Philippine Journal of Science
138 (2): 177-190, December 2009
ISSN 0031 - 7683

Analyzing Online Protocols to Characterize Novice Java Programmers

Maria Mercedes T. Rodrigo[1,*], **Emily Tabanao**[1], **Ma. Beatriz E. Lahoz**[1], and **Matthew C. Jadud**[2]

[1]Department of Information Systems and Computer Science
Ateneo de Manila University, Loyola Heights, Quezon City, Philippines
[2]Department of Computer Science, Allegheny College, Meadville, PA

Computer science educators are concerned and disappointed over students' lack of programming comprehension. This concern has motivated investigations into areas that programming students do not understand, in an effort to provide students with the foundation they need in order to produce correct software. In this study, we built and used several tools to study the behavior of novice programmers—what edits they make, when they compile their programs, and how they respond to errors. The characterization enables computer science educators to identify at-risk students and determine specific interventions. These tools include summarization of student compilation errors, computation of time between compilations, computation of Error Quotients, and a browser that enables educators to view successive compilations. This paper describes how these tools are used and what implications educators can infer from the data that they present.

Key Words: Java, novice programmers, online protocols, BlueJ

INTRODUCTION

Computer programming is a fundamental skill that students in computer science and related disciplines are expected to learn. Computer science educators, though, are concerned and disappointed over first-year students' lack of programming comprehension. In Australia, as many as 35% of students fail their first programming course (Ebrahami & Schweikert 2006). McCracken, et al.'s (2001) ACM-funded multi-national, multi-institutional study found that approximately 30% of computer science students in the United Kingdom and the United States do not understand programming basics. They could not write syntactically correct programs. This means that the programs the students wrote were grammatically incorrect, and could not be executed by the computer. Even when they produced error-free code, their programs produce incorrect results. Subsequent studies (Fitzgerald et al. 2005), including a multinational study by Lister et al. (2004), corroborated these results and found that students have a fragile grasp of programming and were unable to read, analyze, and trace through short fragments of code.

Studies like these, while daunting to the educator, provide fertile ground and many questions for the researcher. The premise of related research undertakings is that it is necessary to give novices the proper foundation in programming if they are to produce correct software later on. A valid analysis of novice problems can help educators adjust the pace and emphasis of lectures and laboratory exercises. From this analysis, it may also be possible to identify at-risk students early and provide specific interventions (Garner et al. 2005). Furthermore, studies of novice programmers may provide computer

*Corresponding author: mrodrigo@ateno.edu

scientists and computer science educators with insight into designing better programming languages, programming environments, and programming curricula (Ebrahami & Schweikert 2006).

Goals, Research Questions, and Hypotheses

Our work attempted to help faculty better monitor their programming classes, study their students' programming difficulties, and help them decide on appropriate intervention strategies. To these ends, we attempted to answer the following questions:

Can we identify at-risk novice programmers by analyzing logs of their interactions with the compiler as mediated by their programming environment? We hypothesize that a student's Error Quotient (or EQ; discussed later) would be predictive of student achievement. We believe that a student with a high EQ would do poorly in the midterm, while a student with a low EQ would do well.

Can browsing through student compilation logs help faculty identify student misconceptions? We hypothesize that enabling a faculty member to browse through student compilation logs can assist that faculty member with student formative assessment. A cursory examination of student compilation logs would indicate whether a student was progressing well or poorly towards the solution of a given programming problem. A more detailed examination of student compilation logs would give the faculty members deeper insight into the specifics of what concepts a student did or did not understand. These examinations can direct faculty members' subsequent lectures and exercises towards concepts that students found problematic.

Significance, Scope and Limitations

As of the time of this writing, this paper was the first publication to demonstrate the successful use of EQ as a predictor of student achievement in computer programming. We did not attempt to correlate EQ with other student characteristics such as achievement in other subjects, nor did we attempt to hypothesize about what other traits might be relevant to the interpretation of EQ or novice programmer behavior.

This paper was also the first publication that described the development and prospective use of the BlueJ Browser as a tool for formative assessment. At the time of this writing, the BlueJ Browser was used principally for the log file analysis that led to this publication. The Browser, as reported here, had not yet undergone extensive user tests.

By testing the veracity of EQ as a predictor of achievement and the BlueJ Browser as an assessment tool, we hope to add to the teaching resources available to faculty responsible for novice programmer education.

Review of Literature

Since the late 1980s, researchers have been using instrumented environments to gather data regarding novice difficulties. Instrumented environments are add-ons to existing integrated development environments that students use to write, compile, and test their programs. The augmented environment enables automated data collection at a grain size and level of detail that researchers can specify.

Several research groups (cf. Jackson et al. 2005; Jadud 2005; Sison et al. 2000; Truong et al. 2004) gathered and collected students' online protocols, that is, all files submitted to the compiler representing the student's path through the solution space. An examination of these paths is highly revelatory about what a student chooses to compile and when, and what a student's most common errors and behaviors are (Spohrer & Soloway 1989). Researchers can then mine these paths for omissions, malformations, or erroneous arrangements.

Examinations of these logs have revealed insights into novice programmer learning, e.g., a few types of bugs account for the majority of novice errors (Spohrer & Soloway 1989). Novices tended to repeat the same programming errors throughout a programming exercise (Chmiel & Loui 2004). They also tended to get frustrated easily and to depend on others for assistance. As evidence of this frustration, some novices have been found to move away from problematic code, whether or not they have solved the problem (Jadud 2005).

These findings have several implications in the teaching of programming. For example, educators may have to specifically address clusters of related concepts considered to be difficult. Understanding programming structures, learning syntax, designing a program, and dividing the program into classes are all examples of activities found to be correlated (Lahitnen et al. 2005). Either students will tend to understand them easily or will tend to struggle through them all. Providing students with more support in one or more of these areas may ease the assimilation of related concepts.

Debugging is a skill that educators must foster in students through formal training (Chmiel & Loui 2004). Teaching programming alone is insufficient. Students need help in developing skills to locate and address program errors. Related to this, students must be taught that the end goal is not clean program compilation but correct program

execution (McCracken et al. 2001). Hence, debugging should take place not just during compile-time but during run-time as well.

These findings also imply a need for the development of better automated methods and tools for assessing programming skill. These range from tools for judging student programming structure (cf. Truong et al. 2004) to cataloging errors (cf. Jackson et al. 2005) to identifying student intentions (cf. Lane & VanLehn 2005). These tools semi-automate the assessment process and can assist teachers in identifying problem areas that need to be addressed as well as at-risk students who need greater assistance.

Previous studies on student online protocols have generally been conducted by researchers using specially designed tools. Our study differs from previous work in that we aim to make the tools for the browsing and analysis of novice programming behavior available to computer science teachers who may or may not be part of this research community.

Scientific Framework

Perkins et al. (1988) classified students into three stylistic categories: Movers, Stoppers, and Extreme Movers. Movers are students who persist in experimentation and testing. They try one solution after another, using feedback effectively to progress in their programming tasks. Eventually, Movers succeed in arriving at a solution.

At the opposite end of the spectrum are the Stoppers. Stoppers are students who are unable to proceed because they have simply given up. These tend to be students who are frustrated or else had negative experiences with programming. They are unsure of themselves and lack confidence in handling the programming language and are not confident about how to get the machine to do what they need it to do.

Extreme Movers are those who tend to ignore feedback or use it ineffectively. They make program changes at random and hence do not progress effectively in their programming tasks. They tend to move around in circles or else go from one unworkable course of action to another instead of moving progressively towards a real solution.

These classifications of programmer behaviors are of interest to computer science educators because they afford us with the opportunity to diagnose learning or non-learning behaviors and possibly intervene in order to promote greater learning. We are especially concerned with at-risk novices who may well decide to disengage from information technology-related courses altogether, opting instead for something less technical in nature. The challenge is to formalize these classifications into computationally tractable models.

To this end, Jadud (2006a) was able to quantify the compilation behavior of students in a given session. The type of syntax error committed and how often it was repeated is an indicator of how well or poorly a student was progressing. A penalty was assigned to behaviors that did not move a student towards the goal of having an error-free code.

Jadud developed a quantification of the student's compilation behavior through a grounded theoretic process. He called it the Error Quotient or EQ. Every record in the data logs represented one compilation event. Stored in each record is the error message (if any), the location of the error, and the source code. The EQ is a function of the error message, the line number, and the text of the source code. Given every two successive compilation events, it determines if the same error occurs on the same line, and whether the source code of the edit location is unchanged. Every pair of compilation events yielded a normalized score. These scores were then averaged to get the final EQ score of the session.

An EQ score ranges 0 to 1.0. An EQ score of 0 means that the student alternated between successful and unsuccessful compilations; or, put another way, a student can make syntax errors, but as long as these errors are fixed in the subsequent compilation, the EQ score will come out as zero. An EQ score of 1.0 means that every compilation resulted in the same syntax error all the time.

In Jadud's original study, data was collected from willing participants whenever they were working on code on campus. As the data generated from this kind of collection was previously unknown, this “opportunistic” data collection differs from our extension of Jadud's work in several ways. First, we gathered compilation logs from students only during formal class sessions. Unlike Jadud's initial work, this was the only data used in computing the EQ of the novice programmers taking part in the study. With a more “constrained” dataset, we were able to successfully demonstrate a correlation between students' EQ and their achievement on the midterm exam.

METHODOLOGY

This study was conducted at the Department of Information Systems and Computer Science (DISCS) of the Ateneo de Manila University on the First Semester

of School Year 2007-2008. The participants of this study are the students enrolled in CS21 A-Introduction to Computing I, also referred to as CS1 in the literature. A total of 143 students agreed to participate in the study, 17% were female and 83% were male. The DISCS has seven computing laboratories that also serve as lecture rooms. The CS 21A classes were held in three of these laboratories. The machines are connected in a local area network and the Internet. All the machines were installed with the same operating system, a recent version of Java, and BlueJ.

Students in this study used BlueJ in performing their programming exercises. BlueJ is an integrated development environment (IDE) for Java specifically designed to support introductory programming. BlueJ's main window provides students with a graph depicting the classes within a program and their relationships to one another (Figure 1).

Double-clicking a class (depicted by rectangle in Figure 1) allows the student to edit its source code (Figure 2).

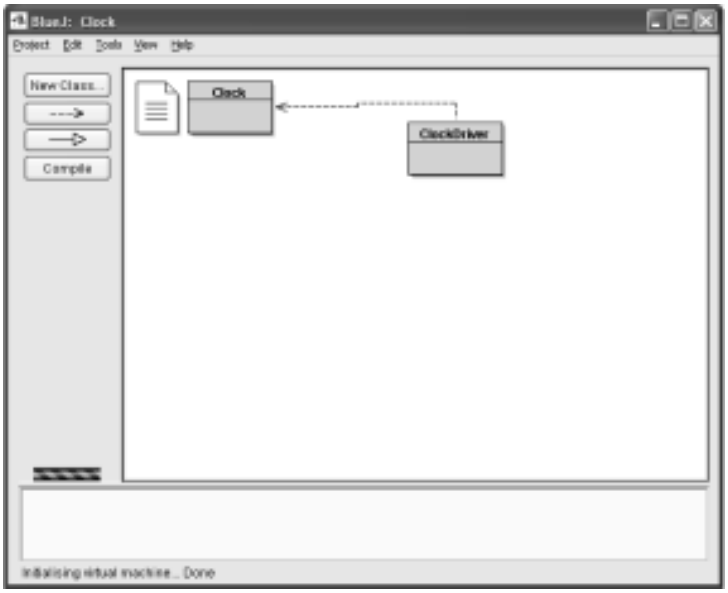

Figure 1. BlueJ's Main Window

Figure 2. BlueJ's Editor with Compile Button Encircled

The student compiles the program by clicking on the Compile button (encircled in Figure 2) or by pressing Ctrl-K. The act of compiling their program converts their source code into an executable form; this is also when the syntax, or grammar, of their program is checked. In the event that any errors are present, the IDE only displays one error message at a time (so as not to overwhelm the student) in a frame below the editor window (Figure 3). In the edit window, the line in question is highlighted (Figure 3).

Note that there are instances in which errors can be misleading. In Figure 3, for example, the highlighted line of code in BlueJ reads:

```
minute = 0;
```

The first error reported by the Java compiler, reported back to the student, is:

```
';' expected
```

As is often the case with compilers for many programming languages, the erroneous line is incorrectly indicated. The line that actually needs correction is the previous line. It is missing a semicolon, which is the statement terminator in the Java programming language:

```
hour = 0
```

These kinds of subtle errors—trivial for experienced programmers—tend to disrupt the learning of novices.

Using BlueJ's extension API, Poul Henriksen and Matthew Jadud of the University of Kent instrumented the BlueJ IDE to capture the behaviors of novice programmers. Each time a student compiles his/her program within this IDE, the IDE sends a snapshot of the code and any error messages to the server, where it is stored in an SQLite database table.

In the study presented here, each student computer had a separate database named after the room and computer number, e.g., F223_01 is the log from computer 1 of room F223. We kept separate logs for each student for each lab period. The data captured included the timestamp of the compilation event, the name of the file compiled, its contents, an indicator of whether the compilation was successful or not, the error message number, the error message, and the line at which the error occurred.

Five laboratory exercises were designed to be performed by all the students on their scheduled laboratory day. Data was collected during these lab exercises. The exercises were designed to be finished in a one-hour laboratory session. The exercises were given to the students on the day of their scheduled laboratory. Code was provided to the students that would, when executed, test the program they had written.

Databases

The instrumented BlueJ environment stored data in two tables in an SQLite database: CompileData and

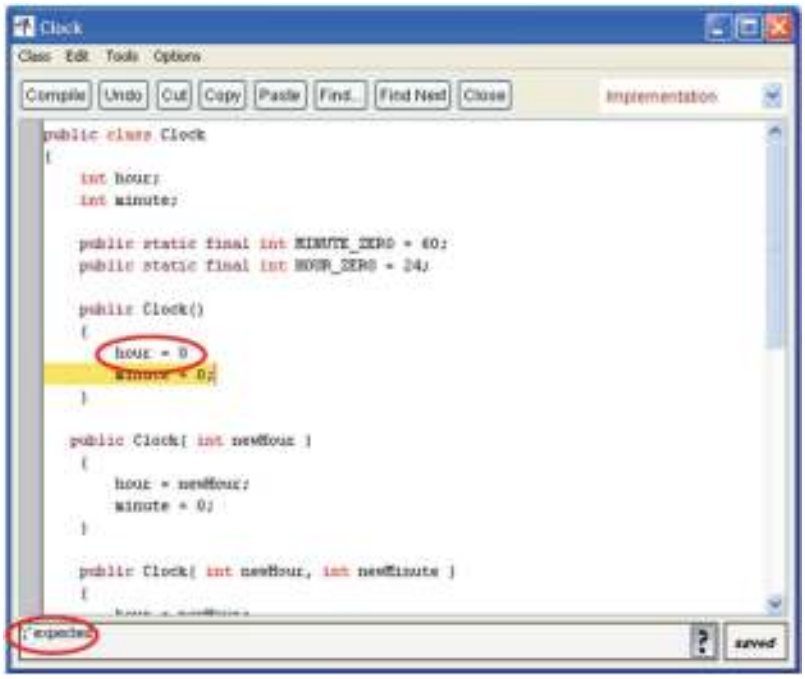

Figure 3. Highlighted Line With the Error and Error Message Encircled

InvocationData. The CompileData table allowed us to answer questions about student compilation behavior in the BlueJ programming environment. The InvocationData table stored data each time a student invoked or executed a program. This study focused on the CompilationData table only. A compilation event data is captured every time a student clicked the Compile button in the instrumented BlueJ environment.

Procedure

On the first day of classes, students were informed about the study. A consent letter was given to each of the students for them to indicate whether or not they were willing to participate in the research undertaking. They were not obliged to join the study. They did not need any kind of preparation or follow any procedure during the study itself. All they had to do was attend their classes, complete the laboratory exercises, and behave as normally as possible.

Data gathering of the compilation logs was completely automated. Data was gathered only on the scheduled laboratory schedules when the standard exercises were given. The laboratory session lasted for an hour to one hour and a half. The lab exercises followed the lectures of the topics covered in the exercises. The data was collected during the first nine weeks of the semester.

Data Cleaning and Extraction

When the server is turned on, all events inside BlueJ are captured. Sometimes students work on sample programs before working on their assigned laboratory exercises. All compilations and invocations on those sample programs were captured by the server. Data recorded that were not part of the data to be studied were removed from the database.

The fields extracted from the database for data analysis included: a unique session identifier, the start and end time of the edit sequence, the name and contents of the edited file, and any error messages that occurred during compilation. The extracted data were stored in a text file which was later used in generating summaries.

Generating the Error Quotient

The Error Quotient was formulated to be the simplest possible algorithm based on behavioral data that would differentiate successfully between students. It has no known precedent in the literature regarding programming behavior.

Jadud's formulation of the Error Quotient began with an iterative, qualitative process not unlike a document analysis, using tools for querying and browsing not unlike those described here (Jadud 2006b). Then, using those characteristics of student behavior that were most consistently tagged across the members of the original population, the EQ algorithm was postulated, and through a brute-force search, parameters were selected that maximized the distance between subjects in the population. Put simply, the weights attached to the repetition of errors, their types, and location (strictly behavioral quantities) were selected to maximize the standard deviation between the sessions recorded for students taking part in the study.

The EQ of each student per laboratory exercise and over all laboratory exercises was computed based on the following algorithm. Given a session of compilation events e_1 through e_n:

Collate Create consecutive pairs from the events in the session, for example, $(e_1,e_2),(e_2,e_3)...(e_{n-1},e_n)$.

Calculate Score each pair according to the algorithm presented in Figure 4.

Normalize Divide the score assigned to each pair by 9 (the maximum value possible for each pair).

Average Sum the scores and divide by the number of pairs. This average is taken as the EQ for the session.

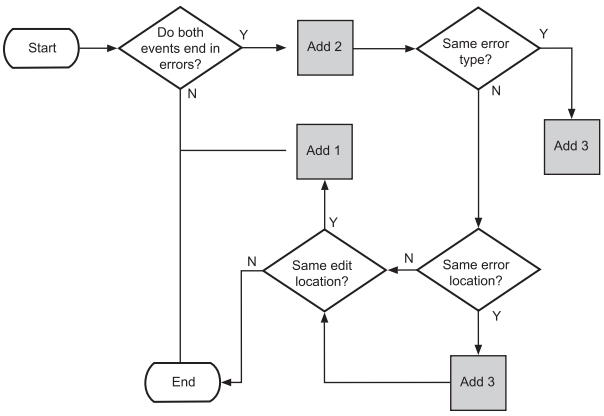

Figure 4. Flowchart showing the EQ calculation process.

The student's achievement in midterm exam was then correlated with their EQ score using Pearson's R.

RESULTS AND FINDINGS

The five laboratory sessions produced a total of 28,386 compilation events. Of the 28,386 compilation events that we collected, 59% or a total of 16,631 compilation events

generated an error. There were a total of 52 different error types encountered. It was noted that the top ten errors accounted for 76% of all these errors (Table 1). This means that majority of the students' time was spent correcting only a few different error types. This was consistent with the findings from earlier studies (cf. Jackson et al 2005; Spohrer & Soloway 1989). Compared to earlier studies (cf. Jackson et al 2005; Jadud 2005), we found that our subjects encountered errors similar to those of their counterparts abroad.

Table 1. Top Ten Errors Encountered

Error Type	Percentage
1. cannot find symbol – variable	20%
2. ';' expected	13%
3. ')' or ')' or'[' or ']' or '{' or '}' expected	10%
4. missing return statement	8%
5. cannot find symbol – method	6%
6. illegal start of expression	6%
7. incompatible types	4%
8. <identifier> expected	4%
9. class, interface, or enum expected	3%
10. 'else' without 'if'	2%
Total	76%

Though it may seem that the top three errors are simple, their causes can be quite difficult to trace—a missing bracket, inadvertent capitalization, undeclared variables, etc. Jadud (2005) found that a missing semicolon can take to as much as thirty minutes or more for a novice to correct. This is a cause for concern because students can get disheartened or frustrated if they encounter errors every time they compile. One possible way of improving the debugging ability of novices is to inform them of the common errors encountered by beginning programming students and discuss to them when these errors occur and how to solve them.

The time between compilations gave us an idea of how much time students spent correcting or editing their code. Fifty-five percent of all compilation events occurred in less than 30 seconds after the previous event. We questioned whether rapid-fire compilation was a quantitative description of Perkins et al.'s (1988) definition of Extreme Movers, and therefore indicative of a non-learning behavior. We also noted that 15 percent of the time was spent by students working at least two minutes on their code between compilations. We again questioned whether this time lag is a quantification of Movers' behavior and whether it implies careful examination and reflection upon code. These are questions that we will explore in future work.

Interestingly enough, 15 percent of the compilations happened two minutes after the previous compilation. We have just begun analyzing the behavior of students when they were able to successfully compile their program—that is, when they manage to eliminate all of the syntactic errors from their code. Very little happened at this point for some students, while others were actively engaged in testing and interacting with the code they wrote. This behavior, along with data pertaining to the time spent between compilations, will likely further aid in catching students who are struggling early in the learning process.

Error Quotient (EQ)

Students who encountered many syntax errors and failed to fix them from one compilation to the next are characterized by high EQs. On the other hand, students who had few syntax errors, or who corrected their syntax errors between compilations, received low EQ scores. In other words, a high EQ score implied that the student struggled with one error after another. A low EQ score implied that the student found and corrected syntax errors quickly.

We took the mean EQ score of each of the participants in five lab sessions. The lowest average EQ score was 0.05. The highest was 0.58. On average, students had a mean EQ score of 0.27 with a standard deviation of 0.12. We then correlated the mean EQ score with the midterm exam scores, we arrived at a moderate but significant correlation value $R = -0.54$ ($p < 0.001$). This implied that the lower a student's EQ, the higher his or her midterm exam score.

On the one hand, it may seem obvious that students who consistently struggle with syntax errors do not master the conceptual material presented both in class and in the laboratory. That said, we know of no other metric capable of providing real-time insights into the behavior of novices that may, in whole or in part, have some predictive power. Even if future refinements fail to produce a strong correlation, a real-time EQ calculation is still a practical tool, as any student incapable of successfully writing an error-free program is likely in need of assistance. Being able to aid these students in a timely fashion, long before any kind of midterm exam, would be a powerful thing.

The BlueJ Browser

To help us analyze the logs further, we developed the Browser as a server-side web application. The current version of the Browser has three screens: the List Screen

(Figure 5), the Table Summary Screen (Figure 6), and the View Code Screen (Figure 7). Each screen and its features are described below.

The List Screen displays all the logs within a given folder. Each item in the list is a hyperlink that will take the user to the corresponding Table Summary View.

Figure 5. List Screen.

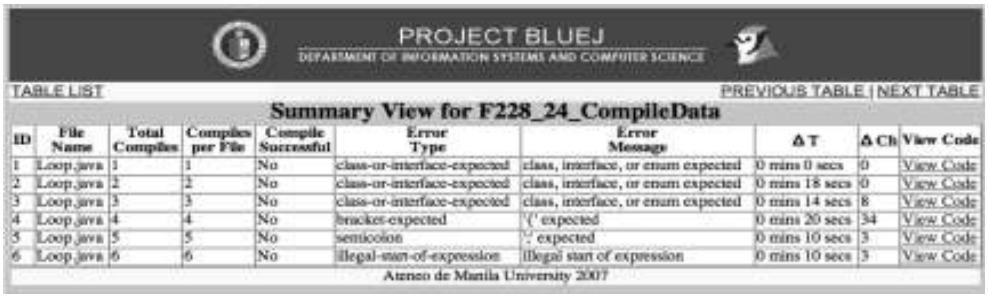

Figure 6. Table Summary View

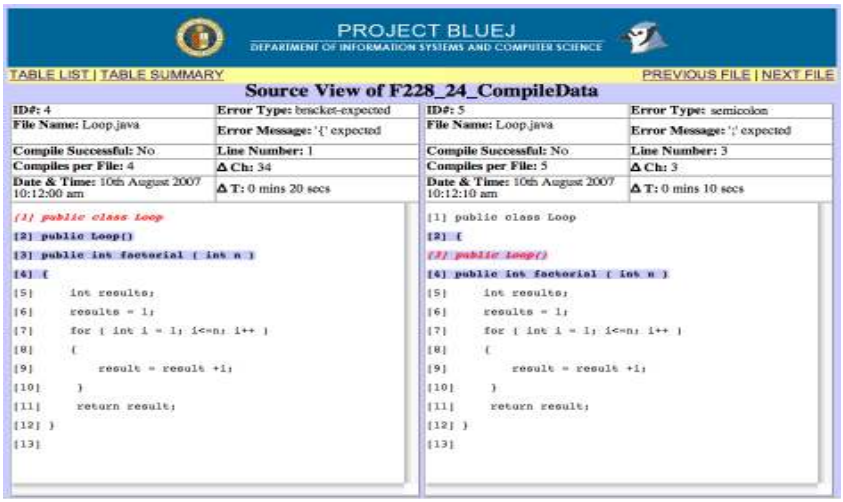

Figure 7. View Code Screen

The Table Summary View gives a summary of the compile data of one student's logs. The fields include:

ID is the index of the records available for the programming session.

File Name shows the name of the file/s that the novice has been working on.

Total Compiles shows the number of times the novice has compiled the program for the duration of the programming session.

Compiles per File shows the count of the compilations of a particular file.

Compile Successful displays whether or not the code compiled properly.

Error Type shows the classification of the error according to the error classifications cited in Jadud's (2005) paper.

Error Message displays the actual error message generated by the compiler if any.

ΔT (Delta Time) is a calculation of the amount of time between consecutive compilations.

ΔCh (Delta Character) is a calculation of the Edit Distance or the number of characters changed by the student between consecutive compilations.

View Code is a hyperlink that will take the user to the corresponding View Code Screen for the Compile Data Record.

This view enables a faculty member to view a student's compilation history at a glance. The faculty member can tell how many compilations it took for a student to complete the laboratory exercise and what errors he/she made in the process. Examining the types of errors that a student made, the frequency of the errors, and the order in which he/she made them (Did the same errors occur consecutively?) may also tell the faculty member which mistakes students are making and may give the faculty some clues as to how to correct them.

The View Code Screen was designed to provide faculty members with a detailed view of how students edit their code. The view displays two consecutive compilations side-by-side. Data such as ID, File Name, Compiles per File, Error Type, Error Message, ΔCh and ΔT are also found here as well the additional data cited below:

Date & Time is a system time stamp showing the date and the time that the code was compiled. ΔT is calculated making use of these values.

Line Number is the line number generated by the compiler when the compiler detects an error.

The **Code Windows** on the bottom part of the screen show the actual code written by the novice for that compile. All the errors are identified by red italicized text such as line 1 in record ID 4 and line 3 in record ID 5 shown in the figure above. The edits or the differences between the two pieces of code are highlighted in light blue.

By examining two consecutive compilations side-by-side, the faculty member can investigate the quantity and quality of a student's edits between compilations. This gives faculty clues as to whether the student actually progressed in the programming task, tried to address the error or simply tried to work around it– whether the student was editing purposefully or was just guessing.

Characterizing students using the browser

We now attempt to analyze novice programmer behavior using the BlueJ browser. As discussed in the theoretical framework, we were interested in three types of programmer behaviors as defined by Perkins et al. (1988): Mover, Stopper, and Extreme Mover. What are the features that distinguish one type of programmer from another? We suspected that Stoppers and Extreme Movers would have higher-than-normal EQs. Other than that, we examined the following vignettes without categorical rules. We attempted to operationalize Perkins et al.'s definitions by identifying the following behaviors that may be symptomatic of the categories.

To identify Stoppers, we looked for students who gave up. At some point during the laboratory exercise, they decided not to proceed without completing the assigned work. We suspected that Stoppers would have relatively short Table Summary Views whose last record still has an error message. Stopper behavior might also be indicated by long lulls between compilations. We take these to be indicators of either "rest periods" or possibly consultations with peers or teachers to better grasp what to do next.

To identify Extreme Movers, we looked for students who were not able to correct errors very effectively. That is, they tended to make the same mistakes in succession. While they may have made attempts at correction, the modified code only added new errors or perpetuated the old ones. When looking for Extreme movers, we looked for Table Summary Views that showed the same error or errors repeated over time, possibly in differing locations. Drilling down into the code, we tried to see how the student responded to the errors—whether he/she was able to address the error effectively or not. We looked for malformations in the code that might reveal a lack of

understanding of syntax or semantics, or an inability to translate intention to syntax.

Finally, we identified suspected Movers by looking for Table Views that show a variety of errors, more often than not, in different locations. Upon examining the code more closely, Movers showed an ability to find and correct errors.

Note that these categories were neither mutually exclusive nor absolute in any way. For example, it was possible for Movers to behave like Extreme Movers if they worked very rapidly. It was also possible for a Stopper to become a Mover. Consultations with classmates or teachers may lead to "Aha!" moments that then lead to more productive work.

In this section, we used the BlueJ browser to locate students who exhibit Stopper, Extreme Mover, and Mover tendencies. By looking at the changes students made from one compilation to another, we have just begun to appreciate the complexity of novice programming behavior. What follows are just three examples taken from a substantial corpus of data, but our observations do not yet represent a rubric that might be shared amongst multiple readers and used to obtain a measure of consistency. These are included here as representative exemplars to provide a sense for the wealth of data contained within our online protocols.

We first illustrate behaviors we associated with a Stopper, that is, someone who gave up without finishing the programming task. The first sign that this student was having difficulty was that he has an EQ of 0.61 for this lab exercise. This is much higher than the overall mean of 0.27 for all students in all labs. It is even higher than the mean for the student's section for that particular lab period (0.24, SD=0.22). Figure 8 illustrates the student's complete logs for the entire lab period. The student worked on the program for approximately 20 minutes out of a 50-minute lab period. Note that he edited and compiled his program 19 times and then does not go on. Of the 19 compilations, only one compilation is error free. The 19th compilation still has an error: *; expected*.

In Figure 9, note that line 36 of the program was:

```
[36] System.out.printf("Credits
     left:P%6.2f \n", Credits
     left:P );
```

Summary View for sy0708_lab1_a_070629_f223_39_compiledata

ID	File Name	Total Compiles	Compiles per File	Compile Successful	Error Message	Delta T	Delta Ch	View Code
4	MobilePhone.java	4	1	No	invalid method declaration; return type required	19717074 mins 43 secs	1043	View Code
5	MobilePhone.java	5	3	No	cannot find symbol - variable load	1 mins 20 secs	22	View Code
6	MobilePhone.java	7	4	No	';' expected	0 mins 41 secs	1	View Code
7	MobilePhone.java	8	5	No	cannot find symbol - variable load	0 mins 37 secs	47	View Code
8	MobilePhone.java	9	6	No	cannot find symbol - variable P	0 mins 41 secs	20	View Code
9	MobilePhone.java	10	7	No	missing return statement	2 mins 22 secs	9	View Code
10	MobilePhone.java	11	8	No	missing return statement	0 mins 29 secs	1	View Code
11	MobilePhone.java	12	9	No	missing return statement	0 mins 46 secs	0	View Code
12	MobilePhone.java	13	10	Yes		4 mins 23 secs	204	View Code
13	MobilePhone.java	14	11	No	';' expected	3 mins 12 secs	33	View Code
14	MobilePhone.java	15	12	No	';' expected	0 mins 50 secs	3	View Code
15	MobilePhone.java	16	13	No	';' expected	0 mins 38 secs	2	View Code
16	MobilePhone.java	17	14	No	';' expected	2 mins 6 secs	126	View Code
17	MobilePhone.java	19	15	No	';' expected	0 mins 33 secs	2	View Code
18	MobilePhone.java	20	17	No	';' expected	0 mins 20 secs	7	View Code
19	MobilePhone.java	21	18	No	';' expected			

Figure 8. Table Summary View of Logs Showing Stopper Tendencies

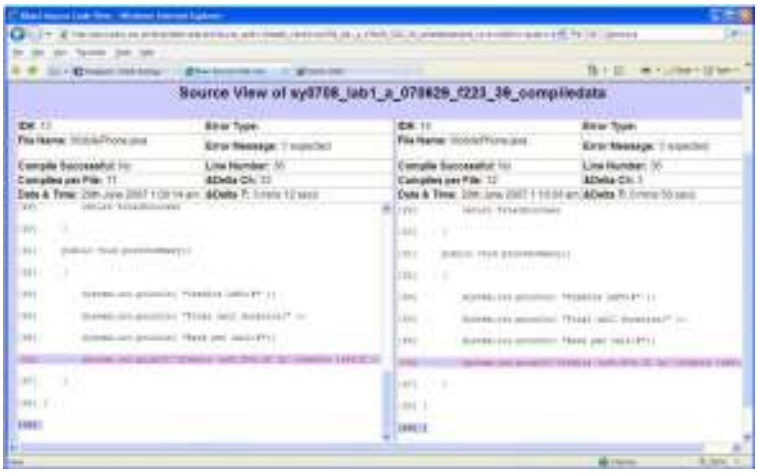

Figure 9. View Code of Compilations 13 and 14

An examination of the erroneous line shows that the source of the error is the malformation of the variable name "CreditsLeft". The student put a space between the words "Credits" and "left"; "left" should begin with an uppercase "L", but the student used a lowercase "l" instead; finally, there is an extraneous ":P" after the word "left". This line remained unchanged for three compilations.

The error message:

```
')' expected
```

was not effective at communicating the problem to the student.

By the 19th compilation, the student had attempted to solve the problem by deleting the erroneous line. The error occurs in a different line, line 33, which reads:

```
[33] double Credits left:P = pesos -
     RatePerCall;
```

However, the malformation still persisted and the error remained unsolved.

In later laboratory sessions, the same student exhibited the behaviors we associated with an Extreme Mover. For this lab session, the student had an EQ of 0.47. This is still higher than the mean of 0.27 for all students for all labs and is more than 3 standard deviations higher than the mean for this section for this lab (0.11; SD=0.11).

Figure 11 shows how the student moved from one part of the code to another, making changes but still failing to correct the errors displayed in previous compilations. For this example the recurring errors are *incompatible types-found char but expect Java.lang.String, unknown variable-*

Figure 10. View Code of Compilation 19.

Figure 11. Table Summary View of Showing Extreme Mover Tendencies

variablestudentName and *illegal start of expression. illegal start of expression* occurred 10 times (compilations 5, 10, 13, 14, 16, 18, 19, 21, 23, and 25) , four times (compilations 15, 17, 20, and 22) and two times (compilations 7 and 8).

In compilation 5, the first occurrence of the *incompatible types-found char but expect Java.lang.String* error:

```
[29] nameBorrower = '-';
```

In compilation 6, the student did something that would have never been seen anywhere in a textbook or in the classroom. We would suspect that their edits are a guess; certainly, they may have seen the code they used *somewhere*, but not in the context in which they applied it:

```
[29] char - = java.lang.String
[30] nameBorrower = '-';
```

The left-hand side of an equals sign should be occupied by a variable to which a value can be assigned. The right-hand side should be an expression that evaluates to a value. Neither of these is true, and, as a result, the best the Java compiler can do is to say that line 29 is "not a statement."

The original problem was subtle, and one easily missed by a student unaccustomed to reading program code. The variable `nameBorrower` is of type `String`, but they attempt to store the value `'-'` in it. A string in Java is denoted with double quotes (`"like this"`); something of type `char` is indicated by enclosing a single character in single quotes (`'x'` is a value of type `char`, for example). Again, the Java compiler has done its best, but the student's edits do not imply a deep understanding of what the error means, or how it might come about.

While reading through his code, we asked ourselves: was the student paying attention to the error messages from the compiler? If he was, did he understand what the error meant and how to fix it? Were the error messages confusing or misleading? Was he just guessing? If the student was guessing or if he was fishing for more clues about what he was doing wrong, we can liken this behavior to gaming the system (Baker et al. 2004) which is negatively correlated with learning gains.

From an affective point of view, this behavior implied a measure of desperation—desperation to understand what is wrong, desperation for help, and desperation to succeed. An intervention from a teacher or a fellow student at this point would probably have helped this student move closer to the goal.

Finally, we examine programming behaviors that we associated with that of a Mover. The student selected for this example had an EQ of 0.05, much lower than the average for all students for all labs of 0.27. This EQ was also low compared with the average EQ of this section for this lab (0.11; SD=0.11). The student moves through the code, programming or correcting errors effectively.

In his Table Summary View (Figure 12), note that successful compilation were interspersed with unsuccessful compilations. The errors he encountered varied from compilation to compilation, both in terms of the error type and the error location.

His first error occurred during the first compilation, in line 26:

```
[26] currentBorrower = '-'
```

The student was able to correct the error by compilation 2; the fix was the addition of a semicolon at the end of the statement:

```
[26] currentBorrower = '-';
```

However, a new error was reported, *cannot find symbol – variable intiTitle*:


PROJECT BLUEJ
DEPARTMENT OF INFORMATION SYSTEMS AND COMPUTER SCIENCE

TABLE LIST PREVIOUS TABLE | NEXT TABLE

Summary View for F223_38_CompileData

ID	File Name	Total Compiles	Compiles per File	Compile Successful	Error Type	Error Message	ΔT	ΔCh	View Code
1	Book.java	1	1	No	semicolon	';' expected	0 mins 0 secs	0	View Code
2	Book.java	2	2	No	unkown-variable	cannot find symbol - variable intiTitle	0 mins 12 secs	1	View Code
3	Book.java	3	3	Yes			0 mins 39 secs	1	View Code
4	Driver.java	4	1	Yes			0 mins 8 secs	703	View Code
5	Library.java	5	1	Yes			0 mins 7 secs	1372	View Code
6	Library.java	6	2	No	illegal-start-of-expression	illegal start of expression	7 mins 33 secs	264	View Code
7	Library.java	7	3	Yes			0 mins 18 secs	11	View Code
8	Driver.java	8	2	Yes			0 mins 10 secs	1640	View Code
9	Library.java	9	4	No	else-without-if	'else' without 'if'	2 mins 29 secs	1799	View Code
10	Library.java	10	5	Yes			0 mins 8 secs	11	View Code

Ateneo de Manila University 2007

Figure 12. Table Summary View Showing Mover Tendencies

```
[16] bookTitle = intiTitle;
```

The problem is because of a typo—a transposition of the "t" and "i" in "init", yielding "inti". In issuing its error, the compiler is claiming that the student has never declared the variable `intiTitle` in their code. The compiler is correct, because the variable-with-typo is unique in their program. They correct this error quickly, and on the next compilation, line 16 reads:

```
[16] bookTitle = initTitle;
```

In Figure 22, compilation 6, line 41 is said to have the error *illegal start of expression*.

```
[40]

[41] public void printSummary()
```

This is an example of an error message that is somewhat misleading. What caused the error was a missing close curly bracket in line 40. Despite the difficult error message, the student was able to make the necessary correction in 18 seconds. By compilation 7, the code reads:

```
[40] }

[41]

[42] public void printSummary()
```

The problem was solved. It is rare for a student to exhibit this level of ability with the Java programming language; many more students have experiences that look like our previous example, the Extreme Mover, where it appears they are using guesswork in an attempt to correct their code.

SUMMARY AND CONCLUSION

Most novices find programming difficult. In this study we attempted to understand students programming difficulties by capturing and analyzing our students' online protocols while performing their laboratory exercises. From their online protocols we derived the errors they encountered, computed for the time between compilations, and computed their EQ.

We found that students who have high EQ scores tends to get lower midterm exam scores. When EQs are used in conjunction with the instrumented BlueJ environment, we gained a means of assessing student behaviors. High EQ scores may indicate that students exhibit Stopper or Extreme Mover tendencies and may be candidates for remediation. Consider the example used to illustrate Extreme Mover tendencies. The ways in which the student attempted to address the error *incompatible types-found char but expect Java.lang.String* implied she was uncertain about what the error meant and did not know how to correct it. In such cases, interventions from faculty or more able peers may be appropriate.

This tool allowed us to determine who among our students know how to debug a program and who do not. From a teacher's viewpoint, if a significant number of students struggle with debugging, it may be necessary to spend more class time discussing common errors, error messages, and ways in which they can be addressed.

From the viewpoint of an IDE developer, it may be necessary for error messages to be more descriptive and less ambiguous. In the vignettes, we found that students do not always understand error messages. The student with Mover tendencies was able to work past the somewhat misleading error *illegal start of expression*, but the same was not true for the student with Extreme Mover tendencies.

We are still in the process of maximizing the potential informativeness of the BlueJ Browser. We consider questions such as: Does the browser present too much or too little helpful information? Which of the views and their subsequent details are essential and which may be excluded? Is a compilation-by-compilation view of student programming really necessary to assess a students' mastery? How will knowing a student's learning trajectory factor into an educator's assessment of that student? What other student characteristics might be important or significant in the interpretation of programming behaviors? These are questions that may be answered in subsequent research that tracks the deployment and use of the tool.

ACKNOWLEDGEMENTS

We thank the Ateneo de Manila University, the Department of Information Systems and Computer Science and Mindanao State University-Iligan Institute of Technology for their support. We also thank Anna Christine Amarra, Andrei Coronel, Darlene Daig, Dr. Fabian M. Dayrit, Jose Alfredo de Vera, Geraldine Kate Dumayas, Dr. Ma. Regina Estuar, Michael Hughes, Dr. Emmanuel Lagare, Sheryl Ann Lim, Ramon Francisco Mejia, Christian Mercado, Fr. Bienvenido F. Nebres, SJ, Darwin Santos, Jessicca Sugay, Dr. John Paul Vergara, the technical and secretarial staff of the DISCS, the Ateneo's CS 21 A students of school year 2007-2008 and those of MSU-IIT for their contributions to this study. This study was made possible through a grant from the Department of Science and Technology's

Philippine Council for Advanced Science and Technology Research and Development entitled *Modeling Novice Programmers' Behavior Through Analysis of Logged Online Protocols* and a 2008-2009 Advanced Research and University Lecturing Fulbright Scholarship provided by the US Department of State, the Philippine American Educational Foundation, and the Council for International Exchange of Scholars

REFERENCES

BAKER RS, CORBETT AT, KOEDINGER KR. 2004. Lectures Notes in Computer Science. In: Intelligent Tutoring. Germany: Springer Berlin/Heidelberg. p.531-540.

CHMIEL R, LOUI MC. 2004. Debugging: from novice to expert. In: SIGCSE '04. Proceedings of the 35th SIGCSE Technical Symposium; 1-3 March 2004; Norfolk, Virginia, USA: Special Interest Group on Computer Science Education (SIGCSE). p.17-21.

FITZGERALD S, SIMON B, THOMAS L. 2005. Strategies that students use to trace code: an analysis based in grounded theory. In: ICER '05. Proceedings of the 1st International Workshop on Computing Education Research; 1-2 October 2005; Seattle, WA, USA: ACM. p.69-80.

GARNER S, HADEN P, ROBINS A. 2005. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In: ACE '05. Proceedings of the 7th Australasian Conference on Computing Education; 31 January – 4 February 2005; Newcastle, NSW, Australia: Australian Computer Society, Inc. p.173-180.

JACKSON J, COBB M, CARVER C. 2005. Identifying top Java errors for novice programmers. In: Proceedings of the 35th ASEE/IEEE Frontiers in Education Conference; 19-22 October 2005; Indianapolis, Indiana: ASEE/IEEE. p.24-27.

JADUD MC. (2005). A first look at novice compilation behavior using BlueJ. Computer Science Education, 15(1), 25-40.

JADUD MC. 2006a. Methods and tools for exploring novice compilation behaviour. In: ICER '06. Proceedings of the 2nd International Workshop on Computing Education Research; 9-10 September 2006; Canterbury, United Kingdom: ACM. p.73-84.

JADUD MC. 2006b. Exploring novice compilation behavior in BlueJ. [Doctoral thesis]. Retrieved from http://www.jadud.com/people/mcj/files/20060408-jadud-dissertation.pdf.

LAHITNEN E, ALA-MUTKA K, JARVINEN HM. 2005. A study of the difficulties of novice programmers. In: ITiCSE '05. Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education; 27-29 June 2005; Caparica, Portugal: ACM. p.14-18.

LANE CH, VANLEHN K. 2005. intention-based scoring: An approach to measuring success at solving the composition problem. In: SIGCSE '05. Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education; 23-27 February 2005; St. Louis , Missouri, USA: Special Interest Group on Computer Science Education (SIGCSE). p.373-377.

LISTER R, ADAMS ES, FITZGERALD S, FONE W, HAMER J, LINDHOLD, M, MCCARTNEY, R, MOSTROM, JE., SANDERS K, SEPPALA O, SIMON B, THOMAS L. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin, 36*(4), 119-150.

MCCRACKEN M., ALMSTRUM V, DIAZ D, GUZDIAL M, HAGAN D, KOLIKANT Y.B-D., LAXER C, THOMAS L., UTTING I. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4):125-140.

PERKINS DN, SCHWARTZ S, SIMMONS R. 1988. Instructional strategies for the problems of novice programmers. *Teaching and Learning Computer Programming: Multiple Research Perspectives.* 153-178.

SISON RC, NUMAO M, SHIMURA M. 2000. Multistrategy discovery and detection of novice programmer errors, *Machine Learning, 38*, 157-180.

SPOHRER JC, SOLOWAY E. (1989). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM, 29*(7), 624-632.

TRUONG N, ROE P, BANCROFT P. 2004. Static analysis of students' Java programs. In: ACE '04. Proceedings of the 6th Conference on Australasian Computing Education; 18-22 January 2004; Dunedin, New Zealand: Australian Computer Society, Inc. p.317-325.