

Methods and Tools for Exploring Novice Compilation Behaviour

Matthew C. Jadud
Computing Laboratory
University of Kent
Canterbury, Kent, UK
matthew.c@jadud.com

ABSTRACT

Our research explores what we call *compilation behaviour*: the programming behaviour a student engages in while repeatedly editing and compiling their programs. This edit-compile cycle often represents students' attempts to make their programs syntactically, as opposed to semantically, correct. Over the course of two years, we have observed first-year university students learning to program in Java, collecting and studying thousands of snapshots of their programs from one compilation to the next. At the University of Kent, students are introduced to programming in an objects-first style using BlueJ, an environment intended for use by novice programmers.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education—*Behavior, Research*

General Terms

Experimentation, Human Factors, Languages, Measurement

Keywords

novice, compiler, compilation, behavior, Java, BlueJ

Roadmap

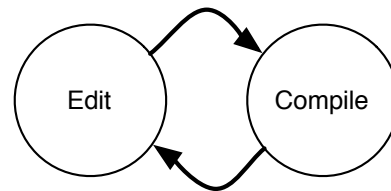
This paper begins with an exploration of the data collected in the course of our study, and a discussion of the most immediate conclusions safely drawn from that data. From this data-driven middle ground, we then explore ways to apply our results toward improve the tools and methods employed in teaching novices. Lastly, we discuss how our research relates to other work regarding novice programmers, and close with possible applications of the tools we have developed, as well as future research directions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICER'06, September 9–10, 2006, Canterbury, United Kingdom.
Copyright 2006 ACM 1-59593-494-4/06/0009 ...\$5.00.

1. COMPILATION BEHAVIOUR IN THE AGGREGATE

It is not uncommon for novices to engage in iterative cycles while learning to program. There are large cycles (one assignment after another), and small cycles (edit-compile-run). Our observations of novice programmers focused entirely on the “edit-compile” interactions students had with the Java compiler.



The first-year students from the University of Kent who took part in our studies were working in BlueJ, a pedagogic programming environment intended to support the learning of Java from an objects-first perspective[20, 21]. Their assignments always began with some template code to get them started, and they often had between three and four weeks to complete any given assignment. 62 students were observed in the Fall of '03, 56 in the Spring of '04 (31 carrying over from the Fall), and 68 students in the '04-05 academic year.

Students who participated in our study had “snapshots” of their programs taken every time they compiled their program if they were using BlueJ and working on campus in a public computing laboratory. When a student compiled their program, a complete copy of the source of their program, any output from the compiler, and additional useful meta-data were captured and shipped to a database. (A complete description of the data captured can be found in Appendix A, and directions for obtaining tools to capture and analyse compilation behaviour data from BlueJ in Appendix C.) As a matter of pedagogic principle, BlueJ only reports one syntax error at a time, meaning students in our study only saw one syntax error at a time.

This information taught us a great deal about novice programmers and the way they go about writing programs. In the simplest case, it told us which errors they encounter most often when learning to program in Java (Figure 1). Furthermore, it told us that the majority of a student's time correcting syntax errors is spent dealing with only a few different error types. We then looked at the time students spent between

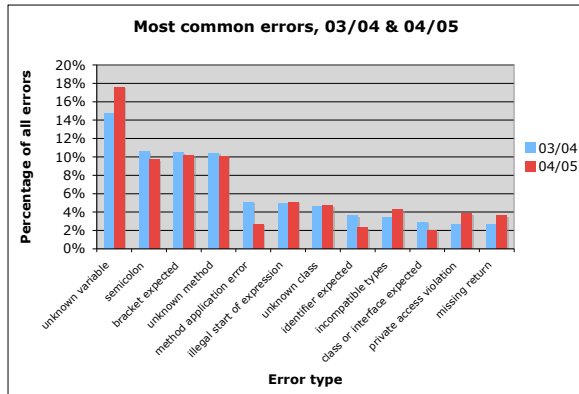


Figure 1: Most common errors encountered by students.

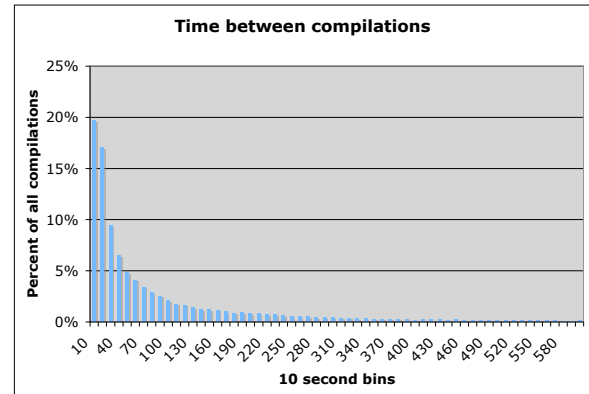


Figure 2: Time between compilation errors.

one compilation and the next; in general, students tend to recompile their programs very quickly (Figure 2). This distribution shows that it is more likely for a student to tweak and recompile their program in less than 10 seconds than it is for them to spend between 11 and 20 seconds, between 21 and 30 seconds, and so on. Their behaviour appears to depend (in aggregate) on the current state of their program (Figure 3). When their program contains a syntax error (F), it is common for students to spend only a handful of seconds reading a syntax error message, editing their code, and re-compiling. However, when they have a program that is syntactically correct (T), they are likely to spend a great deal of time (minutes) adding new code to their programs. This kind of aggregate data was explored more fully in [14].

The students in our study would appear to write lots of code in a sitting, and then try and correct all of the syntax errors at once. This is in keeping with the behaviour we have observed students exhibit in the classroom. We believe this strategy is less than effective for many of our students.

2. A VIGNETTE

We captured over 42,000 distinct compilation events in the course of our study, forming roughly 2100 distinct sessions. A session implies that a student edited and compiled their program at least seven times (a tested and reasonable cut-off for defining the length of a session), and then quit. If they restarted BlueJ within five minutes of quitting, we consider this part of the same session, thus accounting for any unforeseen crashes or otherwise spurious behaviour on the part of the development environment. Because of the nature of our collection, each snapshot in a session is only slightly different from the previous, and often syntactically incorrect.

What follows is a portion of a compilation session from Neville, a student who was a first-year during the 2004-2005 academic year. Captured in October, these code fragments give a sense for what the compilation behaviour of one of our weaker students looks like. This vignette serves as a precursor to our discussion of tools for visualising and making sense of a novice's compilation behaviour. While visual

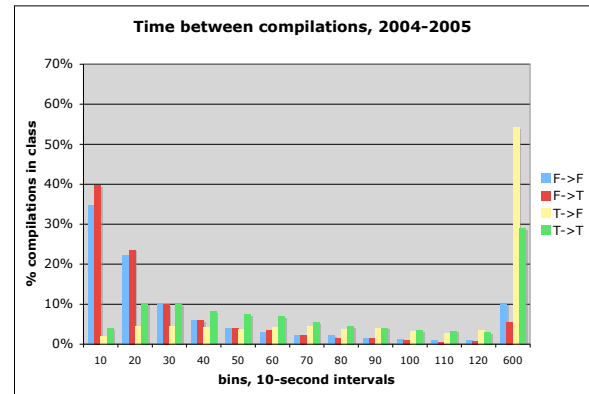


Figure 3: Time between compilations and current program error state.

and statistical tools are powerful, these were inspired by our detailed, qualitative readings of student programming sessions.

2.1 The Notebook

Neville's session captures some of his efforts on the Notebook project. This project comes directly from chapter four of *Objects First with Java* [20]. The Notebook assignment is interesting to us, as every student in our study had to complete it during their first semester. This means we get a picture of their programming behaviour early in the learning process. Perhaps most importantly, students are expected to add a minimal amount of code to their program as part of the Notebook project—we can focus our investigation without having to artificially ignore large sections of code written by the students.

The Notebook project exposes students to the notion of using an object to hold several pieces of data, and then collecting up those objects into a Collection of some sort (in this case, an `ArrayList`). From *Objects First with Java*:

We shall model a personal notebook application that has the following basic features:

- It allows notes to be stored.
- It has no limit on the number of notes it can store.
- It will show individual notes.
- It will tell us how many notes it is currently storing.

In working on this code, Neville encountered many syntax errors. The code presented represents one-third of one session, captured in October of 2004. Specifically, the first 15 minutes of a 45-minute programming session are presented, focusing entirely on just a few lines of code.

2.2 Snapshots of Neville

From the start of the session to the first compilation, Neville added nine lines of code to his program. Specifically, he wrote the `removeNote` method in its entirety. He then made one edit, recompiling his code just five seconds after completing this block: he added a semicolon on line 66.

<i>Before</i>	
059	<code>//A class to remove a note</code>
060	<code>public void removeNote(int noteNumber)</code>
061	<code>{</code>
062	<code>if(noteNumber < 0) {</code>
063	<code> // ... do nothing.</code>
064	<code>}</code>
065	<code>else if(noteNumber < numberOfNotes()) {</code>
066	<code> notes.remove(noteNumber)</code>
067	<code>}</code>
068	<code>}</code>
<i>After</i>	
059	<code>//A class to remove a note</code>
060	<code>public void removeNote(int noteNumber)</code>
061	<code>{</code>
062	<code>if(noteNumber < 0) {</code>
063	<code> // ... do nothing.</code>
064	<code>}</code>
065	<code>else if(noteNumber < numberOfNotes()) {</code>
066	<code> notes.remove(noteNumber);</code>
067	<code>}</code>
068	<code>}</code>

It might seem impressive that Neville could add an entire method to his program and be almost 100% correct in doing so. However, this method is actually provided on page 85 of *Objects First with Java*. We don't know if Neville was referencing the text when he wrote this code—but we suspect that he must have been, as his later compilation behaviour is not nearly so correct or succinct.

It is the addition of the `listAllNotes()` method that interests us more: thirteen lines written in four minutes. In particular, we will focus in on lines 71 through 74, although there are troubling errors elsewhere in Neville's code.

```

070 //List all the notes in the arraylist
071 public void listAllNotes()
072
073 int indexNum
074 {
075     if(noteNumber < 0) {
076         // ... do nothing.
077     }
078     else
079         while (indexNum < numberOfNotes) {
080             System.out.println(notes.get(indexNum));
081             indexNum++ ;
082         }
083

```

If we look closely at lines 71 through 74, we see that Neville has introduced the `indexNum` outside of the opening `{` of the method `listAllNotes()`; it is not likely that an experienced Java programmer would make this mistake. The code could be made more correct by swapping lines 73 and 74. It takes Neville thirty minutes, and many edits, to discover this for himself.

<i>What Neville wrote...</i>	
070	<code>//List all the notes in the arraylist</code>
071	<code>public void listAllNotes()</code>
072	
073	<code>int indexNum</code>
074	<code>{</code>
<i>... and the correction needed.</i>	
070	<code>//List all the notes in the arraylist</code>
071	<code>public void listAllNotes()</code>
072	
073	<code>{</code>
074	<code>int indexNum</code>

Neville does not make this correction, and furthermore, the compiler is not helping. In response to what he wrote, a `' ; '` expected error is reported by the compiler on line 72. In response to this error, Neville adds a semicolon to his code—on line 73.

<i>Before</i>	
070	<code>//List all the notes in the arraylist</code>
071	<code>public void listAllNotes()</code>
072	
073	<code>int indexNum</code>
074	<code>{</code>
<i>After</i>	
070	<code>//List all the notes in the arraylist</code>
071	<code>public void listAllNotes()</code>
072	
073	<code>int indexNum;</code>
074	<code>{</code>

Perhaps Neville doesn't believe the compiler, or perhaps he missed the error—any bump of the keyboard or click of the mouse in BlueJ would clear the error message from the status bar in his editor. Whatever his reasons, Neville recompiles his program; again, a `' ; '` expected error is reported on line 72. In three seconds, Neville looks at the error, the location of the error, his code, and then adds another semicolon to his program at the end of line 71. Sadly, this is *syntactically correct* in some contexts, but it is unlikely that a novice Java programmer would ever need to declare a method header (by itself) and terminate it with a semicolon.

```

Before
070 //List all the notes in the arraylist
071 public void listAllNotes()
072
073 int indexNum;
074 {
After
070 //List all the notes in the arraylist
071 public void listAllNotes();
072
073 int indexNum;
074 {

```

Because this addition is, potentially, a syntactically correct location for a semicolon, the compiler yields a rather uncommon and unhelpful syntax error:

```
missing method body, or declare abstract
```

After one minute and seven seconds, Neville recompiles his program without making any changes to the source code—again, perhaps to refresh the error, or perhaps in hopes that he’ll end up with a syntax error that implies some obvious way forward. Fifteen seconds after this recompile, he removes one character from his program—the semicolon he had just added to line 71.

As a result, the compiler once again reports ‘;’ expected on line 72. Neville then makes a small diversion elsewhere in his program, and returns to the `listAllNotes()` method, where he decides to remove the semicolon from the end of line 73. Thirteen minutes into the session, this brings Neville back to where he started. Arguably, no progress has been made.

2.2.1 Remove the error

After nearly a quarter of an hour, Neville has not managed to correct what experienced programmers might consider to be an obvious syntax error. After trying the most obvious fixes (adding a semicolon when and where the compiler reports a ‘;’ expected), Neville proceeds to employ a technique that we’ve seen students use time and again; if we were to give this behaviour a name, we would call it “remove the error.”

“Remove the error” manifests itself differently in different sessions. Sometimes students remove one or more lines of code completely, only to paste it back into their program several compilations later. In other cases, students employ block comments to comment out an entire method or methods. In this case, Neville comments out one line only: line 73.

```

Before
070 //List all the notes in the arraylist
071 public void listAllNotes()
072
073 int indexNum
074 {
After
070 //List all the notes in the arraylist
071 public void listAllNotes()
072
073 //int indexNum
074 {

```

“Remove the error” is a common strategy that we observed numerous times in our analysis, but it is uncommon to see it

used successfully. By all appearances, students have been lost and confused when this pattern of code removals and insertions is observed in a session. It is a difficult strategy to use effectively.

In a quarter of an hour, Neville has successfully eliminated, but not fixed, one syntax error from his program without making substantial progress towards a more syntactically correct program. This “fix” has only opened up the doors for other related errors that end up confusing Neville further. Over the next fifteen minutes, he wrestles with errors related to this misplaced variable declaration—adding and removing it, renaming it and other variables in his program—until near the very end of the session, where he “sees” or otherwise “discovers” his true error, and corrects it. In truth, our data collection method does not help us understand how or why Neville made this critical fix: he may have discovered the error himself, or he may have asked a friend or instructor.

2.3 Stoppers vs. Movers?

Another behaviour we have observed in many of our students’ traces is that they will “move on” from a particularly problematic piece of code, regardless of whether they have corrected any syntax error that may be lingering (such as the error Neville was facing). Stronger students will ignore a syntax error to work on some other part of their program, later return to the error, and fix it. With students like Neville, however, this is less common. Often, when they move to some other part of their program without fixing an error, they only manage to introduce a new, unrelated syntax error that they then proceed to get stuck on. In either case, this kind of behaviour reminds us of investigations carried out by Perkins, Hancock, Hobbs, Martin, and Simmons at Harvard regarding the investigation of beginners working in BASIC and LOGO[28].

The explorations of Perkins et al. focused on how students can learn to program (and develop effective problem-solving strategies) without the aid of carefully designed instruction. By observing and interacting with students engaged in writing programs to solve small problems, they developed a theory of *stoppers*, *movers*, and *tinkerers*. Stoppers are students who, when faced with a difficult problem, will give up, or otherwise ask for help without working the problem through themselves. Movers and Tinkerers, however, will explore the problem—sometimes systematically, sometimes successfully—hopefully to keep moving towards a problem solution.

We find their summary to be quite compelling:

... for novice programmers, tinkering has both positive and negative features. On the positive side, it is a symptom of a mover rather than a stopper: the tinkerer is engaged in the problem and has some hope of solving it. With sufficient tracking to localize the problem accurately and some systematicity to avoid compounding errors, tinkering may lead to a correct program. On the negative side, students often attempt to tinker without sufficient tracking, so that they have little grasp of why the program is behaving as it is. They assume that minor changes will help, when in fact the problem demands a change in approach. Finally, some students allow tinkers to accumulate untested or leave them in place even after

they have failed, adding yet more tinkerers until the program becomes virtually incomprehensible[28].

In our exploration of many hundreds of sessions—of which Neville’s is a very condensed example—we find that Perkins et al. have captured our own observations very succinctly. Although they are discussing students attempting to correct the semantics of their programs, our data suggests that students exhibit similar behaviours when struggling with the syntax of the language. It was this behavioural description of novice programmers that captured our imagination early in our work; we hoped to uncover more behaviours like “remove the error,” and perhaps put these to work in understanding and teaching novice programmers.

Reading through sequences of edits or syntactically incorrect programs is fundamentally an exercise in sense-making. To better understand what our students were doing, we developed tools for visualising and navigating the snapshots we captured during their programming efforts. These tools yielded many interesting insights, and may yet prove to be useful to both practitioners and researchers alike.

3. USING COLOUR AND LINE

In approaching the data from a new student and/or session, the same questions were asked over and over regarding the programs students had written:

- Were there any large edits in the session?
- Were there any long edits, where a large amount of time passed?
- Were there any particularly problematic syntax errors?
- Did the student focus their effort on one part of the program only? Or, were their edits scattered throughout their program over the course of the session?

These and other related questions evolved as a side effect of our grounded theoretic approach to understanding the massive amount of code we had available[11, 29, 37]. Once we settled into our document analysis, we began an iterative process in which we would make notes on sessions, collect and compare those notes to other sessions we had observed previously, and begin looking for commonalities between sessions. While we found it difficult to highlight behavioural similarities between students, we did observe that we were asking the same questions over-and-over as our investigation progressed. Furthermore, we noted that many of the questions could be quantified, either directly from metadata in our database, or through the analysis of programs written by students. This ultimately led to a working visualisation of students’ sessions.

Table 1 in Appendix B is a visualisation of the first 40 compilations from Neville’s session that was discussed earlier; it represents the current state of our visualisation’s evolution. Each row represents the changes that took place as the student edited their program in between one compilation to the next. The first column, **ErrType**, tells us which syntax error was reported by the compiler; 86 distinct types of syntax error are recognized in our study. Each syntax error type is assigned a distinct colour, which allows us to quickly glance at this column and see if the students struggled over a sequence

of compilations with one type of syntax error or many different types. The \star represents an error-free compilation.

The second column, ΔT , tells us approximately how much time the student spent working on their code in between compilations. There are five bins: 0-10 seconds, 20-30 seconds, 30-60 seconds, 60-120 seconds, and more than two minutes. ΔCh is the number of characters changed between one compilation and the next; negative numbers mean the student removed code between compilation and the next, while positive numbers indicate they added code.

The **Location** column is incredibly useful. The span of the column (effectively) represents the length of the file. The dot (or dots) show us where the student edited their program in between one compilation and the next. The shaded rectangle shows the location of any syntax errors that were reported in the *previous* compilation. As a result, one can see both where the compiler reported an error and where the student chose to edit their code in response to that error in a one-dimensional plot.

In looking at Neville’s session (Table 1), it can be seen how he spends many compilations wrestling with error type #1 (`' ; ' expected`), and spends the majority of his time editing his program in one place near the end of the file. When he does encounter another type of syntax error, it is in the same place as the `' ; ' expected error`, likely implying that he is still dealing with the same underlying problem.

Visualising novice programming sessions provided us with a powerful tool for exploring the behaviour of more students in less time. Perhaps most importantly, as researchers we were no longer bogged down in reading code one compilation at a time; now, we could quickly scan over dozens of compilations events, and say things like “Wow. It looks like they were stuck!” By rendering these visualisations out to HTML, embedded hyperlinks allowed us to quickly jump into the source code representing a given compilation, and begin our compile-by-compile investigation anywhere in the dataset. They provided a powerful map into our data, and we intend to continue evolving the visualisation as a tool, both for future research and practical use. This will be discussed further in section 5.1.

4. THE ERROR QUOTIENT

If one can algorithmically visualise the quantitative aspects of a novice’s programming behaviour, it seems possible to quantify that behaviour as well. In many ways, quantifying novice compilation behaviour represents one possible conclusion of a grounded theoretic process. The goal of grounded theory is to build theory based on data[11]; our quantification of novice compilation behaviour represents one possible theory regarding the edit-compile cycle that evolved out of our iterative readings, re-readings, and visualisations of novice programming sessions.

With the advent of a visualisation tool, our characterization of programming sessions was no longer grounded only in verbose notes, but was improved by our ability to search for and make note of cues in our visualisation. The kinds of questions we asked about a session became very concise: is there a large block of one colour in the **ErrType** column? If so, they must have wrestled with one error type for a long time. Likewise, do a bunch of dots and rectangles line up in the **Location** column? If so, that means they were stuck on the same piece of code, even if the error type changes. In the case of both conditions, it was clear that the student was

struggling—a fact that could easily be ascertained by quickly reading through the session. Having this high-level map of a session made it far easier to make sense of a student’s code, as opposed to puzzling through their efforts entirely from the ground up.

Beginning with the compilation behaviour that had been observed and considered to be “bad”—that is, behaviour that did not seem to move a student towards syntactically correct code—a simple algorithm was devised by which to score each session. This algorithm takes into account the type and location of syntax errors, as well as their frequency, and yields a single number (normalized 0→1) which characterizes the session. Given a session of compilation events e_1 through e_n :

Collate Create consecutive pairs from the events in the session, eg. $(e_1, e_2), (e_2, e_3), (e_3, e_4)$, up to (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 11 (the maximum value possible for each pair).

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

We call the number that describes a student’s programming session the **error quotient**, or EQ. The error quotient characterizes how much or how little a student struggles with syntax errors while programming. At the extremes, a perfect EQ for a given session would be 0.0; this does *not* imply that the student made no mistakes, or had no syntax errors in their programming process. Instead, it means that at no point did a student encounter a syntax error more than one compilation in a row. However, a session scoring 1.0 would imply that every single compilation ended in a syntax error, and each error was the same in each successive compilation.

The algorithm presented in Figure 4 and the penalties assigned were not the result of random guesswork. Initially, we experimented with how we might weight the penalties of a repeated syntax error vs. two consecutive errors of different types. We found that our choices resulted in a distribution of EQ values for our entire population ranging from 0.0 to 0.6. This seemed too narrow a distribution, and we felt that a better separation of individuals should be possible.

In an attempt to spread the population out, we carried out a brute-force search of the space surrounding the algorithmic parameters originally chosen. We selected the set of parameters that maximized the spread of the EQ distribution while simultaneously minimizing the standard deviation of EQ scores in each session for each individual. This yielded a distribution of EQ scores with less variation for each individual student, while maximizing the distance between individuals.

We found at this point that some filtering of our data was necessary; the full population of 161 subjects included students for whom there was only one session recorded. In moving from a qualitative analysis to a more quantitative analyses, these outliers became troublesome. By limiting our sample to students for whom there were at least two sessions in the database, 16% of the sample was lost; in limiting our sample to students with three or more sessions, 40% of our sample was dropped. Only 96 students, over the course of

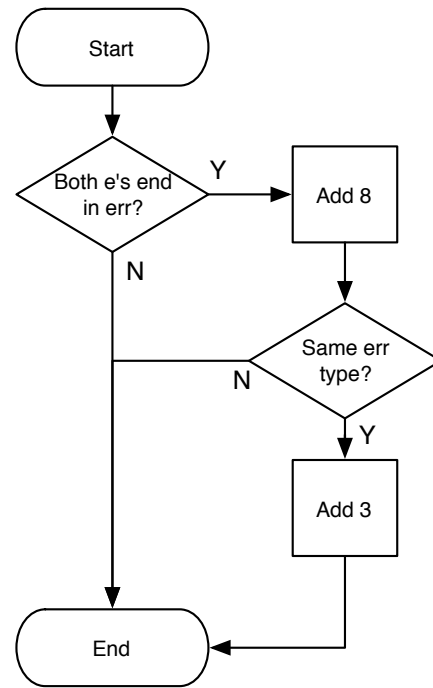


Figure 4: The EQ Algorithm.

two, one-year studies, used the public laboratories three or more times for working on their programs.

The impact of this filtering was significant for our statistical work; see [15] for a complete discussion. Our filtering left us with a Gaussian distribution of EQ scores for the 96 students whom we had enough data to reason about confidently (Figure 5).

5. EQ AS A PREDICTOR?

In examining the relationship between a student’s EQ and their performance on traditional homework- and exam-based grades, we limited our population further to those students enrolled during the 2004-2005 academic year (56 students). We had the most contiguous compilation data for students enrolled during this academic year, and their performance both on assignments and examinations was statistically equivalent to those students who did not take part in our study[15].

There is a distinct correlation between a student’s EQ and both the average grades they receive on assignments and their final exam (Figures 6,7). The low quality of the fit to student assignment data might be explained by any number of factors: given cheating rates reported as high as 10%-15% in the literature[3, 22], and the relatively variable nature of assignments (students deciding not to turn work in, etc.), this is neither surprising nor distressing. While the fit is significant, its quality is poor (R^2 of 0.11). The relationship between a student’s EQ value and their final exam grades is more significant, but again the trend is still very poor ($R^2 = 0.25$). Only a more complete dataset – perhaps including all of a student’s programming behaviour, as opposed to limited, in-lab snapshots – will allow us to further explore the statistical, as opposed to qualitative, relationships between a student’s course mark and EQ.

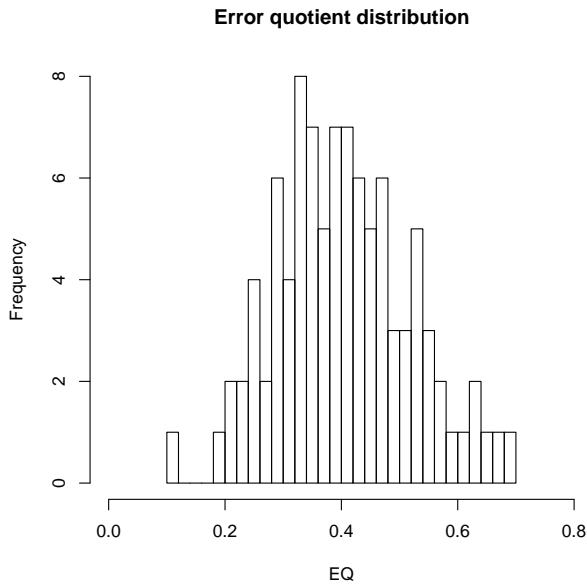


Figure 5: EQ scores, 04/05 filtered.

We cannot actually make any strong claims about whether a first-year student’s compilation behaviour can be used as a predictor for their performance on traditional, exam-based metrics. We are missing an unknown amount of data for each student—how often did they work on their programs at home vs. in a public laboratory? What was their behaviour like there? There are many unknowns, and only further investigation with more complete a more complete dataset will begin to address them.

5.1 Putting tools to use

Our work involved the development of three separate tools for the study of novice compilation behavior.

1. A “code browser” that allows an instructor to easily read successive compilations of a single program,
2. a visualisation that captures the types and frequency of syntax errors encountered by novices in a single programming session, and
3. an algorithm by which we can score sessions and quantitatively compare one session against another.

These are powerful tools for observing novices engaged in the act of programming. In fact, there is a single, critical difference between these tools and traditional measures of student progress like assignments and exams: they are *formative* measures of student progress as opposed to *summative* measures (which are far more commonly employed in CS classrooms)[1]. An assignment may be given to students one week, collected the next, and fed back (graded) a day or week later. However, it is a snapshot in time, and only represents a “finished” product; the instructor has no idea whether a student completed the assignment quickly while watching TV or struggled hour after hour attempting to get their program to compile, let alone be semantically correct.

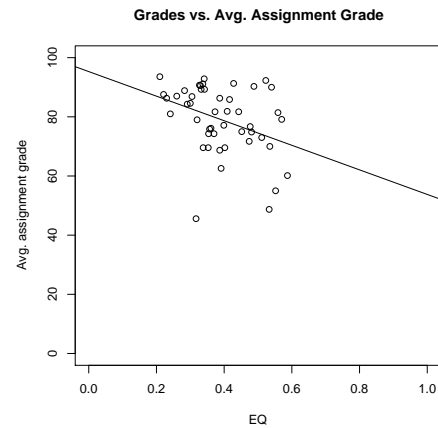


Figure 6: EQ vs. average assignment grades for the 04/05 academic year.

Residual standard error: 10.79 on 45 degrees of freedom
 Multiple R-Squared: 0.13, Adjusted R-squared: 0.11
 F-statistic: 6.86 on 1 and 45 DF, p-value: 0.012

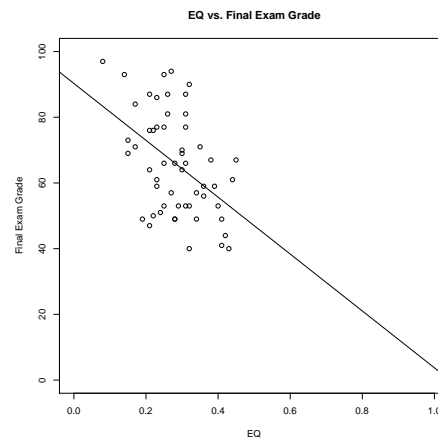


Figure 7: EQ vs. final exam grades for the 04/05 academic year.

Residual standard error: 11.86 on 45 degrees of freedom
 Multiple R-Squared: 0.27, Adjusted R-squared: 0.25
 F-statistic: 16.4 on 1 and 45 DF, p-value: 0.0002

By comparison, it is possible to compute the EQ of a session live and in real-time; certainly, an instructor could look at the EQ and visualisations of student programming sessions easily on a daily basis, even for hundreds of students, and get a sense for how they are doing.

With this kind of window into a student's behaviour, many early interventions can be imagined for students who are struggling. A recommendation that they consider coming in for additional tutoring, or attend the next help session led by a teaching assistant, is only an email away. The instructor might also form peer groups of students and recommend they study together; this kind of group work in programming is often formalized as "pair programming" in CS classrooms[39, 40, 41]. The development of tutorial content or other supporting materials is another possibility—in particular, tools that students can use to practice programming and focus in on what is challenging them at a given moment. Some initial steps have already been taken in extending BlueJ to include focused tutorial content of this nature; see <http://trails.cs-ed.org/> for our initial efforts in this area.

6. THE RELATED AND FUTURE WORK

Practitioners have been trying to support novices in the task of writing syntactically and semantically correct programs for many years. Both DITRAN and WATFOR were FORTRAN compilers developed in the mid-1960s with error handling and reporting mechanisms intended to make life easier for the beginner[4, 27, 31]. As interactive terminals became more common, both flowcharts and structured editors (placing rigid, syntactic constraints on the user) become more common as tools for teaching programming[5, 13, 32, 38]; these tools never quite took hold. The last fifteen years have instead been dominated by pedagogic programming environments that focus more on supporting the student by providing clear syntax errors and a simple IDE interface[10, 21], restricted views into the language (eg. language levels)[6, 9], or visualisations of the run-time behaviour of programs[12, 19, 26].

In terms of research, many studies have focused on students' mental models or the misconceptions students have when actively engaged in writing programs[2, 24, 33, 34]. In the case of Soloway and Spohrer's work, this grew into a theory of *goals* and *plans*[36]. While we collected programs automatically like they did (they referred to this as an *online protocol*), our work differs in one very important way. Soloway and Spohrer, in formulating their goal/plan evaluation of novice programmers, only examined the first syntactically correct program written by the students they were studying. Their argument was that they could observe the "misconceptions" students had about how to solve a given problem by looking at this one snapshot in time.

From our point of view, Soloway and Spohrer worked primarily in a cognitive context. They were concerned with the mental processes that students went through while writing programs and attempting to fix the *semantic* errors they encountered along the way. Their work grew out of a desire to model this "buggy" process algorithmically (in the form of MARCEL), and as a result they collected data and applied analytical techniques appropriate to their question[35]. By way of contrast, our work looks explicitly at the behaviour of novice programmers who are wrestling with *syntactic* concerns in the language. By identifying and understanding

the behaviour of novices learning to program, we hope to build up to later making sound cognitive and constructivist inquiries and recommendations[7].

The work of Myers and Ko regarding interfaces for programming environments—novice or expert—is encouraging work that is driven, first and foremost, by user behaviour and needs[16, 17, 18]. Likewise, distributed, multi-national studies like that carried out by McCracken et al., *Bootstrapping CS Education Research*, *Scaffolding CS Education Research*, and both BRACE and BRACElet provide a model for a community of researchers to ask a question, collect a large amount of (potentially) diverse data, and address a research question from a number of different perspectives[8, 23, 25, 30]. In all of these cases, the researchers are looking not so much to prove a theory as to answer a question, which we feel is an important distinction to be made.

Looking forward, we hope to evolve the visualisation tools we have developed as well as our understanding of novice compilation behaviour. By using these tools live with real students, we can iterate and evolve our visualisation tools with an instructor. In addition, we can explore automatic mechanisms for encouraging students to break out of repetitive error cycles within more controlled, experimental contexts. While this sounds like the Microsoft paperclip ("It looks like you're stuck on a ' ; ' expected error!"), our goal is to encourage students to work more efficiently—and changing their behaviour may be one way to achieve this goal.

In addition to examining how both instructors and students can make use of this kind of behavioural data, we will investigate expanding the scope of our data collection. Currently, we know very little about what students do between compilations. Classroom observation is one way to improve our understanding of what happens from one compilation to the next. Another is to instrument the programming environment to give us more information about what the students do when their program is syntactically correct. Did they create and test objects using BlueJ's object-interactive features? Did they run a suite of unit tests provided by the instructor? These kinds of questions can give us further insight into how students interact with their programs, and lift us out of the purely syntactic view of programming that we have currently.

Acknowledgements

Many thanks to David Barnes and Mathieu Capcarrere for allowing their classrooms to be the subject of our studies; Damiano Bolla and Ian Utting for the BlueJ extensions framework; Damian Dimmich, Poul Henriksen, Christian Jacobsen, and Ed Suvanaphen for their assistance making sense of everything; and Sally Fincher for her support and guidance.

7. REFERENCES

- [1] T. A. Angelo and K. P. Cross. *Classroom Assessment Techniques: A Handbook for College Teachers*. Jossey-Bass, 1993.
- [2] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Commun. ACM*, 26(9):677–679, 1983.
- [3] Janet Carter. Collaboration or plagiarism: what happens when students work together. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE*

- conference on Innovation and technology in computer science education, pages 52–55, New York, NY, USA, 1999. ACM Press.
- [4] D. D. Cowan and J. W. Graham. Design characteristics of the watfor compiler. In *Proceedings of a symposium on Compiler optimization*, pages 25–36, 1970.
- [5] Peter J. Denning. Acm president's letter: smart editors. *Commun. ACM*, 24(8):491–493, 1981.
- [6] Peter DePasquale, John A. N. Lee, and Manuel A. Perez-Quiones. Evaluation of subsetting programming language elements in a novice's programming environment. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 260–264, New York, NY, USA, 2004. ACM Press.
- [7] P.A. Ertmer and T.J. Newby. Behaviorism, cognitivism, constructivism: Comparing critical features from an instructional design perspective. *Performance Improvement Quarterly*, 6(4):50–70, 1993.
- [8] Sally Fincher, Raymond Lister, Tony Clear, Anthony Robins, Josh Tenenberg, and Marian Petre. Multi-institutional, multi-national studies in csed research: some design considerations and trade-offs. In *ICER '05: Proceedings of the 2005 international workshop on Computing education research*, pages 111–121, New York, NY, USA, 2005. ACM Press.
- [9] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: a programming environment for scheme. *J. Funct. Program.*, 12(2):159–182, 2002.
- [10] Stephen N. Freund and Eric S. Roberts. Thetis: an ansi c programming environment designed for introductory use. In *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pages 300–304, New York, NY, USA, 1996. ACM Press.
- [11] Barney G. Glaser and Anselm L. Strauss. *Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, June 1967.
- [12] Mark Guzdial and Elliot Soloway. Teaching the nintendo generation to program. *Commun. ACM*, 45(4):17–21, 2002.
- [13] Herman D. Hughes. A tool designed to facilitate structured programming. In *SIGCSE '77: Proceedings of the seventh SIGCSE technical symposium on Computer science education*, pages 26–30, New York, NY, USA, 1977. ACM Press.
- [14] Matthew C. Jadud. A first look at novice compilation behavior. *Computer Science Education*, 15(1):25–40, 2005.
- [15] Matthew C. Jadud. *An exploration of novice compilation behavior in BlueJ*. PhD thesis, University of Kent, May 2006, unpublished.
- [16] Andrew J. Ko, Htet Aung, and Brad A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 126–135, New York, NY, USA, 2005. ACM Press.
- [17] Andrew J. Ko, Htet Aung, and Brad A. Myers. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1557–1560, New York, NY, USA, 2005. ACM Press.
- [18] Andrew J. Ko and Brad A. Myers. Designing the whyline: a debugging interface for asking questions about program behavior. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 151–158, New York, NY, USA, 2004. ACM Press.
- [19] M. Kolling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Journal of Computer Science Education*, 13(4), 2003.
- [20] Michael Kolling and David J. Barnes. *Objects first with Java: A practical introduction using BlueJ*. Prentice Hall, 2nd edition, 2005.
- [21] Michael Kolling and John Rosenberg. Tools and techniques for teaching objects first in a java course. In *SIGCSE '99: The proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, page 368, New York, NY, USA, 1999. ACM Press.
- [22] Thomas Lancaster and Fintan Culwin. Towards an error free plagiarism detection process. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 57–60, New York, NY, USA, 2001. ACM Press.
- [23] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Mostrom, Kate Sanders, Otto Seppolo, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 119–150, New York, NY, USA, 2004. ACM Press.
- [24] Richard E. Mayer. A psychology of learning basic. *Commun. ACM*, 22(11):589–593, 1979.
- [25] Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. In *ITiCSE-WGR '01: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 125–180, New York, NY, USA, 2001. ACM Press.
- [26] Andres Moreno, Niko Myller, Erkki Sutinen, and Mordechai Ben-Ari. Visualizing programs with jeliot 3. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 373–376, New York, NY, USA, 2004. ACM Press.
- [27] P. G. Moulton and M. E. Muller. Ditrán—a compiler emphasizing diagnostics. *Commun. ACM*, 10(1):45–52, 1967.
- [28] D. N. Perkins and Fay Martin. Fragile knowledge and neglected strategies in novice programmers. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 213–229, Norwood, NJ, USA, 1986. Ablex Publishing Corp.
- [29] M. Piantanida, C. Tananis, and R. Grubs. Generating grounded theory of/for educational practice: The journey of three epistemorphs. *International Journal of*

- Qualitative Studies in Education*, 17(3):325–346, 2004.
- [30] Kate Sanders, Sally Fincher, Dennis Bouvier, Gary Lewandowski, Briana Morrison, Laurie Murphy, Marian Petre, Brad Richards, Josh Tenenberg, and Lynda Thomas. A multi-institutional, multi-national study of programming concepts using card sort data. *Expert Systems*, 22(3):121–128, July 2005.
- [31] Peter W. Shantz, R. A. German, J. G. Mitchell, R. S. K. Shirley, and C. R. Zarnke. Watfor—the university of waterloo fortran iv compiler. *Commun. ACM*, 10(1):41–44, 1967.
- [32] Ben Shneiderman, Richard Mayer, Don McKay, and Peter Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Commun. ACM*, 20(6):373–381, 1977.
- [33] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: an empirical study. *Commun. ACM*, 26(11):853–860, 1983.
- [34] J. C. Spohrer and E. Soloway. Alternatives to construct-based programming misconceptions. In *CHI '86: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 183–191, New York, NY, USA, 1986. ACM Press.
- [35] James Clinton Spohrer. *Marcel: a generate-test-and-debug (gtd) impasse/repair model of student programmers*. PhD thesis, 1989.
- [36] James G. Spohrer and Elliot Soloway. Analyzing the high frequency bugs in novice programs. In *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*, pages 230–251, Norwood, NJ, USA, 1986. Ablex Publishing Corp.
- [37] A. Strauss and J. Corbin. *Basics of qualitative research: grounded theory procedures and techniques*. Sage Publications, Newbury Park, Calif., 1990.
- [38] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.
- [39] Lynda Thomas, Mark Ratcliffe, and Ann Robertson. Code warriors and code-a-phobes: a study in attitude and pair programming. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 363–367, New York, NY, USA, 2003. ACM Press.
- [40] Tammy VanDeGrift. Coupling pair programming and writing: learning about students' perceptions and processes. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 2–6, New York, NY, USA, 2004. ACM Press.
- [41] Laurie Williams and Richard L. Upchurch. In support of student pair-programming. In *SIGCSE '01: Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 327–331, New York, NY, USA, 2001. ACM Press.

APPENDIX

A. DATA CAPTURED

Data captured in the course of our study was posted via HTTP to a webserver (so as to circumnavigate firewalls), and was then injected into a Postgres database. It would then be processed and copied into a set of “cleaned,” or other-

wise finished, tables. Only the tables used for analysis are described here.

A.1 The metas table

The metas table contained nine columns: (int) index, (int) cindex, (int) session, (txt) uname, (int) result, (int) client.start, (int) server.receive, (txt) hostname, and (txt) host.type. The index provides a unique index to the event captured, while cindex provides a session-specific counter, reset whenever a student engages in a new programming session. The uname field is the student's university-wide unique username, and result is actually binary (0 or 1), and tells us whether a particular compilation event was error-free or not. The client.start field is generally disregarded; for consistency, all of our timing data is based on the server.receive field; any network latency effects are likely lost in our rounding of all timing data up to the nearest second. Currently, hostname and host.type do not factor into our analyses, but are important metadata to have associated with each compilation event.

A.2 The errors table

Each entry in the metas table may have one or more entries in the errors table associated with it. This is because it is possible for the student to have more than one file open when they compile their program, each which may generate an error or warning when compiled. The errors table contained eleven columns: (serial) index, (int) meta, (int) sess, (txt) uname, (int) etype, (int) etime, (txt) emsg, (int) eline, (txt) project, (txt) fname, and (txt) file.

We are rarely interested in the particular index into the table; our queries typically involve joining across the meta field, which is an index into the metas table. Both the sess and uname fields are copied out of the other table, largely because we are not database experts. The etype indicates whether the event was the result of an error, a warning, or a successful compilation; the etime comes from the client, not the server. The emsg is the full message reported by the compiler, if any, and eline the line in the file (again, as reported by the compiler). The project field gives us the full path to the BlueJ project, which therefore gives us the project name—students will sometimes do a “Save As,” explore an idea, and then switch back to the first project. This kind of renaming of projects means we cannot simply use the filename (fname) as a unique identifier for a file. The file field is the largest in the database, as it contains the complete source for a students program as captured at the time of compilation.

B. NEVILLE'S FIRST SESSION









































The partial session depicted in Table 1 represents our current working visualisation for novice compilation sessions. We have two output formats: HTML and \LaTeX . The version included here lacks the rich hyperlinking that our HTML version provides—the presentation differences between the two, however, are minimal at this time.

C. OBTAINING TOOLS

The tools described in this paper exploring novice compilation behaviour in BlueJ, as expert tools, were not fit for distribution. However, they will ultimately be available for download under the bluej.org domain.

Table 1: Neville, Session 1

Project: notebook1 **File:** Notebook.java
Duration: 42m53s **EQ:** 0.6
Date: Thursday, November 4th, 2004 11:09am

#	Err Type	ΔT	ΔCh	Location
1	1		201	
2	*		1	
3	1		239	
4	1		1	
5	25		1	
6	25		0	
7	1		-1	
8	1		12	
9	1		0	
10	1		0	
11	1		-1	
12	2		2	
13	1		-11	
14	2		2	
15	2		-1	
16	1		-2	
17	1		-1	
18	1		1	
19	1		-73	
20	1		13	

Neville, Session 1, Continued...

#	Err Type	ΔT	ΔCh	Location
21	1	█	0	
22	1	█	-10	●
23	1	█	0	
24	1	█	0	
25	1	█	1	●
26	1	█	-1	●
27	1	█	2	●
28	2	█	2	●
29	1	█	-4	●
30	1	█	0	
31	1	█	-9	●
32	25	█	1	●
33	1	█	-1	●
34	1	█	0	
35	1	█	0	
36	1	█	459	●
37	1	█	0	●
38	1	█	17	●
39	1	█	0	
40	14	█	2	●