A first look at novice compilation behavior using BlueJ

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

Keywords: POP-II.A. novices, POP-III.D. pedagogic IDEs, POP-V.B. observation, POP-IV.E. CS-Ed research

Abstract

Syntactically correct code does not fall from the sky; for beginning programmers, the process that leads to a student's first executable program is not well understood. We have begun to explore the behaviors that students exhibit while authoring code by focusing on when and what they choose to compile. By examining these *compilation behaviors*, we have determined the most common errors encountered in-class by students using BlueJ in our introductory course on object-oriented programming at Kent, how they tend to program when in supervised laboratory sessions, and 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 it's use in the classroom.

Introduction

This paper presents a first look at novice compilation behavior of students learning object-oriented programming using the BlueJ pedagogic programming environment. The goals of our explorations of observable programmer behavior are to help inform the teaching of programming and the development of pedagogic programming environments. In this paper, we explore some gross behavioral 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 two, our data collection methods in section three, in section four an analysis of the data collected during the autumn term of 2003, a discussion of our work in section five, and close with future research directions based on these analyses.

1. 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 current understanding of *compilation behavior*. 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 behavior, or how we might shape that behavior to improve programming practice.

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). Characterizing 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 the researchers have already decided that there is nothing interesting about the process students go through in developing a program that is syntactically correct. These studies typically *begin* with the students' first syntactically correct programs, ignoring the observable behavior of novice programmers, or focus on extrapolating cognitive explanations for the behaviors observed.

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 they tend to appreciate the simpler interfaces, errors tend to persist over fewer compiles with "reduced" or "subsetted" languages, and 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.

Types and rates of error occurrence

Although not strictly labeled as *behavioral* studies, research into error message design and syntactic errors in programming languages are 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) provide 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 behavior 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.

Characterizing Novices

Perkins, Hancock, Hobbs, Marin, and Simmons (1986) observed young programmers working in LOGO, and based on their observations classified them 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 behavioral understanding and characterization of novice programmers should enable us to detect (using Perkins's categorization) stoppers and extreme movers automatically.

2. Methodology

We began our work with an automated observation of novice compilation behavior as it naturally occurred in classroom tutorial sessions. These sessions met once a week for one hour in a public computing lab on campus, are limited to approximately sixteen students, and are overseen by either a member of the faculty or graduate 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 behavior over time.

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.

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 stand at this point in the term.

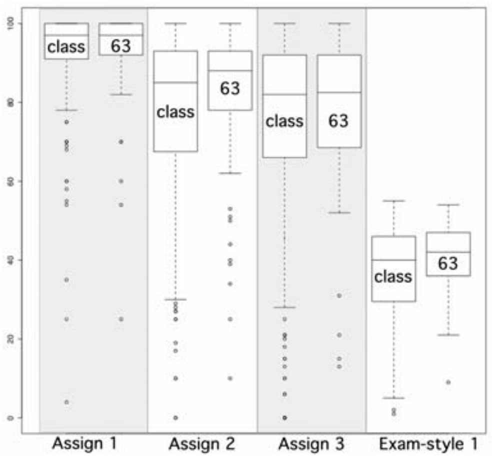

Figure 1: Grade distributions for assessments 1, 2, 3, and “exam”

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 our sub-

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

population does not perform, on the assessments given, better or worse than their peers. In this respect, we can take them to be representative of the class as a whole.

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 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 t he attendance sheet. This means a student can easily mark a 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 who agreed to take part in the study come from their actual interactions with the computer in class, and therefore are potentially more accurate.

3. 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 behavior apparently exhibited by the students is one where 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 behavior.

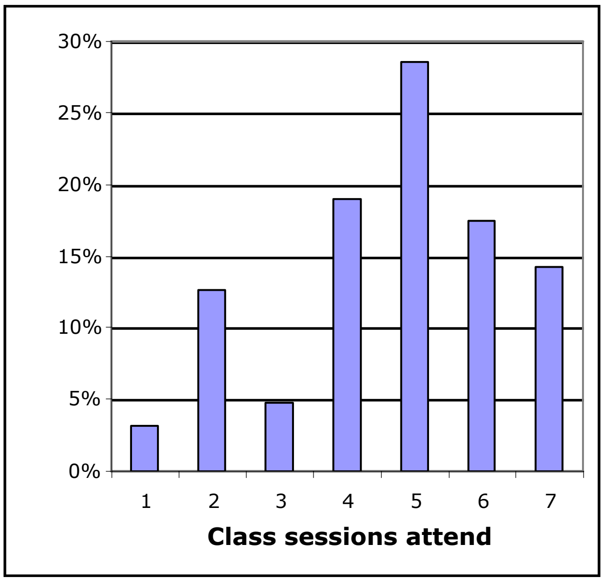

Figure 2: Attendance of students taking part in the study

Error types and distribution

Figure 3 presents the distribution of the twenty most common of the 1926 errors encountered by students using BlueJ during laboratory sessions in the Spring term of 2003; this data can also be found in table 4 at the end of the paper. 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 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 a student must deal with after compiling their code plays a significant role in determining their consequent behavior. These errors are just one example of the constantly shifting *context* in which any one compilation event (or pair of events) take place in. This makes characterizing novice programming behavior difficult, as the context is not fixed, but stateful, and that state is easily influenced by the students own actions (intentional or otherwise).

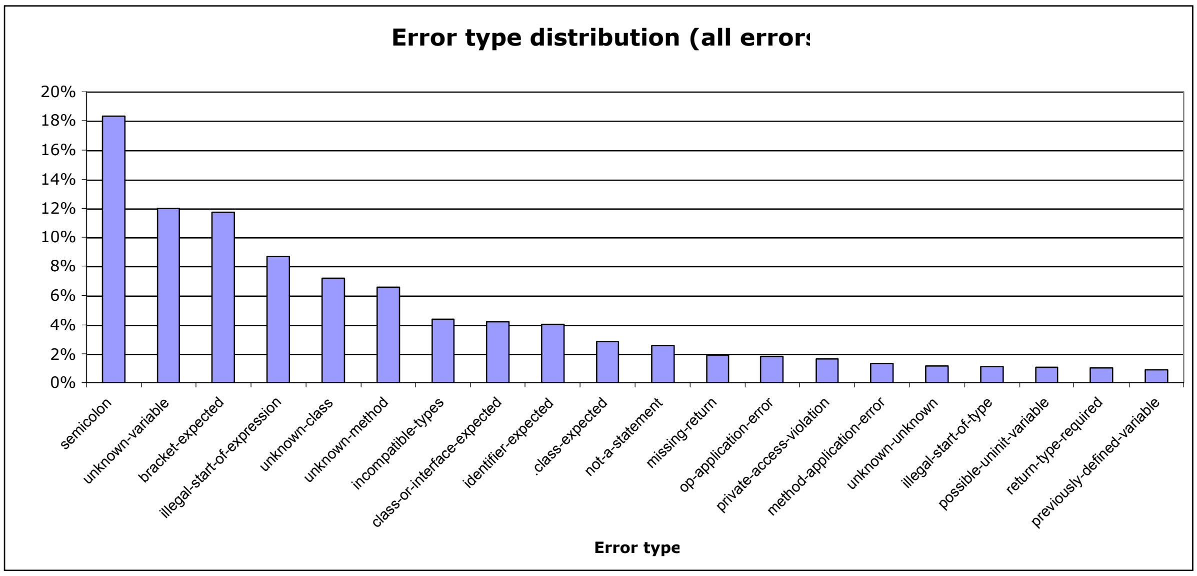

Figure 3: Distribution of top twenty errors generated by students in the study

Time between compilation events

Our first explorations into the behavior 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.

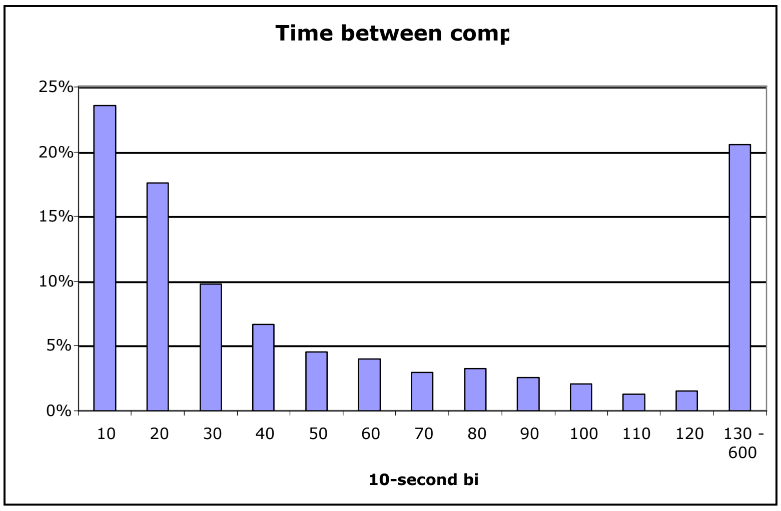

Figure 4: Time between compiles, 10 second intervals

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 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.

Table 1: Distribution of compilation event pairs

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

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

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.

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).

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.

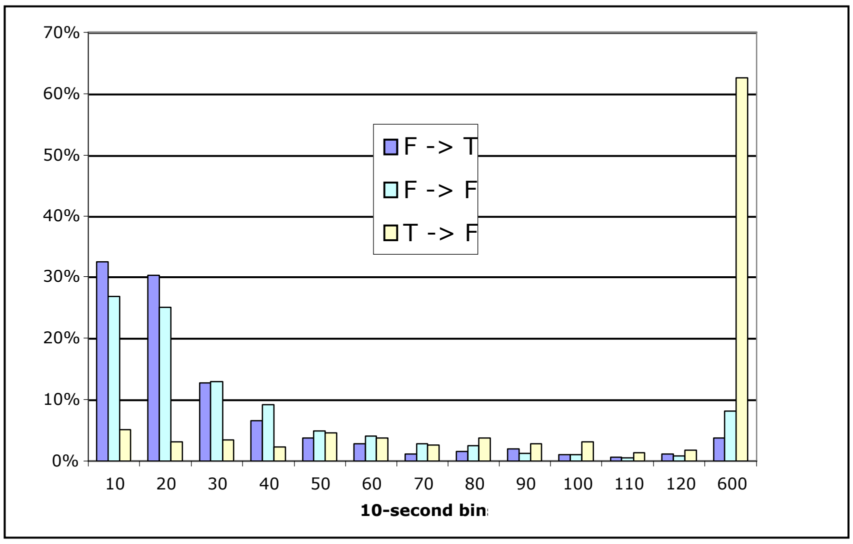

Figure 5: Time between compiles for three types of compilation pairs (F indicates syntax error, T indicates no syntax error)

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.

Table 2: Time spent after the three most common syntax errors

	Min	1 Q	Median	Mean	3 Q	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: Characters changed after the three most common syntax errors

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

4. Discussion

Our eventual goal is to determine if there are different, characteristic compilation behaviors exhibited by students learning to program. As can be seen, we can infer interesting things about our students programming behavior in the classroom using the BlueJ. This analysis of the all of the students who took part in our study sets the stage for future explorations examining the behavior of individuals as opposed to the population as a whole.

It would appear that the typical behavior of students in our study is to make a significant number of changes, and then come back and correct all the syntax errors 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 behavior, we can now begin to think about questions regarding the *shaping* of that behavior. Can we modify the environment in such a way as to change programmer behavior—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 behavior changes with this modified version of BlueJ: a simple, single-case experimental design (Leslie, 2002).

Even if changes like those proposed appeared to “improve” novice programmer behavior 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 think that students who really **learn** to program can later learn to program in *any* programming language. Our goal is that nay behaviors 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 behavior, we must always remember to ask whether our actions are in the students' best interest. In behavioral terms, we still do not have a way of determining when we are observing "good" or "bad" programming behavior. 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? It it isn't, what programming behaviors should we encourage instead?

5. Future Work

We have presented an overview of some broad analyses applicable to our subjects and their behavior. From this analysis, a rough sketch of our novice's behavior 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 and its effect on their behavior. We have not, however, begun to unravel the question of whether different students exhibit different programming behavior, and what we can observe to detect those differences, if they exist.

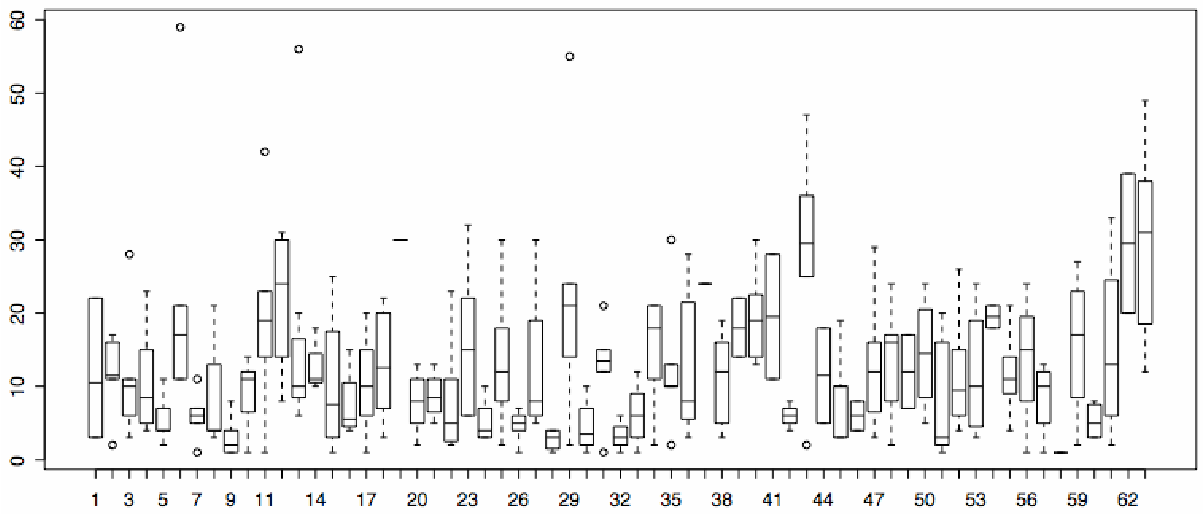

Figure 6: Distribution of session lengths on a per-student basis

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 number 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 behavior who merit further examination. We might, from this graph, infer three types of behavior based only on the number of times a given student compiles their code while programming in-class:

Normal behavior While there is variation between students, we can quickly see that most students compile between four and 18 times per session; there is significant overlap in this region for most students.

Deviant behavior Only four students have extreme outliers: Dave (6), Adria (11), Noel (13), and Jesus (29). These students are, at a glance, different in that they have apparently “normal” behavior within the population, but one session where they compiled their code many more times than at any other point in time.

Different behavior Three students clearly compile more often than the rest of the students: Clarisa (43), Sirena (62), and Rosana (63). In addition to these students who compile more than most other students, it is clear that there are some who only compile a few times in a given session, and they are very, very consistent in this; by inspection, Daisy (9), Florence (26), and Fidel (28) are examples of students that fall into this class of behavior.

A collection of box-and-whiskers plots like this is a crude visualization that allows us to quickly determine what kind of variance exists in our data on a per-subject basis. Based on these kinds of visualization, we can see that it might be worth examining the distribution of errors each of these “interesting” students generated individually, or examine how much time they spent between compilation events. We did not begin with this kinds of analyses, as they are expensive—students cannot be easily reduced to a single number for comparison, and therefore require visually comparing histograms, scatter plots, star charts, and other visual tools for exploratory data analysis before we know what (if anything) is significant in our search for distinctive compilation behavior in novice programmers.

Table 4: 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

Acknowledgements

Many thanks are due the BlueJ team at the University of Southern Demnark, 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 archetecture, thus making this research possible. Thanks also to David J. Barnes and Mathieu Capcarrere for making their course availble 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., and Stoler, B. (2002). DrJava: a lightweight pedagogic environment for Java. In *Proceedings of the 33rd SIGCSE technical symposium on Computer Science Education*, pages 137–141. ACM Press.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18(6):543–554.

Brown, P. J. (1983). Error messages: the neglected area of the man/machine interface. Communications of the ACM, 26(4):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., and 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*, pages 300–304. ACM Press.

Gannon, J. D. and Horning, J. J. (1975). The impact of language design on the production of reliable software. In *Proceedings of the International Conference on Reliable Software*, pages 10–22.

Green, T. R. G. and Petre, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174.

Heeren, B., Leijen, D., and van IJzendoorn, A. (2003). Helium, for learning Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 62–71. ACM Press.

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

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

Litecky, C. R. and Davis, G. B. (1976). A study of errors, error-proneness, and error diagnosis in Cobol. *Communications of the ACM*, 19(1):33–38.

M. Kolling, B. Quig, A. P. and Rosenberg, J. (2003). The BlueJ system and its pedagogy. *Journal of Computer Science Education*, 13(4).

McIver, L. (2001). Syntactic and Semantic Issues in Introductory Programming Education. PhD thesis, Monash University.

Moulton, P. G. and Muller, M. E. (1967). DITRAN: A compiler emphasizing diagnostics. *Communications of the ACM*, 10(1):45–52.

Perkins, D., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. (1989). *Conditions of Learning in Novice Programmers*. Lawrence Erlbaum Associates.

Ramalingam, V. and 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*, pages 124–139. ACM Press.

Rist, R. (1986). Plans in programming: definition, demonstration, and development. *Empirical 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*, pages 168–172. ACM Press.

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

Spohrer, J. C. and Soloway, E. (1986b). Alternatives to construct-based programming misconceptions. In *Proceedings of the SIGCHI conference on Human Factors In Computing Systems*, pages 183–191. ACM Press.

Spohrer, J. C., Soloway, E., and Pope, E. (1985). Where the bugs are. In *Proceedings of the SIGCHI conference on Human Factors In Computing Systems*, pages 47–53. ACM Press.

von Mayrhauser, A. and 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*, pages 158–163. IEEE Computer Society Press.