Computer Science Education
Vol. 15, No. 1, March 2005, pp. 25–40

Routledge
Taylor & Francis Group

A First Look at Novice Compilation Behaviour Using BlueJ

Matthew C. Jadud
Computing Laboratory, University of Kent, Canterbury, Kent, UK

Syntactically correct code does not fall from the sky; the process that leads to a student's first executable program is not well understood. At the University of Kent we have begun to explore the *compilation behaviours* of novice programmers, or the behaviours that students exhibit while authoring code; in our initial study, we have focused on *when* and *what* they choose to compile. By examining these behaviours, we have determined the most common errors encountered by students using BlueJ in our introductory course on object-oriented programming, how those students tend to program when in supervised laboratory sessions, and we have identified future directions of study driven by our initial observations. Our goal is to apply this research to the future development of BlueJ and instructional methodologies involving its use in the classroom.

1. INTRODUCTION

This paper presents a first look at novice compilation behaviour of students learning object-oriented programming using the BlueJ pedagogic programming environment. The goal of our explorations of observable programmer behaviour is to help inform the teaching of programming and the development of pedagogic programming environments. In this paper, we explore some gross behavioural characteristics exhibited by a population of first-year students learning Java while utilizing BlueJ, a pedagogic integrated development environment for programming in Java. We begin by presenting related work in this area in section one, our data collection methods in section two, in section three an analysis of the data collected during the autumn term of 2003, a discussion of our work in section four, and we close with future research directions based on these analyses.

2. PREVIOUS WORK

Bits and pieces of research in the areas of computer science education research, the psychology of programming, and human-computer interaction contribute to our

*Corresponding author. Matthew C. Jadud, Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK. E-mail: matthew.c@jadud.com

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/05/010025-16 © 2005 Taylor & Francis Ltd
DOI: 10.1080/08993400500056530

current understanding of *compilation behaviour*. At the least, studies regarding novice programming, syntax errors and error rates, compiler and error message design, debugging, pedagogical programming environments, pedagogic programming languages, and language subsets all have some measure of relevance. Typically, these studies explore the cognitive psychology of novice programmers, probing what they *understand*. Very few of these studies explore the programmer's behaviour, or how we might shape that behaviour to improve programming practice.

2.1. Misconceptions, Planning

Studies regarding logical, run-time (post-syntactic) errors and ''misconceptions'' represent one class of cognitive research (Spohrer et al., 1985; Spohrer & Soloway, 1986a, 1986b). Characterising the planning process, problem-solving process, and comprehension of written programs are yet other themes in the research (Brooks, 1983; Rist, 1986; Klahr & Carver, 1988; Mayrhauser & Vans, 1994; Ramalingam & Wiedenbeck, 1997). This type of research typically sheds little or no light on our own explorations, as it does not address the syntactic problems students have as they engage in the programming process. These studies typically *begin* with the students' first syntactically correct programs, ignoring the observable behaviour of novice programmers, or focus on extrapolating cognitive explanations for the behaviours observed.

2.2. Pedagogic Integrated Development Environments

There is a growing body of literature regarding programming languages designed for novices and environments to support those languages (Freund & Roberts, 1996; Findler et al., 1997; Allen et al., 2002; Patterson et al., 2003). There are few studies regarding the use of these kinds of environments by students; from them, we know that they tend to appreciate the simpler interfaces, that errors tend to persist over fewer compiles with ''reduced'' or ''subsetted'' languages, and that the students clearly interact with the pedagogic environments differently than professional integrated development environments (DePasquale, 2003; Heeren et al., 2003). This is a young area of study, however, and a great deal more work needs to be done.

Green and Petre (1996) provide a model for evaluating programming environments in their analysis of two visual programming languages, LabView and Prograph, and their associated programming environments. A primary goal of their work was to provide an example of the use of Green's *cognitive dimensions* framework as a non-specialist tool for evaluating whole environments. In the course of her Ph.D. work, Linda McIver (2001) evaluated a number of programming languages using the cognitive dimensions framework, although this evaluation was not carried out as an empirical investigation, and did not take into account the environment in which the programming might take place—an important part of the novice's programming experience.

2.3. Types and Rates of Error Occurrence

Although not strictly labeled as *behavioural* studies, research into error message design and syntactic errors in programming languages is pertinent to our work (Brown, 1983; Schorsch, 1995). Gannon's (1975) work evaluating TOPPS and TOPPS II (a pair of statically and dynamically typed languages developed at the University of Maryland for teaching programming and studying the design of programming languages) provides a starting point for the systematic comparison of two different (but syntactically similar) programming languages. These studies provide models and ideas for analyses of the process students go through while programming, as well as approaches to analyze the programs generated themselves.

Several studies have been carried out that are methodologically similar to ours, with interesting results. In evaluating the effectiveness of their new computing center, Moulton and Muller (1967) provide some numerical and anecdotal reports on error rates and programmer behaviour at the University of Wisconsin. Litecky and Davis (1976) carried out a study of 73 novices programming in COBOL, focusing entirely on the syntax errors generated. Zelkowitz's (1976) research also focused largely on errors, examining all the PL/I programs compiled and executed on the University of Maryland's mainframes.

2.4. Characterizing Novices

Perkins, Hancock, Hobbs, Marin, and Simmons (1986) observed young programmers working in LOGO, and based on their observations classified the students as either **stoppers** or **movers**. In their characterization, *stoppers* were students who would, while working on a program in class, constantly ask for help every step of the way. *Movers*, on the other hand, would muddle through problems on their own, and *extreme movers* were students who would perhaps pay too little attention to the feedback the compiler and programming environment provided, hacking madly with no apparent sense of where they had been. At some level, a behavioural understanding and characterization of novice programmers should enable us to detect (using Perkins's categorization) stoppers and extreme movers automatically.

3. METHODOLOGY

We began our work with an automated observation of novice compilation behaviour as it naturally occurred in classroom tutorial sessions. These sessions met once a week for one hour in a public computing lab on campus, were limited to approximately sixteen students, and were overseen by either a member of the teaching staff or a postgraduate teaching assistant; they are typified by a minimum of lecture-style content, and often involve the students working through one or more problems to help illustrate concepts from that week's lecture.

We instrumented the BlueJ programming environment to report at compile-time the complete source from students' programming sessions, as well as an assortment

of relevant metadata. This data included the username reported by the OS[1]; the research site (e.g., "KENT"); a client-side index indicating compilation number in the current sequence, where the first compilation after startup is zero, the next is one, etc.; the compilation result (syntax free or an error); the filename of the file being compiled; when the client initiated the compile; when the server received the information; the IP address and hostname (as reported by the host); the OS name, architecture, and version; the compilation result type (error-free, warning, or error); and the line number of any errors. Every time students compiled their code, this collection of information was packaged up and shipped to a server for storage and later analysis.

In the classroom, 63 of the 206 students enrolled in our first programming course gave us their consent to capture information regarding their programming habits. This sub-population provides us with the ability to ask questions that span multiple compile events and multiple sessions—questions ultimately leading to whether individuals exhibit detectable patterns of compilation behaviour over time.

3.1. Population Characterization

Our sub-population of 63 students appears to be representative of the larger population, given course marks from the first term and the available attendance data.

*3.1.1. Course Marks*During the Michaelmas (autumn) term of 2003, the students had four marked assessments; three were take-home coursework, while the fourth piece of coursework was an in-class "exam," intended to provide both the students and instructor a sense of where they stood at this point in the term.

The first pair of box-and-whiskers in Figure 1 represent the distribution of marks on the first assessment of the term for the entire class (the first box), and the 63-student sub-population who consented to be part of the study (the second box). For both the entire class and the sample population, the first quartile, mean, and third quartiles are for all intents and purposes identical. There is no way to discriminate between the populations on assessment one.

In looking at the rest of the assessments, we see that we cannot discriminate between the marks reported for the class as a whole and those in our sample population; in the case of all of the assessments, we have $p < .05$ agreement between the two populations. This tells us that the students taking part in our study do not perform significantly better or worse than their peers on the assessments given. In this respect, we can take them to be representative of the class as a whole.

*3.1.2. Attendance*In the programming courses at Kent, attendance records are kept for the laboratory sessions. The absence rate in these laboratory sessions is roughly one class session in eight. During the time observed, the students in the study missed class significantly more often than the coursewide average (Figure 2). For the 63 student

[1]Data is only collected in the event that the student agreed to take part in the study.

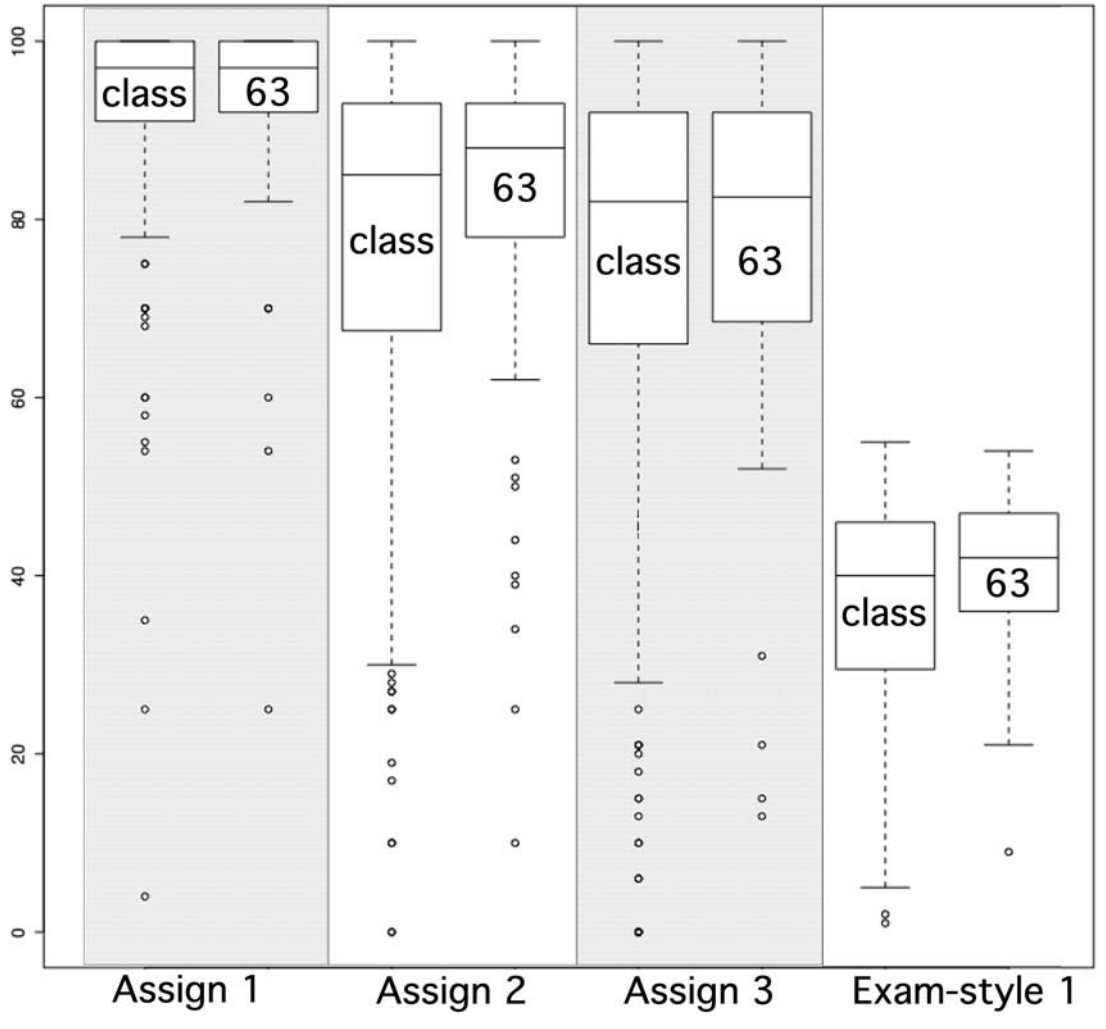

Fig. 1. Grade distributions for assessments one, two, three, and "exam" one.

sub-population, one class in two was missed on average, and only one in six students managed to attend seven of the nine recorded class sessions.

It would appear that the sample population attends class less frequently, on average, than the rest of the students enrolled in the course. One possible explanation could be that students took part in the study hoping it would, somehow, help their overall course grade. Another possible explanation comes from the way attendance is taken in our laboratories: while some instructors mark students present or absent themselves, it is not uncommon for instructors to pass around the attendance sheet. This means students can easily mark their friend present, or mark themselves present from weeks previous they may have missed—either way raising the *apparent* attendance figures for the course. The attendance figures we present for students

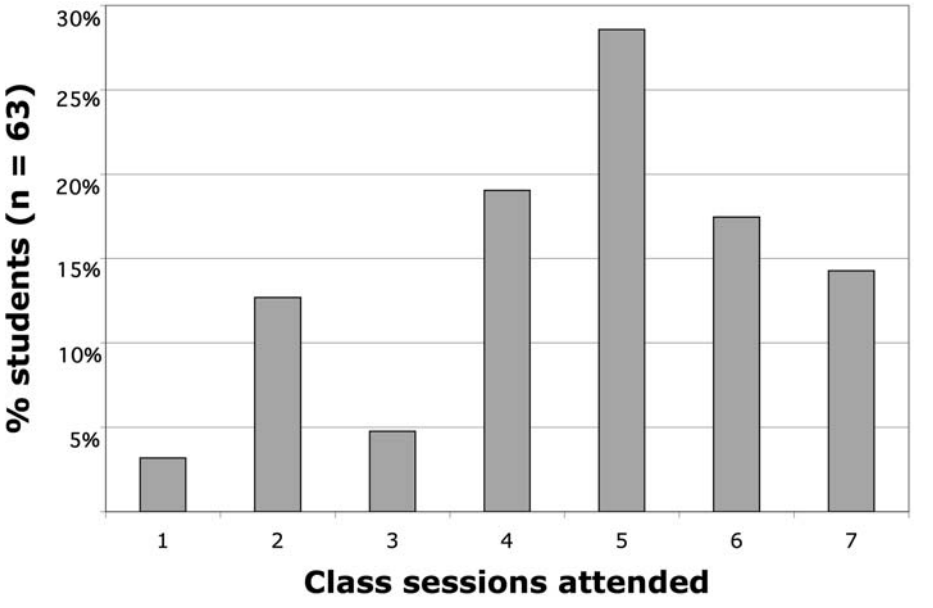

Fig. 2. Attendance of students taking part in the study.

who agreed to take part in the study come from their actual interactions with the computer in class, and therefore are potentially more accurate.

4. ANALYSIS OF RESULTS

To date, we have observed that a minority of different types of syntax error account for the majority of errors dealt with by students. Because these errors are relatively simple to fix, the typical programming behaviour apparently exhibited by the students is one in which they write some code, and then rapidly fix a sequence of one or more trivial errors; this rapid-fire repair of their code can easily lead the casual observer to believe they are ''just letting the compiler do their thinking for them.'' The types and distribution of errors encountered provide a window into understanding other aspects of the novice programmer's behaviour.

4.1. Error Types and Distribution

Figure 3 presents the distribution of the twenty most common of the 1,926 errors encountered by students using BlueJ during laboratory sessions in the Spring term of 2003. Of the 42 different types of error encountered, the five most common errors account for 58% of all errors generated by students while programming: missing semicolons (18%), unknown symbol: variable (12%), bracket expected (12%), illegal start of expression (9%), and unknown symbol : class (7%). Typically, unknown

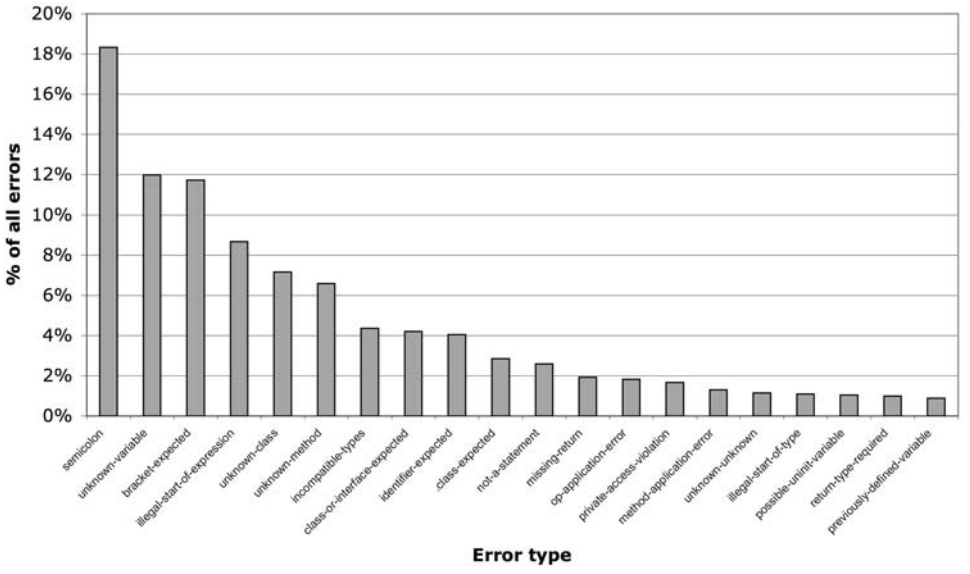

Fig. 3. Distribution of top twenty error types generated by students in the study.

variable errors are generated by typographic errors; unknown class errors are generated for similar reasons, as well as failing to import a package containing the class in question. Bracketing errors take into account all types of bracket—‘ ( )’, ‘{ }’, and ‘[ ]’ —and illegal start of expression errors are often caused by bracketing and missing semicolons.

The type and number of syntax errors students must deal with after compiling their code plays a significant role in determining their consequent behaviour. These errors are just one example of the constantly shifting *context* in which any one compilation event (or pair of events) takes place. This makes characterizing novice programming behaviour difficult, as the context is not fixed, but stateful, and that state is easily influenced by the students' own actions (intentional or otherwise).

4.2. Time Between Compilation Events

Our first explorations into the behaviour of the sample population involved examining the time students were spending between successive compilation events, as well as how often those events resulted in syntax-error free code. In Figure 4, each bar of the histogram represents a ten second window; 51% of all compilation events occurred less than 30 seconds after the previous event. At the same time, we can also see that roughly 20% of all compilation events involved more than two minutes of work time on the part of the student between compilation events.

This picture is only partially useful; at first glance, it implies that students tend to spend very little time working on their code between compiles. While it is true that students recompile often, there is more we can ascertain about why they are doing this, and when in their programming cycle these events occur.

The view of time between compiles presented in Figure 4 is *context-free*: given a measure of the time between two compilation events, it tells us nothing about the *result* of the first compilation, or the *result* of compiling any changes they then make. Did the student begin with a syntax error, and end up with code that was error-free? Did they have a missing semicolon error, and end up fixing it, only to find yet another error waiting for them?[2]

If a compilation event ends in a syntax error, we'll label it **F**; if it is error-free, we'll call it a **T** event. Now, each pair of compilation events can be characterized in one of four ways, as illustrated in Table 1.

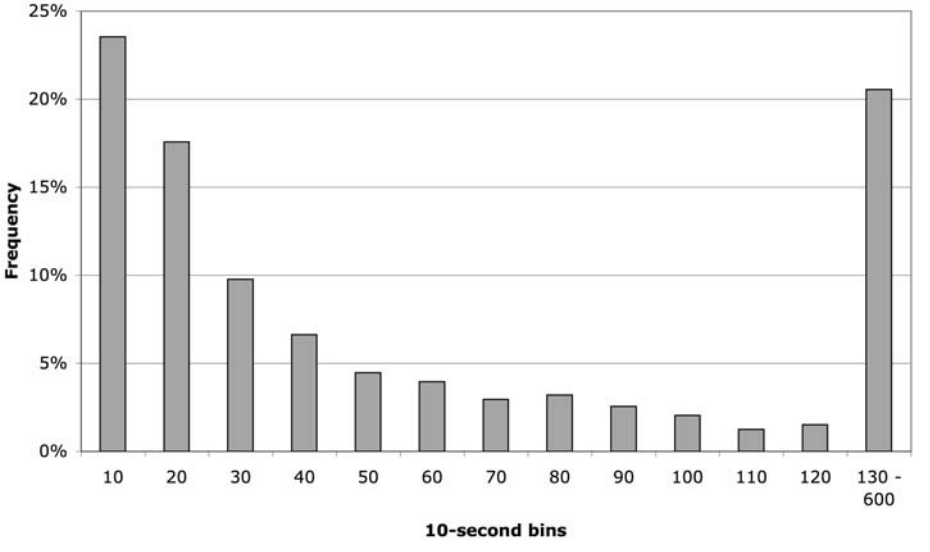

Fig. 4. Time between compiles, ten second intervals.

Table 1. Distribution of Compilation Event Pairs.

	T	**F**
T	30%	10%
F	16%	44%

[2]The developers of BlueJ believe, as a matter of pedagogic principle, that only one error should be presented at a time to beginning programmers.

Reading from left to right, we see that 30% of all pairs of events were successful, followed by another successful event (**T→T**). Similarly, 44% of all events were a syntax error followed by another syntax error (**F→F**). Because there are several ways to invoke the compiler in BlueJ, some of which recompile all files in a given project, it is possible that the **T→T** case is over-represented, and therefore we are skeptical of this number at this time; we have left it out of Figure 5 for this reason. The **F→T** case represents the last syntax error students fix in a given sequence of errors (although, because BlueJ only reports one error at a time, they would have no way of knowing this in advance of compiling their code). **T→F** is perhaps the most revealing of these categories, as it represents the first compile a student makes after they know they have syntactically correct code. Figure 5 tells us something very important about when students recompile their code quickly, and when they do not. When students have just encountered a syntax error, they are likely to recompile quickly. When students have just compiled their code successfully, 60% of the time they follow it by spending more than two minutes working on their code. While we do not know exactly what takes place in those two minutes, we can say that one-third of all compilation events that occurred more than two minutes after a successful compile involved substantial edits in the source code (100 + characters modified). We use the word ''substantial'' in comparison to the amount of work conducted by students correcting ''missing semicolon'' errors (one or two characters inserted, removed, or modified).

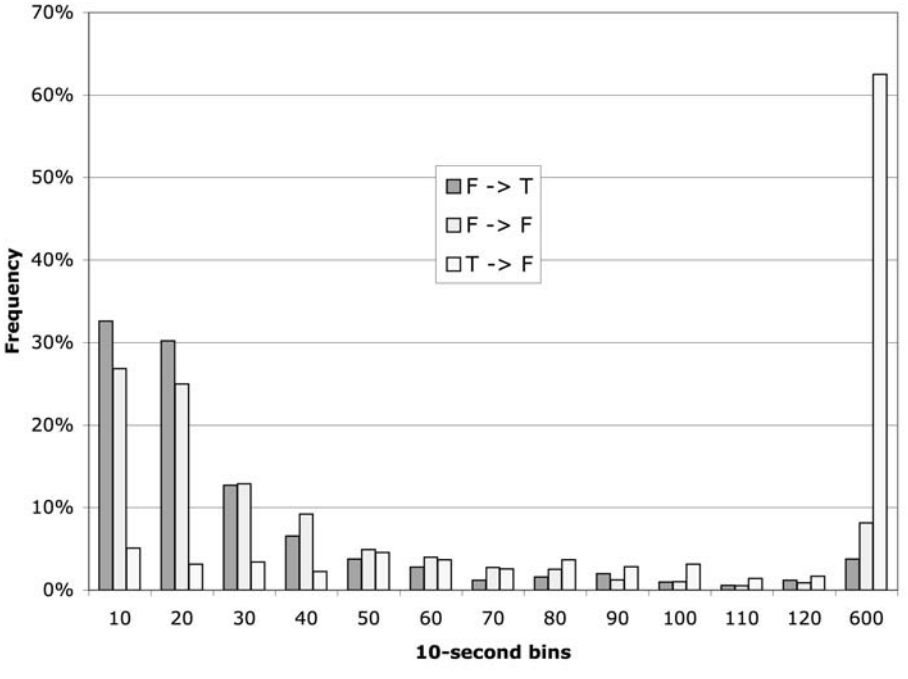

Fig. 5. Time between compiles for three types of compilation pairs.

*4.2.1. Time Between Events as a Function of Error Type*We have reason to believe that despite the fact that students are recompiling very quickly most of the time, they are not doing substantial work in that time. The majority of their code modifications seem to be coming on the heels of successful compilation events (Figure 5). What are they doing when they recompile 12 seconds after their previous compilation event?

We can look at the time and number of characters changed for each of the three most common error types: missing semicolon, unknown symbol : variable, and bracket expected. What we find is that, for the three most common classes of error encountered by students, very little time is spent fixing those problems, and a minimum of characters must be added, deleted, or changed in their source to enact those changes. Tables 2 and 3 summarize the number of seconds and number of characters changed (respectively) that follow the most common syntax errors encountered by students.

In the case of both time and characters changed, we can see that the mean is easily affected in significant ways by one or two large outliers (as given by the "max" value). In the case of characters changed, for example, we see that the mean always lies well outside the third quartile, telling us that the majority of all compilation events following a given error type is very tightly clustered around the *median*, and not the mean. Based on this, we can say that the most common syntax errors encountered by students are typically handled in less than thirty seconds, and require adding or removing only a few characters.

5. DISCUSSION

Our eventual goal is to determine if there are different, characteristic compilation behaviours exhibited by students learning to program. As can be seen, we can infer

Table 2. Seconds Spent After the Three Most Common Syntax Errors.

	Min	1Q	Median	Mean	3Q	Max
missing ;	1	5	8	20	16	265
unknown var	2	13	22	39	41	346
bracket	3	8	14	25	25	350

Table 3. Number of Characters Changed After the Three Most Common Syntax Errors (Seconds).

	Min	1Q	Median	Mean	3Q	Max
missing ;	1	1	1	5	2	148
Unknown var	1	1	3	8	7	191
Bracket	1	1	2	7	2	254

interesting things about our students' programming behaviour in the classroom using BlueJ. This aggregate analysis of the population provides a necessary context for the next steps in our work, which will involve investigating the behaviour of individuals within the population and exploring potential interventions for shaping novice compilation behaviour.

For example, Figure 6 shows the distribution of session lengths from the entire population on a per-student basis. A ''session'' represents the sequence of compiles from one class period. A typical student compiles (on average) 10 times per session. This collection of box-and-whisker plots supports that analysis, as the mean numbers of compilations per session for many of the students are centered around 10. It also tells us, at a glance, that there are also a few students who do not fit this typical behaviour who merit further examination. We might infer from this graph three types of behaviour based only on the number of times a given student compiles their code while programming in class: those students who are typical, students who compile more often than average, and students whose behaviour cannot be adequately described by this one graph.

Students 43, 62, and 63 all compile far more often in a single session (on average) than the typical subject in our study. These students might be more meticulous than their peers, insisting on checking that each line they write compiles. Another possible explanation is that these students are sloppy in their interaction with the BlueJ user interface: any interaction with their program source will cause the most recently reported error to disappear, leaving the students to wonder exactly what error they were about to attempt to fix. To rediscover what error was being reported, they must recompile their code.

A third possible explanation for this increased rate of compilation is that some students do not trust the error message reported by BlueJ. Instructors of the introductory Java courses observed this ''distrust'' in their own classrooms: in their experience, it was not uncommon to witness students recompiling their programs without attempting to ''fix'' or otherwise ''understand'' the error they received from

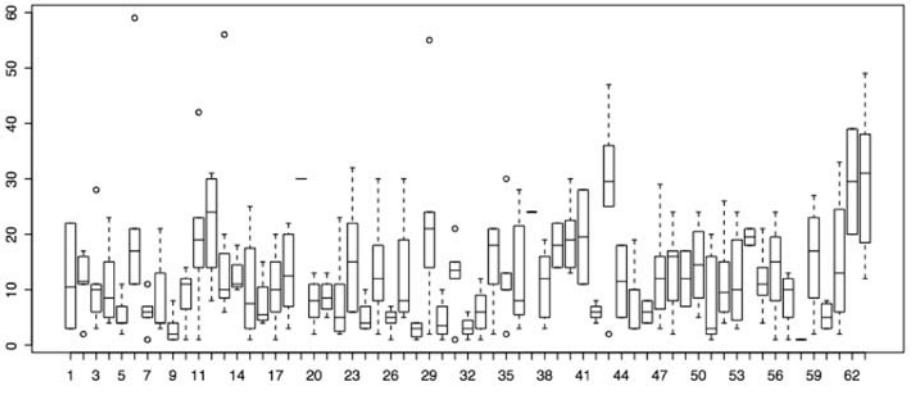

Fig. 6.

the compiler. Instead of taking in the error or consulting additional help regarding the meaning of the error, the students would immediately hit the ''compile'' button a second time. The instructors postulate that the students simply ''don't believe'' the compilation error BlueJ is reporting to them in many cases.

While it would be nice to believe that students are meticulous in their programming practice, the data collected in our study would seem to indicate some combination of awkward interactions with the BlueJ interface and students' belief (or lack thereof) in the messages reported to them by the compiler. These phenomena manifest themselves in the data as a repetition of errors (Table 4). Based on our sample population, there is a one in two chance that any missing semicolon error will be followed by another missing semicolon error. Unfortunately, 21% of the time the next error observed is the exact same error (no changes to the source, and the same error is reported on the same line number).

Also of interest are those subjects whose behaviour defies easy classification: for example, subjects 6, 11, 13, and 29 all are ''typical'' in their compilation behaviour, but exhibit one session that is, for the subject and the entire population, an extreme outlier: they compile forty, fifty, sometimes close to sixty times in a single one-hour laboratory session. We have no explanation at this time for this apparently exceptional behaviour.

A collection of box-and-whiskers plots like this are a crude visualization that allows us to quickly determine what kind of variance exists in our data on a per-subject basis. Based on this kind of visualization, we can see that it might be worth examining the distribution of errors each of these ''interesting'' students generated individually, or examining how much time they spent between compilation events. We did not begin with these kind of analyses, as they are expensive—our search for distinctive compilation behaviour in novice programmers will require a careful and reasoned exploration using a variety of data visualization techniques.

5.1. Shaping Behaviour

It would appear that the typical behaviour of students in our study is to make a significant number of changes, and then come back and correct all the syntax errors

Table 4. Error Types that, when Repeats Occur, Involve a New (Unique) Error, or are an Exact Repeat.

Error	new	repeated
Missing	28%	21%
Unknown var	30%	17%
Bracket	31%	20%
Illegal start of exp	34%	30%
Unknown class	35%	14%

Table 5. Tabular Summary of Figure 3, Distribution of Errors Encountered by Subjects.

Rank	Error Type	Ratio	Rank	Error Type	Ratio
1	semicolon	.1833	22	unreachable statement	.0078
2	unknown variable	.1199	23	else without if	.0073
3	bracket expected	.1173	24	package does not exist	.0057
4	illegal start of expression	.0867	25	missing body or abstract	.0042
5	unknown class	.0717	26	unclosed comment	.0031
6	unknown method	.0659	27	method ref. in static context	.0026
7	incompatable types	.0436	28	file I/O	.0026
8	class or interface expected	.0421	29	no return for void method	.0021
9	identifier expected	.0405	30	dereferencing error	.0021
10	.class expected	.0286	31	loss of precision	.0016
11	not a statement	.0260	32	empty character literal	.0016
12	missing return	.0192	33	unclosed character literal	.0016
13	op application error	.0182	34	inconvertible types	.0016
14	private access violation	.0166	35	illegal escape character	.0005
15	method application error	.0130	36	protected access violation	.0005
16	[uncharacterized]	.0114	37	type mismatch	.0005
17	illegal start of type	.0109	38	cannot assign to final	.0005
18	possible uninitialized variable	.0104	39	class public in file	.0005
19	return type required	.0099	40	bad modifier combination	.0005
20	previously defined variable	.0088	41	illegal character	.0005
21	unexpected type	.0083	42	array dimension missing	.0005

that resulted from the most recent addition of code. The majority of the errors the students encounter represent a minority of the total possible number of errors they might encounter: students are typically adding in missing semicolons, correcting spelling mistakes and typographic errors, and correcting unbalanced parentheses or brackets.

In framing our work in terms of behaviour, we can now begin to think about questions regarding the *shaping* of that behaviour. Can we modify the environment in such a way as to change programmer behaviour—perhaps encouraging them to make fewer ''missing semicolon'' errors, or be more attentive to the various kinds of brackets used to delineate code? For example, we might introduce improved highlighting of bracket pairs, or perhaps highlight places where semicolons *should* be when they are missing. Then, we would observe how student behaviour changes with this modified version of BlueJ: a simple, single-case experimental design (Leslie, 2002).

Even if changes such as those proposed appeared to ''improve'' novice programmer behaviour in some way, we don't want to condition students the way Tom Schorsch (1995) and his colleagues did in 1995 at the United States Air Force Academy. They developed CAP (Code Analyzer for Pascal), a tool that enforced institutional programming style and pointed out many common programming errors. Schorsch describes an unwanted side-effect of requiring all students to use CAP:

> We also wanted students to learn to use a correct programming style as a matter of habit. We assumed that with CAP continually telling the students to fix their programming style, eventually they would learn to do it correctly from the beginning. Unfortunately, we believe that many students began using CAP as a crutch to merely get by. Rather than incorporating the required programming style rules into their programming habits, some students ignore style altogether knowing that CAP will annotate their code with all the corrections that are necessary (Schorsch, 1995).

We want our students to be able to graduate from any initial programming tools we use in the classroom. Computer science educators like to think that students who really **learn** to program can later learn to program in *any* programming language. Our goal is that any behaviours our students learn in an initial, pedagogic environment improve their programming practice in the absence of such scaffolding—for example, in commercial tools they might encounter outside of the classroom.

Regardless of how we consider shaping novice programming behaviour, we must always remember to ask whether our actions are in the students' best interest. In behavioural terms, we still do not have a way of determining when we are observing ''good'' or ''bad'' programming behaviour. Is it ''good'' or ''bad'' that novices seem to be programming in long spurts followed by a rapid sequence of relatively simple syntax error corrections? If it isn't, what programming behaviours should we encourage instead?

6. FUTURE WORK

We have presented an overview of some broad analyses applicable to our subjects and their behaviour. From this analysis, a rough sketch of our novices' behaviour in the classroom has begun to emerge, and from even this outline we can begin to ask questions about the effect of the environment they are using on their programming behaviour. We have not, however, begun to unravel the question of whether different students exhibit different programming behaviour, and what we can observe to detect those differences, if they exist.

ACKNOWLEDGEMENTS

Many thanks are due the BlueJ team at the University of Southern Denmark, Deakin University, and the University of Kent; in particular, to Ian Utting and Damiano Bolla at Kent for providing compilation callbacks in the BlueJ extensions architecture, thus making this research possible. Thanks also to David J. Barnes and Mathieu Capcarrere for making their course available for study, to David Barnes, Sally Fincher, and Bob Keim for their continued conversation and feedback on this work, and to Dr. Richard Little at Baldwin Wallace College for his time and input regarding the statistical assumptions made herein. All errors remain my own.

REFERENCES

Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: a lightweight pedagogic environment for Java. In *Proceedings of the 33rd SIGCSE technical symposium on computer science education* (p. 137 – 141). New York: ACM Press.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*, 543 – 554.

Brown, P.J. (1983). Error messages: the neglected area of the man/machine interface. *Communications of the ACM*, *26*, 246 – 249.

DePasquale, P.J. (2003). Implications on the learning of programming through the implementation of subsets in program development environments. PhD thesis, Virginia Polytechnic Institute and State University.

Findler, R.B., Flanagan, C., Flatt, M., Krishnamurthi, S., & Felleisen, M. (1997). DrScheme: A pedagogic programming environment for Scheme. *Programming languages: Implementations, Logics, and Programs*, *1292*, 369 – 388.

Freund, S.N., and Roberts, E.S. (1996). Thetis: An ANSI C programming environment designed for introductory use. In *Proceedings of the twenty-seventh SIGCSE technical symposium on computer science education* (pp. 300 – 304). New York: ACM Press.

Gannon, J.D. & Horning, J.J. (1975). The impact of language design on the production of reliable software. In *Proceedings of the international conference on Reliable software* (pp. 10 – 22).

Green, T.R.G. & Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, *7*, 131 – 174.

Heeren, B., Leijen, D. & van IJzendoorn, A. (2003). Helium, for learning Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell* (pp. 62 – 71). New York: ACM Press.

Klahr, D. & Carver, S. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology*, *20*, 362 – 404.

Leslie, J. (2002). *Essential behaviourism*. Hodder Headline Group.

Litecky, C.R. & Davis, G.B. (1976). A study of errors, error-proneness, and error diagnosis in cobol. *Communications of the ACM*, *19*, 33 – 38.

Kolling, M., Quig, B., Patterson, A. & Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13.

McIver, L. (2001). Syntactic and semantic issues in introductory programming education. PhD thesis, Monash University.

Moulton, P. G. & Muller, M. E. (1967). DITRAN: A compiler emphasizing diagnostics. *Communications of the ACM*, *10*, 45 – 52.

Perkins, D., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989). *Conditions of learning in novice programmers*. Lawrence Erlbaum Associates.

Ramalingam, V. & Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Papers presented at the seventh workshop on empirical studies of programmers* (pp. 124 – 139). New York: ACM Press.

Rist, R. (1986). *Plans in programming: Definition, demonstration, and development*. Emperical Studies of Programmers.

Schorsch, T. (1995). CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education* (pp. 168 – 172). New York: ACM Press.

Spohrer, J.C. & Soloway, E. (1986a). *Analyzing the high-frequency bugs in novice programs*. Empirical Studies of Programmers.

Spohrer, J.C. & Soloway, E. (1986b). Alternatives to construct-based programming misconceptions. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 183 – 191). New York: ACM Press.

Spohrer, J.C., Soloway, E. & Pope, E. (1985). Where the bugs are. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 47–53). New York: ACM Press.

von Mayrhauser, A. & Vans, A. (1994). *Program understanding – a survey*. Technical report, Colorado State University.

Zelkowitz, M.V. (1976). Automatic program analysis and evaluation. In *Proceedings of the 2nd international conference on software engineering* (pp. 158–163). IEEE Computer Society Press.

NCSE	
Manuscript No.	NCSE105636
Author	
Editor	
Master	
Publisher	

Computer Science Education
Typeset by Elite Typesetting for

Routledge
Taylor & Francis Group

ELITE TYPESETTING TECHNIQUES LTD
www.elitetypesetting.com

QUERIES: to be answered by AUTHOR

AUTHOR: The following queries have arisen during the editing of your manuscript. Please answer the queries by marking the requisite corrections at the appropriate positions in the text.

QUERY NO.	QUERY DETAILS	QUERY ANSWERED
1	Please supply caption for Figure 6.	
2	Table 5 supplied, but not mentioned in text. Please confirm	