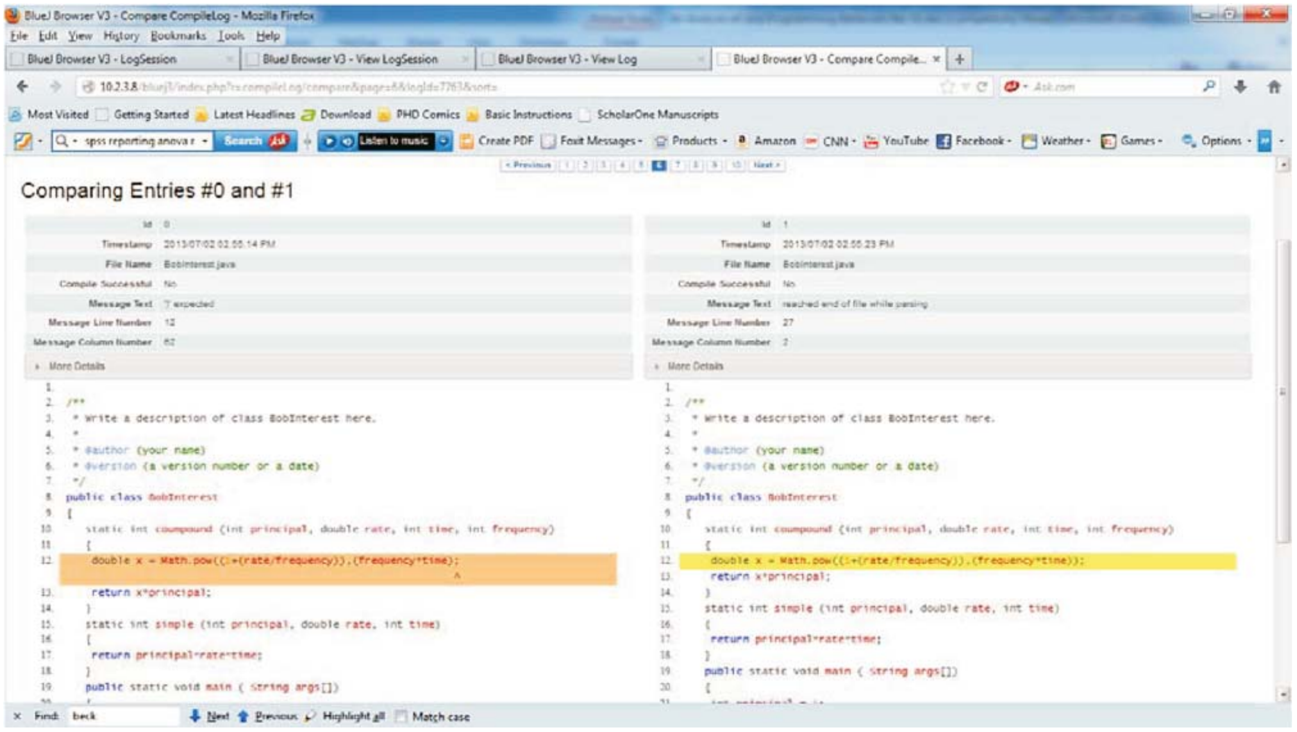
Since 2006, the Department of Information Systems and Computer Science has been using an instrumented version of the BlueJ Interactive Development Environment (IDE) for Java (Kolling & Rosenberg, 1996). This version stored student online protocols on a central server. The student compilation logs were stored in SQLite format and contained a student ID, copy of the students' program as compiled at that moment, a time stamp, any errors the compilation produced, and some other information.

To enable us to access the logs quickly and easily, and to generate reports of interest, we updated the BlueJ Browser discussed in Rodrigo et al. (2009). In the 2009 version, users could only perform data processing and visualization after the laboratory session ended. The current version of the Browser, version 3, can now collect and process data in real-time. Via the Browser, we can see the students' source code for each compilation they did, as well as view their source side-by-side to how their work progresses (Figure 2). For a higher-level view, the browser can also generate reports containing EQs, confusion rates, and error counts (Figure 3). These reports can be viewed at different grain sizes such as per student, per section, or per course.

For our analysis, we made use of five sources of data: student compilation logs, EQ (Jadud, 2006), confusion rate (Lee, 2011), student midterm grades, and an ease-of-learning student survey. The student compilations, EQs, and confusion rates were accessed using the BlueJ Browser. Each type of data will be discussed in turn.

Student Compilation Logs

The CS21A classes were conducted in classroom laboratories with a 1:1 student-to-computer ratio. During the classes, the students were engaged in hands-on work. Each class session required students to perform non-graded hands-on exercises. Every 2 weeks, students were given a graded lab in which


BlueJ Browser V3 - Compare CompileLog - Mozilla Firefox
File Edit View History Bookmarks Tools Help
BlueJ Browser V3 - LogSession
BlueJ Browser V3 - View LogSession
BlueJ Browser V3 - View Log
BlueJ Browser V3 - Compare Compile...
Most Visited
Getting Started
Latest Headlines
Download
PHD Comics
Basic Instructions
ScholarOne Manuscripts
Search
Listen to music
Create PDF
Foxit Messages
Products
Amazon
CNN
YouTube
Facebook
Weather
Games
Options
« Previous
Next »

Comparing Entries #0 and #1

Id	0
Timestamp	2013/07/02 02:55:14 PM
File Name	BobInterest.java
Compile Successful	No
Message Text	')' expected
Message Line Number	12
Message Column Number	62

▸ More Details

```
1.
2.  /**
3.   * Write a description of class BobInterest here.
4.   *
5.   * @author (your name)
6.   * @version (a version number or a date)
7.   */
8.  public class BobInterest
9.  {
10.    static int coumpound (int principal, double rate, int time, int frequency)
11.    {
12.     double x = Math.pow((1+(rate/frequency)),(frequency*time);
                                                              ^
13.     return x*principal;
14.    }
15.    static int simple (int principal, double rate, int time)
16.    {
17.     return principal*rate*time;
18.    }
19.    public static void main ( String args[])
```

Id	1
Timestamp	2013/07/02 02:55:23 PM
File Name	BobInterest.java
Compile Successful	No
Message Text	reached end of file while parsing
Message Line Number	27
Message Column Number	2

▸ More Details

```
1.
2.  /**
3.   * Write a description of class BobInterest here.
4.   *
5.   * @author (your name)
6.   * @version (a version number or a date)
7.   */
8.  public class BobInterest
9.  {
10.    static int coumpound (int principal, double rate, int time, int frequency)
11.    {
12.     double x = Math.pow((1+(rate/frequency)),(frequency*time));
13.     return x*principal;
14.    }
15.    static int simple (int principal, double rate, int time)
16.    {
17.     return principal*rate*time;
18.    }
19.    public static void main ( String args[])
20.    {
```

Figure 2. Side-by-side comparison of successive compilations using the BlueJ browser.

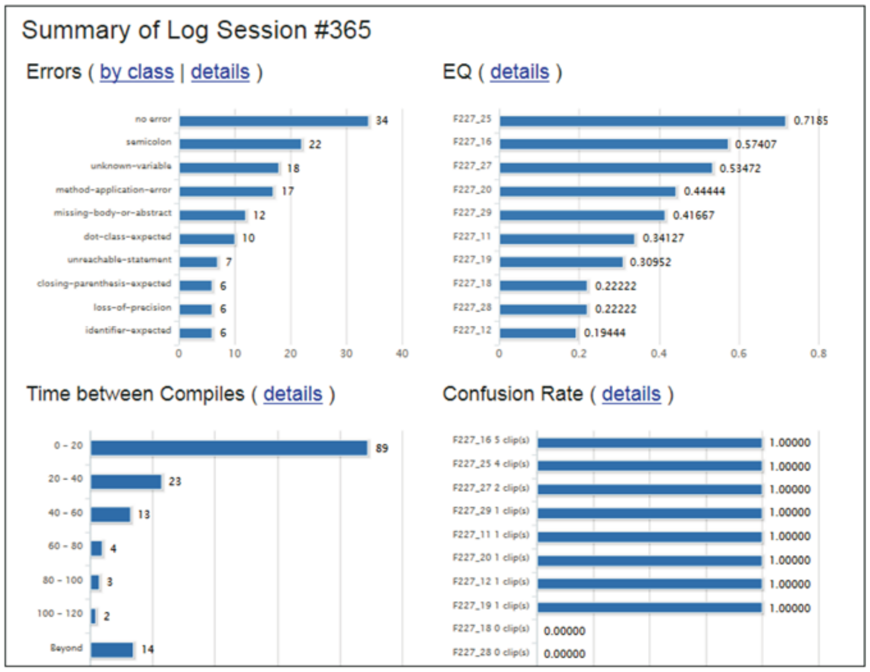


Figure 3. Reports generated by the BlueJ browser.

final programs had to be submitted to the course instructor. Twice a semester, the students were also given hands-on exams. The difference between the hands-on exams and the graded labs was that students were expected to work independently during the former. During the latter, the students were allowed to ask the teacher or fellow students for advice.

For the purposes of this analysis, we only examined the logs from school years 2009-2010, 2010-2011, and 2011-2012. Table 1 shows the breakdown of the student population for those three school years. Note that we were unable to capture 100% of student compilations. There were times when the server was off during a lab session, leading to lost opportunities for data collection.

Error Quotient

There were two quantitative metrics used to analyze the log files: EQ and confusion rate. Jadud's (2006) EQ was a quantification of a student's capacity to cope with syntax errors. The EQ of each student per laboratory exercise and over all laboratory exercises was computed based on the following algorithm:

Table 1. Student Population Breakdown per School Year

	School year		
	2009-2010	2010-2011	2011-2012
No. of CS21A sections	6	5	5
No. of CS majors	61	65	54
No. of MIS majors	92	12	90
Number of log files	52403	21848	116049

Given a session of compilation events e_1 through e_n,

1. Collate: Create consecutive pairs from the events in the session, for example, $(e_1,e_2),(e_2,e_3)...(e_{n-1},e_n)$.
2. Calculate: Score each pair according to the algorithm presented in Figure 4.
3. Normalize: Divide the score assigned to each pair by 9 (the maximum value possible for each pair).
4. Average: Sum the scores and divide by the number of pairs. This average is taken as the EQ for the session.

An EQ score ranges from 0 to 1.0, where 0 means that the student alternated between successful and unsuccessful compilations. A student's EQ is a proxy indicator of how efficiently he or she finds and resolves syntax errors. An EQ of 1.0 means that the student encountered the same error on the same line all the time. Succeeding studies showed that EQ negatively correlated with midterm exam score (Rodrigo et al., 2009). A student with a poor average EQ was likely to do poorly on the midterm exam.

Confusion Rate

Lee's (2011) confusion rate was a machine-learned model of student confusion. The algorithm first segregated student compilations into subsets or clips of eight transactions each. The level of confusion within a clip was a function of the number of compilations with errors, the maximum time between compilations with errors, the average time between compilations, and the number of pairs of compilations with the same error. The model returned a 1 if it judged the clip to exhibit confusion. It returned a 0 otherwise. These decisions were totaled and averaged across all clips for that lab session to determine the student's confusion rate. The confusion rate therefore ranged from 0 to 1.0 where 0 meant that none of the student's clips exhibited confusion. A confusion rate of 1.0 means that all clips were judged to exhibit confusion. A subsequent study by Lee et al. (2011) showed that when students' confusion is successfully resolved, it is positively

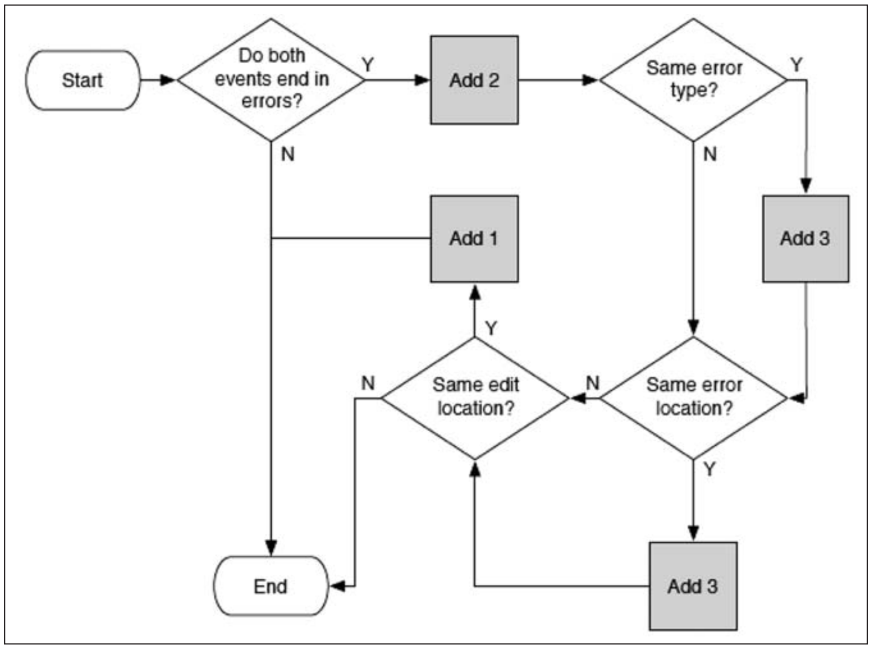


Figure 4. Flow chart showing the scoring method.

correlated with their midterm scores. Persistent confusion, on the other hand, was negatively correlated with the midterm scores. These findings supported D'Mello and Graesser's (2012) model of cognitive disequilibrium in deep learning environments. The model asserted that confusion was a useful affective state if the student purposefully attempts to resolve cognitive conflict. If students were unsuccessful, though, the result was likely to be frustration, boredom, and disengagement with the learning task.

Student Midterm Grades

In the preceding text, we mentioned several times that we correlated our measures with midterm scores. We used midterm scores instead of final grade or final exam scores because CS21A only used BlueJ up to the midterm. After the midterm, the students switch IDEs to JCreator. We therefore do not have log files for programming problems and exercises after the midterm exam.

Ease-of-Learning Student Survey

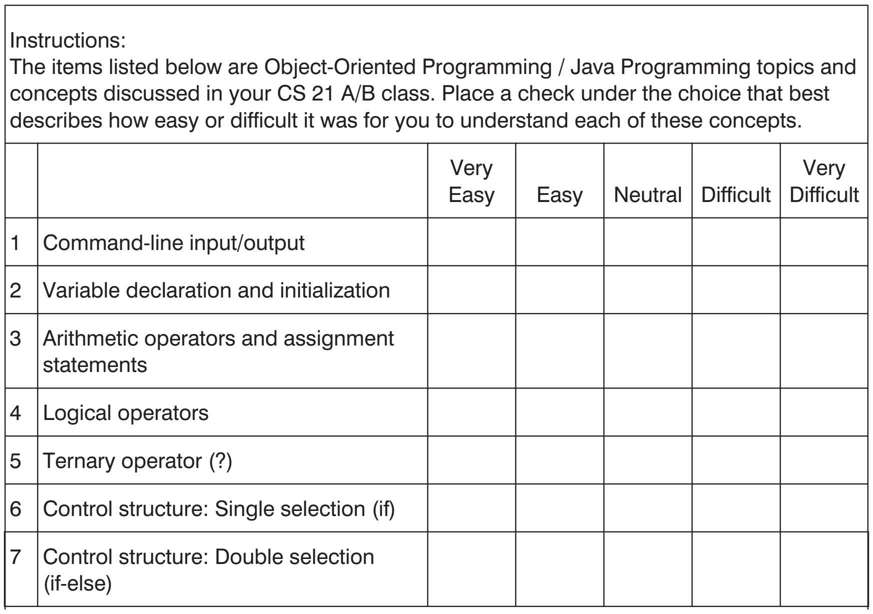
In 2001 (Joint Task Force on Computing Curricula, IEEE Computer Society, & ACM, 2001) and 2008 (Interim Review Task Force, Association for Computing Machinery, & IEEE Computing Society, 2008), the Association of Computing

Machinery (ACM) and IEEE formed a task force to review the computer science curriculum. One of the outputs of this task force was an articulation of the fundamental constructs that students must learn from an introductory computer science course. These include fundamental constructs of programming, algorithmic problem solving, data structures, recursion, event-driven programming, and object-oriented programming. These concepts are typically taught over two to three semesters.

We created a survey form that listed the general topics that they were expected to learn from the introductory course, as identified by the ACM and IEEE. We then asked all currently-enrolled Computer Science sophomores, juniors, and seniors and Management Information Systems juniors and seniors to indicate whether they found a topic easy or difficult to learn (see Figure 5). Out of 294 students in 3 cohorts, 199 responded, for a response rate of 68%.

Final Data Set

We limited our analysis to students with complete data sets, i.e., survey respondents with test scores, and whose logs we could definitively match. Of the 294 respondents, only 147 students' data was complete.



Instructions:
The items listed below are Object-Oriented Programming / Java Programming topics and concepts discussed in your CS 21 A/B class. Place a check under the choice that best describes how easy or difficult it was for you to understand each of these concepts.

		Very Easy	Easy	Neutral	Difficult	Very Difficult
1	Command-line input/output					
2	Variable declaration and initialization					
3	Arithmetic operators and assignment statements					
4	Logical operators					
5	Ternary operator (?)					
6	Control structure: Single selection (if)					
7	Control structure: Double selection (if-else)					

Figure 5. An excerpt from the Ease-of-learning Student Survey.

ANALYSIS

Homogeneity of the Cohorts

We first attempted to determine whether each cohort varied significantly in terms of midterm scores and ease of use. We divided the data by cohort (all seniors, all juniors, all sophomores). A single factor analysis of variance (ANOVA) showed that there was no significant difference among the three groups ($F(2, 144) = .535$; $p = .587$).

To determine whether the cohorts differed in terms of their perception of the ease or difficulty of CS21A, we converted the ease-of-learning responses to numerical scores (e.g., Very Easy = 1, Easy = 2, and so on), and computed the average ease-of-learning score per student. We once again used a single factor ANOVA to compare the three cohorts and found among them ($F(2, 144) = 1.336$, $p = .266$). This meant that the cohorts' perception of the level of difficulty of CS21A was uniform.

Differences among Terciles

We then divided the same data set into terciles based on the midterm score. From Table 2, we see that the "high" tercile, is composed of 50 students with the highest midterm score while the "low" tercile has the 48 students with the lowest midterm scores.

When we compared the midterm scores of these three groups using a single-factor ANOVA, the differences between groups was significant ($F(2, 144) = 276.866$, $p < .01$). A Tukey HSD post-hoc analysis confirmed that each tercile varied significantly from the other. To determine whether the groups differed in the way they perceived the relative ease or difficulty of CS21A, we again used a Single-Factor ANOVA to compare their ease-of-learning scores. We found that the groups did vary significantly ($F(2, 144) = 12.735$, $p < .01$). A Tukey HSD post-hoc analysis showed that high-achieving and average students' perception of CS21A's ease-of-learning was not significantly different. Both groups' perception differed significantly, however, from the perception of the low-achieving

Table 2. Profile of Students in the Three Terciles

		Midterm scores				Ease-of-learning scores			
Tercile	*N*	Mean	Min	Max	*SD*	Mean	Min	Max	*SD*
High	50	82.48	73.50	99.00	6.92	2.28	1.14	4.00	6.92
Average	49	66.78	62.00	73.00	3.29	2.48	1.20	3.63	0.69
Low	48	53.16	30.00	61.00	7.48	2.94	1.33	4.63	0.67

students. Low-achieving students reported having a harder time with the subject matter than the students in the other two groups.

We selected the middle one-third of the students, or 17 students per tercile, for closer examination. We collected the EQs and confusion rates of each lab session of each of these students. Since students had a variable number of labs, we averaged the EQ and confusion scores. Table 3 shows that the high performers typically had lower EQs than the low performers and that the high performers' confusion rates were lower than the other two groups. When we compared these scores using a Single-factor ANOVA, though, the scores were not significant, whether for EQ ($F(2, 47) = 1.574$, $p = .218$) nor confusion ($F(2, 46) = 1.212$, $p = .307$).

This implies that the degree to which the groups struggled with the subject matter was similar.

Problematic Topics

Upon examination of the survey results, we determined which topics the students from the sample rated as "Hard" or "Very Hard," and found that a majority of the students had a hard time understanding event-driven programming, especially threads ($n = 30$), exception ($n = 15$), and event handling ($n = 15$). Another area that students found difficult involved concepts related to object-oriented programming. These concepts include polymorphism ($n = 22$), inheritance ($n = 20$), and encapsulation ($n = 15$). Finally, the survey revealed that some students had a hard time learning how to declare and manipulate multidimensional arrays ($n = 21$), which fell under the general topic of data structures.

Unfortunately, we were only able to view logs which spanned only the first half of the semester. This meant that only encapsulation and multidimensional arrays will fall under the scope of our analysis. However, part of our analysis was to confirm whether or not those topics which students found to be easy or average in difficulty were indeed easy or average for all groups, or were only so for one or two of the groups.

Table 3. Average EQ and Confusion Scores of the Three Groups

	EQ score		Confusion score	
Groups	Mean	Variance	Mean	Variance
Low	.31	.15	.60	.30
Average	.22	.21	.50	.34
High	.23	.14	.47	.27

Qualitative Evaluation of the Data

The student compilation logs that we inspected for this part of the analysis only spanned the first half of the semester. This means that we had to limit our analysis to topics covered in that time. Table 4 maps the ACM/IEEE recommended topics against those topics covered in the logs we were able to examine. In the course of the inspection, we found that the representatives of the various groups tended to make similar mistakes.

Basic Syntax and Semantics of a Higher-Level Language

Representatives from all groups missed semi-colons, misspelled their variables, or forgot opening or closing curly braces. There were instances when the students forgot that Java is a case-sensitive language: they spelled variable Key**B**oard as Key**b**oard or get**B**alance as get**b**alance (differences bolded). For some students, these small errors can be difficult to find. One student attempted to initialize an array of strings with literals, missing a '`,`' between literals. She received 10 consecutive "`'}' expected`" errors.

The code snippet in Figure 6 of the student's first compilation produces the error mentioned above, which again was caused by a missing "`,`" between the strings "`i`" and "`j`." Throughout the next nine compilations, the student tried various permutations of modifying the array. One of these was to put all the elements of the array into a single line rather than splitting them into two. Another was removing the strings "`SPACE`" and "`ENTER`" as elements of the array. Eventually she correctly located the source of the error, and fixed it accordingly.

The scope and lifetime of a variable was a concept that was not easily understood. Students attempted to access variables local to one method from another method, or variables declared inside a loop outside of the loop, or variables local to one block of a control structure being referenced by another block (e.g., `if-else`). In Figure 7, we see an example of the third instance, where the variable is declared and initialized within the `if`-block, which makes it inaccessible in its corresponding `else`-block.

```
String [] labels = {"a", "b", "c", "d", "e", "f",
  "g", "h", "i" "j", "k", "l", "m", "n", "o",
  "p", "q", "r", "s", "t", "u", "v", "w",
  "x", "y", "z", "SPACE", "ENTER"};
```

Figure 6. Code snippet showing a missing comma between "`i`" and "`j`". **Bold** added for emphasis.

Table 4. A List of the ACM/IEEE Topics against Topics Covered in the Log Files

ACM/IEEE topics	Covered
Fundamental constructs of programming	
Basic syntax and semantics of a higher-level language	X
Variable types, expressions, and assignment	X
Simple input and output	X
Conditional and imperative control structures	X
Functions and parameter passing	X
Structural decomposition	X
Algorithmic problem solving	
Problem-solving strategies	X
Roles of algorithms in problem solving	X
Implementation strategies for algorithms	X
Debugging strategies	X
Data structures	
Representation of numeric data	X
Range, precision, and rounding errors	X
Arrays	X
Representation of character data	X
Strings and string processing	X
Runtime storage management	
Pointers and references	
Linked structures	
Implementation strategies for stacks, queues, and hash tables	
Implementation strategies for graphs and trees	
Strategies for choosing the right data structure	
Recursion	
Event-driven programming	
Event handling	
Event propagation	
Exception handling	
Object-oriented programming	
OOP design	X
Encapsulation and information hiding	X
Separation of behavior and implementation	
Classes and subclasses	X
Inheritance	
Polymorphism	

```
public double sellSeats( int regSeats,
 int discountSeats, int premiumSeats )
{
 if ( availSeats < regSeats +
  discountSeats + premiumSeats )
 {
   int flightCost = 0;
 }
 else
 {
   flightCost = ( regSeats *
     regPrice ) + ( discountSeats * ( 0.9 * regPrice ) )
     + (premiumSeats * ( 1.1 * regPrice ) );
     . . .
 }
 totalSales = totalSales + flightCost;
 return totalSales;
}
```

Figure 7. Code snippet showing the variable `flightCost` is out of scope. **Bold** added for emphasis.

In this example, however, the student was able to fix the error quickly, declaring `flightCost` outside of the `if`-block and initializing it within the `if`- and `else`-blocks.

Variable Types, Expressions, and Assignment

The syntax of a declaration and assignment take time to learn. One student intended to declare the variable money as a double using the syntax:

```
money = double money;
```

Students are sometimes unaware that their operations change the values of the variables. In Figure 8, the student uses `j++` when what she really means is `j+1`.

We see that the intent of the student was to compare a cell of a two-dimensional array with an adjacent cell. The student however, mistakenly increments the variable `j` by using `j++` as its array index. Although the code was syntactically correct, it was logically wrong and caused an array index out of bounds error during run time.

Simple Input and Output

It was difficult for students to discriminate between the use of `print`, `println`, and `printf`. We found an instance in which a student used a

```
for ( i = 0; i < 9; i ++ )
{
 for ( j = 0; j < 9; j ++ )
 {
  if ( sudoku[i][j] != sudoku[i][j++] )
  {
   validRows = true;
  }
  else
   validRows = false;
 }
}
```

Figure 8. Code snippet showing a use of `j++` instead of `J+1`.
Bold added for emphasis.

`println`, but with the format specifiers and argument vectors of the `printf`. The student was confused because the error message he received was "`cannot find symbol, method println( java.lang.String, double)`."

Another instance was the case in which the student used `printf` with string concatenations, forcing it to function like a `print` (see Figure 9).

Although the compiler found no error for the code was syntactically correct, it was a semantically inappropriate use of the construct. This misleads the student, making him think that his current mental model of Java syntax is correct because there is no appropriate feedback being given that says otherwise.

Some students solved their compilation errors through experimentation (Perkins et al., 1986). Figure 10 shows a student's attempts at finding the correct placement for the '`\n`'. He first tries several permutations of '`\n`' ('`\n`' in Figure 10a, '`/n`' in Figure 10b, and '`n\`' in Figure 10c) outside of the quotation marks before finally arriving at the correct syntax, which was to place '`\n`' within the quotes.

Conditional and Imperative Control Structures

Students are sometimes confused between the use of '=' and '=='. We found at least two instances in the average tercile in which students used '=' as a comparator (see Figure 11).

In this code excerpt, it is clear that the student's intention in the code line

```
if (ePass = true)
```

```
    System.out.printf("| Gate 1 | Php " + "%2.2f" + " | "
          + gate1.getSJT() + " | " + gate1.getSVT() + " | "
 + gate1.getBonuses() + " |\n", gate1.getCollections());
```

Figure 9. Code snippet showing a use of `printf` like as `print`.
Bold added for emphasis.

```
System.out.printf(\n"TOLL LANE 2: Php %.2f, %d cars, %d e-pass users,
        %d violators.", lane2.getCash(), lane2.getCars(),
        lane2.getEpassUsers(), lane2.getViolators());
```

Figure 10a. Code snippet showing a use of '`\n`' outside the quotes.
Bold added for emphasis.

```
System.out.printf(/n"TOLL LANE 2: Php %.2f, %d cars, %d e-pass users,
        %d violators.", lane2.getCash(), lane2.getCars(),
        lane2.getEpassUsers(), lane2.getViolators());
```

Figure 10b. Code snippet showing a use of '`/n`' outside the quotes.
Bold added for emphasis.

```
System.out.printf(n\"TOLL LANE 2: Php %.2f, %d cars, %d e-pass users,
        %d violators.", lane2.getCash(), lane2.getCars(),
        lane2.getEpassUsers(), lane2.getViolators());
```

Figure 10c. Code snippet showing a use of '`n\`' outside the quotes.
Bold added for emphasis.

```
public void letCarPass ( boolean ePass, double
  ePassBalance, double tollFee )
{
  if ( ePass = true )
  {
     if ( ePassBalance >= tollFee )
        ePassBalance = ePassBalance - tollFee;
     return ePassBalance;

     else ( ePassBalance < tollFee )
       ePassBalance = ePassBalance –
          epassBalance;
  }
 ...
}
```

Figure 11. Code snippet showing a use of '=' as a comparator instead of '=='. **Bold** added for emphasis.

was to check if the value of the `boolean` value `ePass` is equal to `true`. Note, however, how the student did not seem to notice the difference in his use of the '=' operator in the abovementioned line and, for instance, the line

```
ePassBalance = ePassBalance - tollFee;
```

where the function of the operator was for assignment and not comparison. By using '=' instead of '= =', the student unknowingly performs an assignment function rather than a comparison, consequently making the `if`-statement always `true`.

Program control flow was difficult for some students to grasp. Some students put statements after a `return` (Figure 12), resulting to an '`unreachable statement`' error thrown by the compiler. This particular error lasted 19 compilations and took about 5 minutes to resolve.

Functions and Parameter Passing

It takes students time to learn to invoke methods and use method parameters correctly. They forget to use the dot operator or forget to use a '( )' after the name of a method with no parameters. When a method does have parameters, they often invoke methods with the wrong parameter list (Figure 13).

The following code examples, the snippets from `RiceTrader` and `RiceTraderTester` show a mismatch between the parameter list when the method `buyOneSack` was invoked by the objects `fernando` and `julio` and the actual number and kind of parameters required in the method definition.

```
if (donutTotal > 0)
{
  donutTotal -= totalDonutsOrdered;
  donutSales += (donutCost *
    totalDonutsOrdered);
  donutSold += totalDonutsOrdered;
  return donutSales;

  if (oneFreeDonut) || (loyaltyCard == 12)
  {
    donutFree += 1;
    donutSales -= totalDonutsOrdered;
   }
}
```

Figure 12. Code snippet showing a misplacement of the ‘`return`’ statement. **Bold** added for emphasis.

```
//in class RiceTrader:
public void buyOneSack(double pricePerSack)
{
   CashOnHand = CashOnHand - (pricePerSack
      * amount);
   RiceLeft = RiceLeft + amount;
}

//in class RiceTraderTester:
public static void main( String args[] )
{
   RiceTrader fernando = new
     RiceTrader();
   RiceTrader anita = new RiceTrader();
   RiceTrader julio = new RiceTrader();
   fernando.buyOneSack( 1, 598 );
   anita.sellRice ( 1, 739.69 );
   julio.buyOneSack( 1, 624.43 );
   fernando.sellRice ( 7, 28.38 );
}
//in class RiceTester:
System.out.println(“Fernando has sold” +
  getKilosSold() + “ kilos of rice.”);
```

Figure 13. Code snippet where the student invoked a method with the wrong parameter list. **Bold** added for emphasis.

The code snippet from `RiceTester`, on the other hand, shows the student forgetting to use the '.' operator when invoking the method `getKilosSold( )`.

Students also make the mistake of assigning values to parameters instead of using the parameters to initialize field variables, as seen in Figure 14.

OOP Design

Students resist the idea of using a constructor to initialize their variables. Instead of creating a constructor, they opt to initialize fields directly (Figure 15). In the process, they also leave some fields uninitialized.

```
public Segment(double px1, double py1, double
px2, double py2)
  {
     px1=1;
     px2=1;
     py1=1;
     py2=1;
  }
```

Figure 14. Code snippet showing wrong initializations.
Bold added for emphasis.

```
public class SodaTrader
{
double cashOnHand = 500;
int cansSold;
int cansLeft;

...
public void printReportC()
{
  System.out.println("Crispin has sold " +
  cansSold + " cans. He has Php" +   cashOnHand + " and "
  + cansLeft + " cans left.");
  System.out.println("END OF REPORT");
}
}
```

Figure 15. Code snippet showing direct initialization of fields.
Bold added for emphasis.

From the low-achieving tercile, there was an example in which the student did not understand how to declare and instantiate an object. In Figure 16, what the student intends to do is declare and instantiate a variable page to be of type `Page`. Instead, she declares `page` as an `int` and attempts to perform operations on it that can only be performed with an object of type `Page`.

Encapsulation and Information Hiding

Access modifiers and the use of accessor methods are a challenge for students to understand. We have seen code in which students skirt around the issue by declaring all fields as `public` and not using accessor methods at all (Figure 17).

In this code fragment, the student, except for the constants, erroneously declared all of the class' attributes as `public`. By doing this and not using accessor methods, the student fails to learn the concept of encapsulation. Apart from this, the student does not realize that other objects that reference this class' attributes are open to change these values unknowingly, and often, accidentally.

```
public double sellSpace( int page, double size )
{
  // invokes the sellSpace method on the page
  // object indicated in the first
  // parameter, page
  Page needed;
  if (page.equals(“page1"))
      needed = page1;
    else if (page.equals(“page2"))
    needed = page2;
   else if (page.equals(“page3"))
      needed = page3;
    else if (page.equals(“page4"))
      needed = page4;
  else
   {
     System.out.println(“I don’t have that
         product");
     return 0;
 }
}
```

Figure 16. Code snippet showing wrong type declaration for `Page`.
Bold added for emphasis.

```
public class Flight
{
    public double rglrPrice, fuelPrice, seatCost,
         fuelCost, money;
    public int seatsAvailable, rglrSeats,
         premSeats, discSeats, totalSeats;
    ...
}
```

Figure 17. Code snippet showing a refusal to use access modifiers. **Bold** added for emphasis.

Others

In addition to the observations we noted using the topics specified in the ACM/IEEE curriculum, we also had additional observations about how students approach their programming tasks. For instance, in one of the high-performing student's logs, we found the use of helpful comments, as shown in Figure 18.

Through the use of these comments, the student was able to articulate how each of the code blocks should work, thus reducing the probability of errors in logic.

On the other hand, some of the students in the low-performing tercile included useless text in their code. We saw at least one instance of a random keyboard bang. There was another instance when a student inserted the text "`ewan something`" (*ewan* is an informal Filipino phrase for "I don't know") in her code.

DISCUSSION AND CONCLUSIONS

There are three ways in which we wish to consider the results of this and previous studies regarding the behavior of novice programmers. First, there are the direct results and implications of the work: our "take away" regarding novice programmers, their behaviors, and the results of their efforts. Second, we consider the ways in which data regarding programmer behavior and affect can be used by students and educators alike to shape our practices as learners and educators. Finally, we consider the potential impact that large scale data collection and analysis might have on educational research and practice.

```
if (ePass)
// if car is using an epass..
{
  if (ePassBalance < tollFee)
  // if epassbalance is less than the toll fee,
   // add car to the violators
  {
    ...
  }

  else
  // let car pass through, then subtract the
   // toll fee from the e-pass balance. Which
  // is pointless
  {
    ...
  }
}

else
// if car is not using epass...
{
   ...
}
```

Figure 18. Code snippet showing a helpful use of comments.

Direct Results Regarding Student Behavior, Affect, Perceptions, and Errors

We summarize the findings of this study in terms of the research questions:

How do the groups differ in terms of EQ? EQ does not differ significantly between groups. However, high-achieving students, on average, have lower EQs than low-performing students.

How do the groups differ in terms of confusion rate? High-achieving students are generally less confused than the low-achieving and average students. However, once again, this difference was not statistically significant.

As EQ is a quantification of students' ability to cope with syntax errors, the findings imply that all groups struggled to a similar degree. However, the average EQ scores in Table 3 show that high-achieving and average students generally resolve syntax errors more effectively than low-performing students. Previous studies support this finding: students with low error quotients spend less time fixing syntax errors, often doing so correctly on the "first try," whereas peers with higher error quotients and confusion rates often confound the error with additional layers of syntactic error, failing to correct the original problem (see Jadud, 2006; Lee et al., 2011). We would argue that this allows the higher performing students to spend more time on task, interacting with and testing the code they have written, as opposed to "thrashing" on syntactic issues. We feel this is reflected in the students' self-reporting regarding their perception of difficulty in CS21A: high-performing students reported having an easy time learning the course material, while students in the low-performing tercile reported having a more difficult time.

How do groups differ in their perception of the ease or difficulty in learning introductory computer science topics? We found that high-achieving and average students' perception of the difficulty of the subject matter was not statistically significant. Low-achieving students, on the other hand, reported having a statistically significantly harder time with the subject matter than the two other groups.

What errors or misconceptions is each group more prone to make? All three groups tended to make the same types of errors. They had difficulty locating typographical errors. The concepts of variable scope and lifetime were difficult to internalize. The subtle differences between the equality comparator and the assignment operation were easy to overlook. The different forms of print statements were often interchanged. Object-oriented constructs such as the use of constructors, the notion that classes are data types, and encapsulation were difficult to grasp.

For computer science researchers and educators, the findings have implications on instruction. The common errors describe concepts that students have difficulty understanding and should therefore warrant more explanation during instruction. Instructors also have to have an eye out for low-achieving students. Their perception that introductory programming is difficult could lead to discouragement or frustration.

One approach to aiding students in learning to deal with syntactic errors could be to focus more on the process they are learning, helping students to become more self-aware and reflective in their work (as encouraged by Lewandowski's process journals). However, the frustration and confusion that students experience when programming are emotions and confusion *in the moment*. To this end, pedagogic approaches that directly support students in meaningful ways while they are engaged in programming (and, therefore, learning) are, we believe, critical.

At the 2013 Symposium for Computer Science Education, Bailey-Lee, Parris, Porter, Spacco, and Simon presented a series of papers surrounding the application of peer instruction and pair programming to the introductory computing curriculum at the University of California at San Diego. They found that peer instruction halved their failure rates (Porter, Lee, & Simon, 2013), improved their retention rate by as much as 30% (Porter & Simon, 2013), and that instructional method (peer instruction vs. traditional lecture) had a significant impact on traditional outcome measures (Simon, Parris, & Spacco, 2013). While it is dangerous to infer too much from a single set of studies, there is growing evidence across many disciplines that pedagogic approaches that increase student-student interactions have significant, positive impacts on student learning.

Reactive and Anticipatory Support Environments

In the section on prior work, we discussed the development of anticipatory, reactive tools that support novice programmers. Some of these systems are currently being used in professional development environments. From their website, the Stanford HCI group describes "opportunistic programming" as "a method of software development that emphasizes speed and ease of development over code robustness and maintainability" (Standford HCI Group, n.d.). HelpMeOut, mentioned in the prior work, is an embedded context-aware search tool in the software development environment, making it easy for programmers to search the WWW for examples relevant to the rapid development of programs in Adobe Flex (Hartmann et al., 2010). This tool carries the trade name *Blueprint*, and is implemented in Adobe Flash Builder 4.

We suggest that syntax-directed editors, perhaps best typified today by systems like Scratch, might be considered *anticipatory-proactive*. Scratch makes it difficult, if not impossible, for a young programmer to write a program with a syntax error, because of its syntax-directed nature. It might also be argued that this is simply a restriction, but given that the compiler will only accept syntactically correct programs, we find the rhetorical question hard to resist: why do we even allow novices to construct syntactically incorrect programs in the first place, given the negative impact we feel it has on student learning? Systems like Mentor have been around since the 1970s, and even today, syntax directed systems come and go for a variety of languages and novice programming environments (Lang, 1986; PLT, n.d.).

Consistently, syntax-directed systems are found to be good for novice programmers, yet we as educators continue to educate our students with tools that exist in the strictly *reactive-defensive* and *reactive-responsive* paradigm. From personal conversation with the authors of novice programming environments, we know that these decisions are rarely made with data: instead, deeply held beliefs about how students should learn to program, or how the environment's author learned to program are strong design and development motivators, not fundamental research regarding the usability of their tools or the learning of the students who use those environments.

Impact of Large Scale Educational Data Collection and Analysis

Our last discussion point is the use of large scale data collection and analysis to inform educational theory and practice. Educational data mining is a growing field of study concerned with “the use of large-scale educational data sets to better understand learning and to provide information about the learning process” (Romero, Ventura, Pechenizkiy, & Baker, 2011). Educational data mining research generally falls in three broad categories: the development of tools and techniques that are suited for working with very large data sets, the analysis of the data to answer questions about learning, and the presentation of meaningful findings to educational stakeholders.

The tools and methods used in this article are exemplars of educational data mining in novice programmer research. The gathering of online protocols produces a fine-grained record of student programming activity. These data lend themselves to the computation of EQ or confusion rate—metrics developed specifically for the quantification of novice programmer behavior and affect. The BlueJ Browser automatically computes these and other metrics, and presents the information to faculty and researchers. As a result, we know more about what students do and how they feel as they program, and we gain evidence-based clues as to how to help them.

Using this information to help the learner is, we believe, the next big challenge in the space of novice programming environments. In the simplest case, visualizing this data in an easy-to-digest form could allow instructors to quickly engage in an automated, formative assessment of their students. This would allow instructors and teaching assistants to provide additional learning support to students in a timely, supportive manner, as opposed to waiting for evaluation points (like examinations) to provide guidance.

In online learning contexts, we can imagine that data and analyses of the sort described here could be used to automatically engage students in timely questions, tutoring, or contextual error support that goes beyond what the compiler alone is capable of providing. Initial work in this direction has been explored previously, but to the best of our knowledge never applied on a large scale (Marceau, Fisler, & Krishnamurthi, 2011).

In the context of Decker and Anderson’s scale from that which is reactive to anticipatory, we believe it will take educators and developers working together to actively transform the way we write programs if we want to provide (technological) *anticipatory-proactive* support for novice programmers. Some developers and researchers are already thinking this way: Bret Victor, a former Apple design software engineer, has demonstrated some very engaging active manipulation tools for the Processing programming environment (Victor, 2012), and Hundhausen et al. have demonstrated through sound research that direct manipulation interfaces are effective in supporting novice programmers in learning to program (Hundhausen, Brown, Farley, & Skarpas, 2006).

While it is true that we are interested in finding ways to support more students, in more places, through the use of data and technology, we currently have no data regarding novice programmer behavior from educators employing pair programming in the classrooms (McDowell, Werner, Bullock, & Fernald, 2006;

Williams & Kessler, 2000). Given the amount of research regarding the benefits of pairwise collaboration of students in introductory computing contexts, we believe that a fine-grained analysis of pair programmer behavior could be extremely illuminating when it comes to developing new tools to support students in a proactive manner.

ACKNOWLEDGMENTS

The authors thank our colleagues Jessica Sugay and Walfrido David Diy for their helpful comments and insights, and the CS21A faculty and their students who participated in the study.

REFERENCES

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *Proceedings of the 10th annual SIGCSE conference on Innovation and Technology in Computer Science Education – ITiCSE '05* (pp. 84-88). New York, NY: ACM Press. doi: 10.1145/1067445.1067472

Bonar, J., Khrlich, K., Soloway, E., & Rubin, E. (1982). Collecting and analyzing on-line protocols from novice programmers. *Behavior Research Methods & Instrumentation, 14*(2), 203-209.

Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming. *Proceedings of the 10th ACM SIGACT-SIGPLAN Syposium on Principles of Programming Languages – POPL '83* (pp. 10-13). New York, NY: ACM Press. doi: 10.1145/567067.567069

Dann, W., Cooper, S., & Pausch, R. (2008). *Learning to program with Alice* (2nd. ed.). Upper Saddle River, NJ: Pearson Education Inc.

Decker, D., & Anderson, C. (1989). From reactive to proactive: A continuum. *Journal of Extension, 27*(3). Retrieved from http://www.joe.org/joe/1989fall/f1.php

de Leon, M., Espejo-Lahoz, M. B., & Rodrigo, M. M. (2008). An analysis of novice programmer doodles and student achievement in an introductory programming class. *Philippine Information Technology Journal, 1*(2), 17-21.

Denny, P., Luxton-Reilly, A., & Tempero, E. (2012). All syntax errors are not equal. *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education – ITiCSE '12* (p. 75). New York, NY: ACM Press. doi: 10.1145/2325296.2325318

D'Mello, S., & Graesser, A. (2012). Dynamics of affective states during complex learning. *Learning and Instruction, 22*(2), 145-157. Elsevier Ltd. doi: 10.1016/j.learninstruc.2011.10.001

Dy, T., & Rodrigo, M. M. (2010). A detector for non-literal Java errors. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research – Koli Calling '10* (pp. 118-122). New York, NY: ACM Press. doi: 10.1145/1930464.1930485

Freund, S. N., & Roberts, E. S. (1996, February). Thetis: An ANSI C programming environment designed for introductory use. *Proceedings of the trail verison of the twenty-seventh SIGCSE technical symposium on computer science education* (pp. 300-304), February 15-17, 1996, Philadelphia, Pennsylvania.

Goldenson, D. R. (1989). The impact of structure editing on introductory computer science education: The results so far. *SIGCSE Bulletin, 21*(3), 26-29.

Hartmann, B., Macdougall, D., Brandt, J., & Klemmer, S. R. (2010). What would other programmers do: Suggesting solutions to error messages. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems – CHI '10* (pp. 1019-1028). New York, NY: ACM Press. doi: 10.1145/1753326.1753478

Hughes, M. C., Jadud, M. C., & Rodrigo, M. M. T. (2010). String formatting considered harmful for novice programmers. *Computer Science Education, 20*(3), 201-228. doi: 10.1080/08993408.2010.507335

Hundhausen, C. D., Brown, J. L., Farley, S., & Skarpas, D. (2006). A methodology for analyzing the temporal evolution of novice programs based on semantic components. *Proceedings of the 2006 International Workshop on Computing Education Research – ICER '06* (p. 59). New York, NY: ACM Press. doi: 10.1145/1151588.1151599

Interim Review Task Force, Association for Computing Machinery, & IEEE Computing Society. (2008). *Computer Science curriculum 2008: An interim revision of CS 2001.* Retrieved from http://www.acm.org./education/curricula/ComputerScience 2008.pdf

Jackson, J., Cobb, M., & Carver, C. (2005). Identifying top Java errors for novice programmers. *Frontiers in Education 35th Annual Conference – FIE '05* (pp. T4C-24–T4C-27). doi: 10.1109/FIE.2005.1611967

Jadud, M. (2006). *An exploration of novice compilation behaviour in BlueJ*. Doctoral dissertation. Retrieved from Kent Academic Repository.

Joint Task Force on Computing Curricula, IEEE Computer Society, & ACM. (2001). *Computing curricula 2001*. Retrieved from http://www.acm.org/education/curric_vols/cc2001.pdf

Kolling, M., & Rosenberg, J. (1996). Blue—A language for teaching object-oriented programming. In K. J. Klee (Ed.), *Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education (SIGCSE '96).* New York, NY: ACM Press (pp. 190-194). doi: 10.1145/236452.236537. Available at http://doi.acm.org/10.1145/236452.236537

Kummerfeld, S. K., & Kay, J. (2003, January). The neglected battle fields of syntax errors. In *Proceedings of the fifth Australasian conference on computing education* (Vol. 20; pp. 105-111). Australian Computer Society, Inc.

Lane, H. C., & VanLehn, K. (2005). Intention-based scoring: An approach to measuring success at solving the composition problem. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education – SIGCSE '05* (pp. 373-377). doi: 10.1145/1047124.1047471

Lang, B. (1986). On the usefulness of syntax directed editors. In R. Conradi, T. M. Didricksen, & D. H. Wanvik (Eds.), *Advanced programming environments: Proceedings of an international workshop* (pp. 47-51). London, UK: Springer-Verlag. doi: 10.1007/3-540-17189-4_87

Lee, D. (2011). *Detecting confusion among novice programmers using BlueJ compile logs*. Unpublished master's thesis. Ateneo de Manila University, Quezon City, Philippines.

Lee, D. M., Rodrigo, M. M. T. R., Baker, R. S. J. D., Sugay, J. O., & Coronel, A. (2011). Exploring the relationship between novice programmer confusion and achievement. In S. D'Mello & A. Graesser (Eds.), *ACII 2011, Part 1. LNCS 6974* (pp. 175-184). Berlin Heidelberg, Germany: Springer-Verlag.

Lewandowski, G. (2003). Using process journals to gain qualitative understanding of beginning programmers. *Journal of Computing Sciences in Colleges, 19*(1), 299-310.

Lewandowski, G., Gutschow, A., McCartney, R., Sanders, K., & Shinners-Kennedy, D. (2005). What novice programmers don't know. *Proceedings of the 2005 International Workshop on Computing Education Research – ICER '05* (pp. 1-12). New York, NY: ACM Press. doi: 10.1145/1089786.1089787

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin, 36*(4), 119-150. New York, NY: ACM Press. doi: 10.1145/1041624. 1041673

Lynch, W. C. (1967). Description of a high capacity fast turnaround university computing center. *Proceedings of the 1967 22nd National Conference – ACM '67* (pp. 273-288). New York, NY: ACM Press. doi: 10.1145/800196.805997

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education, 10*(4). New York, NY: ACM Press. doi: 10.1145/1868358.1868363

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011). Measuring the effectiveness of error messages designed for novice programmers. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education – SIGCSE '11* (pp. 499-504). New York, NY: ACM Press. doi: 10.1145/1953163.1953308

Matthews, J. (2011). Assessing organizational effectiveness: The role of performance measures. *The Library Quarterly, 81*(1), 83-110. Chicago, IL: The University of Chicago Press. doi: 10.1086/657447

McCartney, R., Moström, J. E., Sanders, K., & Seppälä, O. (2005). Take note: The effectiveness of novice programmers' annotations on examinations. *Informatics in Education, 4*(1), 69-86. Vilinus, Lithuania: Institute of Mathematics and Informatics.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125-180. New York, NY: ACM Press. doi: 10.1145/572139.572181

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM, 49*(8), 90-95. doi: 10.1145/1145287.1145293

Miller, P., Pane, J., Meter, G., & Vorthmann, S. (1994). Evolution of novice programming environments: The structure editors of Carnegie Mellon University. *Interactive Learning Environments, 4*(2), 140-158.

Murphy, C., Kaiser, G., Loveland, K., & Hasan, S. (2009). Retina: Helping students and instructors based on observed programming activities. *Proceedings of the 40th ACM Technical Symposium on Computer Science Education – SIGCSE '09* (pp. 178-192). New York, NY: ACM Press. doi: 10.1145/1508865.1508929

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research, 2*(1), 37-55. Amityville, NY: Baywood. doi: 10.2190/GUJT-JCBJ-Q6QU-Q9PL

PLT. (n.d.). DivaScheme. *Brown Computer Science*. Retrieved from http://cs.brown.edu/research/plt/software/divascheme

Porter, L., Lee, C. B., & Simon, B. (2013). Halving fail rates using peer instruction: A study of four computer science courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)* (pp. 177-182). New York, NY: ACM Press. doi: 10.1145/2445196.2445250. Available at http://doi.acm.org/10.1145/2445196.2445250

Rodrigo, M. M., Tabanao, E., Lahoz, M. B., & Jadud, M. (2009). Analyzing online protocols to characterize novice Java programmers. *Philippine Journal of Science, 138*(2), 177-190.

Romero, C., Ventura, S., Pechenizkiy, M., & Baker, R. S. J. d. (2011). *Handbook of educational data mining*. Boca Raton, FL: Taylor & Francis Group.

Simon, B., Parris, J., & Spacco, J. (2013). How we teach impacts student learning: Peer instruction vs. lecture in CS0. In Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13) (pp. 41-46). New York, NY: ACM Press. doi: 10.1145/2445196.2445215. Available at http://doi.acm.org/10.1145/2445196.2445215

Sison, R., & Chen, J. J. (2005). Intention-based diagnosis of novice C programmer errors in a web-based tutoring system. *Journal Research in Science, Computing and Engineering, 2*(1), 34-41.

Stanford HCI Group. (n.d.). Opportunistic programming: Helping people prototype, ideate, and discover by building software. *Stanford HCI Group*. Retrieved from http://hci.stanford.edu/research/opportunistic/

Thomasson, B., Ratcliffe, M., & Thomas, L. (2006). Identifying novice difficulties in object oriented design. *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education – ITICSE '06* (pp. 28-32). New York, NY: ACM Press. doi: 10.1145/1140123.1140135

Traver, V. J. (2010). On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction, 2010,* 1-26. doi: 10.1155/2010/602570

Truong, N., Roe, P., & Bancroft, P. (2004). Static analysis of students' Java programs. *Proceedings of the Sixth Australasian Conference on Computing Education – ACE '04* (pp. 30, 317-325). Darlinghurst, Australia: Australian Computer Society, Inc.

Vee, M. N. C., Meyer, B., & Mannock, K. L. (2006). Understanding novice errors and error paths in object-oriented programming through log analysis. *Proceedings of the Workshop on Educational Data Mining at the 8th International Conference on Intelligent Tutoring Systems – ITS 2006* (pp. 13-20). New York, NY: ACM Press.

Victor, B. (2012). Learnable programming: Designing a programming system for understanding programs. *Bret Victor*. Retrieved from http://worrydream.com/#!/Learnable Programming

Williams, L., & Kessler, R. (2000). All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM, 43*(5), 108-114. New York, NY: ACM Press. doi: 10.1145/332833.332848

Worsley, M., & Blikstein, P. (2013). Programming pathways: A technique for analyzing novice programmers' learning trajectories. In K. Yacef et al. (Eds.), *AIED 2013, LNAI 7926* (pp. 844-847). Berlin, Germany: Springer-Verlag.

Direct reprint requests to:

Ma. Mercedes Rodrigo
Ateneo Laboratory for the Learning Sciences
Department of Information Systems and Computer Science
Ateneo de Manila University
Quezon City 1108, Philippines
e-mail: mrodrigo@ateneo.edu