An Exploration of Novice Compilation Behaviour in BlueJ

A thesis submitted to

the University of Kent at Canterbury

in the subject of Computer Science

for the degree

of Doctor of Philosophy

By

Matthew C. Jadud

October 2006

Abstract

Our research explores the process by which beginning programmers go about writing programs. We have focused our explorations on what we call *compilation behaviour*: the programming behaviour a student engages in while repeatedly editing and compiling their programs in an attempt to make them syntactically, if not semantically, correct. The students whose behaviour we have observed were engaged in learning to program in an objects-first style using BlueJ, an environment designed for supporting novice programmers just starting out with the Java programming language.

The significant results of our work are two-fold. First, we have developed tools for visualising the process by which students write their programs. Using these tools, we can quickly obtain valuable information about their process, and use that information to inform further research regarding their behaviour, or apply it immediately in a classroom context to better support the struggling learner. Second, we have proposed a quantification of novice compilation behavior which we call the *error quotient*. Using this metric, we can determine how well (or poorly) a student fares with syntax errors while learning to program. This quantity, like our tools for visualisation, provides a powerful indicator for how much or little a student is struggling with the language while programming, and correlates significantly with traditional indicators for academic progress.

Contents

List of Figures

List of Tables

Acknowledgements

I have given up trying to list all of the truly wonderful people who have helped shape my education to date. However, all of you: thank you.

My parents and my wife's parents have been incredibly supportive in so many ways as Carrie and I pursued our respective Ph.D.s. There are not thanks enough.

I have had excellent supervision at each stage of my academic career: Tim Sullivan at Kenyon, Jonathan Mills, Ed Robertson, and Thom Schwen at Indiana, and Sally Fincher at Kent. My thanks especially to Sally—it was fun.

There are many whom I am fortunate to count amongst my friends, but Robin Blume-Kohout and Christian Jacobsen deserve special thanks.

And lastly, my supportive and loving wife, Carrie. I am glad and proud to be your husband.

Chapter 1

What students do

Our research focuses on what students do while learning to program. Anyone who has attempted to program (or even used a computer!) has almost certainly been baffled by a cryptic error message. Compilers—the tools that take a program written by a student and turn it into something executable by the system—are veritable gold mines for cryptic and confusing error messages.

It is our belief that there is a great deal to be learned by observing a student's behaviour in response to these messages, not their invisible mental state (their "understanding" or "comprehension"). At the least, their behaviour is a powerful indicator for instructors, allowing for timely interventions to aid their students' learning process. At best, a student's behaviour in response to these cryptic error messages my be an observable indicator of learning and programming mastery.

1.1 "They aren't thinking!"

"Students today aren't thinking when they're programming—they're letting the compiler do their thinking for them!" I first heard these words while a postgraduate at Indiana University Bloomington; a Professor was complaining to me about how his students simply couldn't write programs the way they used to. I thought about what I had observed in my own students, and agreed; students clearly liked to beat on the compiler.

It was during my last year at Indiana that I saw something that made me think twice. In our introductory course, a rich feedback mechanism had been hooked into the web-based handin system. Students would submit their programs, and they would be analysed (syntactically) as well as run against a set of unit tests. The results of these analyses was presented to the students; feedback ranged from the syntactic ("This problem called for use of a `cond` statement; your solution employs a nested `if`") to the semantic ("Your code passes 8 of 10 tests; have you checked all of the boundary conditions?"). The system kept all of the students' submissions.

Some students submitted their code four or five times before the deadline. Many more students submitted between 10 and 15 times. Some students, however, submitted their code to the web-based handin system *more than 100 times*. The web-based submission mechanism was slow and tedious to use (I wrote it); to submit an assignment over 100 times represented a significant amount of time and serious determination on the students' part.

These were not the actions of students who "weren't thinking." They received a quality of feedback from the web-based system that they didn't from their IDE; as they were programming in Emacs and compiling their Scheme programs on the command line, a tool that performed syntactic analysis as well as executing a unit test suite was very powerful. That said, their code often contained basic syntax errors—it appeared as if they were completely ignoring the compiler on their own machine in lieu of a more powerful, web-based tool. At the same time as we are attempting to explain this behaviour, we are not trying to defend it—these actions appear to be those of a student desperate for any help they can get. Or perhaps not; but desperate or not, these students were *thinking*, and making use of whatever tools they had available to complete their assignments successfully.

1.2 Of *stoppers* and *movers*

We were entirely too ready to characterise our students as unthinking, button-pressing zombies. After examining the programs submitted by students from Indiana during that one, unique semester (students were limited to a maximum of 10 submissions the next term), it was clear that students were engaging in some very complex and interesting behaviours. Unfortunately, we did not have the feedback from each submission available, so we had no way of knowing exactly what changes were made in response to what feedback. We had half a picture—and it was a very interesting picture indeed.

We began looking at research regarding the use of compilers and environments designed for novice programmers. We were particularly taken by work by Perkins, Hancock, Hobbs, Marin, and Simmons; in a study of novice programmers working in LOGO, they had characterised students broadly as either "stoppers" and "movers"—students who would quickly give up in the face of adversity while programming (stoppers), and students who would tinker their way around, through, or away from a problem (movers)[PHH+89]. In addition, they were particularly interested in the behavior of some movers—"tinkerers," as they were called. These students would poke, and tweak, and otherwise manipulate their code in a variety of small ways. Sometimes they would make progress towards their ultimate goal of a working program, sometimes not.

The notion that we could characterise the behaviour of our students became central to our thinking about students learning to program. We were familiar with many studies conducted through the 1970's, 1980's, and 1990's that explored the "mental models" students employed while reading and writing programs (see [RW97, Ris86], Chapter 2); the Perkins study was one of the first we had encountered that focused in on the *what* of student behaviour as opposed to the invisible and more elusive question of *why*.

1.3 Novice compilation behaviour

Novice compilation behaviour is the study of a student's interactions with their compiler while learning to program. This may be mediated by some kind of development environment, or it may be that they do their programming with a simple text editor and a compiler invoked on the command line. Authoring software involves more than just getting the code to compile successfully; as we have discovered in our own studies, however, this alone is often a majour accomplishment for our students. For this reason, we have focused our investications on only one part of the "edit-compile-run" cycle that so many students engage in while learning to program (Figure 1.1).

To explore this cycle, we instrumented BlueJ, a development environment designed for stu-

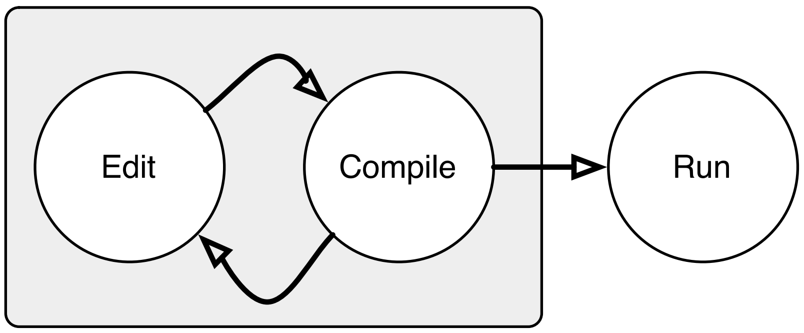


Figure 1.1: The edit-compile-run cycle.

dents learning to program in an object-oriented style using the programming language Java (Figure 1.2,[KQPR03]). Every time subjects in our study invoked the compiler, we took a snapshot of their program, as well as any messages emitted by the compiler. This resulted in a significant amount of data; roughly 42,000 distinct snapshots of student programs were captured, representing over 2000 "sessions." A session implies, at the bare minimum, that a student 1) opened their editor, 2) wrote some code (or edited an existing program), 3) compiled it, and 4) quit.

1.3.1 Exploring the data

We began sifting through these thousands of sessions with the intention of discovering some sort of "pattern" to novice programming behaviour. Sadly, one type of behaviour dominated all others: the overwhelming repetition of errors. Students seem to make lots, and lots, of syntax errors. Many students would spend 10, 20, 30 or more minutes wrestling with errors in the syntax of their program, and never actually execute what they had written (Chapter 3). This implied to us that many of our students were spending long periods of time failing to write code that would meet even the simplest of learning outcomes for our course modules.

We looked at the quantifiable aspects of this "thrashing" behaviour. As a student wrestles with a piece of code, we can quantify 1) the type of syntax error or errors they encounter, 2) where in their program those errors are reported, and 3) where they edit their program in response to these errors. These three quantities, observed over time, form the basis for what we

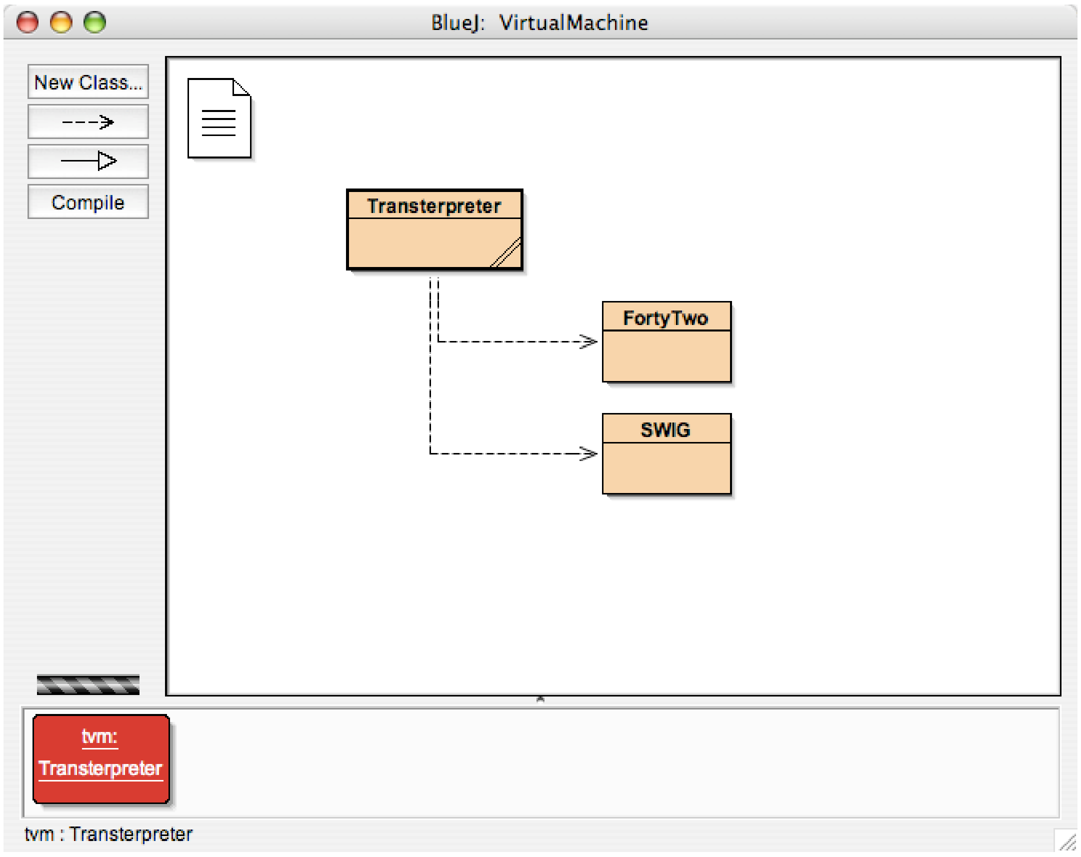


Figure 1.2: BlueJ, a programming environment for novices.

have called the **error quotient** (Chapter 4). A student's error quotient (EQ) is a value, averaged over a session, that tells us whether they are dealing quickly and efficiently with syntax errors (a low EQ), or if they are stuck, making changes that do not actually fix the syntax error in question, and possibly introducing new ones (a high EQ).

1.4 Tools for teaching

Our work suggests that introductory programming students at Kent with low error quotient scores tend to do well on traditional coursework assessments and end-of-year examinations. Similarly, it would appear that students who spend the semester or year wrestling with the syntax of the Java programming language do poorly on their assignments and examinations. While this may seem obvious—if they cannot write programs successfully, how can they possibly do well on programming assignments?—we believe this represents a significant outcome of our research.

It was clearly wrong to suggest students might be "letting the compiler do their thinking for them." Some students appear to be desperately hoping for some help or clue from the compiler that will help them understand the syntactic morass they are in while trying to accomplish what they believe should be a simple programming assignment. And, while further work is necessary, we believe the data we collected and the analytical techniques we developed for analyzing it may prove useful for automatically monitoring and providing feedback to novice programmers (and their instructors) while the initial teaching of programming is taking place.

Chapter 2

Review of Literature

The fundamental interactions between a programmer and their compiler are poorly understood. There have been several, paradigmatic shifts in programming languages since the 1950's—from assembly, to procedural/structured programming, to object-oriented programming and design—yet the edit-compile-run cycle remains. Despite this constancy, there has been little systematic research regarding the edit-compile-run cycle.

Our research has demonstrated that a great deal can be learned by studying how a programmer interacts with their code over many successive compilations. Throughout the literature on environments, tools, and languages, as well as the research regarding novice programmers, this interaction is implicit and rarely acknowledged directly.

2.1 The tools of the trade

While the fundamental edit-compile-run cycle has changed little over the last twenty years, some of the underlying technologies have changed remarkably. Compiling and executing a program in the punched-card era could take all day; now, it takes milliseconds. Despite massive increases in the computational power casually available to instructors and students, the tools used to write and compile programs have changed minimally in the last fifty years.

2.1.1 Punched cards forever

If instructors today claim that their students compile their programs too early, too often, they are themselves to blame. The race to provide ever-faster compilation began in the 1960's, and continues to this day. In 1967, the Univac 1107 system running at Case University (later to become Case Western Reserve University) in Cleveland, Ohio, boasted an impressive turnaround-time of 12 hours on most jobs. Lynch reports that "A user can submit a deck at 8:00 a.m., have it on the input tape by noon, his job completed by 12:35, and the output returned to him by 5:00 p.m. in the evening."[Lyn67] Compared to students today (who work in laboratories equipped with many machines, each thousands of times more powerful than the Univac 1107), the 9-hour turnaround time seems to approach infinity; but Lynch goes on to say that "It should be noted that only 10-15 percent of the runs are the first of the day, 85-90 percent are repeats, and about a third of the runs have a circulation time of less than 5 minutes. It is often possible to debug a moderate size program in less than an hour by gaining frequent access to the computer."

This early report on programmer behaviour seems to imply that having access to rapid feedback from a computer regarding the syntax and semantics of a program was a valued interaction style. Not only is this behaviour exhibited by students today, but tools like Eclipse[1] continuously re-compile the programmer's code, highlighting errors as the programmer develops their code in real-time; this is the "rapid compilation cycle" taken to its natural limit.

Tools to teach with

The WATFOR (WATerloo FORtran) system from the University of Waterloo[SGM$^+$67] and the DITRAN (DIagnostic forTRAN) compiler from the University of Wisconsin Madison[MM67] were probably the first compilers for "mainstream" languages to be written with the student in mind. Both WATFOR and DITRAN lived and thrived for years; indeed, the WATFOR compiler grew into a family of tools over the next decade at the University of Waterloo, constantly evolving with the advent of new hardware and environments.

> Comprehensive error diagnostics are provided at both compile time and run time. WATFOR detects the standard syntax errors during the compilation of a source pro-

[1]http://www.eclipse.org/

> gram. For each such error detected, a coded message is printed which relates the error detected to the statement in which it occurred by means of a serial line count.
>
> An important feature of the WATFOR compiler is that it generates object code which detects certain logical errors during the execution phase of a FORTRAN IV job. In this way, inconsistencies such as undefined variables, subscript values not within the bounds declared for a subscripted variable, and the redefinition of constant parameters within a called subprogram, are detected at run time. Run-time error messages have the same format as compile-time messages.[SGM$^+$67]

Ignoring when this was written, and the particular languages involved, this quote might describe any programming environment in existence today. The authors describe compile-time and run-time errors, consistently formatted, linked back to a source line number in all cases. Programmers today working with GCC (the Gnu C Compiler) don't even get line numbers in run-time errors; they have to invoke a separate debugger process and specifically set breakpoints in their code to get debugging information this useful.

In addition to describing programming environments similar to those we work with today, Moulton and Muller's paper about DITRAN is also interesting due to the presence of some simple usage statistics at the end of the paper. While their inclusion is almost presented as an afterthought, they provide some critical insight into how students and other programmers on campus interacted with the compiler. First, they provide us with some general statistics, which indicate the size and number of programs they observed (Table 2.1). Furthermore, based on this information, it would appear that most programs are short, and probably make heavy use of intermediate variables for mathematical calculations (Table 2.2).

We are most interested in their reports of compilation errors (Table 2.3). Based on our own analysis of Java programs and syntax errors, it would appear that common linguistic idioms (like the semicolon at the end of a line in C and Java) and the misspelling of variable and method names are a common source of syntax errors. Similarly, the most common statement in the language observed by Moulton and Muller is also the most frequent source of syntax error.

Table 2.1: General observations regarding DITRAN usage

Number of students	234
Number of batch runs	208
Total number of programs	5158
Average compilation time per program	3 seconds
Average execution time per program	3 seconds
Average core requirements per program	793 words
Average number of statements per program	38

Table 2.2: Percent of total errors for common DITRAN errors.

Type of Statement	**% of total**
Arithmetic Assignment	45.6
WRITE	9.1
DO	8.5
IF	7.4
GOTO	4.9
FORMAT	4.8
END	3.1
READ	3.0
DIMENSION	3.0
CONTINUE	2.7
All others	7.9

Table 2.3: Syntax errors in DITRAN programs

Number of programs with compilation errors	1859
Average number of compilation errors per program	3.8
Statement type	**% total errors**
Arithmetic Assignment	26.0
Statement format and sequence	22.0
Identifiers	14.8
DO Statements	6.8
General Punctuation	6.6
I/O Statements	5.6
Reference and Definition	5.4
FORMAT Statements	4.6
GOTO, IF Statements	2.4
All others	5.8

Looking forward

There are some similarities between programming systems developed in the mid to late sixties and those in use today. However, a significant difference between computing in the late 1960's and today is the cost associated with every program written and executed. In the sixties, programs were punched into cards, and output was through a line printer. Cards and paper both cost money, and supporting an entire school—and a department of computer science, in particular—was expensive.

> The major problem of student running now lies in he sheer bulk of cards and paper needed. It is expected that the BRUIN compiler, when coupled with a number of high-speed remote work stations to keep all that paper away from the computing laboratory, will at last bring the growing monster of student programming under control.[Mun69]

Munk's observation regarding the cost of programming is interesting for more than its financial insights. What is important here is that Munk is looking forward to technologies that will not only reduce the costs of student computing at the university, but also allow computer science instructors to make programming accessible to more students. While the "growing monster of student programming" was tamed long ago, the basic practice of writing code in plain text, compiling that code, and executing it has not changed.

2.2 Editing and editors

During the 1970's, with increasingly powerful computers and networks, the question of how one writes a program shifted from the economics of punched cards to the utility of the on-line editor. Nearly seven years later, the WATFOR project was still alive and well at the University of Waterloo, and continued to set the pace of innovation regarding educational tools for computer science.

> Another area of research has been student-editing and job-entry systems. A simple editor, the WIDJET system, was implemented and augmented with commands which allow submission of jobs for execution. ... One of the prime considerations was that students could use the editor to write their first computer program after their first lecture. **We feel strongly that novice students should run programs immediately rather than possibly losing interest during several "theoretical" lectures.** The command language was designed to be very simple to learn. Basically, the editor is a context editor whose commands apply to a current line. The current line may be changed by repositioning commands. **The original editor could only move the current line pointer forward in a file; in order to move backward the pointer had to be repositioned at the beginning of the file. In response to student suggestions, the editor was modified so the pointer could be moved either forward or backward.**[CDGW76] (Emphasis added)

In looking back at Cowan's discussion regarding software development on the DEC PDP-11, it is important to note that the developers of educational technology felt that the ability for students to try ideas out, quickly and easily, was important. That we would hear lecturers today say that students have too much power at their fingertips—because they recompile their programs *too often*—is an interesting turnabout. In truth, tools are available (in the form of powerful personal computers) that are not being exploited to their full effectiveness in their classroom.

To illustrate the underutilization of resources available to us as educators, consider that students had to tell the developers of the WIDJET editor to allow the user to scroll backward through a program listing. This is a seemingly obvious feature; computer users are accustomed to non-linear editing of text, sound, and video on modern computers. Just as this seemingly obvious feature was missing from an early programming environment for novices, it is likely that many other useful features are missing from today's environments—features that there are now plenty of spare CPU cycles to support.

2.2.1 Structured editors

Along with big hair and leg warmers, the eighties were the golden age of *structured editing*. Structured editors restricted programmers to editing within the grammar of the language—they dodged the issue of confusing syntax and reporting syntax errors simply by making it impossible to make these kinds of mistakes. These "smart" or "intelligent" editors typically took the form of highly structured (and sometimes visual) systems, or would instead make use of the blossoming knowledge of AI researchers to provide an "expert system" that would guide the user, helping them catch and correct mistakes.

In 1981, Peter J. Denning, then President of the ACM, wrote in his "President's Letter" that "A new breed of editors is emerging. They will revolutionize programming and document preparation. They are 'smart': not only because they talk back to their clients, but because they intimately understand their clients' tasks and can assist them in doing them correctly. These editors are not people, they are programs." This is a captivating idea—as the notion of powerful, timely, and relevant computer-aided instruction is—that the computer will make learning easier by observing your context and providing startlingly relevant help along the way.

The Cornell Program Synthesizer was a well-known structured editor in it's time[SJ81, TR81]. Very clearly, the authors state their beliefs: "Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint." There is no *research* reported by the team developing the Cornell Program Synthesizer that indicates that programmers think about their programs this way. This trend continues for some time; fifteen years later, in 1996, Brad Myers decries this disconnect between the practice and research of computing education:

> Over the past twenty years many research studies have discovered useful information about novice programmers, and identified good and bad aspects of today's programming systems, both visual and textual. However, this body of research is widely distributed throughout the literature and is not well organized, making it difficult to use in guiding the design of new systems. The result is that these research results generally have not been systematically fed back into the design of new programming systems. Instead, the design of new languages and environments has most often been driven by technical objectives, such as ease of parsing, ease of generating fast code, closeness to the machine, ease of proving correctness, etc. Even systems that were designed for novice users or for teaching have not attempted to broadly survey this body of research before making critical decisions about the metaphor or model that the language is based on, the notation that is used in the language, and the environment.[PM96]

Myers and Ko have continued to explore this disconnect between programmers and their environments. In their study "Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks," they focused on how expert programmers go about *understanding* their programs. They found that experts, when faced with a maintenance task, spent their time engaged in any one of eight behaviours: reading a segment of code, editing code, navigating between dependencies, searching for names (using search tools in the IDE), testing, reading documentation, switching between environments (eg. the IDE and application), and reading task descriptions.

One fifth of the programmers' time was spent reading code; another fifth was spent editing, and one sixth of their time was spent navigating between relevant segments of code. Based on their analysis, Myers and Ko propose a structured editing environment that explicitly supports programmers in these high-level tasks[KAM05]. Their experimental IDE is task-based, and aids programmers in tasks they carry out most often—tasks that were derived from empirical research of real programmers engaged in authentic tasks.

The work done at Cornell and the work carried out by Myers and Ko both resulted in a structured editor—a tool that constrained the way users interact with code, and help them in the task of writing syntactically and (hopefully) logically correct code quickly, the first time. But there is a massive difference between these projects. The developers of the Cornell Program Synthesizer were of the mind that ”programs are not text; they are hierarchical compositions of computational structures...” This was not, as reported, based on any empirical research into the behaviour or cognition of programmers; instead, it was based on a deeply held belief by the authors about what programming was, and how people did and should engage in the practice of writing programs.

Myers and Ko took a very different approach; they began by observing what real programmers do when confronted with an authentic task. They set their subjects a task, and looked to see how they could support their subjects best through the use of software. When they were done with their structured editor, it was tailored to the tasks of editing and maintaining code—based on rigorous and repeatable study of programmers engaged in an authentic maintenance task.

The lesson drawn from the research surrounding the structured editing of code is simple: if the state of initial programming environments is to be improved, we need to understand how learners go about the programming task, characterize their behaviour, and propose testable solutions that target specific problems based on observable phenomena.

2.2.2 Initial programming environments

Structured editing environments mostly died along with the 80’s; however, the notion that programs might be written in more than just a text editor was a compelling one. Over the course of the 1990’s, a number of programming environments intended for novice programmers were developed, but few in conjunction with a research agenda as rigorous as that exemplified by Myers and Ko. That said, they were generally developed by practising educators who, informally, continuously evaluated their product and the way their students use it.

Thetis

Thetis, an environment for learning the C programming language, is an exemplar of initial learning environments intended for use by novice programmers.

> In 1991, the Stanford Department of Computer Science decided to abandon Pascal in its introductory computer science courses and to adopt ANSI C as the language of instruction. We based this decision on several factors: the inadequacy of standard Pascal as a base for teaching modern programming concepts, the need to prepare our students for more advanced courses in which they will be expected to use C for programming projects, and increasing pressure from students and faculty throughout the School of Engineering for instruction in a language that has become the industry standard. We also believe that it is not reasonable to expect students to learn C on their own; students must receive instruction in C in order to become good C programmers. C has several known deficiencies that make it a challenging language to teach.[Rob93]

Thetis was developed to address what Roberts et al. perceived as the particular needs of students learning C for the first time. In particular, they found a number of problems (from a novice's point of view) with the existing C tool chain:

> **Shortcomings of existing compilers**
>
> The idea that weaknesses in the programming environment complicate the learning process for beginning students is not a new one; such shortcomings have been recognized in earlier papers. The underlying problem is that most commercial compilers—particularly for languages like C that cater to a large audience of programmers—are designed for experts rather than novices. As a result, most of these compilers are poorly suited to student use.
>
> In our experience, commercial compilers suffer from the following problems when used in an introductory course:
>
> - *The error messages generated by commercial compilers are often uninformative and sometimes misleading.* New programmers tend to make certain mistakes more

often than others. For example, omitting a required semicolon or right curly brace is a very common error in C programs. Unfortunately, compilers sometimes respond to this error by reporting a seemingly unrelated syntax error several lines below the actual mistake, presumably because the parser does not detect the problem until that point in the source file. Expert programmers understand the problem and know where to look; novices are completely baffled.

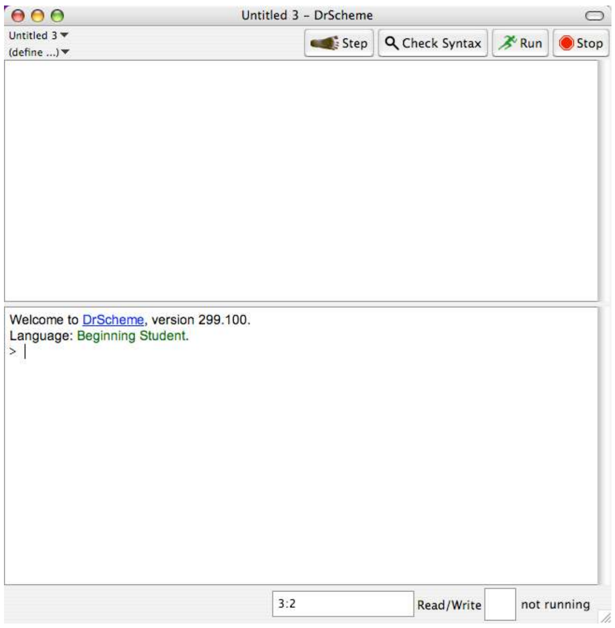
- Interactive debuggers typically require students to understand advanced concepts before they are ready to assimilate them...
- Commercial compilers offer a bewildering set of options and special features that are useless to the novice and occasionally cause consistency problems...

The authors of Thetis reduced the complexity of the C programming language in the simplest and most direct way possible: they rewrote the tools used to write, compile, and execute programs written in the language. They then structured their instruction to take advantage of the improvements they had made in the language's implementation. The combination resulted in what must have been a greatly improved environment for the novice C programmer.

DrScheme

Like Thetis, DrScheme provides a simplified environment for the novice programmer exploring the Scheme programming language. The default view of the environment provides a space for code to be defined, for the student to interact with that code, and four buttons: "Step," "Check Syntax," "Run," and "Stop" (Figure 2.1). Three of these (Check Syntax, Run, and Stop) should be obvious as to what they do in the context of a programming environment; "Step," however, executes a student's program one expression at a time, allowing them to see exactly how their code is evaluated[Gro].

DrScheme provides a series of "language levels" for novice programmers[GF03, Fla02]. These language levels are each subsets of the Scheme programming language; they limit the syntactic forms available to the novice programmer, and provide error messages that can be much more specific in the context of a reduced language. From the tool-writer's perspective, knowing certain linguistic constructs are (or are not) possible in a given syntactic context al-



```
Welcome to DrScheme, version 299.100.
Language: Beginning Student.
> |
```

Figure 2.1: The DrScheme ILE.

lows for clearer and more specific feedback to be generated for the novice programmer. While there are four distinct levels (or sub-languages) provided for students ("Beginning Scheme," "Intermediate Scheme," "Intermediate Scheme with Lambda", and "Advanced Scheme"), the concept is very much the same as that implemented in Thetis: students should learn to program using simplified versions of the full programming language.

This notion of "subsetting" a language was the subject of recent dissertation work by Peter DePasquale; in "Evaluation of Subsetting Programming Language Elements in a Novice's Programming Environment," the conclusions reported were that reducing the complexity of the environment, and also the language itself, were good for novice programmers[DLPQ04]. Based on focus group discussion, students felt the environment eased their transition into programming, and appreciated the fact that they were restrained from moving too quickly or wandering too far when attempting to solve a programming problem. From the data collected during the process, students tended to compile their programs less often than those using a professional development environment (Microsoft Visual Studio .NET), and fared no better or worse than their peers in making the transition from an initial learning environment to a professional IDE.

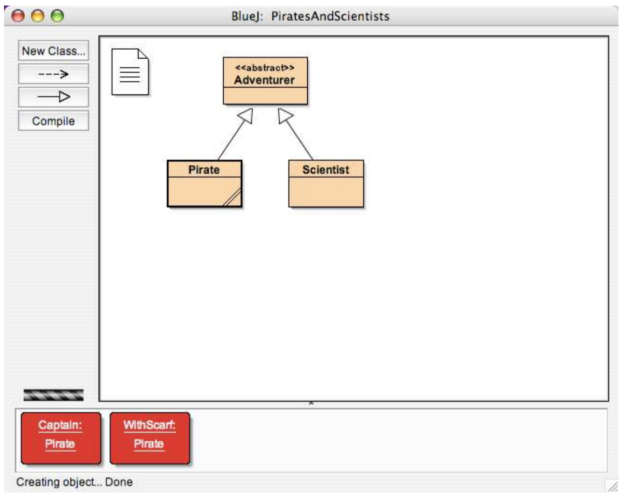


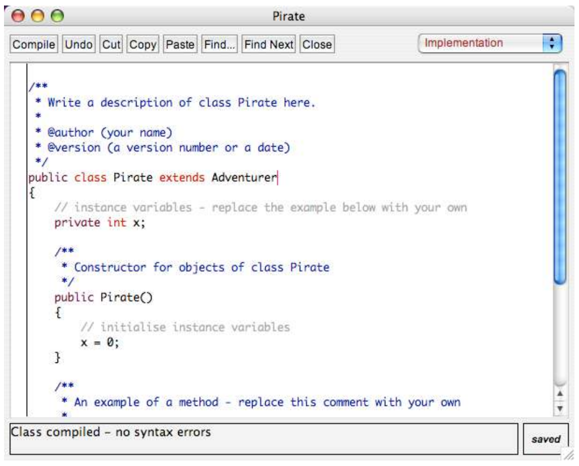
Figure 2.2: The BlueJ Diagram window.

BlueJ

While both Thetis and DrScheme attempt to hide the complexity of the full language from the novice, BlueJ takes a different approach to introducing students to object-oriented programming in Java. Simply stated, both Thetis and DrScheme are text editors with improved tools for manipulating and reporting on programs written by students. While the BlueJ environment does have a simplified/minimal interface, much like Thetis and DrScheme, the BlueJ environment does not attempt to simplify the Java programming language, or the feedback that comes from the underlying tool chain (`javac`) as provided by Sun Microsystems. Instead, BlueJ provides a basic visualization of students' programs in the object-oriented paradigm. Using a subset of the industry-standard UML notation, students interact both with the visual (structural) representation of their program as well as the textual (logical) representation of their program[KQPR03].

Figure 2.2 presents the diagrammatic view of code that students are greeted with when they begin working in BlueJ. By double-clicking one of the classes (denoted by an orange rectangle in this diagram), students can then edit the text of their program (Figure 2.3).

While BlueJ provides visual abstractions for the structure of their program, there is nothing particularly spectacular about how BlueJ supports the novice writing Java code. The language



```
/**
 * Write a description of class Pirate here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Pirate extends Adventurer
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class Pirate
     */
    public Pirate()
    {
        // initialise instance variables
        x = 0;
    }

    /**
     * An example of a method - replace this comment with your own
     *
```

```
Class compiled – no syntax errors
```

Figure 2.3: The BlueJ code window.

is not restricted in any way, nor are the error messages tailored for the beginning programmer. The only simplification provided by BlueJ is related to the volume of errors reported; if students were to invoke the Java compiler (`javac`) on the command line, it is likely that many errors would be reported simultaneously. By comparison, BlueJ only reports one error to the programmer at a time. Therefore, a typical student session when working with BlueJ will involve the authoring of code, compiling of that code, and then the student responding to the first syntax error found in their program.

Once a student has eliminated all the syntax errors from their code, they can then return to the visual representation of their program and interact with it. The work presented in this thesis focused on students' interactions with the errors reported by the compiler, and ignored the interactions that took place at run-time after successfully eliminating all the (syntactic) errors from their programs. The way novices interact with syntax-error free code has been the subject of a significant amount of research. For example, the work of Perkins et al., of Bonar, Soloway, and Spohrer[SBE83, SS86b], and of McCracken et al.[MAD^{+}01]. all nicely frame students' interactions with syntactically correct code. These work of these groups are discussed further in sections 2.2.3, 2.2.4, and 5.4, respectively.

2.2.3 Initial programmers

The work of Perkins, Hancock, Hobbs, Martin, and Simmons addressed questions regarding motivation and behaviour: why do some students plough on in the face of errors and adversity, and others quickly throw their hands up in defeat and ask for help? Informally, many of the behaviours described by Perkins et al. have been observed in students attempting to correct syntax errors in their programs. This can be contrasted with the research carried out by researchers like Bonar, Soloway, and Spohrer: their work largely ignores the syntactic struggle that leads students to the very programs being studied. There is no easy way to reconcile the gap that exists between the behavioral and cognitive research carried out to date.

Stoppers and movers: towards the concrete

"Conditions of Learning in Novice Programmers" describes some of the work carried out in the Educational Technology Center at Harvard University by Perkins, Hancock, Hobbs, Martin, and Simmons[PHH^{+}89]. Their explorations focused on how students can learn to program (and develop rich problem-solving strategies) *without* carefully designed instruction informed by cognitive science and the research on learning and programming. A cynic might say that it would be just as interesting to study how students learn to program *despite* all the cognitive theory that is incorporated into our curricula.

> The point remains that, even without carefully designed instruction based on cognitive science and in-depth study of the particular domain, some students learn. What might be called "bootstrap learning," where students go significantly beyond what they have been taught, does occur. Not only is this apparent from the occasional student with a flair for programming, but our clinical inquiries teach us that even students who are not doing that well overall can occasionally invent programming tactics for themselves.

Perkins et al. would sit with students engaged in programming tasks, working in either LOGO or BASIC. While they did not employ think-aloud protocols, they would ask questions of the students (and sometimes answer simple questions that came up as a result) while they were working through problems at the computer. For example, they discuss their observation of "close tracking" of code by novice programmers—this is a process which students in their study were taught to use, which involves carefully reading their program to better understand precisely what it does (as opposed to what they think or hope it will do). While their research differed from that presented here, there are significant similarities in the behaviours observed. The following, extended quotation captures one of the interactions observed by Perkins et al. that is so reminiscent of the data we present in Chapters 3 and 4.

> Although in principle close tracking is a mechanical procedure, in practice it often proves a source of difficulties. Students commonly neglect to do it when they need the information close tracking provides to untangle a problem. For example, Fred was working on the square-of-stars problem in BASIC. He had written the following code:

```
10 n$ = "*"
20 input "how many stars per side";n
30 for x = 1 to n
40 print n$;
50 print n$
60 print n$;
70 next x
```

> When he ran the program he got this output:

```
**
***
***
***
*
```

> He repaired the program by changing line 40 to:

```
40 print n$,
```

so that the cursor would move to the next print zone. When he ran the program again he got this output:

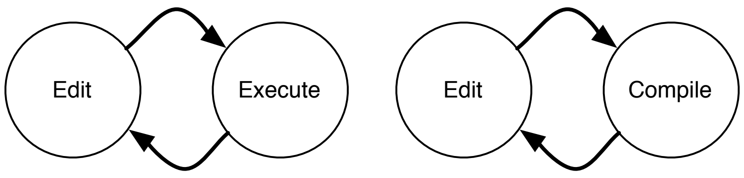
```
*          *
**         *
**         *
*
```

Now the program was at least *looking* more like two sides of the square and he seemed to think that he was getting closer to the solution. Close tracking might have helped Fred to realize that his approach could not possibly work in the general case, because it allowed the width of the figure to be determined by print zones instead of n, the desired width. But, rather than track to see what the program was actually doing, Fred persisted in making many small repairs to lines 40, 50, and 60; he became stuck in a cycle of diagnosis and repair that could never get him closer to the correct programming solution.

Perkins et al. say that Fred does not do a good job of "close tracking" in this situation—that is, he doesn't carefully step through his code in an attempt to understand how it really works. Instead, Fred engages in a series of quick edits in an attempt to fix the program as it stands. As an expert one can clearly see that Fred misunderstands the problem, and his process is not likely to get him to a correct solution. As an external observer, the struggle Fred is going through is clear, even if it might not seem that way to the novice.

This "just one more tweak" behaviour that students seem to employ in the Perkins study strike us as being remarkably similar to the behaviours observed in our own studies, as students struggle to write a syntactically correct program in Java (Figure 2.4). While the languages and situation differ, the fundamental behaviour seems remarkably similar:

In the Perkins vignette, the student was observed making quick changes to his program that produced known results that *appeared* to bring him closer to the desired solution. Likewise, in our own observations, students will often make quick changes in the face of syntax errors: if a semicolon is reported missing, they will add a semicolon. It doesn't matter that the code in



(a) Perkins's Students (b) Our students

Figure 2.4: The fundamental edit cycle in Perkins's studies and our own.

question is so syntactically incorrect that it is clear the student has no basic grasp of the grammar of the language—they'll follow the compiler's advice quickly (in a handful of seconds), perhaps hoping that this will be "the fix" that gets them moving again.

One might characterize the behavioral parallel observed between these populations as one of "hack-n'-act" (Figure 2.5). In both, as instructors, we might be inclined to say there is a missing step: "think." Harsh as this sounds, this is exactly the phenomenon Perkins et al. were curious about: why, in the face of so much research and cognitive psychology, presented as best we know how to our students, do they persist in developing their own "losing" strategies for programming? Or, in the absence thereof, how do they develop successful ones?

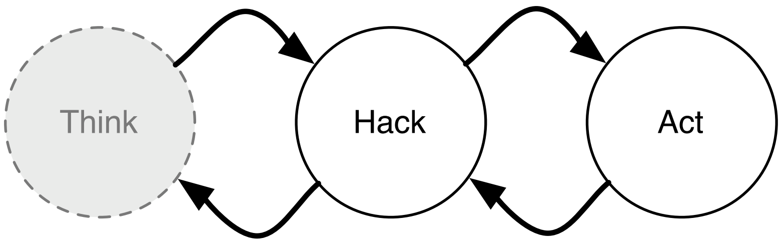


Figure 2.5: The hack-n'-act cycle, *sans* thinking?

Perkins, Hancock, Hobbs, Martin, and Simmons go on to describe this behaviour in terms of *stoppers* and *movers* (as well as the subgroup of "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. Tinkerers, however, will explore the problem—sometimes systematically, sometimes successfully—hopefully to keep moving towards a problem solution.

> ... 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.

2.2.4 Models and mistakes

The work of Bonar, Soloway, and Spohrer does not consider the code development process that Perkins et al. focused on. They explored semantic errors made frequently by student programmers, as captured by the first syntactically correct program a student manages to execute. Methodologically, they ignore the process that led to the creation of that program, and the process that follows the creation of this first syntactically correct artefact. Their premise is that the "goals" and "plans" a student was using (or attempting to use) in developing their program can be inferred from this snapshot in time.

Their work explores possible sources of error and misconception in novice programmers, and searches for an over-arching model that is representative of how all novices go about writing programs. The most comprehensive source of reports we have for this work are the sequence of papers that appeared in *Studying the Novice Programmer*, edited by Soloway and Spohrer[SS88]. "Programming Knowledge: A Major Source of Misconceptions in Novice Programmers," "A Goal/Plan Analysis of Buggy Pascal Programs," and "Novice Mistakes: Are the Folk Wisdoms Correct" appear together in this volume, and will be discussed here as a single work.

Bonar, Soloway, and Spohrer motivate their research into the mistakes novices make based on their work in [BS83]:

> While solving a programming problem (writing a program), novices will encounter some aspect of the problem they don't understand (an impasse).
>
> In order to move beyond the impasse, novices cast about for a way to resolve the aspect of the problem they don't understand (a patch). Frequently, that resolution involves an appeal to their knowledge of natural language step-by-step procedures that would be applicable in a similar situation.
>
> In implementing the patch, a bug is introduced.

There is no data presented, or referenced, to explain the student problem solving process. It is stated as fact that students "frequently" appeal to their natural language understanding of a problem and its solution. While this is not unreasonable, it is also the foundation upon which they will build a cognitive theory of novice programmers: the planning process they go through in writing a program, and the (semantic) errors they make along the way.

In "Preprogramming Knowledge," the two critical components of the Bonar/Soloway model of novice programming are described as being an extension of "repair theory," which can be traced through VanLehn's work on complex problem solving in the early 1980's to the Cascade project and beyond[BV80, VJ93]. Bonar and Soloway claim that programming, being a more complex domain than elementary arithmetic, must have two important components:

> 1. We characterize the knowledge that allows a novice to form a bridge between programming language syntax and semantics and higher level design concerns. This is information about how the language constructs are used to accomplish standard programming tasks. We represent this information as schema-like structures called *programming plans.* We discuss the programming plans for both the introductory programming language Pascal and natural language step-by-step procedures.
> 2. We characterize impasses arising from missing or misapplied programming plans needed in the course of developing a program. We propose that many bugs arise out of novice strategies for patching an impasse and continuing a problem solution. We call these strategies *bug generators*. We focus on the bug generators that patch an impasse by using the knowledge of how the problem

would be solved with natural language step-by-step procedures. This knowledge is used to supply missing programming language knowledge.

Bonar and Soloway employ think-aloud protocols while students write a small program in Pascal. Based on the analysis of the transcribed version of these interviews, they characterize the step-by-step natural language knowledge a student brings to the task, as well as the programming, or Pascal-specific, knowledge a student brings to the task. This information helps them determine what *plans* are made by the students as part of the programming process, and in a second analysis phase they evaluate the bugs in the student's problem-solving process[BS83].

This work is continued in "A goal/plan analysis of buggy Pascal programs." In this paper, they extend their lower-level theories regarding the role of natural-language and programming-specific knowledge into a theory of *goals* and *plans*. Goals are smaller tasks a programmer might attempt to achieve along the way to completing a larger plan. This work is also described and extended in "Analyzing the High Frequency Bugs in Novice Programmers," where they combine the detailed bug analysis of [BS83] with their higher-level goal/plan approach[SS86a].

One of the last papers in this particular thread of research is Spohrer and Soloway's "Novice Mistakes: Are the Folk Wisdoms Correct?"[SS86b]. They contend that, for the most part, how we decide to teach programming is based not on empirical research, but instead on folk wisdoms regarding novice programmers. Whether or not this is the case, their arguments are sound:

> As we observed in the introduction, a not unreasonable assumption is that instruction can be improved when educators gain a better understanding of what students do and do not know. From our analysis we conclude that computer science instructors should strive to familiarize themselves with specific high-frequency bugs, and to learn as much as possible about the origins of all bugs. Both high- and low-frequency bugs seem likely to occur when students are unable to coordinate and integrate the goals and plans that underlie program code. Although additional studies must be carried out to test the stability of our results, our data nevertheless begin to pinpoint specific areas in which students have difficulty learning to program.

So, while we take issue with some of the leaps made in the formation of their study, their conclusions are (generally) reasonable. That is, we agree completely that educators would do well to understand better what their students do and do not know, as well as the kinds of mistakes their students typically make while engaging in programming tasks.

The importance of syntax

In all of their work, Bonar, Spohrer, and Soloway did not address the syntactic issues their subjects had with the languages. Their theory regarding mental models of novice programmers *does not account for syntax errors*. While they did study students writing a variety of programs in a first year course (and focused closely on three in particular), their work always begins from the first syntactically correct program a student manages to produce. From "Analyzing the High Frequency Bugs in Novice Programs,"

> Sixty-one students' first syntactically correct programs for the three problems were selected for detailed analysis. We selected the first syntactically correct version because we are primarily interested in non-syntactic errors, and the first version contains more bugs than later versions (i.e., the novices had not yet debugged their programs). Of the 183 programs chosen in this manner, twenty-five were excluded from the analysis because the students had not tried to solve enough of the problem in their first attempt (e.g., merely printing an introductory message).

We can only assume that this is indicative of all of their selection criteria, as we do not receive such a detailed account elsewhere. Here, we see they have excluded 14% of their sample because the students did not write enough of the program in the first go. As a result, their study systematically ignores the process that leads up to a syntactically correct program, it ignores the process that occurs after the first program is written, and it drops a significant percentage of their population because their methodology is not capable of handling students who write a minimal amount of code—code that can be executed—before progressing on to the task they have been given.

The problems with syntax errors

At the same time as Spohrer and Soloway are ignoring syntax in their studies, elsewhere we hear others calling for more attention to be paid to the errors that systems produce. Coming out of the 1960's, it was a big step in compiler technology for run-time errors to be reported in terms of the *source line*, as opposed to the location of the error in the *generated assembly code*. However, compiler error messages still suffered from obscure, numeric lookup codes and other unusable reporting measures throughout the 1970's. In 1983, Peter Brown suggested that an *integrated development environment* might provide a framework in which some of these problems can be solved in his paper "Error messages: the neglected area of the man/machine interface."

> There are also two more recent developments that could lead to a further improvement: the integrated programming environment and the high-resolution display with windowing. Each of these is just beginning to become available to the average user in the field, as distinct from the researcher. The average user, of course, needs good error messages even more than the researcher.
>
> The effect of an integrated programming environment is twofold. First, most such environments remove the traditional rigid separation between editing, compiling, and running; it is all the more important, therefore, to have a uniform approach to errors so that a run-time error is not presented to the user as something radically different from, say, a lexical error.
>
> Second and more exciting, some programming environments remove the need for certain error messages altogether. Two well-known examples are the Cornell Program Synthesizer and the Interlisp system.[Bro83]

Unfortunately, little empirical work in this area has been carried out in the intervening decades. Brown was one of the few researchers to highlight the problems and possible solutions inherent in dealing effectively with syntax errors. Isa, Boyle, Neal, and Simons highlight this in their own studies:

> Although many guidelines and articles have been written about improving error messages, only a handful of empirical results have been reported in the literature.

> In separate experiments involving COBOL compiler messages, text editor messages, and job-control language messages, Shneiderman reports experimental results that support the conjecture that the wording and content of messages can impact user performance and attitude.[IBNS83]
>
> ...
>
> Testing of error messages is clearly something that must be done if designers are to be sure that users will be able to successfully correct their mistakes. For any given user error situation, program developers must assure that a typical user can make the appropriate correction. However, testing messages in the context of product use or while performing other more general product usability tests poses two problems: first, not all messages will get exercised; and second, when multiple errors occur, it is difficult, if not impossible, to know which error a user is trying to correct.[IBNS83]

With the continued development and study of novice programming environments — environments that through one mechanism or another hope to attack the kinds of problems that Isa, Boyle, Neal, and Simons point out — both theories and methods will play important roles in this evolving field of research. However, it is work like that of Myers and Ko that may most productively be taken as an example of data-driven inquiry for exploring programming behaviour[KAM05].

2.2.5 Notational systems and cognitive dimensions

Our own research focuses on the behaviour of students while engaged in the act of writing programs. Particular attention has been paid to the compile/edit cycle, a process in which the student interacts with and reacts to the compiler and its responses to the syntax they feed it. If a theory regarding how novices deal with syntax (a notational system) is to be developed, then we may eventually turn to existing work regarding how people understand notational systems as a starting point.

The research on Cognitive Dimensions represents an interesting way of framing research regarding how novices struggle with the syntax of a language, and their larger interactions with the initial learning environment. Cognitive Dimensions of notational systems are proposed in much the same way that much of physics is reducible to the dimensions of mass, length, and time[Gre89a]. They provide the language designer and researcher with a framework in which to discuss, compare, and evaluate notational systems. As suggested by Green, a "system" is a notation and an environment taken together:

> Indeed, the relationship between the notation and the environment is such that the notation cannot be used except in some kind of environment of use. Trying to just use a notation, outside of any environment, would be like trying to just talk to someone without being in some kind of social situation. We may catch ourselves thinking that paper-and-pencil, by its familiarity and blandness, is not really an environment at all, but if so we are making a mistake: paper and pencil is an environment with its own particular contributions--for instance, it offers unrivalled support for recording hesitations and commitments, makes it easy to see large amounts of text with little effort, and supports instant action with very little delay in finding the right place, choosing the right mode, etc. In all these respects it is superior to typical computer-based environments.[Gre89a]

1. Hidden/Explicit dependencies	6. Diffuseness
2. Viscocity/Fluidity	7. Consistency
3. Premature commitment	8. Discriminability
4. Role-expressiveness	9. Action slips
5. Hard mental operations	10. Perceptual Cues

Table 2.4: A working set of cognitive dimensions.

Each of the cognitive dimensions presented in Table 2.4 is suggested based on a body of research carried out by Green and others regarding programming languages and their environments. For example, we might say that a language like Lisp scores highly on the dimension of *consistency*, as its uniform syntax allows for easy and unambiguous notation of logical expressions. This might be contrasted with the C programming language, where the programmer must concern themselves with the order of precedence in logical operators—entirely an artefact of the infix notation—which represents a source of notational inconsistency.

The Cognitive Dimensions framework has been used in a number of contexts for evaluating programming languages and their environments. Green and Petre used it for evaluating

LabView and Prograph, as well as the visual complexity of programs written in these visual languages[GP96]; likewise, Yang et al. applied these dimensions to the design of visual programming languages[YBDZ97], and Clarke has applied these ideas to the design of languages and programming APIs at Microsoft[Cla01]. This might begin to give a sense for how broadly, in terms of languages and notational systems, the Cognitive Dimensions framework has been applied.

While not carried out as an empirical study, Linda McIver explored the Cognitive Dimensions of programming languages used in introductory programming courses as part of her dissertation research regarding the development of a programming language for novices[McI01]. Her work applies the Cognitive Dimensions framework broadly across a large number of languages from a variety of programming paradigms, and then uses this analysis to inform the creation of new programming languages intended for beginning programmers.

What this, and other work regarding Cognitive Dimensions tells us, is that it is not only the notational system that matters. The entire environment—whether it is a text editor, initial learning environment, or a professional, integrated development environment—must be considered as well. Many studies regarding novice programmers have ignored this larger context. For example, it would be difficult to discover exactly what version of Pascal, and what version of the VAX OS, students in Spohrer and Soloway's studies were working with. However, with a theoretic framework like Cognitive Dimensions, we are able to make recommendations regarding tools for supporting novice programmers that may, in some way, begin to transcend the specifics of what language and which editor students were struggling, working, and learning with.

Chapter 3

Into the Data

Good data tells a story. In exploring novice compilation behaviour, we found that the hidden interactions between students learning to program and their compiler provide a rich source of material for interesting stories. Like so many frames in a movie strip, we captured a copy of their work in progress every time they compiled their code. That makes for a lot of data.

To understand the story captured in our data, we begin with the context in which it was collected: who, where, and when. To make clear our method, we then describe what data we collected and how we went about that collection. These basic questions provide a background for the interesting story that the data starts to reveal: a glimpse into the hidden battle between novice programmers and their compiler. First, we consider our students in the aggregate, looking at trends that hold across the entire population. Then we focus in on the individual, and discover that there is no less to be told about the individual than the entire population.

3.1 Students at Kent

Our explorations begin with our students. We suspect that university students in their first year are, by-and-large, the same the world over. They are young adults living on their own, often for the first time in their lives. These students come to us excited about new opportunities, meeting new people, and taking on the challenges that their course of study will bring them.

3.1.1 The place

The University of Kent was granted its Royal Charter in 1965; to this day, the college system continues, with all incoming first-years (“Freshers”) being assigned to either the Rutherford, Eliot, Keynes, or Darwin colleges. They end up studying in one of seventeen departments arranged into three faculties: the Faculty of the Humanities, the Faculty of Science, Technology, and Medical Studies, and the Faculty of Social Sciences. Historically, the colleges played a more important role in the students’ lives; now, they are effectively dormitories, but with an added sense of history and social structure.

The centre of campus is home to the majority of the academic departments, with the Colleges and student accommodation arranged about the periphery. While Canterbury is a popular tourist destination, the University and City provide no more or less distraction for the students than any other University. The students at Kent find many productive distractions from study as well—they engage in all manner of sports through the University fitness centre and various club sports. Likewise, political, religious, musical, and other societies exist to fill the time students spend when not in class or studying.

The Computing Laboratory

The Computing Laboratory is centrally located on campus, providing space and facilities that many of the students enrolled in computing degrees make use of over the three years they are at Kent. The Lab’s facilities that are restricted to those students who are in the degree are located in the Octagon. Named for its distinctive, 8-sided shape, the Octagon houses some departmental teaching spaces as well as specialized labs. These include the Multimedia lab, with high-spec machines for dealing efficiently with digital media, as well as the Unix lab, housing Sunray thin-clients to Raptor, a 4-processor Sun 480 provided by the department for undergraduate use.

Despite its central location on campus, the layout of the department itself does not encourage students to move through the halls and accidentally interact with the faculty. The majority of the department’s offices are located on the second floor; as there are no teaching spaces in this area, undergraduates are unlikely to be found in these hallways unless they are specifically seeking out a member of the faculty for a particular reason. Similarly, the amount of interaction between the graduate students and the undergraduate population is likewise minimal.

The degree

Students in the Computing Laboratory fall into one of a number of distinct courses of study, although all of them share a great deal in common (in terms of coursework, etc.). Students encounter their first module in programming, "Introduction to Object-Oriented Programming," in each of nine different courses of study. While these courses of study diverge most significantly in the second and third year of the degree, we have focused here only on the similarities and differences that students encounter in the first year of the program.

Computer Science

This course of study is the most general of the degree courses, and is notable in that students encounter two paradigms of programming—object-oriented programming in Java and functional programming in Haskell—in their first year. Additionally, they cover the basics of information systems, hardware and protocols, discrete mathematics, and statistics.

Computer Science with a Year in Industry

Each of the paths through the undergraduate degree offer an option of a year in industry. This highly competitive program is popular with the students, as it gives them a chance to get real-world experience working with Sun, IBM, or other partners in industry that the department has cultivated relationships with.

Computer Science and Management Science

The cross between computing and management science varies little in the first year—students in both programs take many of the same courses. However, the management students trade the exposure to functional programming for an introductory course in management science. Other than this, they take the same foundational courses in theory, hardware, and object-oriented programming as the students in the department's main course of study. This course offers a "year in industry" option as well, and exhibits greater variance in the second and third years in terms of courses offered.

Computer Science and Business Administration

As per the cross with Management Science, this course of study differs little from the base course of study in the first year.

Mathematics and Computer Science

In lieu of any courses on information systems, hardware, and protocols (that are taken by students in the Computer Science degree), these students take more courses in calculus, discrete mathematics, probability, and algebra. Over the three years of the degree, students in this degree program focus most intensely on the programming courses offered in the department, while content knowledge (courses like operating systems, etc.) are replaced by maths of one sort or another.

The one module that is consistent across all of these courses of study is the introduction to programming in the object-oriented style. This module has been taught by two faculty members in the time we were collecting data; in the 2003-2004 academic year, it was co-taught by David Barnes and Mathieu Capcarrere, and during the 2004-2005 academic year, it was taught entirely by Mathieu Capcarrere. In both cases, the module text remained the same (*Objects First with Java*,[KB05]), and the assignments remained relatively consistent between the two years, as they followed the text quite closely.

The most likely source of differences between the students enrolled in these different degree courses will come from their *expectations* regarding the first course in programming. For the students in computer science, they see Java as a critical part of their education, as it is widely used in industry. For students in some of the other programs, however, it is often viewed as being less essential, or less practical: Visual Basic, and the ability to write macros in Excel, is certainly more practical. This research does not delve into the expectations and desires students bring with them to the classroom, but instead focus on their programming behaviours regardless of their particular course of study.

Facilities

All students at Kent have access to the University-provided computing labs. In addition, the Computing Laboratory provides additional hardware and services for students enrolled in one of its degree programs. From the Computing Laboratory's departmental web-pages:

> The department has several well equipped laboratories for the sole use of its own students, including a lab equipped with 16 Sun Rays for access to the department's Sun servers running Solaris, and several rooms equipped with PCs running Windows XP; typically 2.4GHz Pentium 4 with 512MB of memory and 19 inch screens.
>
> In addition, the department has a number of servers donated by Sun Microsystems which can be used by students in their final-year projects, a Sun server (with 16GB of memory and four SPARC processors) for general undergraduate use, and additional servers for postgraduate use.
>
> All this equipment is available during term time, weekends and vacations.
>
> We provide a number of printers and you will find that the cost of printing is somewhat cheaper than when you use the university printers.[1]

As many an undergraduate will attest, these statements are true... in part. As much of this equipment is in the Octagon, it is actually inaccessible after 9:30 PM during term time, and is not accessible at all weekends. There is no 24-hour computing facility maintained by the Laboratory that students can make use of, and students are clearly expected to rely on their own computing equipment to some degree during their course of study.

Public computing laboratories

The public computing laboratories from which the majority of our observations come are not maintained by the Computing Laboratory, but instead by the University's Computing Service. These laboratories typically have between 15 and 30 machines, arranged around the periphery of the room they are located in. The machines in these spaces are all 2GHz+ Dell machines running Windows XP. From the desktop, the students have access to the Microsoft Office suite and a host of other tools; most notably, for our study, BlueJ. They have a 25MB network share that they can store files on, which many students find to be inadequate; it is not uncommon to see them mounting drives from the computers in their own room, using it to stream MP3s, and even provide a complete remote desktop while working in the labs.

[1] http://www.cs.kent.ac.uk/students/facilities.htm, accessed on 20050605

3.1.2 The students

For a number of years, the Computing Laboratory has issued a survey to all of its incoming students. While this survey is not comprehensive, it provides us with an overview of the computing background and experience of students entering the department. This survey has been issued every year, but we will only present here the data from the year 2003-2004 survey, as it is the most recent survey conducted, and the years 1999-2003 were not appreciably different.

To begin with, the students were asked to rank their familiarity with a number of operating systems and common software packages, with values ranging from 1 ("never used") to 5 ("have used a lot") (Table 3.1). 132 students completed the questionnaire.

Table 3.1: Year 2003 incoming CS survey: OSes and Applications

	Mean	Median	1Q	3Q
Web browsing	4.84	5	5	5
Email	4.78	5	5	5
Windows 98, ME, XP, etc.	4.77	5	5	5
Word Processor (e.g. MS Word)	4.60	5	5	5
Instant Messengers (e.g. MSN, ICQ)	4.44	5	4	5
Spreadsheet (e.g. MS Excel)	4.00	4	3	5
Database (e.g. MS Access)	3.56	4	4	5
Creating web pages/sites	3.23	3	2	5
Usenet news	2.82	3	2	4
Unix (Linux, Solaris, etc.)	1.70	1	1	2
MacOS	1.60	1	1	2

This particular background is not surprising: the incoming students at Kent are very comfortable with the Windows operating system, have little experience working with Apple computers or Unix systems, and are very comfortable with email, the World Wide Web, and standard office applications like word processors and spreadsheets.

Using the same scale, the students expressed their familiarity with a variety of programming languages (Table 3.2).

Based on their self-assessment, none of our incoming students are familiar with any particular programming language (Table 3.2); if they have been exposed to anything in school, it is most likely that they have encountered some form of VB, and even then only to a limited extent. It isn't uncommon for students to confuse the creation of graphical user interfaces in

Table 3.2: Year 2003 incoming CS survey: Languages

	Mean	Median	1Q	3Q
Basic (including VB)	2.94	3	2	4
Any other language(s)	2.04	1	1	3
Java	1.85	1	1	2
Pascal (including Delphi)	1.82	1	1	2
C or C++	1.60	1	1	2
Prolog	1.42	1	1	1

Visual Basic with programming; as such, it seems to say that the vast majority of our students have little real experience in any programming language when they enter their first year of a degree in computer science.

The question we were most interested in from this survey was how many of these students owned their own computer. Of the students surveyed, 95% reported they owned their own PC. This was further broken down by a series of yes/no questions: 81% reported that their computer was recent (1GHz or faster), 93% were running some version of Windows, 13% had Linux on the PC. 74% of these students had their computers on the campus network, meaning they had 100 megabit ethernet in their study bedroom, and another 20% would have their computer on the Internet via some other ISP. This implies that roughly 20% of the incoming 2003 class lived off-campus, where a significant number of them would have commuted to school from home.

On their way out

While we have some information regarding what our students were like when they entered, we can also look at their experiences during the year as reported on exit questionnaires from their introduction to object-oriented programming. From the course home-page, we find a brief summary by Mathieu Capcarrere of the feedback questionnaires completed by the students:[2]

> The feedback form has been analysed, but I didn't find the time to write a proper page. In Brief: the results are obviously biased as it is only representative of people still coming to the lectures after 10 weeks... Overall, almost all of you had no experience of Java before the course, and no or little experience of any other language.

[2]Accessed June 2005. No longer available online.

> You agree (limited std deviation) that the course has been interesting and that it definitely resulted in an improvement in your knowledge [thank god]. A vast majority thinks the workload has been reasonable, but a tiny and determined minority seems to think that torture would have been more pleasant. On the lectures, you kind of agree that they were clearly structured, going at the right speed, and the visual aids were helpful, but it is not a plebiscite neither. For help, the anonymous question page wins with a 10 to 1 margin on other means. Classes were far less convincing but there is still a majority to say they were OK, and a (small) majority of people to say that classes should be devoted to assignments. On that latter question, the opinions tend to be clear-cut (large std dev.). Finally BlueJ was found to be easy to use and install, but you would not warmly recommend the textbook, nor advise against it: a definite maybe.

This summary comes at the end of the 2004-2005 academic year. Given the plethora of ways students can and do interact with lecturers in the department—email, newsgroups (Kent has a thriving newsgroup culture in the Computing Lab), and of course in person—these are not necessarily the students' preferred mechanism for interacting with their lecturers regarding course content. Interestingly, the students found the anonymous question pages to be the most useful form of feedback in the course. The Computing Laboratory maintains a web-based, anonymous feedback system for almost all of their courses. Using this system, students can leave comments regarding lectures, homeworks—anything pertaining to the course, really—without having to identify themselves to their instructor or their peers. Furthermore, the anonymous question pages are public, and archived: the questions students ask, and the instructor's answers, are available for all to read at that time and for years to come. While students found this system to be a valuable part of their experience in this module, the most pertinent comments filtered through to us are the students' reflections on the use of BlueJ and the course textbook, *Objects First with Java*. These are, directly and indirectly (in the case of the textbook) the subjects of our study. While students were largely unimpressed by the environment and text, it is important that the nature of both the programming environment and the course text are understood before we continue further.

3.2 BlueJ and *Objects First with Java*

Our research is concerned with how novices interact with the compiler, and the compile-edit cycle resulting from that interaction. The techniques we employed are not language specific, nor are they bound to any one particular programming environment. That said, the students who took part in our study were all using one programming environment (BlueJ) with one course text.

3.2.1 BlueJ: an initial programming environment

BlueJ is an initial programming environment for beginners learning to program in Java. Such environments typically provide a simplified mechanism by which students can edit, compile, and then execute their programs. BlueJ is no exception: it has a minimal interface, and encourages an "objects-first" approach to programming. This means that the BlueJ environment encourages students to think about, first and foremost, *objects*. Objects are an abstraction used by computer scientists for representing a combination of data and methods that operate over that data; they are not unique to Java, but they are the primary mechanism by which Java programmers organize the structure of their programs.

The first thing students see when they start BlueJ is the diagram editor (Figure 3.1). This view allows them to see the larger, object-oriented structure of their code. In the centre of the editor is a diagram made up of rectangles (Classes) and any relationships between classes; our screenshot shows three classes. On the left are four buttons; the bottom button, `Compile`, is one of the three ways students can invoke the compiler, and therefore generate the data we use in our study. At the bottom of the screen is the *object bench*: here, students can instantiate the code they have written, and interact with it—this is how students execute, or "run", their programs. While simple in it's design, BlueJ provides many opportunities for students to interact with their programs via the diagram editor—our study was not concerned with any of these interactions.

If a student double-clicks a class, they are presented with a simple text editor in which they can write or modify programs. If there is a syntax error in their program (as we have introduced in Figure 3.2), a student might discover this by pressing the `Compile` button on this window, or by pressing the keyboard shortcut for invoking the compiler. As a matter of principle (held

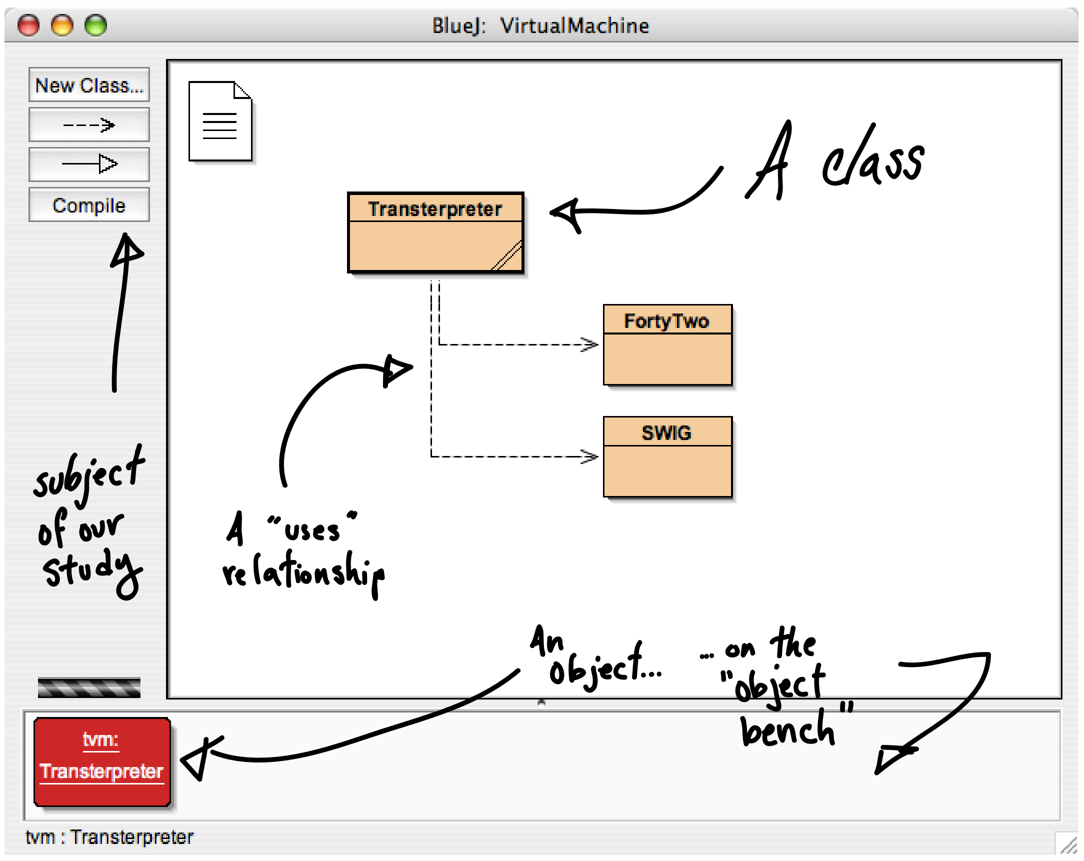
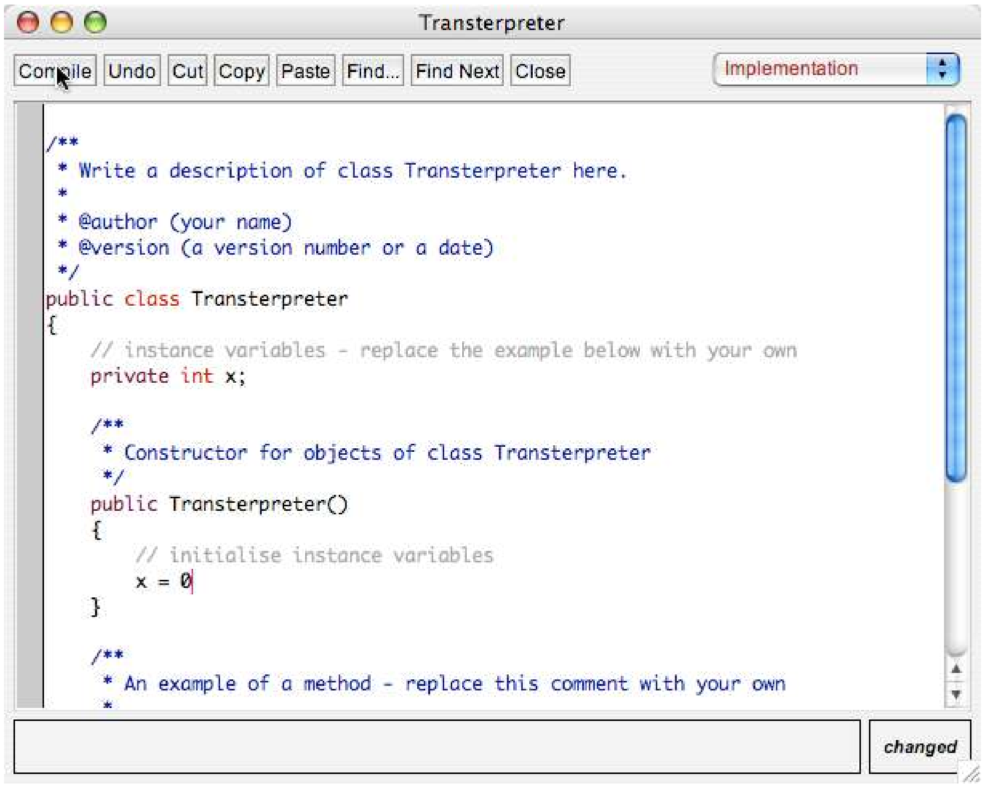


Figure 3.1: BlueJ's Diagram Editor

by those responsible for authoring BlueJ), *only one syntax error at a time is presented to the student*. So, in Figure 3.3, we see how BlueJ reports a `';' expected` error. The line in question is highlighted, and an error message is reported in the status bar at the bottom of the editor window.

Our research was unconcerned with the pedagogic features BlueJ offers regarding the visualization and interaction with objects; instead, we focused entirely on the compile-edit cycle that is ubiquitous in the programming world. However, our analysis was greatly simplified by the fact that BlueJ only reports one syntax error at a time.



```
/**
 * Write a description of class Transterpreter here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Transterpreter
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class Transterpreter
     */
    public Transterpreter()
    {
        // initialise instance variables
        x = 0
    }

    /**
     * An example of a method - replace this comment with your own
     *
```

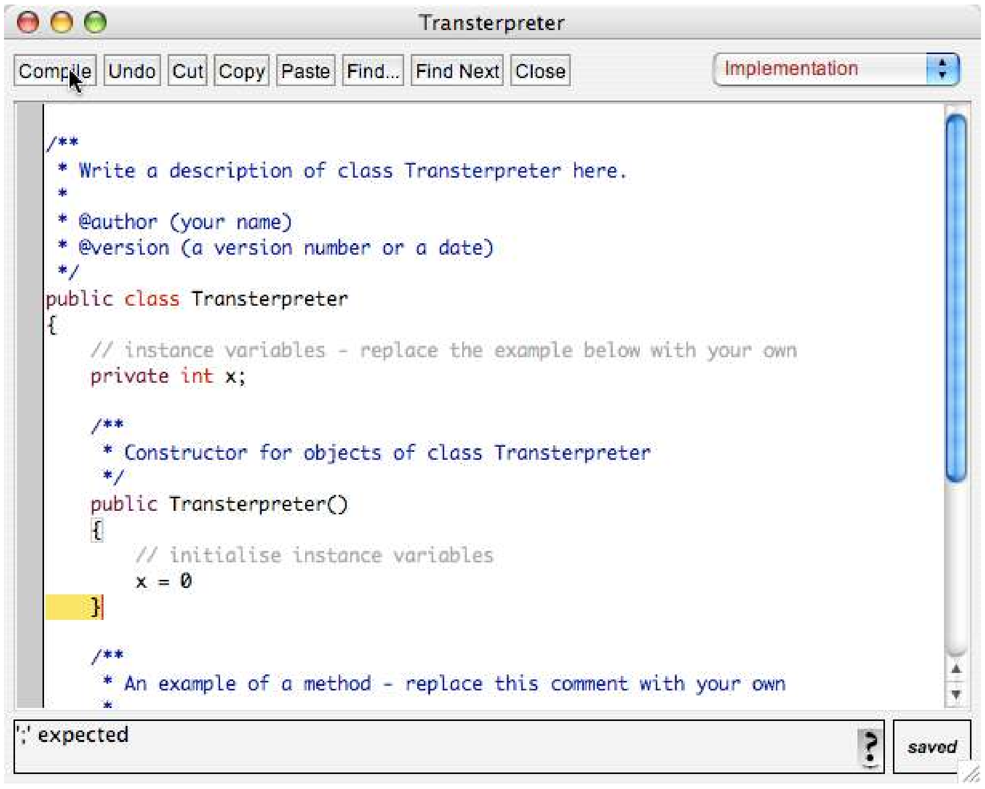
Figure 3.2: BlueJ's Text Editor

3.2.2 *Objects First with Java*

The text used by students in the course modules we studied all made use of the same course text, which is tightly bound to the BlueJ programming environment—an instructor would be hard-pressed to use this text without having their students make use of the BlueJ programming environment. Like BlueJ, the text encourages an objects-first approach to learning to program in Java. As a course text, it is quite popular; in fact, *Objects First with Java* was selling 10 copies per week in January 2006[3]. According to the BlueJ web page, it is used at hundreds of universities and high schools around the world for introducing students to programming in Java.

Practically speaking, the textbook's focus on "objects first" means that students are exposed to concepts in the third chapter of the book that are often reserved for the last few chapters of other textbooks used in Java programming classes. This means that less time is spent, early

[3]Based on an Amazon.com sales rank of 24,978 on January 14th, 2006, and information found at http://www.fonerbooks.com/surfing.htm



```
/**
 * Write a description of class Transterpreter here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */
public class Transterpreter
{
    // instance variables - replace the example below with your own
    private int x;

    /**
     * Constructor for objects of class Transterpreter
     */
    public Transterpreter()
    {
        // initialise instance variables
        x = 0
    }

    /**
     * An example of a method - replace this comment with your own
     *
```

```
';' expected
```

Figure 3.3: BlueJ reporting a `';' expected` error.

in the text, on topics that are typically taught to novice programmers: branching constructs (`if/then/else`), looping constructs (`while`, `for`), and so on. Instead, students begin focusing on what might be considered *architectural* concerns: how their programs are organized, and how different parts of their program might interact, or communicate, with each-other.

While the course text is not directly related to our research, many of the programming assignments our students attempted were taken from this textbook. While we cannot give a complete overview of the text here, we will mention two assignments in particular: the *notebook* project and the *horse race*.

The notebook

The notebook project comes directly from chapter four of the course text. We focus on this assignment for a number of reasons; the most important reason is that we have the largest amount of data for this assignment, as every student in both the 2003-2004 and 2004-2005 academic years did the assignment. Second, students attempt the assignment in the first term, allowing us to examine novice programming behaviour early in the learning process. And lastly, students are expected to add a minimal amount of code to their program—we can focus our investigation without having to artificially ignore large sections of code written by the students.

In chapter four of *Objects First with Java*, students are introduced to the modelling and programming task:

> 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.

Chapter four and the notebook project bring together concepts from chapter three (objects referring to, or otherwise containing, other objects) with the use of Java libraries—in particular, the `ArrayList` class. Students are given an overview of the code involved (also provided on the CD in the back of the textbook), and their assignments centre on extending and evolving this code.

Horse race

The horse race project was new in the 2003-2004 academic year; it was (in hindsight) a decidedly challenging assignment. Tackled early in the second term of the academic year, this project involved students modifying, writing, and extending code representing various objects you might find involved in a horse race: horses, jockeys, and so on.

The students spent large amounts of time on this project, and significant amounts of traffic were generated on the anonymous question pages. In the data we collected, we have sessions that are hours long from some students on this project, as they struggled with solving the various parts of the problem set by their instructor.

3.3 Capturing behaviour

We carried out our observations of novice programming behaviour over a two-year period spanning the 2003-2004 and 2004-2005 academic years. During that time, only one change was made to our observation technique, at the midpoint of the 2003-2004 academic year.

3.3.1 The data at a glance

During the Fall of 2003, we observed students when they were programming during a scheduled class time. This works out to one, one-hour session per week, which would typically take place in one of the public computing spaces located within the Computing Laboratory (CC01, CC02, CC03, or CC04). This resulted in the collection of 3462 pairs of compilation events from the 62 students who agreed to take part in our study. We report *pairs* of events because they are much more interesting than events taken singly, as successive pairs of events allow us to observe the evolution of a student's program over time.

In the Spring of 2004, we expanded our observations to include any time they were working in BlueJ in one of the public computing labs on campus. This means that we captured their behaviour when interacting with the compiler (through BlueJ) both when they were in class, and when they were working on their own somewhere on campus; as a result, we captured 7202 compilation pairs during the Spring of 2004. Our human ethics forms for the Fall of 2003 were written to only cover one term, we reissued these forms in the Spring 2004 term, requesting students to join or otherwise continue to take part in our study. Therefore, we had 56 students who allowed us to collect data on their compilation behaviour during the Spring term. 31 of the students from the Fall continued their participation in our study; as a result, we have a relatively small population for whom we have continuous data over the entire 2003-2004 academic year.

During the 2004-2005 academic year we observed 68 students any time they were engaging with BlueJ in a public computing laboratory. We continued to limit our observation to times students were working in campus computing laboratories largely due to the significant technical and ethical hurdles involved in studying the students' programming behaviour on their own machines located "out in the world." Over the course of the year, we captured 24,852 pairs of compilation events—more than twice the number captured during the 2003-2004 academic year, providing us with a large, contiguous data set for the 68 students who took part in our study.

3.3.2 Tools for data collection: BlueJ

Our exploration of novice compilation behaviour relied extensively on the frequent, automatic collection of source code written by students while programming. By some, our methodologies might be referred to as an *on-line protocol*[SS86c], as our observations did not rely on a researcher observing the subject, but instead a computer automatically recording the student's interactions with it. In the case of our work, we required a careful orchestration of extensions to BlueJ, CGI applications on web servers, and databases to make our research possible.

From the students' perspective, BlueJ is the software that they use for authoring their Java programs. From our perspective as researchers, BlueJ is a critical component in our research infrastructure, due entirely to its highly extensible nature. By extensible, we mean that BlueJ was designed so that individuals could add new functionality without needing to modify the application itself.

The BlueJ extensions framework, developed at the University of Kent by Damiano Bolla and Ian Utting, provides developers with a way of responding to a wide variety of events in the user interface, as well as the ability to extend top-level and contextual menus throughout the environment in a controlled manner[BU]. We extended BlueJ's functionality in a non-visible and non-invasive way; our data gathering extension sat quietly in the background, and responded to three classes of event. The first of these events is generated when students press the "Compile" button in BlueJ, or invoke the compiler through a keyboard shortcut. The second class of events are generated by the compiler itself, and alert us as to whether there was warning or error related to the students code. The third class of event alerts us as to when the compiler has completed its work.

Our extension recognizes when the compilation was begun to the nearest millisecond (later rounded to the nearest second), gathers up the warnings and errors generated during the compilation process, and notes when the compiler finished compiling. We then gather some additional metadata from the system—the student's username, the type of operating system they are running on, and the unique network name of the machine they are using. Lastly, we ship this metadata to a remote server, along with the source code developed by the student.

Granularity

One of the most difficult challenges in developing an on-line protocol is deciding at what level of granularity, or detail, data collection will occur. For example, Spohrer and Soloway applied their theory of goals and plans to the first syntactically correct program written by university students[SS86c]; they chose to ignore the process leading up to this program, as well as any subsequent revisions made by the student. At the other extreme, the GRUMPs project at the University of Glasgow explored the practical limits of instrumenting the Java virtual machine, and captured every event and method call executed by the VM while students were learning to program in an IDE for Ada written in Java[GMD$^+$04]. This is far below the level of of the keystroke; the GRUMPs group had to contend with millions upon millions of events. At this level of granularity, it becomes a project unto itself not only to manage the data, but to mine it for information relevant to the study.

The choice of triggering our collection of data on the act of compilation was not arbitrary. From our own experience in teaching novices to program, we have observed that the compiler is invoked often, sometimes faster than seems possible—if one assumes that students read the error, examine their code, and consider what is the best or right way to fix their code before recompiling. Additionally, we have prior experience authoring code submission systems for use in introductory computing contexts[Jad]. We have observed (indirectly) that some students will submit an assignment dozens of times—sometimes upwards of eighty and ninety times—to get feedback from the hand-in system. At the same time, some students would only submit their code four or five times. This suggests that different students engage very differently in the process of writing, compiling, and submitting programming homeworks.

Having seen this first-hand, and having spent time in the classroom teaching beginning programmers with BlueJ, we had a good sense for how much data we would capture using the compilation event as our key event. Performing some quick "back of the envelope" calculations indicated that we would have a reasonable amount of data to deal with. Assuming we might have 100 students "on-line" at any one time, each with two 2K files open, recompiling every

10 seconds, our web server would be dealing with 600 requests per minute, and the network would be carrying 1.2 megabytes of data per minute, when it is capable of carrying (at least) 12 megabytes per second. These estimates led us to believe we could have researched a population many times larger using the same protocols and infrastructure.

In short, we required no special provision, technologically, to carry out this research; we could have handled a population ten times larger using the same infrastructure, and suffered no loss of data.

3.3.3 Tools for data collection: WWW

The WWW—in particular, the HyperText Transport Protocol and web servers—provided a critical bridge between our students writing their programs and the databases in which we would store our data. In developing the software to support our explorations, it was our intent that the students being studied could be anywhere in the world, while the servers harvesting that data might be on the same campus, or might be halfway around the world.

We exploited these properties in replicating the storage of student compilation data; out of paranoia, we maintained two separate repositories of data, and each compilation event resulted in a copy of the data being sent to our local repository (at Kent) and a second to a server that was located on another continent. The reason we could not insert student data directly into either database was due to the existence of firewalls.

Firewalls take their name from their physical analogue; whereas a real-world firewall is intended to help slow or stop the spread of fire through a building, a firewall on the network is intended to prevent intruders from obtaining access to protected computing resources. Database servers are typically a resource that a department wishes to protect from arbitrary access. As a result, we designed our data collection tools to take advantage of network configurations like that in Figure 3.4, where a web server is exposed to the outside world, and the database server is exposed to the web server—but not to the Internet at-large.

To exploit this common network architecture, we wrote a script that would run on the web server, playing a simple, yet critical role. It would accept data from our BlueJ extension over the HTTP protocol, verify that the data was "reasonable," and then insert that data into the

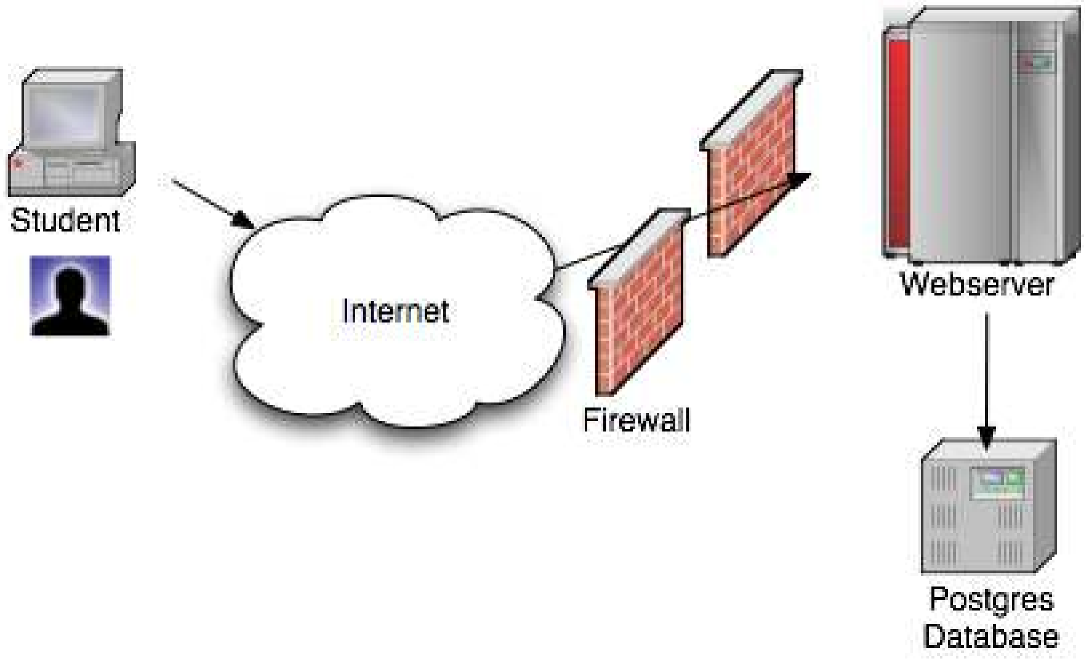


Figure 3.4: Systems involved in data collection

database. In structuring our data collection this way, we allowed for the future contingency that both the students we study and the servers used to store our research data might be located anywhere in the world.

3.3.4 Databases

Our data collection effort involved two tables in a PostgreSQL database[4]; the metadata, or `metas` table, allowed us to answer very simple, but useful, questions about student compilation behaviour in the BlueJ programming environment. The second table collected the specific `errors` that were encountered by BlueJ as it compiled each file in a given project.

Metadata

The SQL statement that was used to create the `metas` table is given in figure 3.5. This statement is not strict ANSI SQL, although the Postgres-specific extensions it employs have analogues on

[4]PostgreSQL is a robust, freely available, open-source database server. http://www.postgresql.org/

all major database servers. In particular, the two data types `serial` and `text` are not part of the ANSI standard. The `serial` type is an integer type that is guaranteed to be monotonically increasing for the life of the table; this is a simple way to provide an increasing, unique index. The second, `text`, is a character field limited to 2GB when PostgreSQL is running on 32-bit operating systems.

```
CREATE TABLE metas (
    "index" serial NOT NULL,
    uname text,
    homedir text,
    result integer,
    client_start integer,
    client_duration integer,
    server_receive integer,
    ip_address text,
    os_name text,
    os_arch text,
    os_version text
);
```

Figure 3.5: SQL to create the `metas` table on a Postgres database server

Many simple but important questions can be addressed with the metadata table alone. By looking at the `result` column, we can tell whether the student compiled code that was error-free (1) or contained a syntax error (0). We can also tell how long the compilation took (`client_duration`), and what time of day they initiated the compile, using either the system time on their machine (`client_start`) or the time at the server when the data was sent to the web server (`server_receive`). With this information, we can also develop a rough sense for how often students in our sample population pressed the "compile" button in BlueJ.

Errors

The `errors` table (Figure 3.6) contains one primary key (`index`) and one foreign key (`meta`). As it is possible for multiple errors to be associated with each row in the metas table, the link from the `errors` table to the `metas` table is essential. When students are working on a project with just one source file, BlueJ will only generate one syntax error at a time. When they have multiple files in a project, it is possible that each file might contain one or more syntax errors.

Therefore, for a project with five files, a single compilation event may generate up to five errors; the foreign key lets us know which errors resulted from a single compilation event.

```
CREATE TABLE errors (
    "index" serial NOT NULL,
    meta integer,
    uname text,
    etype integer,
    etime integer,
    emsg text,
    eline integer,
    fname text,
    file text
);
```

Figure 3.6: SQL to create the `errors` table on a Postgres database server

In the `errors` table, the `etype` takes on one of three values, indicating whether we captured an error-free compilation event, a warning, or an error. The `etime` field tells us when this event was generated on the client. If a warning or error message was generated by the compiler, it is captured in `emsg`, and the line number the compiler reported as being erroneous is stored in `eline`. The `fname` is the fully-qualified name of the BlueJ project. Capturing the full path lets us see if students copied and renamed an existing project (perhaps as a crude type of version control while exploring an idea); using this information, we could track the original source and the "branch" produced by the copy, separately.

The last column is the largest and richest in our database: `file`. This column contains the full text of the student's source code at compile-time. Each row contains the complete source from one class in a BlueJ project. Regardless of whether there was an error present in the code or not, we obtain a complete snapshot of the student's work with every compilation event.

3.3.5 Data preparation

Every time a student pressed the "Compile" button in BlueJ, we captured one event in our `metas` table, and one or more events in the `errors` table. While these raw tables could allow us to answer some questions about student compilation behaviour, we prefer to process this data before handling, as it provides an input format for our data handling scripts that is guaranteed to be self-consistent.

Processing of the `metas` table

Our data is more usable when processed. The `metas` table is reduced from its raw form (Figure 3.5) to a table with fewer columns of interest (Figure 3.7).

```
CREATE TABLE metas_processed (
    index integer,
    cindex  integer.
    session integer,
    uname text,
    result integer,
    client_start integer,
    server_receive integer,
    hostname text,
    host_type text
);
```

Figure 3.7: SQL to create the processed version of the `metas` table

Some elements of the table are copied over without processing: `index`, `cindex`, `result`, `client_start`, `sreceive`, and `hostname`. It is important to note that the `index` column is *not* regenerated in the `metas_processed` table, as this value serves as a foreign key to the `errors` table.

One of the additions to the table includes the explicit numbering of programming `session`s. We begin by denoting the start of a session as any time that the `cindex` (or `client_index` in the raw table) resets itself to zero. The only way this value can be reset is to restart BlueJ. We use this to denote the beginning of sessions. However, because we cannot tell the difference between a graceful shutdown of BlueJ and a crash, we merge sessions that begin and end, on the same machine, less than ten minutes apart into one session. We found that there was no difference between using a window of three, five, or ten minutes for the purpose of merging one session into another if they followed rapidly on from one-another.

The `uname` field is not copied from the `metas` table to the `metas_processed` table as well. While we are guaranteed uniqueness within our population, we do not necessarily want to use that identifier in our own analytic process and/or reporting. Therefore, we obtained a list of several thousand of the most popular baby names in the USA from the year 2000, and have randomly assigned our subjects names from this list. While a key is maintained (for the eventual matching up of course marks and other student performance metrics), those keys are stored separately from the database, and generally play no role in the day-to-day analysis of the data.

Lastly, the `host_type` is reduced from the three fields in the raw data (`os_name`, `os_arch`, and `os_version`) to one field that encapsulates all this information. For example, we acknowledge *w2k*, *xp*, *linux*, *sun*, and *macosx* as distinct operating system tags. However, the only four that enter into our research are *w2k*, *xp*, *linux*, and *sun*, as all of our data comes from the public computing laboratories. Therefore, they are either using the machine in its standard configuration (running Windows 2000 in 2002-2003, and Windows XP beginning in the summer of 2004), or they have connected remotely to a machine in the department running Linux or Solaris.

Processing of the `errors` table

Like the `metas` table, the `errors` table is also processed before it is used for analysis (Figure 3.8). The most important thing we do is copy the `meta` column directly from the source table to the new table, as this is our index into the `metas` (and now `metas_processed`) table.

```
CREATE TABLE errors_processed (
    "index" serial NOT NULL,
    meta integer,
    session integer,
    uname text,
    etype integer,
    etime integer,
    emsg text,
    eline integer,
    project text,
    fname text,
    file text
);
```

Figure 3.8: SQL to create the processed errors table.

The `etype`, `etime`, `emsg`, and `eline` columns are copied directly from the raw `errors` table. The `project` column is extracted from the file name captured in the `errors` table, as it is the only part of the file name we are interested in; it also helps eliminate the possibility of keeping a username in our database that might linger in a BlueJ project file-path. Likewise, the `fname` gives us the last of the "interesting" information that can be extracted from a full, canonical file name.

The `file` column contains the entire program. This code is *not* sanitized or processed in any way. As a result, our programs may contain a student's name in the comments. While we tend to focus on the portion of the document containing errors—the majority of our data could safely be placed on the web for download by other researchers interested in our work, except for the file column. The `file` column is, from a human ethics perspective, unsafe for general distribution. As a result, we have no immediate plans for attempting to release this data for other researchers to utilise.

Handling student source code

Traditionally, program code has been represented in the American Standard Code for Information Interchange (ASCII) character set. This 7-bit standard allowed the representation of 128 distinct characters on terminals and in print; this standard later evolved to a full 8-bit character set, allowing for 256 distinct characters. This isn't a significant problem for the English language, as the alphabet only has 26 characters (52 counting upper and lowercase), ten digits, and approximately 28 characters representing punctuation, white space, and so on. Add in a handful of control characters (newline, carriage return), and it becomes apparent that even ASCII had space left over for odd widgets, control codes for changing terminal colour, all outside of the subset necessary for writing in the English language.

The ASCII character set has always been a problem for languages with significantly larger alphabets; for example, Mandarin, Thai, Japanese, and other languages have alphabets consisting of thousands of unique characters. The Unicode standard[5] proposed expanding the representation of characters from 8 bits to 16 bits, allowing for the unique representation of 65,536 items. Furthermore, the standard allows for the composition of characters, meaning that accents and other typographic decorations can be added to base characters as a combination of one or more Unicode "code points."

The default text editors shipped with BlueJ 1.3, 1.3.5, and 2.0 are Unicode text editors. This

[5]http://unicode.org

means that it is possible, intentionally or otherwise, for students to enter characters into their programs that are outside of the standard ASCII character set. This is not a problem if every link in our data collection process is capable of handling Unicode.

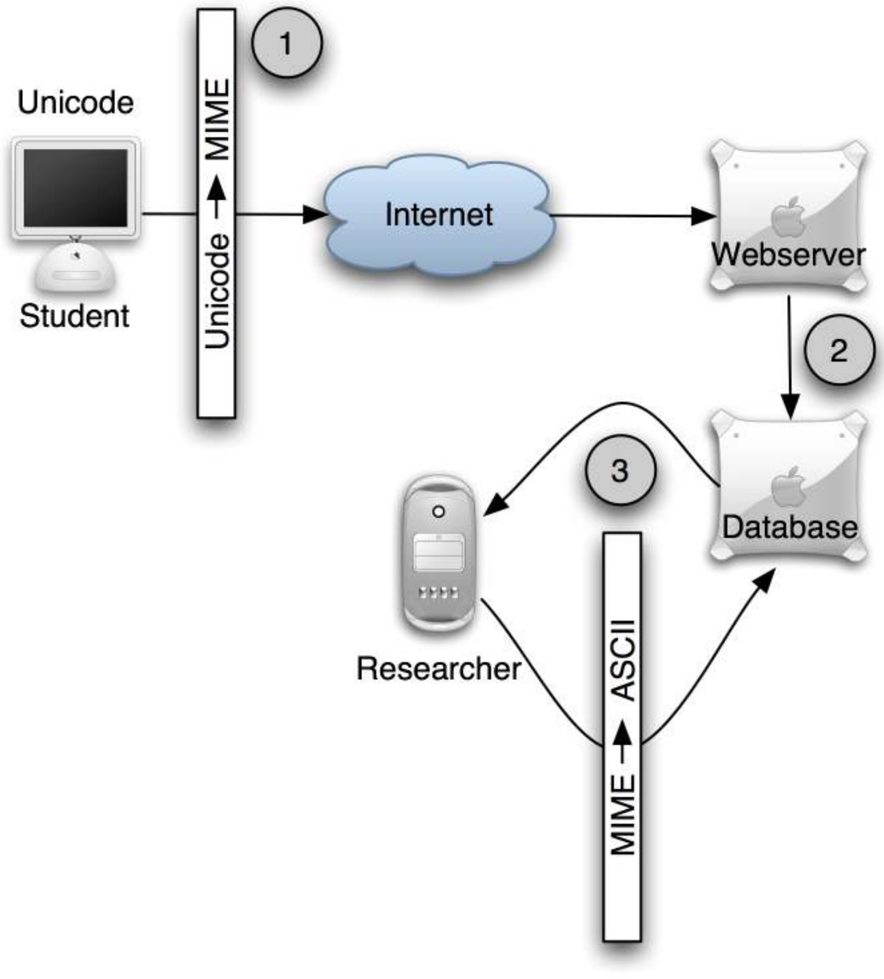


Figure 3.9: Collecting Unicode data requires careful consideration during data collection.

Figure 3.9 outlines some of the critical steps involved in handling Unicode text when developing an on-line protocol like the one employed in our own research.

The first, critical step is the export of data from BlueJ to the web server (Figure 3.9, #1). BlueJ is written in Java; internally, BlueJ has no trouble working with Unicode characters in its programs. Likewise, our data collection extension to BlueJ can easily extract the complete source for a student's program—but this data must be shipped over the Internet to the web server. In particular, it is shipped as an HTTP POST to a CGI application (Common Gateway Interface, or in other words, an application running on a server) . The protocol allows for the transport of Unicode character data; our initial solution for shipping data over the network involved no translation of characters, and instead shipped the students' source directly to the CGI application. As a result, the Java classes used for making the HTTP POST silently encoded any Unicode characters found into entities of the form &#x6C34 (this happens to be the Chinese character for "water").

Along with much of our data processing code, our CGIs were written in the programming language Scheme, using the freely available and open-source PLT implementation of the language[Fla05]. While our intermediary CGI application was *incapable* of handling Unicode characters, this did not effect us during our first semester of data collection. Our Scheme code did not convert the Unicode characters from their HTML entity representation into Unicode code points; instead, it processed them as ASCII text, and inserted the HTML entity representation directly into the database (Figure 3.9, #2). Likewise, our database was not prepared to handle Unicode data—but we were able to easily store the HTML entities in the UTF-8 character set employed by our Postgres database.

During our first semester of data collection, we noticed nothing problematic about our data collection. It is true that any accented characters in a student's code—like an `ö`, for example—would be converted to &#246. Fortunately, this does not effect any of our analyses. If we were interested in re-executing our students' code, we would need to take steps to process and repair these files. This is something that has not been critical to our analysis, and these problems were dealt with in the second semester of data collection and the 2004-2005 academic year.

Looking forward, we can take several steps to handle Unicode properly in the future. First, our data collection extension to BlueJ can perform its HTTP POST in the Unicode character set. Updating our web server (that executes our collection CGI applications) to Apache 2.0 provides a web server that is capable of handling Unicode. Likewise, updating our Scheme CGI applications to run on version 300+ of `mzscheme` will give us a Scheme runtime that is fully Unicode-aware. Lastly, our Postgres tables can be updated to the Unicode character set; this way, we can eliminate any conversion between character sets, and keep the students' source code in its native state from the start of the process all the way through the process of analysis.

3.4 Aggregate compilation behaviour

The infrastructure we built for collecting data regarding novice compilation behaviour is capable of harvesting large amounts of information over a long period of time with little or no intervention on the researcher's part. While we have found that the most interesting stories come from the in-depth study of individuals, looking at this data in the aggregate helps us understand the larger trends in our population's behaviour.

We collected data on 87 unique subjects during the 2003-2004 academic year (some of whom we only obtained data for the Fall term, and some only the Spring term), and 68 subjects throughout the 2004-2005 academic year. We captured 10,664 pairs of compilations in 2003-2004, and 24,852 pairs in 2004-2005. While each compilation has a unique context, the aggregate compilation behaviour of our students is consistent from one year (and one population) to the next.

3.4.1 Time between compilations

A notion of time and frequency of compilation is implicit in the unfair declaration that helped motivate our studies: "Our students aren't thinking! They're letting the compiler do their thinking for them!" It was interesting to us how much time students spent working on their programs between one compilation and the next. While we knew from our experiences in the classroom that students were prone to recompiling their code quickly, we were surprised by just how quickly, and how often, they would reach for the "Compile" button.

Figure 3.10 summarizes the time students spent working on their code between compilations for the 2003-2004 and 2004-2005 academic years. Reading this chart, we can see that 19% of all compilations captured during the 2003-2004 academic year took place less than 10 seconds after the previous compilation. Of all compilations recorded during that year, 17% were between 11 and 20 seconds after the previous compilation. The right-most "bin" in this figure shows the proportion of compilation events that were separated by a time of 2 or more minutes. The 2004-2005 academic year does not differ significantly from the 2003-2004 academic year; however,

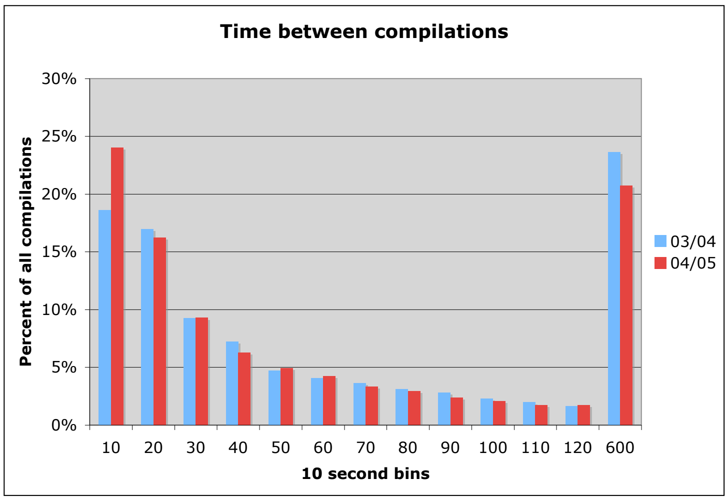


Figure 3.10: The percentage of compilations taking place in a given 10-second window for all compilations recorded during the 2003-2004 and 2004-2005 academic years.

we do think it is striking that 24% of all compilation events captured during the 2004-2005 year followed less than 10 seconds after a previous compilation.

These histograms tell us the same thing about both years: more than half of all compilation events occurred less than 40 seconds after the previous compilation. This tells us that our students are spending very little time between compilations examining their code, and work very quickly to add, remove, or modify their program between one compilation and the next. And while the leading edge of this distribution implies an exponential distribution, student compilation behaviour past the two-minute mark clearly indicates that it is not: roughly 22% of all compilations took place two more more minutes after the previous compilation. A simpler way to describe this behaviour might be to simply say that "a lot" of compilations follow quickly on from the previous compilation.

At first glance, this long tail is not particularly interesting to us, save to point out that student compilation behaviour is not following a typical, exponential probability distribution. Averaging over all of our data, we can look at the percentage of compilations that fell into any given

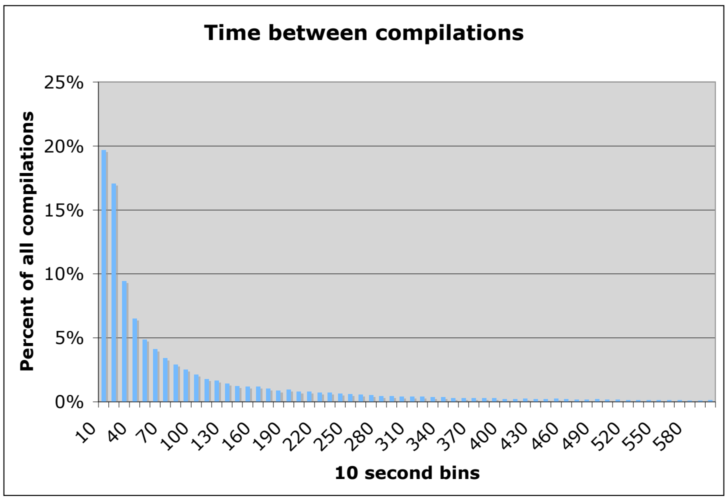


Figure 3.11: The percentage of compilations taking place in a given 10-second window for all compilations recorded in our study.

bin from 10 seconds up through 10 minutes (Figure 3.11). More interestingly, we can compute the continuous probability distribution of this data (Figure 3.12), and see that there is interesting behaviour taking place past the 2-minute mark that cannot be predicted by a crude, statistical model. Were this a strictly exponential distribution, the continuous distribution function would reach a limit (and the number of student compilations reach zero) early in the tail.

Exploring this long tail further, we can break each pair of compilation events into one of four "classes," based on whether the pair began with a syntax error (F) or from syntactically-correct code (T), and whether the student's changes yielded erroneous code (F) or error-free (T) code. Figures 3.13 and 3.14 capture this trend over time.

Both of these charts tell the same story: students spend more time working on their code after an error-free compilation (having eliminated all the syntax errors from their program) than when they are trying to fix a current syntax error. For example, looking at the "F→F" and "F→T" classes, these represent all the compilation pairs that began with code containing a syntax error. The student would not necessarily know if they had fixed the last error in their

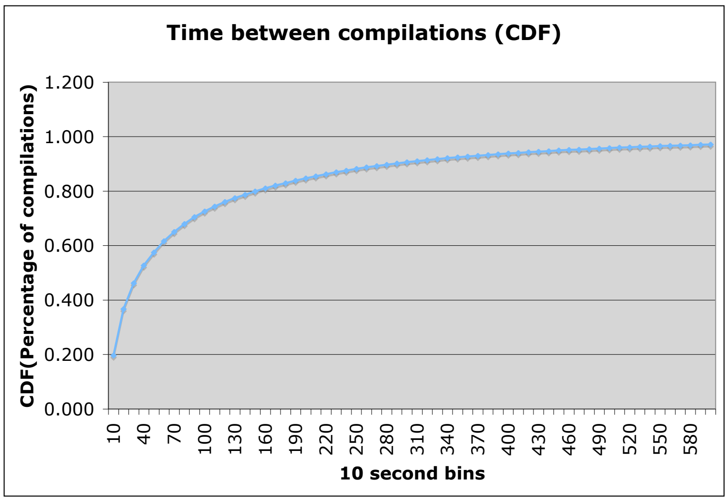


Figure 3.12: Continuous distribution function for the timing distribution of all compilation events recorded in our study.

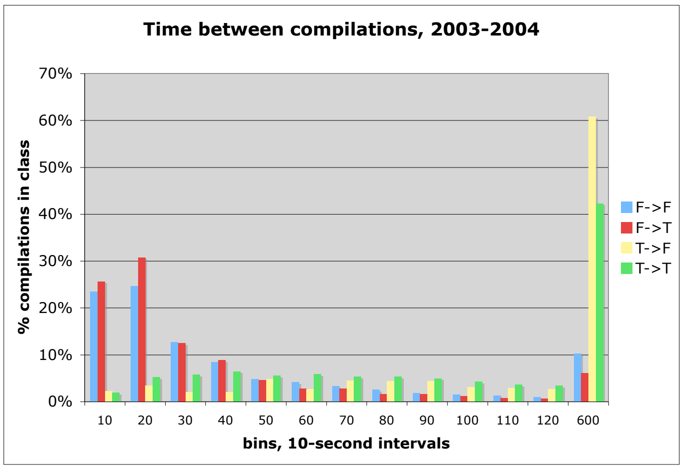


Figure 3.13: Time between compilations 2003/2004, considering whether the student was working from a syntax error (F) or syntactically correct code (T), and how the compilation ended (error = F, correct = T).

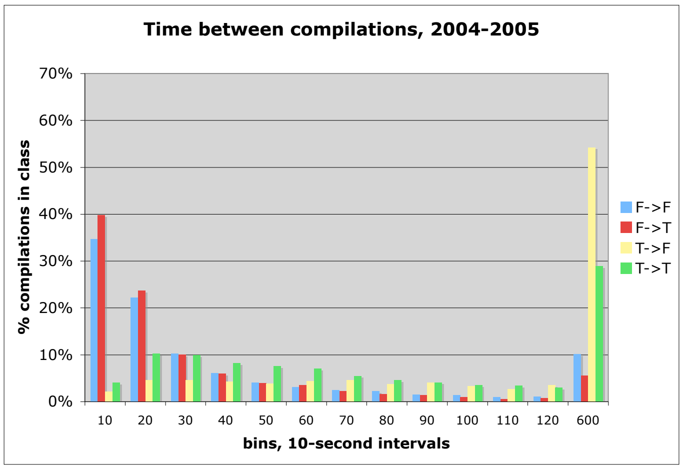


Figure 3.14: Time between compilations 2004/2005, considering whether the student was working from a syntax error (F) or syntactically correct code (T), and how the compilation ended (error = F, correct = T).

code at the time they press the compilation button, so we can easily see that more than 50% of all compilations beginning with a syntax error account for compilations that took place less than 20 seconds after the previous compilation. Stated more simply, if the student was correcting a syntax error, there was a greater-than-50% chance they would recompile their program in less than 20 seconds. Similarly, if the student was working on code that was syntactically correct, there was a better than 50% chance that they would spend at least 2 minutes working on it, possibly more.

From our observations, both in the classroom and through careful document analysis, it is clear that many students write significant amounts of code (10+ lines) at a time, and then attempt to eliminate all the syntactic errors that exist in the code. For example, the following piece of code has more than ten errors in it... or more than one error per line of code.

```
001  int largestValue() {
002     Iterator it = ls. iterator();
003     int temp = 0
004     while (it hasNext()) {
005       Integer intTmp =(Integer) it.next:
006       int num = intTmp. intValue ();
007       if(num >temp); temp = num;{
008         return temp;
009       }
010  }
```

Perhaps the most disturbing sequence of errors are on lines seven and eight; *the code is syntactically correct*. Unfortunately, it is far from *semantically* correct. So while a student may spend several minutes writing code like this, they will then spend at least as many, if not more minutes, attempting to fix the syntax errors. For many of the students in our study, it is unlikely that they can successfully correct these errors given the output of the compiler—that is, it is doubtful the compiler will be helpful enough to actually guide them in fixing all the problems in their programs.

It is code like this that tends to drive us away from coming up with abstract models that might explain the aggregate behaviour of our students. Our aggregate data suggests that students tend to write their programs in large blocks, and then spend significant amounts of time, in very small bursts, attempting to fix the syntax errors that exist in that code. In section 3.6, we look closely at the behaviours of our students as expressed by the code they write. While time-consuming, we found this analysis to be very enlightening.

3.4.2 Syntax errors: type and distribution

Students tend to recompile their programs quickly when it contains a syntax error, and are likely to spend a significant amount of time working on their code when it is error-free. The next question is quite simple: what kinds of syntax errors are the students typically dealing with?

Across the entire corpus, we found that 55% of all compilations ended in a syntax error. We currently recognise 85 different types of syntax error, all of which occur with varying frequency in our data. (A complete table of all syntax errors encountered, and their frequency of appearance, can be found in Appendix B.) Interestingly, the majority of these errors come from a minority of the error types; in [Jad05], we presented this data for the Fall 2003 semester only; we now have data regarding the distribution of syntax errors encountered by our students throughout of the 2003-2004 and 2004-2005 academic years.

Syntax errors in 2003-2004

In Figure 3.15, we can see the most common syntax errors encountered by students during the 2003-2004 academic year. The most common error encountered by students in our introductory programming class during the first year of our observation was the error "unknown symbol," and in particular in an identifier, or variable position. When the Java compiler reports this error, it is saying that the programmer has used an identifier that has not been previously declared; what this means is that the student has likely done one of four things:

1. The student may have misspelled a variable name; for example, they might have declared `foo`, but spelled it `fop` in the body of their code.
2. The student failed to capitalize a variable correctly, perhaps by declaring a variable `Foo`, but using it in their code as `foo`.
3. The student used a variable name, but actually declared something else—for example, declaring `Foo`, and then uses `Bar` in their code.
4. The student used a variable without declaring it.

Ultimately, we are interested in how often students fail to correct a given syntax error. For example, if they have declared a field of a class as

```
int Foo;
```

and attempt to initialize it in the constructor by writing

```
foo = 3;
```

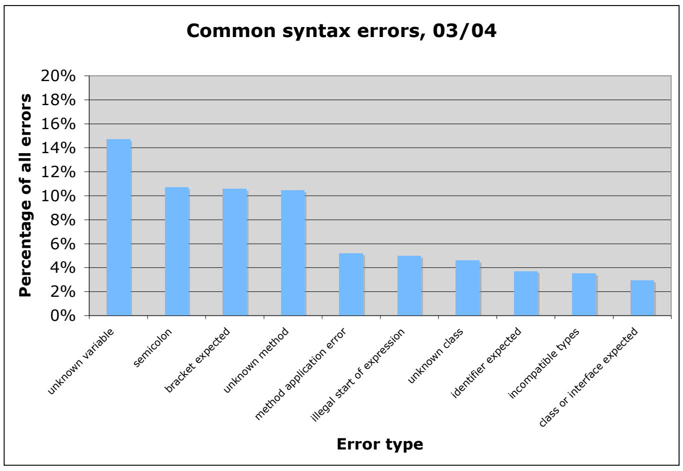


Figure 3.15: Common syntax errors in 2003-2004

the compiler will report `Unknown symbol: variable 'foo'`. Now, if they make a change like

```
foo = 5;
```

as a result of this error, we would say that they failed to correct the syntax error once; if the error persists after another compilation, then they have failed to fix this capitalization error twice in a row.

Conducting an automatic analysis of all programs written by students during the 2003-2004 academic year, 45% of all "unknown symbol: variable" errors were corrected immediately. Looking across all programs, written by all students, this particular syntax error occurs 883 times; 399 of these occurrences are fixed *immediately* (the error does not persist). 16% of the time students encountered this error, it took two compilations to fix the error; 9% of the time it took three recompilations.

This tells us that this particular error—the most commonly encountered error by students during the 2003-2004 academic year—is typically fixed in a small number of recompilations. Interestingly, the three most common syntax errors encountered by students during the 2003-2004 year—`unknown symbol: variable`, `semicolon expected` errors, and bracketing problems—are all fixed in one compilation roughly half the time they are encountered. Between 15% and 20% of the remaining occurrences are fixed in two compilations, roughly 10% of all occurrences require three recompilations for the student to correct the error. This is in contrast to the "illegal start of expression" error that clearly challenges many students, as only 33% of the times this error is encountered is it corrected immediately. Far more often students must wrestle with the error over two, three, or more compilations.

Appendix D includes a complete set of tables for our error repetition analysis for both the 2003-2004 and 2004-2005 academic years. The variation in repetition rates may imply that some errors are "harder" for students to fix than others. While the three most commonly encountered errors tend to be corrected immediately half of the time they are encountered, some errors (like `illegal start of expression`) clearly persist longer when encountered by novices.

Syntax errors in 2004-2005

During the 2004-2005 academic year, we collected twice as many compilation events as in the previous academic year, but the distribution of error types remained remarkably similar. Of the ten syntax errors that were most common during the 2003-2004 year, eight of them reappeared during the 2004-2005 academic year (Figure 3.16).

Unlike the previous year, two of the three most common errors in the 2004-2005 academic year have to do with the spelling/naming of identifiers in the language. Both the introduction of unknown variables (the most common error, responsible for 18% of all syntax errors encountered by students) and methods (10% of all errors) are very similar in nature. This is not entirely surprising; compiler research of the 1960's and 1970's is full of references and research regarding the automatic correction of spelling errors in programs[Mor70]. Unfortunately, it is only in professional development environments like Microsoft's Visual Studio or the open-source editor Eclipse that we see utilities integrated for the correction of spelling, despite spelling-related errors accounting for (up to) 28% of all syntax errors in our student population during the 2004-2005 academic year.

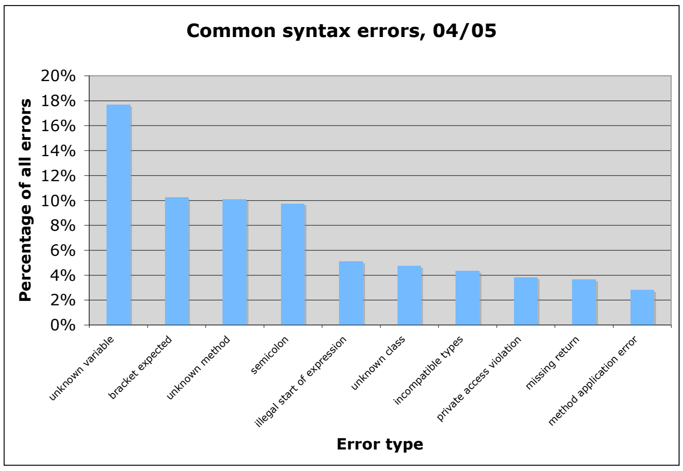


Figure 3.16: Common syntax errors in 2004-2005

Syntax Errors, 2003→2005

The same types of syntax errors dominated both the 2003-2004 and 2004-2005 academic years.

Figure 3.17 combines the top ten from both years of our study, and charts them side-by-side. A two-sample Kolmogorov-Smirnov test indicates that these two distributions are nearly identical ($D = 0.27$, $p = 0.83$)[Dal02]. This is good; the populations we observed over the course of two successive years tended to hit the same syntax errors with the same relative frequency. It is possible that we might even find the correlation would improve if we were to acknowledge superclasses of these syntactic errors. For example, "unknown X" errors (where X is either a variable, method, or class name) might all be classed as syntactic errors stemming from some sort of spelling error. Likewise, semicolon and bracketing errors might be grouped into a class of errors denoting syntactic delimiters. However, such groupings would not change the fact that the students in our study wrestled with the same kinds of errors from one year to the next.

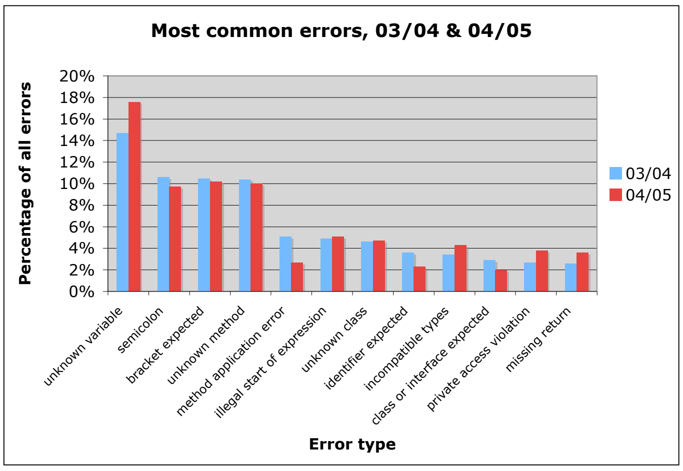


Figure 3.17: Common error rates in both 2003-2004 and 2004-2005.

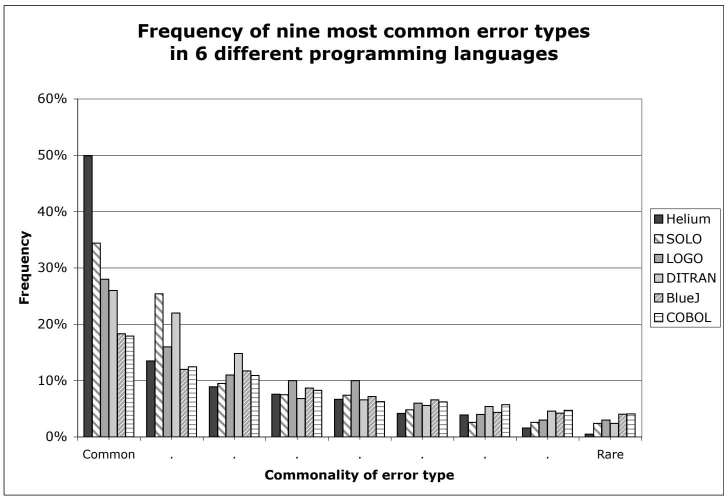


Figure 3.18: Distribution of common syntax errors in nine languages

Error distribution across languages

In the course of our research, we did find some data similar to that we are presenting here regarding syntactic errors encountered by our students during their first year of studying the Java programming language. Often reported as part of a larger investigation, this data is difficult to find, but interesting when brought together in one place.

Across six languages and/or environments, we found that the most common syntax errors followed a similar distribution (Figure 3.18). Helium (a pedagogic Haskell environment) was the most extreme example of this kind of distribution, as 50% of all syntax errors encountered by students using it were of one type. Given the nature of Haskell, we suspect that a refinement of the "type error" would bring this distribution more into line with the other languages. The BlueJ data presented in this chart is based on our Fall 2003 data (which does not differ significantly from either the 2003-2004 or 2004-2005 data); it is almost identical to the data presented in 1976 regarding syntax error rates from students studying COBOL[LD76].

Each of these error types is different; there isn't any particular relation that we can find between the classes of syntactic error captured in each of these languages. For example, the most common error we captured from students programming in Java using BlueJ is quite different from the type errors that were most prominent in students using Helium, and likewise very different from the kinds of errors observed in students using DITRAN or LOGO. While we have no way of evaluating why this distribution seems to play itself out over so many languages, we can easily posit two possible reasons: the programmer, and the grammar. If it is a result of the behaviour of the novice programmer, then that might support some notion that the students in our own study are quite similar from one year to the next, as well as being similar to other novices learning other languages. This is not unreasonable, in its own way, but we have little data to support a conclusion along these lines.

It is also possible that these distributions are inherent in grammars of the languages we program in. For example, there are many points where the Java compiler can fail a parse and throw a `; expected` error. In fact, it is difficult to find points in the grammar where the compiler won't complain about a semicolon expected; it is a prominent terminal throughout the language. Likewise, if you assume programmers are prone to making spelling and capitalization errors, there are countless places where a variable, method, or class name can be spelled incorrectly.

3.5 Visualizing the data

Presented in this section are a series of small vignettes, each exemplifying the compilation behaviour of students that we have observed. These vignettes focus on the programs written by the students, and how they evolved those programs over time. Making sense of these programs is a challenging task; it is easy to get lost from one compilation to the next. Trying to develop a "larger picture" of what a student is doing over five, ten, fifteen, or more successive edits becomes even more difficult.

3.5.1 Reading code: a grounded document analysis

Our single largest source of data regarding the students in our study are the programs they wrote. The easily quantifiable data we captured—the number of compilations they generated, the time between them, and so on—do not provide us with enough information to come to any kind of deep, or meaningful, understanding of an individual student and their programming behaviour. The programs they wrote, especially captured at each compilation, provide a "window" into a dynamic process that has otherwise been invisible to us as educators in the past.

Grounded theory

There are two broad ways to approach to the process of extracting meaning from the programs written by the students who took part in our study. One would be to take an existing theory of novice programmers, and through that focusing lens, study the programs they wrote and how those programs evolved over time. For example, we might begin with a notion of *goals* and *plans* as put forward by Spohrer and Soloway[SS86b, SS86a]. As this theory was originally put forward as a way of characterising the semantic errors students made while programming, we would be forced to reinterpret it in the light of our continuously evolving picture of a student's program over time. We might also begin with Richard Mayer's work on novices learning to program in BASIC, and his theories regarding the psychology of novice programmers[May81, BM83, MDV86]. Or, we could employ the Cognitive Dimensions framework, and attempt to characterise the difficulty, diffuseness, viscosity, and so on, of the constructs students were writing (or attempting to write) in Java at the point of each syntax error[Gre89b].

Each of these approaches would have been valid, and may still be carried out on the documents we have available—using theory as a focusing lens for understanding complex data is a reasonable technique. We felt that a grounded theoretic approach was more appropriate. Grounded theory was put forward by Glaser and Strauss in the late 1960's as way to approach complex, qualitative datasets and let the data guide the development of theory[GS67]. In essence, it is a theoretical framework that casts the researcher in the role of building new theories, where those theories are based entirely on the researcher's experiences in the field, interacting with artefacts, and so on.

Glaser and Strauss eventually parted ways over what they believed/interpreted grounded theory to *really* mean. Strauss and Corbin later wrote *Basics of qualitative research* in an attempt to set the record straight as to what they believed grounded theoretic work to truly mean[SC90]. Indeed, this debate rages on today throughout the qualitative research community. Piantanida, Tananis, and Grubs captured this debate in their article "Generating Grounded Theory of/for Educational Practice: The Journey of Three Epistemorphs"[PTG04].In it, they describe their experiences as qualitative educational researchers, seeking to find an epistemological haven for discussing, describing, and carrying out qualitative research in the educational arena. We can relate to their story as cross-disciplinary researchers; we want to make sound methodological choices without becoming lost in several decades of methodological debate.

Ultimately, our approach was most true to the early descriptions of grounded theoretic work put forward by Glaser and Strauss[GS67]. Our primary source of data was the distributed collection of programs students wrote using BlueJ. Our initial explorations attempted to characterise this data; for example, we read through all of the five most common errors captured in 2004-2005 in an attempt to categorise possible interpretations of the syntax errors reported in a given context. As we became more comfortable with our data, this coding became less interesting, as we were led towards a more in-depth reading of programs on a student-by-student basis.

A document analysis

A grounded theoretic approach to analysing our data involved working to annotate and document the dynamic process of making sense of our data and constantly reflecting new understanding back into our research. While we employed some limited, live observation of students in our study, this was not our primary tool for data gathering and sense-making; we were interested, first and foremost, in the programs written by our students.

The analysis of documents is a powerful tool for looking back in time when conducting research. They transcend, or otherwise "freeze" time, allowing us to look back at what students were doing at any given moment with great clarity[DL05]. Likewise, it is exact (unlike notes from direct observation), and can be interpreted qualitatively or quantitatively as the content allows; in the case of the programs written by students in our study, we have done both. However, as Yin points out, we face problems of biased selectivity; the data itself is incomplete in places, and we as researchers may be selecting "interesting" or otherwise "limiting" documents in our analysis that would otherwise unduly influence the process of sense-making and theory building that we are engaging in[Yin03]. We believe that our mixed qualitative and quantitative approach has provided us with some degree of protection from unnecessary bias in our work. We discuss the tools and techniques that have helped us keep sight of the "bigger picture" in our analysis, as well as the details, in Chapter 4.

Our deeper readings of individual sessions had us searching for differences between subsequent compilations in a given session, and then evaluating the changes students were making in the immediate and larger context of their program and the assignment it related to. Through an ongoing "sense-making" process, our initial goal was to identify compilation behaviours that appeared to be common across students in the population.

We began to see some patterns; for example, it is common for students to guess at the name of object methods. When presented with an `ArrayList`, how does one find the number of elements in it? Is it the object member variable `array.length`? By invoking `array.length()`? Perhaps it is `array.size()`? We've called this "guess the method"; many students seem happy to guess at method names instead of looking them up in the documentation. Likewise, "remove the error" is popular as well; when faced with an error that seems resistant to all of the student's best efforts, it is not uncommon to see them comment out or completely remove code that is problematic. Sometimes they reintroduce the code—sometimes they don't.

As interesting as these small behaviours were, we were making slow progress through the entire data set; with 42,000 pairs of files to examine over many hundreds of sessions, we needed a better way to take an entire session in at a glance. We needed a way of visualizing a single session that would allow us to quickly address many of these questions:

- Were there any large edits in the session?
  - Did they add, or remove, a large piece of code?
  - Was their edit at the beginning of a session, or the end?
  - Was it "idempotent"? (E.g. They removed the code, and then inserted the code again later?)
- Were there any long edits, where a significant amount of time passed?
- Were there any particularly problematic syntax errors?
  - What was the error?
  - Where in the code?
  - How many times did it occur in sequence?
  - How much time was spent working on that error?
  - Did the student fix it, or remove it?
  - Did the student work on other code while the error persisted?
- 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 are the kinds of questions that we found were interesting to ask when presented with a new session. As stated before, finding answers to these questions by working backwards from the source code was a difficult task—reading syntactically incorrect code, attempting to infer intention, and answer complex questions about the structure of the code was taxing. For this reason, we developed a way of visualizing a single session that would answer these questions for us.

Table 3.3 is an example of the tabular representation that we now work with when we begin exploring any given student's programming session. We begin with some basic metadata that helps orient us as to where the student is in the term. The date, project, and file name all tell us where in the syllabus the student is, so we have a rough idea of what concepts they should or should not be attempting to master at that point in time. It also gives us a sense for whether a particular syntactic construction might be new, as we can look at *Objects First with Java* and see if the syntax in question was being newly introduced in the assignment the student was working on. We can also tell at a glance the duration of the student's programming session in BlueJ.

This metadata is useful for quickly orienting us on when and where the student was working, but it is the following table that helps us see how a student spent their time during the session. Each row in the table represents the change between a pair of consecutive compilation events. Taken singly, each column tells a story. From left-to-right, the columns in the table are:

Err Type The error type column tells us two things. First, we can see if a particular compilation pair was successful, implying the compilation was error-free. If it was, we have denoted this with a $\star$. More commonly, we find a number. This number provides a unique index into our table of recognized syntax errors (Appendix B). We have applied colour to this table to make it easier to see the repetition of syntax errors. In our example, Louis was clearly stuck on error type #8 ("Class or interface expected"). Even if we don't look up what error type #8 is, we can see at a glance the large swath of blue that dominates the error type column in this session.

ΔT We know the amount of time (in seconds) that passed between compilations; in our visual overview, we have further reduced this to five bins: 0-10 seconds, 20-30 seconds, 30-60 seconds, 60-120 seconds, and more than two minutes. This provides us with a summary of which compilations were quick and reflexive, and which were the result of (possibly) more thought and work on the student's part.

ΔCh How many characters were added or removed from one compilation to the next? This gives us a sense for the magnitude of the change. A single character might mean the addition of a semicolon; two characters might mean they commented out a single line. The addition of 20-50 lines might mean a new method was written. One to two hundred characters might mean a cut or paste operation was performed. Also, we can "see" when a student has likely removed and added a line or lines to their program; if we look at compilations 10, 11, and 12 in this particular session, we see that Louis removed thirteen characters from his program (which resulted in a successful compilation), he added thirteen characters in (resulting in a syntax error), and then he removed thirteen characters again. While we do not know if those were *necessarily* the same characters, we can now dive into the code at that point and investigate further.

Location The location column represents the extent of a student's program file: the left side of the column is the start of the document, and the right side of the column represents the furthest point a student edited their program. Implicit in this column is a notion of time: the pink rectangle tells us where a syntax error was reported coming into a particular edit, while the black dots tell us where the student edited in response to those syntax errors. If we look at the first compilation in the table, Louis began with a syntax error reported near the end of his program; he added one character to his program *on the same line as the syntax error*. It is not often that we are interested in any one particular compilation when reading this column; instead, we tend to read the column as if it were an unrolled filmstrip. We can see, over time (from one compilation to the next) where the errors were in a student's program, as well as where they edited their code in response to those errors. As we can see from Louis's errors and edits, his attention was focused on the same few lines of code for the majority of the session.

3.5.2 Digging deeper

With a tabular representation of a novice's compilation session, we can, at a glance, tell how many times they were successful in removing all the syntax errors from their program, how much time (generally speaking) they spent working on their code between compilations, how many characters they added or removed, and most importantly, where their attention was probably focused over the course of the session. None of these measures are terribly precise; for example, when a student takes 10 minutes between one compilation and the next, they add one character, and fail to fix a syntax error, we do not know where their attention was focused during that entire ten minutes. However, the table provides an overview of the session that would otherwise be very difficult to build up simply by reading the students source code from one compilation to the next.

Our exploration of students' interactions with the compiler provides a basis for future work exploring further interactions with the BlueJ IDE. For example, from compilation 4 through 9, it is not possible for Louis to have tested or otherwise executed his program. Why? Because the code was never error free. It is not uncommon to see sessions where fifteen, twenty, or more minutes go by where the code is never devoid of syntax errors. This means a student is spending large amounts of time wrestling with syntax errors, not the assignment at hand. And it certainly means they are not in an edit-compile-test cycle, but instead are engaging in an edit-compile-edit cycle that is, perhaps, less instructive in the long run.

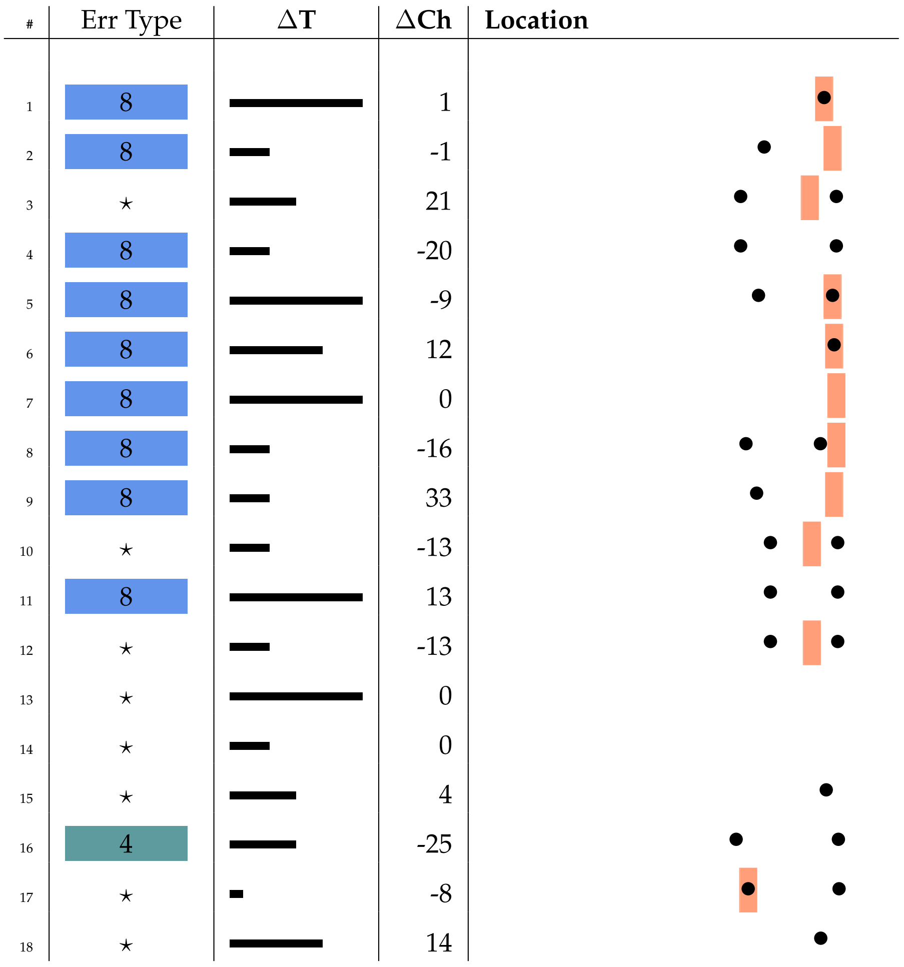
3.6 Vignettes

To help the reader develop a sense for the depth and richness of the data we have captured, we present a series of vignettes, each of which walks through one programming session of one student. Each vignette illustrates trends we have observed throughout the population; each vignette is also representative of a class, or category, of students in our population. Some of the vignettes we have focused on were found through the careful reading of source code. Others were found by reading through our tabular representations, looking for what looked like a problematic or otherwise interesting session.

Each of the vignettes described here captures a sequence of edits made on an assignment from chapter three of *Objects First with Java*. This assignment involves writing code to insert and remove pages from a virtual Notebook; in working on this project, students combine their knowledge of how to define classes of their own definition with lists of data.

Table 3.3: Louis, Session 2

Project:	weblog-analyzer	**File:**	LogAnalyzer.java
Duration:	33m19s	**Error Rate:**	0.25
Date:	Thursday, November 6th, 2003 11:13am		



#	Err Type	ΔT	ΔCh	**Location**
1	8		1	
2	8		-1	
3	⋆		21	
4	8		-20	
5	8		-9	
6	8		12	
7	8		0	
8	8		-16	
9	8		33	
10	⋆		-13	
11	8		13	
12	⋆		-13	
13	⋆		0	
14	⋆		0	
15	⋆		4	
16	4		-25	
17	⋆		-8	
18	⋆		14	

3.6.1 Neville: running in place

The first student we will examine in-depth is Neville, and he struggled a great deal with syntax errors. The code we will examine represents two-thirds of one session, captured in October of 2004. Specifically, we will examine the first 30 minutes of a 45-minute programming session, during which time almost all of Neville's efforts are focused on just a few lines of code.

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.

```
059  //A class to remove a note
060  public void removeNote(int noteNumber)
061  {
062    if(noteNumber < 0) {
063      // This is not a valid note number, so do nothing.
064    }
065    else if(noteNumber < numberOfNotes()) {
066      notes.remove(noteNumber)
067    }
068  }
```

This particular addition of code is just one snapshot that is part of an otherwise long and complex process. We have attempted to capture some of that process in this this chapter by presenting code fragments in pairs. On the left, we will present the state of a student's program before one compilation, and on the right, the changes they made in the time before the next compilation. Due to the constraints of the page, we will occasionally have to stack these *Before* and *After* pairs on top of each-other.

For example, after the first compilation, Neville edited his code for five seconds, and then compiled his code again; in those five seconds, he made one small edit—he added a semicolon to line 66 of his program.

Before

```
059   //A class to remove a note
060   public void removeNote(int noteNumber)
061   {
062     if(noteNumber < 0) {
063       // This is not a valid note number, so do nothing.
064     }
065     else if(noteNumber < numberOfNotes()) {
066       notes.remove(noteNumber)
067     }
068   }
```

After

```
059   //A class to remove a note
060   public void removeNote(int noteNumber)
061   {
062     if(noteNumber < 0) {
063       // This is not a valid note number, so do nothing.
064     }
065     else if(noteNumber < numberOfNotes()) {
066       notes.remove(noteNumber);
067     }
068   }
```

First, we see the program as it was after the first compilation in the session, and then we have highlighted line 66, to help indicate where Neville made edits in the time between the two compilations.

This addition of an entire method over two compilations might seem impressive, considering we have already described Neville as a student who struggles with syntax errors in his programming. Because the authors of *Objects First with Java* believe that students should never start with a blank page, we sometimes need to double-check the code students have added;

in this case, the `removeNote` method is provided on page 85 of *Objects First with Java* (second edition). The fact that Neville inserted nine lines of code, correctly, with only one syntax error, is not terribly surprising.

After adding the `removeNote` method, Neville goes on to add another 13 lines of code in four minutes. At the end of the four minutes, he has added the `listAllNotes()` method; this method will be the focus of the rest of our discussion.

```
070   //List all the notes in the arraylist
071   public void listAllNotes()
072
073   int indexNum
074   {
075     if(noteNumber < 0) {
076       // This is not a valid note number, so do nothing.
077     }
078     else
079     while (indexNum < numberOfNotes) {
080       System.out.println(notes.get(indexNum));
081       indexNum++  ;
082     }
083
```

An experienced Java programmer is unlikely to make the mistake that Neville has made here. If we look closely at lines 71 through 74, we see that he has introduced the variable `indexNum` outside of the opening { of the method. The code could be corrected by swapping lines 73 and 74—or, perhaps we should say that the code could be made "more correct" by way of this switch. We have illustrated what Neville could have done to correct this syntax error in the next pair of code fragments; it takes Neville 30 minutes, and many edits, to discover this for himself.

	What Neville wrote...		*... “corrected” by swapping two lines*
070	`//List all the notes in the arraylist`	070	`//List all the notes in the arraylist`
071	`public void listAllNotes()`	071	`public void listAllNotes()`
072		072	
073	`int indexNum`	073	`{`
074	`{`	074	`int indexNum`

Unfortunately, Neville does not make this correction. As the compiler is currently reporting a `';' expected` error on line 72, he looks at his code, and realizes he is missing a semicolon—on line 73. So, Neville adds a semicolon to the end of line 73.

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

Neville’s addition of a semicolon on line 73 cannot possibly have fixed the error on line 72. While the Java compiler may report many “strange” errors from a novice programmer’s perspective, it is consistent in one regard: it does not report errors before they happen. The compiler may report the location of an error as being somewhere after the actual location of the error—perhaps even the end of the file—but the compiler will never fail due to an error that takes place one, two, or more lines ahead of the reported location.

Neville recompiles his program; the compiler reports, again, a `';' expected` error 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. This time, he adds one to the end of line 71. Because the header of the method `listAllNotes()` is on its own, away from the body of the method, it is easy to see it as a line wanting for a semicolon. However, this is not the case—it is unlikely that Neville, as a novice Java programmer, will ever have any need to add a semicolon to an otherwise empty method header.

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

The result of this rapid addition of a semicolon and recompilation yields a rather uncommon 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—we don't know why, but perhaps it was to refresh the error in the BlueJ environment, as clicking anywhere would have caused the message to disappear. Fifteen seconds after this recompile, he removes one character from his program—the semicolon he had just added to line 71.

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

As a result, the compiler once again reports `';' expected` on line 72. Neville makes a small edit elsewhere in his program, and then returns to the `listAllNotes()` method. In coming back to this piece of code, 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: a `';' expected` error on line 72, and no semicolon at the end of line 71 or 73. This edit follows, below.

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

Remove the error

After nearly a quarter of an hour, Neville has not managed to correct what we would, as experienced programmers, consider 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; we refer to it generally as "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		*After*
070	`//List all the notes in the arraylist`	070	`//List all the notes in the arraylist`
071	`public void listAllNotes()`	071	`public void listAllNotes()`
072		072	
073	`int indexNum`	073	`//int indexNum`
074	`{`	074	`{`

This is a critical point in the development of this code; Neville has effectively *removed* the source of the syntax error in his program. The syntax error that has now consumed a quarter of an hour of Neville's time is, for all intents and purposes, gone. Other problems can now be caught by BlueJ; unfortunately, in his current context, these new syntax error messages seem to have the undesirable effect of leading Neville back to a syntactically incorrect version of this program.

After compiling this code, the compiler reports `cannot resolve symbol - variable noteNumber` on line 75. Looking at the method `removeNote` that Neville added first, and the `listAllNotes` method, we must wonder if the majority of the `listAllNotes` method was actually copied from `removeNote`. The variable `noteNumber`—responsible for the error, as it is not in scope in the method `listAllNotes`—is a hold-over from a copy-paste we suspect Neville made.

There are several ways Neville could fix this error. He could introduce a new field in this class called `noteNumber`, a formal parameter to the method with this name, or a local variable. He could also decide that the variable is incorrectly named, and change it to something that is in scope. If we consider syntax errors as a stimulus, and the resulting edit as a response—a comprehensive and difficult analysis we have not, at this time, carried out—we might say that Neville's edit in this case is a "good" response. He renames `noteNumber` to `indexnum`, and then uncomments line 73.

	Before		*After*
071	`public void listAllNotes()`	071	`public void listAllNotes()`
072		072	
073	`//int indexNum`	073	`int indexNum;`
074	`{`	074	`{`
075	`  if(noteNumber < 0) {`	075	`  if(indexnum < 0) {`

Syntax Blindness

Some of our students, it would seem, are *syntax-blind*. We do not have a better word for what we see in edit traces like this one. Neville is, it would seem, unable to see the syntax error that is present before him, and many other students exhibit similar behaviours when faced with a missing parenthesis, bracket, or other piece of punctuation.

Neville must believe that he is reintroducing the variable `indexNum` to the scope of the method `listAllNotes`; certainly, we would argue that he knows he needs to declare the variable. However, in uncommenting this incorrectly positioned variable declaration, he only manages to reintroduce the syntax error `';' expected`. His response is quick and, unfortunately, reverses the edit he just made: Neville proceeds to comment out line 73.

Before		*After*	
071	`public void listAllNotes()`	071	`public void listAllNotes()`
072		072	
073	`int indexNum;`	073	`//int indexNum;`
074	`{`	074	`{`
075	`  if(indexnum < 0) {`	075	`  if(indexnum < 0) {`

Commenting out line 73 does not fix the problem. It is at this point that Neville realizes that the capitalization of the variable *indexnum* is incorrect. So, he converts it to the mixed "camel case" that is common in Java naming conventions, so it matches the variable declaration two lines previous. After renaming the variable `indexnum` to `indexNum` on line 75, the compiler reports the same `';' expected` error on line 72. Although we are not typically concerned with the time students take between compilations, we would note here that the previous three edits each took Neville less than five seconds.

Before		*After*	
071	`public void listAllNotes()`	071	`public void listAllNotes()`
072		072	
073	`//int indexNum;`	073	`//int indexNum;`
074	`{`	074	`{`
075	`  if(indexnum < 0) {`	075	`  if(indexNum < 0) {`

The next edit Neville makes is to—once again—uncomment line 73. This is reasonable; there is a variable declaration for `indexNum`, the usage of the variable has been correctly spelled, so if he uncomments line 73, everything should be fine, and he can move on. Unfortunately, the compiler is still reporting a syntax error on line 72, as the original problem—the fact that lines 73 and 74 need to be swapped—has not been addressed.

	Before		*After*
071	`public void listAllNotes()`	071	`public void listAllNotes()`
072		072	
073	`//int indexNum;`	073	`int indexNum;`
074	`{`	074	`{`
075	`  if(indexNum < 0) {`	075	`  if(indexNum < 0) {`

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 is facing). With some stronger students we observe this as well—they will ignore a syntax error to work on some other part of their program, and then come back to the error, and fix it. With students like Neville, however, this is less common. More 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 Perkins et. al's investigations of beginners working in BASIC and LOGO[PM86]; Perkins might refer to such a student as a "tinkerer," who keeps moving, but not necessarily for the better.

At this point, Neville makes the largest removal of code we see in the entire session: he removes lines 76 through 79, eliminating the `if` construct in this method completely. This is good, for several reasons. First, the `if` will play no part in a straight-forward, successful solution to this problem. Second, the unbracketed `else` clause, while having reasonable semantics as the code stands now, is a likely source of woe later. Our experience reading through student traces is that the failure to use brackets on `if`, `while`, `for`, and similar constructs is often a source of syntax errors that are difficult to debug.

A large removal of code

```
//List all the notes in the arraylist
public void listAllNotes()

int indexNum;

{
  if(indexNum < 0) {
    // This is not a valid note number, so do nothing.
  }
  else
  while (indexNum < numberOfNotes) {
    System.out.println(notes.get(indexNum));
    indexNum = indexNum  1;
  }

```

Little patterns

A great deal of programming is made up of the repetition of small, idiomatic patterns and structures. What is not explicitly captured in the language is often then expressed in the form of convention, style guides, design patterns, and other ad-hoc tools for organizing code and making it more comprehensible to the programmer and, later, the maintainer of the code.

For example, Neville's next edit captures a typical "little pattern" in many C-like languages: after declaring a variable, it must be *initialized* to some value. Sometimes this is combined into one statement:

```
int x = 3;
```

and sometimes these declarations are separated from each-other:

```
int x;
x = 3;
```

After removing the `if` statement from his code, the next thing Neville does to his program is initialize the variable `indexNum`. Currently, it is declared (unfortunately, in a syntactically incorrect location), but it is not initialized. It is good that he notices that his attempted use of `indexNum` occurs before it is initialized, it is also worth noting that the Java compiler will catch the use of uninitialised variables, and report that to the programmer. Currently, the compiler is not complaining about `indexNum` being uninitialised—it is dying on line 72 because Neville has declared a variable in the syntactic badlands between his method header and the opening bracket of the block of code that follows the header.

Unfortunately, if Neville is "moving on," this is still not helping. In initializing the variable `indexNum`, he makes space for a new line of code—outside the start-of-method bracket!

Before		*After*	
071	`public void listAllNotes()`	071	`public void listAllNotes()`
072		072	
073	`int indexNum;`	073	`int indexNum;`
074		074	`indexNum = 0;`
075	`{`	075	`{`

This kind of editing continues for the remainder of the session. Neville does, at one point, take a short "break" from lines 70-74, modifying some `System.out.println` statements earlier in his program. He introduces a new error there, fixes it, and comes back to this problematic code. Surprisingly, it is at this point that he manages to correct the program. Perhaps by "stepping away" from the point where the problem is he was able to see the syntax error for what it was. Or, perhaps someone helped him? In truth, we cannot know what took place, as we do not know what documentation, examples, or other help he may (or may not) have sought in attempting to debug this particular program.

Neville: stepping back

From our walk through just one of the sessions we have captured of Neville programming, we can see a number of things. First, we suspect that Neville was working too quickly to be reading and comprehending the nature of the syntax error the compiler discovered. Furthermore, his strategies appear to be haphazard—like the addition and removal of semicolons and comments in an attempt to change the output of the compiler. We might even ask how he managed to write as much code as he did, given his apparent lack of ability in debugging the syntax errors that were present in that code. Without a doubt, we would want to recommend one-to-one help for Neville so that he might develop some more advanced strategies for these kinds of syntax errors in the future. His inability to deal with errors quickly and easily will only hinder him in the long run.

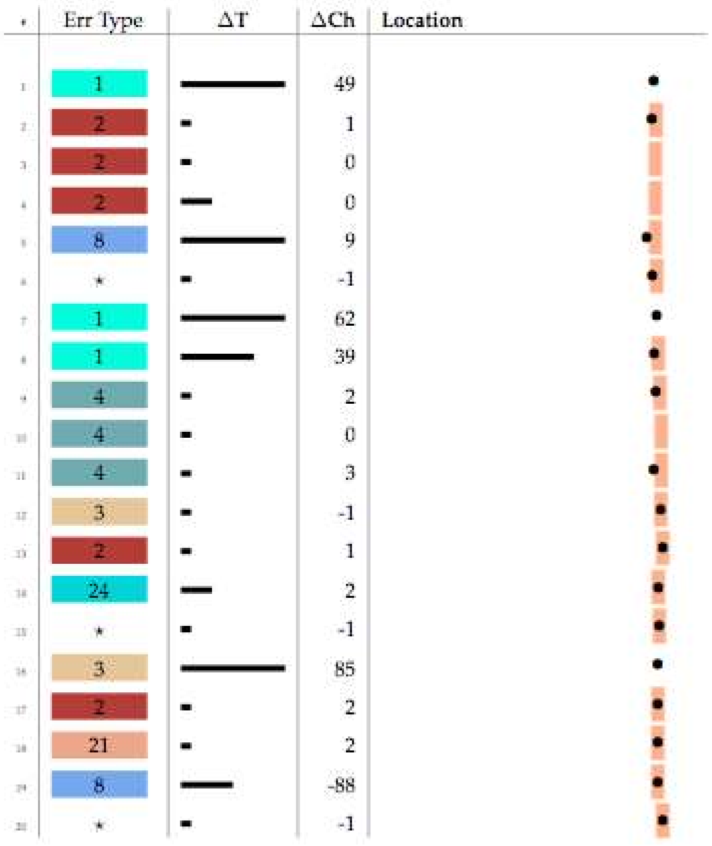
Having worked through the complete sequence of compilations from just one programming session, hopefully it becomes clear how rich and complex the data we have collected is; each session tells a story. While we would like to read every story in our database, it is important that we develop a sense for what our entire population of subjects is doing, and focus on tools that can help us understand our students, both as researchers and as instructors.

Table A.1 (in Appendix A) is the tabular visualization of the program Neville has written. While we know, from reading through the source, that he was stuck on some very basic syntax errors—and all in the same part of his program—we can see the locations of syntax errors (pink boxes) and locations of edits (black dots) make a nearly vertical line. That these locations all line up vertically tell us that Neville did all of his editing in one place, save for a bit at the beginning and end. Also, the kinds of syntax error, and the number of times they repeat, tell us that Neville struggled largely with one type of syntax error; while we saw this, reading through one compilation after another, it is much easier to see the repetition when laid out in a tabular form.

So, at a glance we can use this representation to tell us a great deal about Neville's experiences while working through a session. We can see he spent almost all of his time in one part of his program (indicated by the fact that all of the errors and edits "line up"), that he wrestled with one type of syntax error for the majority of the session, and that this struggle began with two large additions of code, after which he made very small edits throughout the majority of the session as he attempted to fix the syntax error that was plaguing him.

3.6.2 Ron: A picture of motion

A session trace helps us to quickly see how a student spent their time while developing their code. So, instead of attempting to develop a picture of Ron's session working only from the source code he wrote, we might begin by looking at Table 3.19 (also A.2 in Appendix A). This session trace lets us quickly summarize his efforts on the Notebook project.



#	Err Type	ΔT	ΔCh	Location
1	1		49	
2	2		1	
3	2		0	
4	2		0	
5	8		9	
6	*		-1	
7	1		62	
8	1		39	
9	4		2	
10	4		0	
11	4		3	
12	3		-1	
13	2		1	
14	24		2	
15	*		-1	
16	3		85	
17	2		2	
18	21		2	
19	8		-88	
20	*		-1	

Figure 3.19: Ron, hard to distinguish from Neville

What does this session trace tell us? Ron spent most of his time working in roughly the same part of his program as Neville, and like Neville, he did not do much else. The session is shorter—there are fewer compilations, and this particular session is 20 minutes shorter than Neville's. At a glance, we can see that Ron did not spend any long sequences wrestling with just one type of error, but instead he encountered seven different kinds of syntax error during the twenty five minutes he spent working on the project. So, like Neville, Ron was stuck in one place, but unlike Neville, he did not get "hung up" on any one kind of syntax error.

The first thing Ron does is write the method `remove`, which we can see at the start of the trace; the `';' expected` error is dealt with quickly.

Before		*After*	
059	`public void remove(int index)`	059	`public void remove(int index)`
060	`{`	060	`{`
061	`  notes.remove(note)`	061	`  notes.remove(note);`
062		062	

Over the next three compilations, Ron does not make any changes to his program; thirty seconds pass while he hits the "Compile" button, changing nothing. The error is `cannot resolve symbol - variable note`, and one possible fix is to rename the variable `note` in line 61 to `index` (matching the formal parameter to the method `remove`), or renaming the parameter. Either way, this apparently does not occur to him, so he makes a change to his program that lets him continue on with the tasks set before him.

Before		*After*	
059	`public void remove(int index)`	059	`/** public void remove(int index)`
060	`{`	060	`*{`
061	`  notes.remove(note);`	061	`  * notes.remove(note);`
062		062	`*/}`

Ron comments out the entire method. For the remainder of the session, he never comes back to this method. This is significantly different than the strategies employed by Neville; whereas Neville seems to be stuck on the only error in his program, Ron moves on. Perhaps he has decided to come back to it later, perhaps he intends to ask for help from a peer or instructor; either way, we don't know. Regardless of his possible intentions, Ron leaves this particular error behind. We feel this is, generally, a good behaviour—being able to move past one error and work elsewhere, perhaps returning later to the problem code—but we also find it troubling that Ron does not know how to deal with an `unknown symbol` error four weeks into the term.

Ron proceeds to add half of a method:

```
063    public void printAllNotes()
064    {
065      int i = 0;
066      while (i < numberOfNotes);
067      {
068        system.out.println(notes)
069        i = i + 1
070      }
071
072    )
```

Figure 3.20: Ron's first attempt at the `printAllNotes()` method.

```
063    public void printAllNotes()

064    {

065      int i = 0

066      while (i < numberOfNotes)
```

which he compiles, generating a syntax error—and then adds the other half of the method. He ends up with a `printAllNotes()` method that looks reasonable for a first attempt (Figure 3.20).

There are a number of small problems with this code, that Ron manages without too much trouble. He correctly realizes (with prompting from the compiler) that `numberOfNotes` should actually be a method call (e.g. `numberOfNotes()`), and that `system.out.println` should have a capital 'S'. However, the fact that he closed all of his code off with a parenthesis instead of a curly bracket (line 72) takes several `illegal start of expression` errors (compilations 10, 11, and 12) before he coaxes a `'}' expected` error out of the compiler, making his mistake obvious.

Once he has worked through these errors, he makes his next large addition of code, shown below.

Before

```
public void printAllNotes()
{
  int i = 0;
  while (i < numberOfNotes());
  {
    System.out.println(notes);
    i = i  1;
  }



}
}


```

After

```
public void printAllNotes()
{
  if numberOfNotes = 0
  {
    System.out.println("There are no notes in the notebook");
  }
  else
  {
  int i = 0;
  while (i < numberOfNotes());
  {
    System.out.println(notes);
    i = i  1;
  }
}
```

Ron spends several compilations wrestling with this code, after which he decides that it doesn't actually matter—and he removes the new `if` clause completely. We don't know why he removed it, but it could be he decided that didn't actually improve his program or make it function correctly. Given that it never compiled successfully between the time he added it and removed it, he certainly could not have *tested* his program, so the reason for both the addition and removal remains a mystery.

Ron: stepping back

Compared with Neville, it feels like Ron makes much more progress during the time he spent programming. His ability to deal with syntax errors (or, as the case may be, realize that he *cannot* deal with a syntax error) is, we suspect, a more successful strategy in the long-term than that exhibited by Neville. While almost all of the compilations in this session ended in syntax errors, Ron managed to avoid getting hung up on any one error.

We are not trying to claim that one of these two students exhibits compilation behaviour that is particularly "good." Certainly, neither spent any significant amount of time working with syntactically correct code. This means that neither student spent any significant amount of time testing their ideas and solving the problems set for them by their instructor; instead, they spent the majority of their time wrestling with one (Neville) or many (Ron) syntax errors.

3.6.3 Harry: persistence over form

Unlike his other classmates, Harry is nothing if not persistent. However, his methods are not always the most elegant—but that, perhaps, does not really matter. His session trace is captured both in Figure 3.21 and Table A.3 in the appendix.

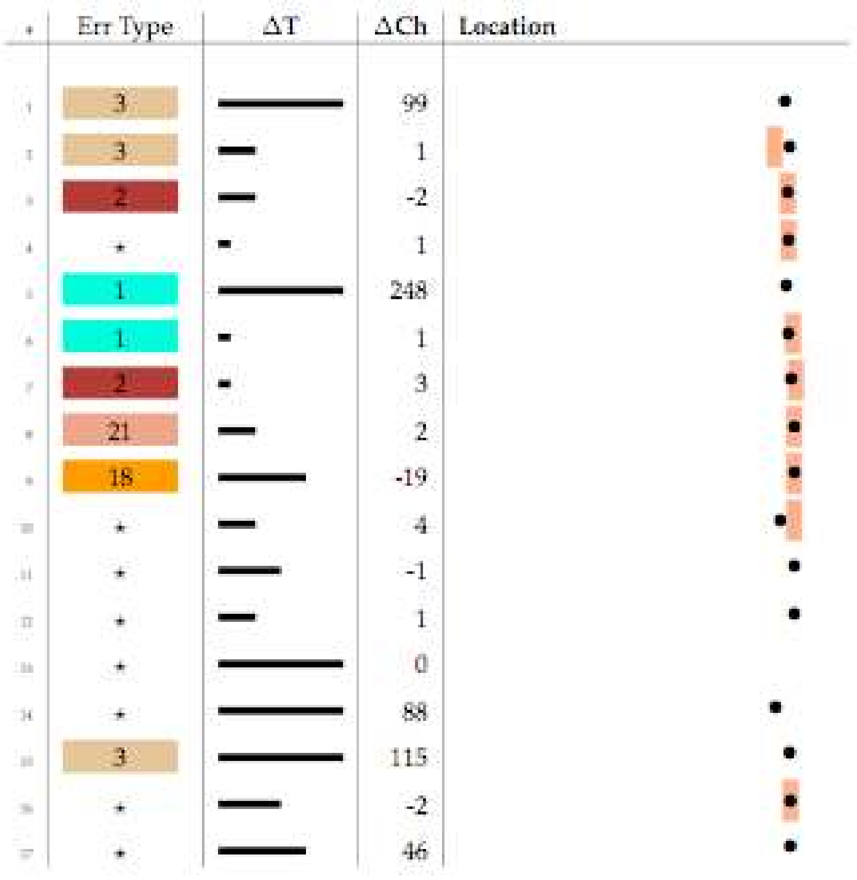


Figure 3.21: A session characterized by Harry's persistence.

We can see at a glance that Harry's efforts on the Notebook project are not dominated by the presence of syntax errors. His editing focuses on the `listAllNotes()` method. However, unlike both Neville and Ron, this session is punctuated by many syntactically correct compilations. This implies to us that Harry was able to test his program. In fact, as can be seen from the trace, Harry makes a number of edits that are syntax-error free near the end of his session; perhaps he was editing, testing, and editing again?

Our attention on Harry's programming will be focused on the `listAllNotes` method, which he introduces between compilations five and six.

```
066   //listAllNotes - prints all stored notes
067   public void listAllNotes() {
068     int i
069     System.out.println ("List of all notes:");
070     while (notes !=null) {
071       while (notes.isEmpty = false) {
072         System.out.println("Note " + i + ":" + notes.get(i));
073         i++
074       }
075     }
076   }
```

These ten lines of code were written over the course of 6 minutes 43 seconds. They are reasonably well formatted, and contain only two minor (obvious) errors—a missing semicolon on lines 68 and 73. The compiler catches each of these, and Harry corrects them without difficulty. More problematic is line 71, which is the test in the inner `while` loop. When Harry first wrote the code, it was `notes.isEmpty = false`, which he changes to `notes.isEmpty() = false`, and then settles on `i <= notes.size()` as being appropriate. He makes his way through these three changes in two minutes, spending a little over a minute on the final edit.

This final form yields the error `variable i might not have been initialized`, so he tweaks line 68 to read `int i = 0;`. The method `listAllNotes()` now looks like the code below:

```
066   //listAllNotes - prints all stored notes
067   public void listAllNotes() {
068     int i = 0;
069     System.out.println ("List of all notes:");
070     while (notes !=null) {
071       while (i <= notes.size()) {
072         System.out.println("Note " + i + ":" + notes.get(i));
073         i++;
074       }
075     }
076   }
```

Harry only makes two more changes at this point; he changes the test on line 71 from $<=$ to $<$, and inserts a missing space in the output of his `System.out.println` statement (a common mistake made by everyone when doing string concatenation). After these two minor changes, he does not revisit this method again for the remainder of the session.

Harry: stepping back

We would like to say that this is an example of an excellent session, but Harry's performance leaves us unsatisfied. First, both at this point and earlier in his program, he makes use of `notes.size()`, which is correct. However, the `Notebook` class contains a method called `numberOfNotes()` that returns the length of the `notes` `ArrayList`; granted, it just returns `notes.size()`, but this may change later if we decide to use an array instead of an `ArrayList`. By using the method `numberOfNotes()`, Harry would make his code more resilient to future changes. So, he scores points on functionality, and looses points for failure to choose to reuse his code where and when it is appropriate.

Secondly, Harry has introduced a very awkward, nested `while` loop. In fact, it introduces an infinite loop into the program. The inner `while` loop correctly steps through the `notes` `ArrayList`, visiting each index from 0 to `size() - 1`. Then, we go to the top of the outer `while` loop, and check the condition `notes != null`, which is still true. We then fail to enter the inner `while` loop (because, at this point, `i` is equal to the length of the `ArrayList`), and instead jump back to the outer `while` loop.

Harry rapidly wrote syntax-error free code—when compared to his classmates Neville and Ron, we want to describe this session as "successful." However, the presence of this infinite loop in his program makes us wish we had a window into what he was doing after successfully compiling his program. Certainly, there was enough time between some of the compilations in the session for Harry to execute his program—and if he did, he somehow managed not to notice that his program was looping infinitely. We executed this code on our own machine, and find that such an oversight would be difficult to make, leading us to believe that it is possible that Harry never actually tested his program.

In any case, we have strayed from the question of *compilation behaviour* to *testing*, which is a run-time concern. Harry wrote his code cleanly, it is well-formatted, and he dealt quickly and efficiently with syntax errors when they arose. He could have reused code better, and perhaps should have wondered about the odd, nested `while` that he wrote. But, overall, we believe his behaviour, when interacting with the compiler, was much more effective than that of his classmates Neville or Ron.

3.6.4 Hermione: success, with elegance

We have seen the compilation behaviour of three students so far; in each case, we have looked at the kinds of errors they faced and how they responded to those errors. In this 15 minute session of Hermione's, only four compilations of twelve captured ended in a syntax error (the session was 75% error-free). None of those errors persisted for any length of time—they were always dealt with quickly and accurately. We can see as much in the session trace in Figure 3.22, or Table A.4 in the appendix.

There are a few things that are striking in the visual representation of the trace. First, the relative lack of colour tells us that there were very few errors in the session; the Err Type column is dominated by stars ($\star$), indicating compilations that were syntax-error free. There are also many edits where Hermione touched multiple lines in the file at a time. Like all of the other sessions, Hermione was adding code to the end of the file, as she was adding a new method to

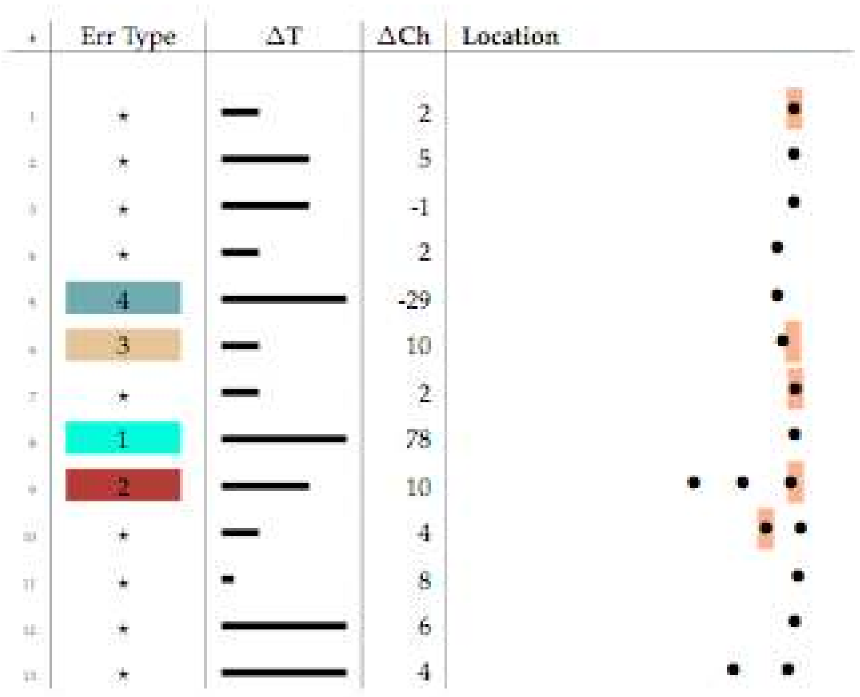


Figure 3.22: A brief, but effective, session from Hermione

the end of her program.

Hermione's first edit fixes a syntax error that was present in her code at the start of the session. Why the code was left dangling in this state previously we do not know. She quickly eliminates the syntax error. We see this change below.

Before

```
for (int i=0; i
  System.out.println(notes.get(i));
}
```

After

```
for (int i = 0; i <= notes.size(); i++) {
  System.out.println(notes.get(i));
}
```

The next edit takes Hermione 1 minute 27 seconds, and involves the removal of just one character; on line 66, she replaces the <= with an =. This is not a syntactic edit, as the program can be compiled successfully with either operator in-place. Instead, this is one of the first examples we have seen of a *semantic* edit. The motivation for this change only has meaning within the context of the executing program. Coupled with the fact that 1 minute 27 seconds pass between compilations at this point, it is entirely possible that Hermione compiled her program, tested it, and discovered that she needed to change the termination case of her `for` loop.

Before

```
066    for (int i = 0; i <= notes.size(); i++) {
067      System.out.println(notes.get(i));
068    }
```

After

```
066    for (int i = 0; i < notes.size(); i++) {
067      System.out.println(notes.get(i));
068    }
```

The craft of programming

The next change is subtle, and is a kind of edit that we haven't seen in our examination of Neville, Ron, or Harry. Whereas many of the sessions we have seen so far have involved students wrestling, to some degree, with errors reported by the compiler (sometimes caused by the poor layout of their own code), here we see a purely *cosmetic* change; Hermione adds two spaces to her program on two different lines.

Before

```
047    if(noteNumber < 0) {
048      ...
050    else if(noteNumber < numberOfNotes()) {
```

After

```
047    if (noteNumber < 0) {
048      ...
050    else if (noteNumber < numberOfNotes()) {
```

On lines 47 and 50, Hermione adds a single space after the `if` and `else if`. Neither of these spaces effects the execution of the program. However, this kind of attention to detail—this *craft* in programming—is part of what is ultimately required for the authoring of truly excellent, robust code. While we do not believe that adding spaces to a program makes the difference between code that executes and performs correctly and code that does not, we do believe that this is the kind of attention to detail that is ultimately needed for writing good unit tests that catch all the expected cases, all of the edge cases, and some of the less expected cases as well. Or, for keeping up on documentation, or anything else about programming that requires attention to detail. Is programming a craft or a science? We don't know. But we do know that Hermione demonstrates far more craft in this one session than any of her peers that we've examined so far.

Testing and user interface interaction?

The next set of changes that Hermione introduces is much larger. She makes changes in the `showNote`, `removeNote`, and `listAllNotes` methods at the same time. This shows up in Table A.4 as a single edit with a relatively large removal of code (Figure 3.23).

This is an interesting cut and edit; Hermione removes a significant amount of code from her program, and generally simplifies the logic of the `showNote()` method in the process. In addition, she has the method output a visible warning when the `else` clause is reached, instead of quietly doing nothing.

At the same time as she made the above changes to `showNote()`, Hermione also added error-checking code to `removeNote()`. Whether she copy-pasted some of the code from `showNote()` into `removeNote()` is difficult to say. This, the second of three changes made in this single compile follows.

Before

```
public void showNote(int noteNumber)
{
  if (noteNumber < 0) {
    // This is not a valid note number, so do nothing.
  }
  else if (noteNumber < numberOfNotes()) {
    // This is a valid note number, so we can print it.
    System.out.println(notes.get(noteNumber));
  }
  else {
    // This is not a valid note number, so do nothing.
  }
}
```

After

```
public void showNote(int noteNumber)
{
  if (noteNumber >= 0 && noteNumber < numberOfNotes()) {




    System.out.println(notes.get(noteNumber));
  }
  else {
    System.out.println("Invalid note index");
  }
}
```

Figure 3.23: A single, large change by Hermione.

Before

```
059   public void removeNote(int noteNumber)
060   {
061     notes.remove(noteNumber);
062   }
063
064
065
066
```

After

```
059   public void removeNote(int noteNumber)
060   {
061     if (noteNumber >= 0 && noteNumber < numberOfNotes()) {
062       notes.remove(noteNumber);
063     }
064     else {
065       System.out.println("Invalid note index");
066   }
```

While the previous two parts of this single edit were interesting, we want to spend a moment on the last change Hermione made in this single, large edit-compile sequence:

Before

```
066     for (int i = 0; i < notes.size(); i++) {
067       System.out.println(notes.get(i));
068     }
```

After

```
070     for (int i = 0; i < numberOfNotes(); i++) {
071       System.out.println(notes.get(i));
072     }
```

On line 66 in the `listAllNotes()` method, Hermione has made one critical change. She has changed the conditions of the `for` loop from

```
for (int i = 0; i < notes.size(); i++)
```

to

```
for (int i = 0; i < numberOfNotes(); i++)
```

While both pieces of code are correct (they are functionally identical), Hermione has made good use of the method `numberOfNotes()` which is defined earlier in the class. In addition to all of the edits she was making, Hermione is reusing a method defined previously, as opposed to constantly invoking the `size()` method on the `notes` object. In this particular instance the reuse probably doesn't matter one whit—but it this kind of awareness and reuse is a habit that will serve her well in the future. Additionally, this awareness can be contrasted with the code we saw Harry write, where he simply used `notes.size()` over-and-over in his code, instead of making calls back to `numberOfNotes()`; it is this kind of awareness that separates the editing and compilation behaviour of Harry and Hermione.

After compiling her code, Hermione discovers an `illegal start of expression` error. In examining many of these traces, an illegal start of expression error typically persists over multiple compilations. In the simplest instance, it often indicates a failure on the part of the student to properly match braces. Hermione catches the error quickly the first time (not shown).

Before

```
068   public void listAllNotes()
069   {
070     for (int i = 0; i < numberOfNotes(); i++) {
071       System.out.println(notes.get(i));
072     }
073   }
```

After

```
065   public void listAllNotes()
066   {
067     for (int i = 0; i < numberOfNotes(); i++) {
068       System.out.println(i + ": " notes.get(i));
069     }
070   }
```

While Hermione tracks down and corrects the bracketing problem she introduced in the `removeNote` method, she also makes cosmetic changes in the `listAllNotes` method as well. As we have seen, students faced with a syntax error typically do little more than tackle the error at hand. Here, Hermione not only dealt with a bracketing problem in her code, but she also went on to make cosmetic changes in the output of her program. In the compilation after this one, she adds the missing + in line 68, which is a common oversight made by many students doing string concatenation in Java, but surprisingly problematic to find in many cases, because the error reported has so little relevance; in this case, the error reported was `')' expected`.

Hermione: stepping back

Hermione's entire session was only 15 minutes long. However, she was very productive in that time, and the errors she encountered were easily dealt with. What is also striking is the amount of time she took between compilations: 20 seconds, 1 minute 18 seconds, 1 minute 27 seconds, 23 seconds, 4 minutes 17 seconds, 27 seconds, 13 seconds, 2 minutes 11 seconds, 1 minutes 13 seconds, 28 seconds, 9 seconds, and 3 minutes.

Whereas half of all compilations recorded by students during the Fall 2003 semester were short—lasting 30 seconds or less—Hermione spends much longer than the average on her edits. The fastest edits were in response to syntax errors, which she handled correctly in each instance without any persistence of the error over time. From the semantics of the changes she was making, we assume that some of this time spent between successful compiles was spent interacting with her program, testing the handling of input and output. Overall, we would have to say that the behaviour that Hermione exhibits in this session is the kind of behaviour we might want all of our novice programmers to exhibit.

Quantifying behaviour

One way to explore the compilation behaviour of novices is to dig deeply into their source code. While observing the changes they make from one compilation to the next does not tell us everything about their thought processes, we can see and infer a great deal about the program they are writing. The observable quantities in this process—the changes they make, the errors that may result from that process, and so on—can be used to summarize a single session, allowing us to develop quickly a sense for how a student spent their time while developing their code.

While our tabular representation does make it easy to see where syntax errors occur in a file and how often they repeat, it is still a difficult representation to work with when hundreds of sessions are involved. One problem in comparing these sessions is that they are all different lengths; one session might be comprised of 20 compilations, another 40. Each of these sessions may have several compilations that resulted in error free code; which one had more, proportionally? Was one session "worse" than the other? Although our tabular summaries of student sessions are a useful overview for a single session, they do not necessarily help us compare one session to another.

In the next chapter, we present a quantisation of our tabular representation. This quantisation is based on many of the criteria we have already discussed: the number of syntax errors made by a student, how often those syntax errors are repeated, and the source location of the error from one compilation to the next. We reduce an entire programming session to one number, which we call the **error quotient**. The error quotient, deterministically generated from observable quantities of a students programming behaviour, gives us a normalised value (zero to one) that allows us quickly and easily to compare sessions: sessions with low error quotients had relatively few syntax errors; sessions with high error quotients had many syntax errors that repeated in the same place over many successive compilations. The details of how we generate this number, and its utility in expanding our analysis of novice compilation behaviour follows.

Chapter 4

The Error Quotient

The error quotient is a measure of how many syntax errors a student encountered during a single programming session. Students who encounter many syntax errors, and more importantly, fail to fix them from one compilation to the next, end up with high error quotients. Conversely, students who have few syntax errors, or who correct their syntax errors quickly (meaning, the error is reported and then repaired) end up with low error quotients.

4.1 Development of the error quotient

The error quotient was not an *a priori* notion we had regarding novice programmers. We knew from experience that students tended to recompile their code quickly and often, we suspected there might be interesting data to be mined from an observation of their behaviour. A great deal of time and effort was expended exploring arbitrary views and slices of this data.

For example, we went looking for relationships between the number of compilation events students generated in each session, as well as the time that elapsed between the compilation events in those sessions. This yielded the fan-like plots in Figure 4.1. We had no idea how to make sense of these figures; even on a log-log scale, there did not seem to be any clear correlation between the number of sessions we had recorded for a student and the average number of compilations we recorded for that student across all of their sessions. (We thought perhaps that students for whom we had captured lots of sessions would necessarily have generated a large number of compilation events.) Confused by this data, we decided to look at total time spent between compilations vs. the total number of compilations captured (Figure 4.2). While we believe there was a good correlation here, we realised that our ad-hoc explorations had led to near truisms—it stands to reason that students who spend more time working on their code will, on average, compile it more often than students who only work in the labs infrequently.

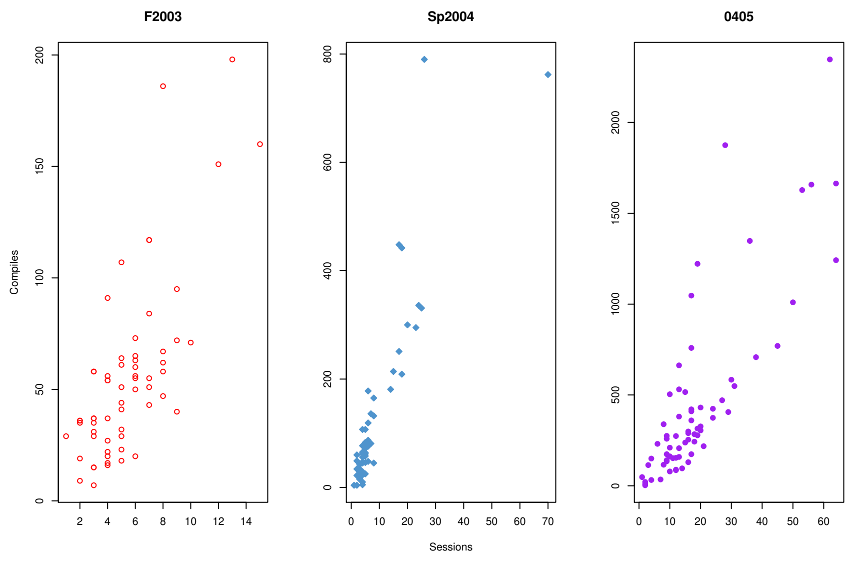


Figure 4.1: Compilations per session, per population.

We came to realise that our blind, statistical explorations were nothing more than grasping at straws. Every attempted correlation required us to hypothesise ahead of our data. For example, we had no reason to suspect that a the number of compilation events in a session might correspond with the total number of sessions recorded—it was nothing more than a guess, or a hunch. By visualising arbitrary combinations of data (as exemplified in Figure 4.1), we were simply hoping that our implicit assumptions will be borne out in practice. These kinds of statistical explorations are a poor substitute for a focused and prolonged examination of the data in a manner true to the story it might tell. In our case, we wanted a better understanding of what novices were doing while repeatedly recompiling their programs—which implies that our time would best be spent reading programs written by students.

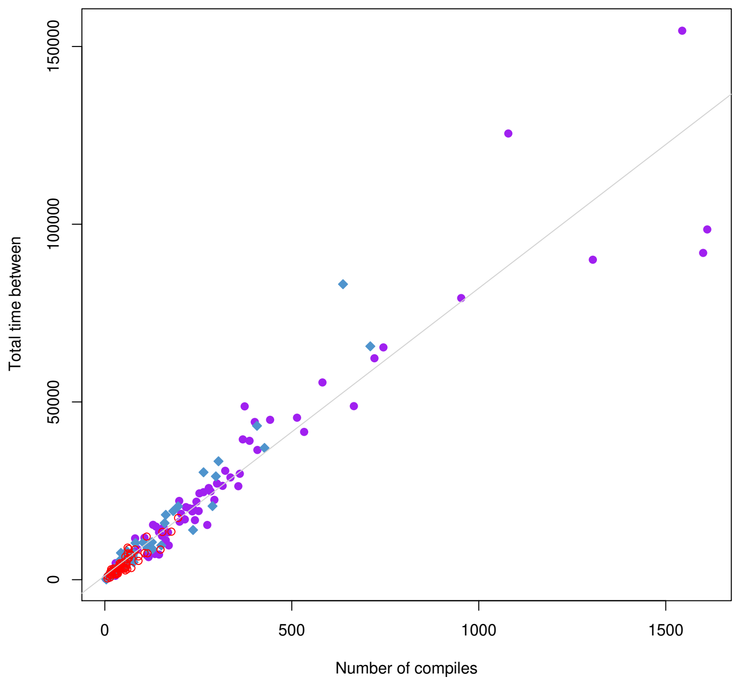


Figure 4.2: Total time between compilations vs. total compilations

4.1.1 Diving into the code

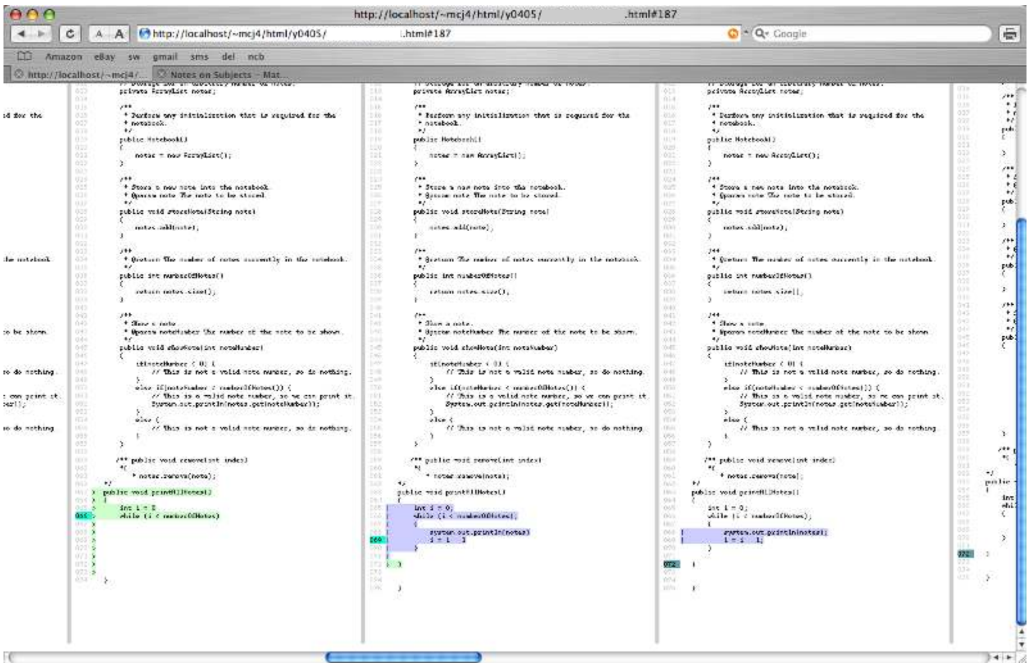
To better understand what the students participating in our study were doing from one compilation to the next, we engaged in an extensive document analysis. Starting with a random selection of sessions from our study (there are over 2000 sessions in our database), we began to read programs. Our initial efforts were spent characterising the kinds of errors students were making, adding notes and commentary to the session one compilation at a time. We turned each session into a narrative, and then discussed these narratives with colleagues. Our goal in this was to tap into the knowledge of educators who had spent years, and sometimes decades, teaching introductory programming.

These discussions were fruitful; from them, we were reminded of behaviours we had observed in our own classrooms, were told about behaviours we had not seen, and developed a deeper appreciation for the kinds of errors students made in their programs. From this process, our next step was to revisit the data with a new focus.

We started with the 2004-2005 academic year and proceeded alphabetically by pseudonym, and began reading the programs students had written. Our goal was to begin characterising the *types* of errors (or sequences of errors) students had made while programming. From our discussions with other educators, we developed an appreciation for the variety of strategies students employ while learning to program, some correct, some not so correct. We approached the data looking specifically for these strategies; our hope was to develop a catalogue of strategies for resolving syntactic errors observed in the sequences of edits students were making to their programs.

This analysis was expensive; it took a great deal of time to read through a session, finding the location of syntax errors, understanding the changes that were made, etc. Very quickly, we realised we needed to develop tools for reading the hundreds upon thousands of subsequent edits that were captured in our database. By rendering sessions to HTML, we could use commonly available tools (like a web browser) to quickly browse through and navigate through our entire database of student programs (Figure 4.3).

Even with improved tools, establishing a catalogue of syntax error recovery strategies was a daunting task. While it may be possible, we found the diversity of strategies to be great, and


http://localhost/~mcj4/html/y0405/ .html#187
http://localhost/~mcj4/html/y0405/ .html#187
Google
Amazon eBay sw gmail sms del ncb
http://localhost/~mcj4/
Notes on Subjects – Mat

```
    private ArrayList notes;

    /**
     * Perform any initialization that is required for the
     * notebook.
     */
    public Notebook()
    {
        notes = new ArrayList();
    }

    /**
     * Store a new note into the notebook.
     * @param note The note to be stored.
     */
    public void storeNote(String note)
    {
        notes.add(note);
    }

    /**
     * @return The number of notes currently in the notebook.
     */
    public int numberOfNotes()
    {
        return notes.size();
    }

    /**
     * Show a note.
     * @param noteNumber The number of the note to be shown.
     */
    public void showNote(int noteNumber)
    {
        if(noteNumber < 0) {
            // This is not a valid note number, so do nothing.
        }
        else if(noteNumber < numberOfNotes()) {
            // This is a valid note number, so we can print it.
            System.out.println(notes.get(noteNumber));
        }
        else {
            // This is not a valid note number, so do nothing.
        }
    }

   /** public void remove(int index)
    *{
       * notes.remove(note);
   */
  public void printAllNotes()
  {
      int i = 0
      while (i < numberOfNotes)
  }
```

Figure 4.3: A browser for viewing successive differences in student programs

consistent patterns were not percolating up through our analysis. For this reason, we went back to the drawing board, exploring our data both in the large and in the small, and developing tools to support those explorations.

4.1.2 Automating a document analysis

The goal of our analysis was to better understand the behaviour of our students while they were learning to program. Unfortunately, our natural inclination was to question our students' *intentions* when writing programs, as opposed to focusing on what, exactly, they had done. As before, we found ourselves theorising ahead of our data, mining it for meaning that we could not possibly verify.

In the previous chapter, we presented a tabular representation of novice programming sessions that provided us with critical insights into how successful or unsuccessful students were in tackling syntax errors while learning to write programs in Java. These tables were the outcome of a detailed document analysis; in reading hundreds of programming sessions, we had developed a sense for the kinds of events that might signal a "strategy" of sorts. For example, "remove the error" was a strategy we had seen time and again, and we highlighted this while discussing code written by Neville. In particular, we realised that we spent the majority of our time looking at the type of syntax error that had been made, and how often it had repeated—this was often our best indicator for how well (or poorly) a student was progressing.

The error quotient is an outgrowth of this automated analysis; once we developed a tabular representation for a student's session, it became obvious that we could quantify the session, as all of the elements of the table were observable, discrete quantities. Employing a strategy much like that employed by some email SPAM filters, we began assigning "penalties" to behaviours that did not move a student towards the goal of having error-free code. What follows is a detailed discussion of how the error quotient is formulated, and how this strictly behavioural quantity relates to student performance in the classroom on traditional homework- and exam-based assessments.

4.2 Line, colour, and repetition

In our tabular representation of programming sessions, we highlighted visual cues that assist us in identifying problematic compilation behaviour. If a student spends the majority of a session stuck in one place in their code, then we see the location of syntax errors and places they edited their code line up in the Error Location column of our visualisation. When students cannot get past a particular syntax error, we see large blocks of the same colour appear in the Error Type column. In both cases, it is the repetition of location or error type that makes the problem more obvious to the observer.

We saw the interplay of these three criteria most obviously in Neville's session from the previous chapter (Table A.1). However, we can browse through our data looking for sessions where errors are repeated in the same part of the program consistently, edits follow those errors, and syntax errors repeat themselves over and over. For example, Terry's 19th session looks remarkably similar to Neville's first session (Table A.5). Likewise, Deborah has more than one session where she gets stuck in one place (sessions 1 and 25, Tables A.6 and A.7).

Examination of the source code in these sessions shows us editing behaviour similar to that which took place in Neville's code—the commenting and uncommenting of "problematic" code, guessing at method names, and so on. To describe these sessions as "similar" required a careful reading of a large amount of source code, however, and judgements are made from an "expert" point-of-view. While our tabular representations of student sessions are convenient for quickly summarising the results of many successive compilation events, we have only established an informal set of working hypotheses regarding what is "good" and "bad" compilation behaviour.

For example, our discussion has often focused on the repetition of errors, both the error type and the location where the syntax error is reported. If one student fails to correct an "illegal start of expression" error over the course of three compilations, and another over 10, is one student three times worse than the other? What if the other student deals with a non-stop string of errors, and the repetition of this particular error is just one of many?

A simple algorithm is presented here, based on our qualitative readings of novice programming sessions, and the development of a visualisation for those sessions. We have attempted to capture the most salient properties of what we consider "bad" compilation behaviour; in particular, we have paid careful attention to how often errors repeat, and whether those errors repeat in the same region of the student's program code.

4.3 Scoring compilation behaviour

Our algorithm for calculating the error quotient of a session is as follows. 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 4.4.

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

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

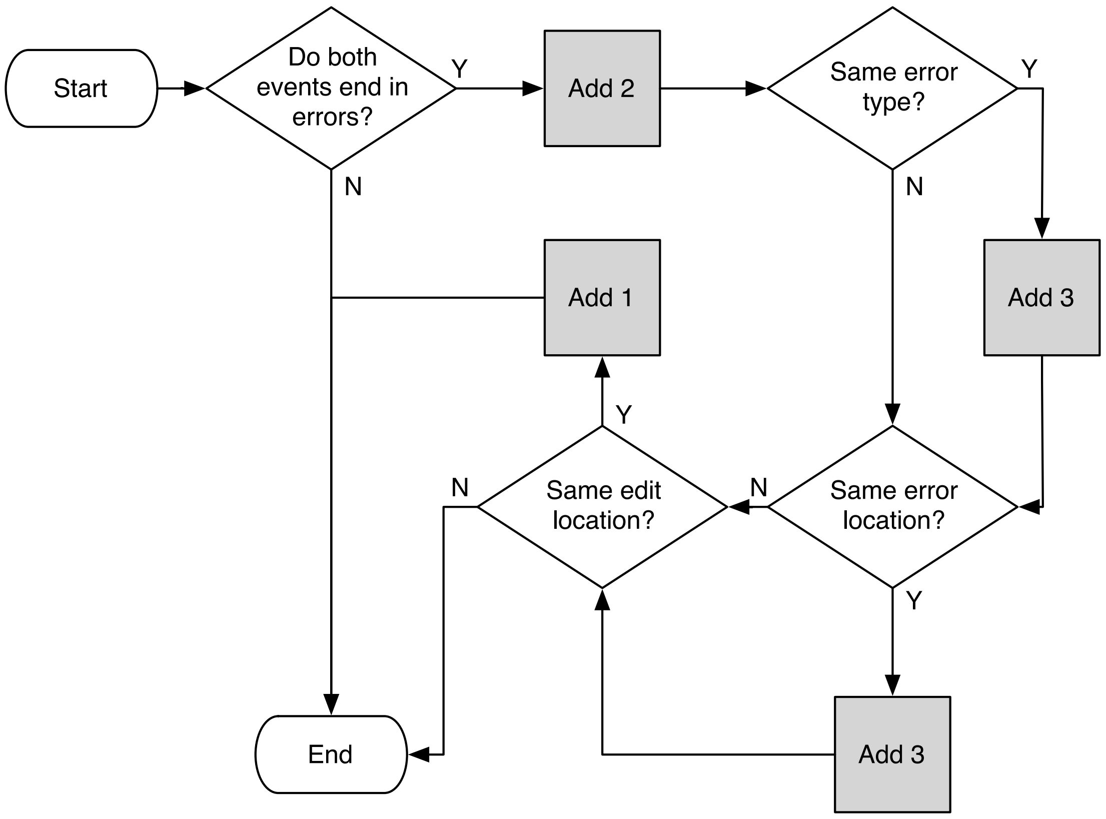


Figure 4.4: To calculate the error quotient of a session, each pair of events is first scored using this algorithm. Those values are then summed and normalized, assuming a maximum score of 8 per pair.

The most important step in this process is the assigning of a score to each pair of events in the session. While the flowchart in Figure 4.4 is reasonably straight-forward, we will demonstrate it's application to four events in a fictitious session trace (Table 4.1).

Our scoring begins by taking each pair of events in turn[1]; in this case, those pairs are events (1,2), (2,3), and (3,4). Taking compilations 1 and 2, we work our way through the flowchart. First, we ask if both events end in a syntax error? Given that the second event has an error type of $\star$ (meaning it was an error-free compilation), we would answer **no** to this question, and assign the pair of events a score of 0.

[1]Compilation pairs with a STRING-EDIT distance of zero are filtered out before any calculation is carried out.

Table 4.1: A fictitious, four-event session

#	Err Type	ΔT	ΔCh	Location	SCORE
1	4	▪	2	●	
2	⋆	▬▬▬▬	13	●	0
3	1	▬▬▬	-27	●	0
4	1	▬▬	-2	●	8

Repeating this process for the pair (2,3), the second question is answered the same way; because the pair of events begins with an error-free compilation, we score the pair as a 0. This is because we believe it is OK for syntax errors to be made by students learning to program—everyone makes mistakes. However, we are trying to penalize the behaviour, where a single error persists over multiple compilations. The inability to deal with syntax errors quickly and concisely is what we intend our algorithm to detect.

In scoring the pair (3,4), we would assign a score of 8. First, they did edit their code, so we would continue scoring the pair. Second, both events were errors, and were the same kind of syntax error (in this case, `';' expected`), so we add 2 and then 3 to the running total for this pair of events. Then, we observe that the error location is the same (we allow a $^{+}/_{-}1$ difference in line number), so we add 3 to the total, yielding 8.

In running our error quotient algorithm over this sample session, we assigned scores of 0,0, and 8 to the pairs (1,2), (2,3), and (3,4). We would then normalize these three values to 0, 0, and .88; the average penalty assigned to the pairs is .88/3, or .29. This average over all the pairs becomes the session's error quotient.

4.3.1 The error quotient and parameter choice

Our choice of parameters in the error quotient calculation is not arbitrary; based on our document analysis, we found that students who were unable to eliminate a given syntax error (thus compiling the same `';' expected` error over-and-over) were struggling more with syntax than students whose explorations at least generated new syntax errors. For this reason, we assign a higher penalty (5) to students who repeat the same syntax error type over multiple compilations than those who explore a larger error space (2).

However, there is a definite parameter space to be explored. Using the parameters we proposed above, we analysed the distribution of the EQ values generated by this method, the relationship between students' EQ to their grades on assignments and exams, and so on; the results were encouraging, as there seemed to be a correlation between the values. This made us curious: were there parameters that would provide us with greater spread between students in our study? That is, were there parameters that provided a better differentiation between those students who tended to deal with very few syntax errors and those who dealt excessively with errors?

Exploring the parameter space

We began by parameterising our algorithm. This involved the introduction of five variables:

- `touched-multiplier`,
- `eline-range`,
- `eline-penalty`,
- `etype-same-penalty`, and
- `etype-diff-penalty`.

The `touched-multiplier` was a multiplicative factor we applied to our scoring when students touched the same line of code from one compilation to the next. `eline-range` allowed us to control what constituted an error on the "same line"; a 0 would literally mean that two subsequent errors would need to be on exactly the same line, while the range [-3,3] would indicate that any error within three lines (in either direction) would constitute being "on the same line." `eline-penalty` is applied when an error occurs on the same line, `etype-same-penalty` when the error type reported is the same as the previous compilation result, and `etype-diff-penalty` when the error type is different.

In an attempt to find "better" parameters, we chose a simple metric: our choice of parameters should, as much as possible, differentiate between students across the population. For

```
(define (generate-all-combos)
  (let ([all-param-combos '()])
    (foreach ([touched '(1.0 1.2 1.4 1.6 1.8 2.0)])
      (foreach ([range '((-1 0 1) (-2 -1 0 1 2)
                         (-3 -2 -1 0 1 2 3))])
        (foreach ([eline '(1 3 5 7 9)])
          (foreach ([etype-same '(3 5 7 9 11)])
            (foreach ([etype-diff '(2 4 6 8 10)])
              (gather all-param-combos
                (cons (mysym 'combo)
                      (list touched range eline
                            etype-same etype-diff)))
              )))))
    all-param-combos))
```

Figure 4.5: Our code for generating parameter combinations.

a given set of parameters, we would calculate the EQ for every session of every student who took part in our study during the 2004-2005 academic year. We stored these results, and then chose the parameter set that had the greatest range, for each student, between the minimum and maximum EQ values recorded, while also minimizing the standard deviation of EQ values calculated for that student.

The space of parameters we searched was not global; we are working from the assumption that our findings from our document analysis of hundreds of programs has provided us with a good initial set of parameters. Therefore, we searched a (coarse) space around our initial set of parameters. These combinations were generated by the code in Figure 4.5. A space of 6174 possible parameter sets resulted.

Table 4.2 summarises the parameters we originally chose, as well as those that we found through a semi-exhaustive search. There are two striking differences: first, our choice to penalize a student for having a syntax error that repeats on the same line does not seem to matter a great deal; second, the ratio between having the same error type (5) versus different error types (2) went from 5/2, or 2.5, to 11/8, or 1.375. Figure 4.6 shows how striking the difference between these two parameter sets is; our original set of parameters gave us a view of the population that was shifted to the left (tending towards low EQ values), and we had a very hard time differentiating between individuals in the middle of the distribution. With the parameters found through search, the population is spread out further (a greater range of EQ values), and we don't have any "hot spots" in the distribution; however, we do have some significant outliers at the two extremes of the distribution.

Table 4.2: Parameter choice for EQ calculation

Parameter	Original	From Search
touched-multiplier	1.0	1.0
eline-range	[-1,1]	N/A
eline-penalty	3	0
etype-same-penalty	5	11
etype-diff-penalty	2	8

Filtering the data

Our data is opportunistic, in a way; we have no control over whether students come into the public laboratories on campus to do their programming or not. Therefore, when doing statistics over the entire population, we do have to wonder: just how much data do we have on each individual? Is it enough to claim we have a "representative sample" of each individual's programming behaviour?

We could control for this in a variety of ways; currently, we require that a "session" have at least seven compilation events, during each of which the student made some kind of change to their code. We might further require that we have a minimum number of sessions for a given student before we're willing to consider them with the rest of the population. If we limit our data this way, we find that we quickly eliminate a significant number of students from our study.

We have 161 students in our study; of these, 84% recorded two or more sessions during either the Fall 2003, Spring 2004, or 2004-2005 academic year. If we limit our data to students for whom we have 3 or more sessions, we lose 40% of our sample; only 96 students, over the two years, made use of the public labs to do their programming three or more times. The impact of this filtering is significant, however.

Figure 4.7 shows four distributions of error quotient scores; the upper-left is a population of 135 students, or those students for whom we have two or more sessions. The upper-right distribution is a population of 96 students, for whom we have three or more sessions. The

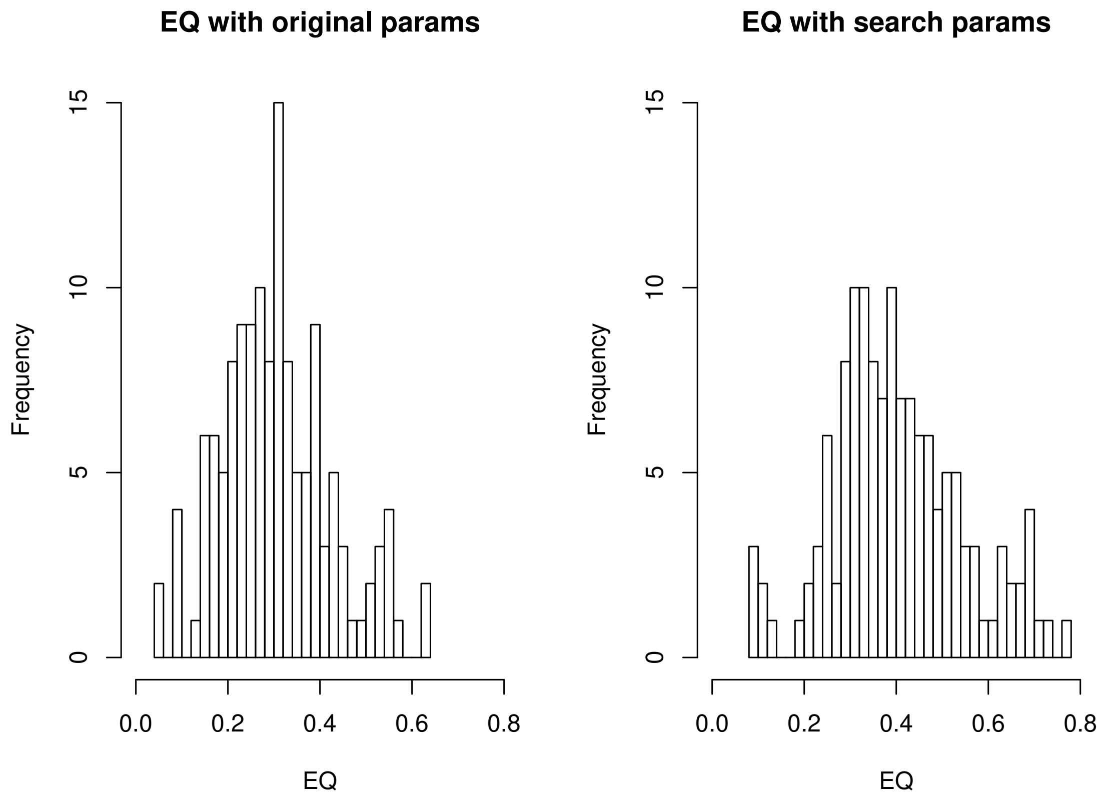


Figure 4.6: EQ distribution with our original parameters (left) and those found through semi-exhaustive search (right).

lower-left is the distribution of error quotients for students who only have one or two sessions in our database, and the lower-right is the distribution of standard deviations for those error quotient scores. What we can see is that students whom we have eliminated for only having one or two sessions do tend towards the extremes of our error quotient distribution; however, the distribution of standard deviations tells us that these values are essentially random—there is not enough data to consider them as being reliable.

The table in figure 4.7 encapsulates our statistical rationale for filtering 40% of our population from our aggregate analyses; when we focus on only those students for whom we have three or more sessions, both our Pearson χ^2 and Anderson-Darling normality tests report, with high confidence, that we have a normal, or Gaussian, distribution of error quotient scores. When we include those students for whom we only have one or two sessions, the distribution is no longer "normal;" the tails of the distribution are dominated by students for whom we have very little behavioural data. We do not think this presents an accurate picture of our population as a whole, and hence we have eliminated them from our analyses in the following section.

4.3.2 Neville, Ron, Harry, and Hermione

In the previous chapter, we examined four sessions by four students: Neville, Ron, Harry, and Hermione. Our exploration of their sessions took us all the way down to the source code. Through the use of our tabular visualisations, and subsequent reading of their source code over many iterations, we developed a notion that each of these four students interact differently with the compiler. Our inclination is to attach a value-laden judgement on the behaviour these students exhibit—that Neville demonstrated "bad" compilation behaviour in comparison to Hermione, whose behaviour appeared to be substantially "better," or otherwise "good." Instead of relying on value judgements, we have proposed a mechanism for automatically scoring individual programming sessions; using this algorithm, we can begin to compare, directly and quantitatively, novice programming sessions. In short, we can ask "Just how did Neville, Ron, Harry, and Hermione do?"

Neville scores the lowest (worst) in this group; Ron and Harry both seem to score in the same

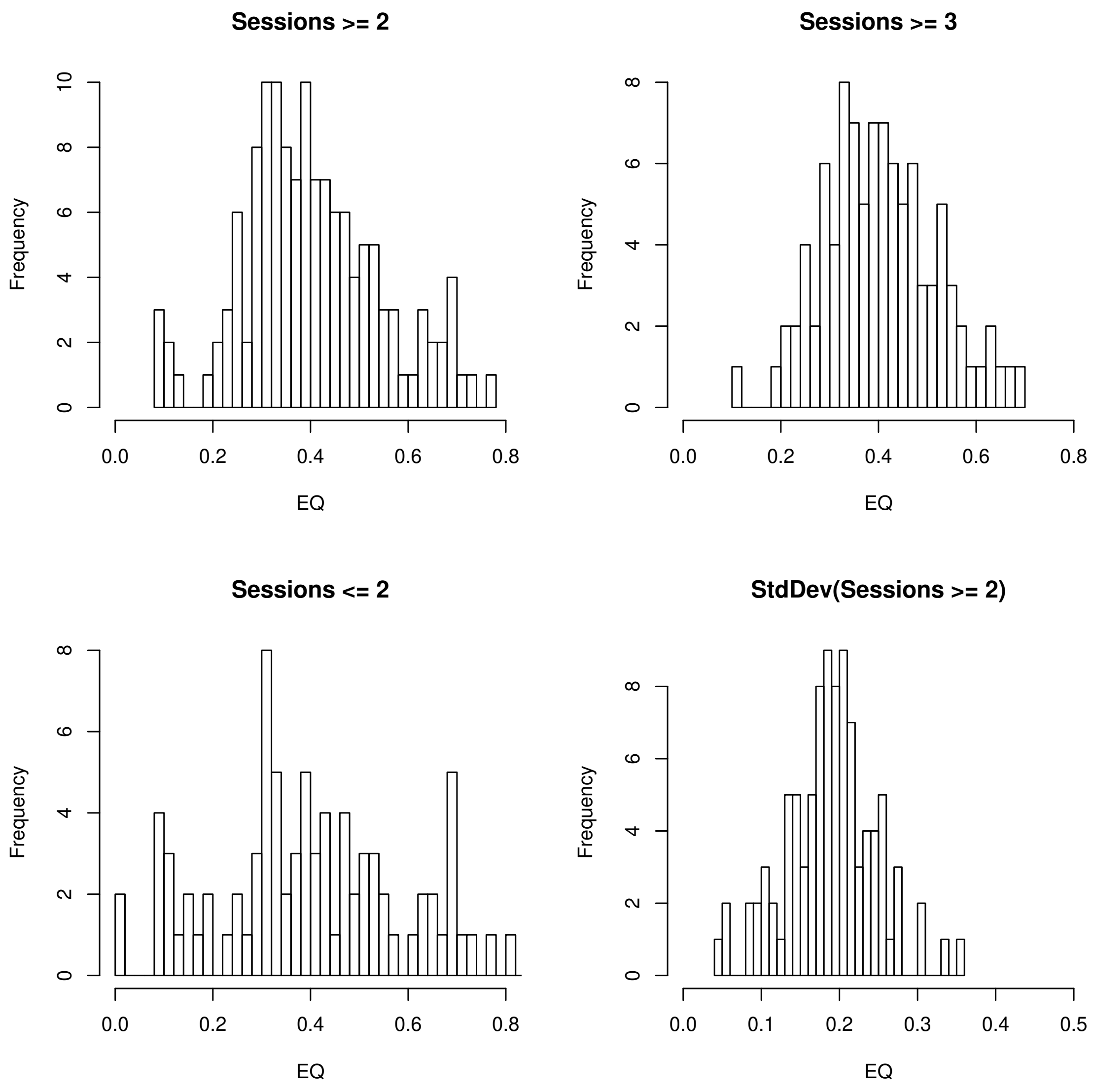


Figure 4.7: EQ distributions based on number of sessions recorded for each subject.

Distribution	# EQs	Min.	Median	Mean	Max.	p (χ^2)	p (A-D)
Sess $>=$ 2	135	0.09	0.39	0.41	0.77	0.26	0.03
Sess $>=$ 3	96	0.11	0.40	0.41	0.70	0.81	0.69
Sess $<=$ 2	65	0.00	0.40	0.41	1.00	0.33	0.34
SD(Sess) $<=$ 2	65	0.04	0.20	0.19	0.35	0.50	0.40

Normalcy of error quotient distributions based on number of student sessions

Table 4.3: Scores of the four vignettes

Student	Session	Score
Neville	1	0.54
Ron	1	0.29
Harry	1	0.22
Hermione	3	0.06

range. Hermione's session scores very well compared to her peers, as she had very few syntax errors in the entire session, few of which are repeated. While the error quotients are interesting values, they do not have a great deal of meaning unto themselves. For example, what does it mean for Neville's session to have an error quotient of .54 and Hermione's session to have an error quotient of .06? The extremes of this range (zero and one) mean one of two things. If there were no repeated syntax errors, the error quotient would be zero. If every single compilation ended with the exact same error type, those errors all occurred on the same line, and the student modified their code between each compilation—failing to correct the error—the error quotient would be one. However, these extremes are not the norm in our population, and so we might look at how Neville, Ron, Harry, and Hermione fared compared to the rest of their classmates.

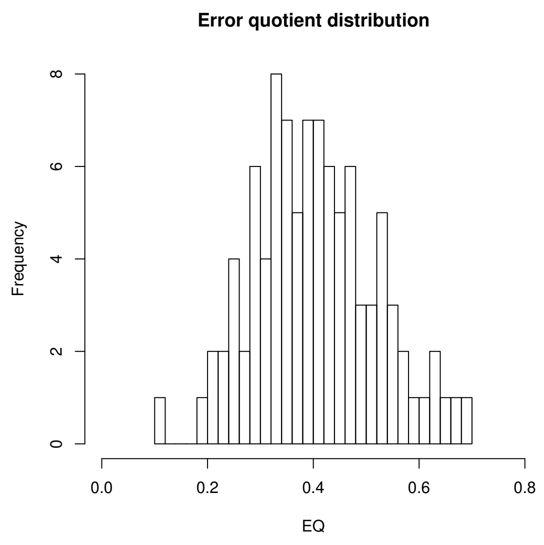
4.3.3 EQ: normally distributed

Testing for normality

The histogram and density plot in Figure 4.8 represent the average error quotient of students who took part in our study from the Fall 2003 semester through to the end of the 2004-2005 academic year. This distribution does represent filtered data; as described in section 4.3.1, we are only including students in this analysis for whom we have three or more sessions on file; this leaves us with 96 students over two years.

Is the distribution of error quotients across our population normal (in a statistical sense)? Using the statistical programming language **R** we can carry out a number of tests on our data to obtain the answer to this question[Dal02].

The histogram presented in Figure 4.8 *appears* to be normally distributed; this first piece of intuition is useful, as it helps us determine if the results coming back from our statistical



Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.11	0.33	0.40	0.41	0.48	0.70

Figure 4.8: Distribution of error quotients for students with 3 or more sessions.

analyses "make sense."

One simple test for normalcy is to perform a Quantile-Quantile (Q-Q) plot. In a Q-Q plot, we plot the n^{th} quantile in our sample against the n^{th} theoretical quantile in the normal distribution. For example, the .42 (or 42%) quantile is the point at which 42% of our data falls below that point, and 58% of our data is above it. A Q-Q plot of two perfectly normal distributions will be a line with a slope equal to one.

In **R**, we can express this test as:

```
qq.check <- function(dist) {
              z.norm <- (dist - mean(dist)) / sd(dist);
              qqnorm(z.norm);
              abline(0,1); }
```

Figure 4.9 is the plot generated by the above **R** fragment when fed the distribution of EQ data from our entire population. As can be seen, the tails of our distribution do not conform to a normal distribution, but the distribution is, by-and-large, normal according to this test.

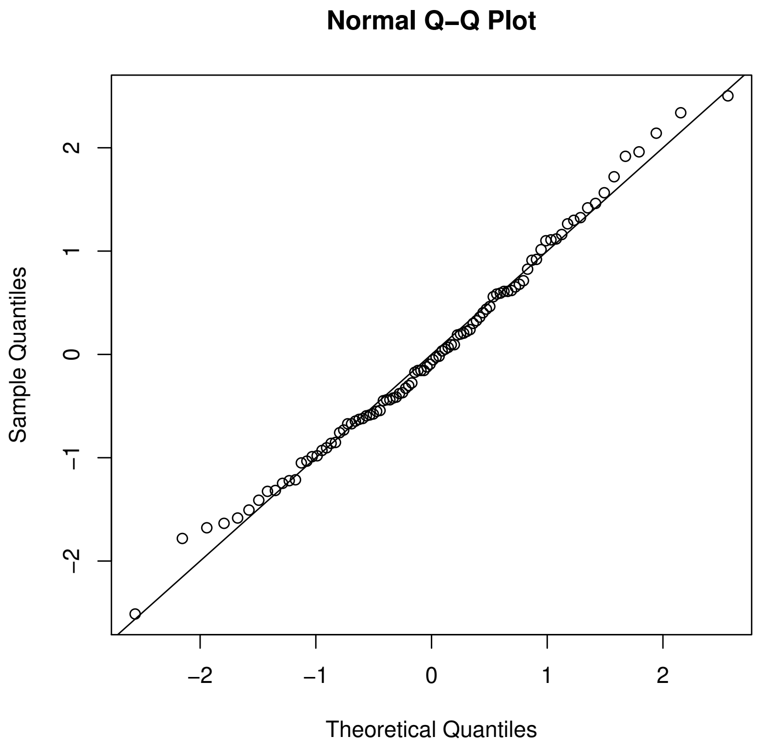


Figure 4.9: Q-Q plot of EQ values from entire study population.

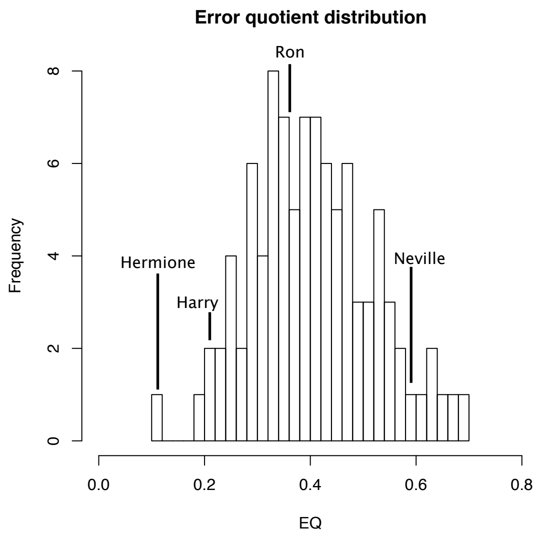


Figure 4.10: Neville, Ron, Harry, and Hermione's average EQ overlaid on the population EQ.

Between the histogram itself and the Q-Q plot, we can be fairly certain that this data is normally distributed. However, we can further apply a Pearson χ^2 goodness of fit test to the data; this parametric test for normalcy will tell us whether we should accept the null hypothesis (that our data is distributed normally) or the alternative hypothesis (that it is not). The Pearson test[2] yields $P = 6$ and a $p - value = 0.81$, which leads us to believe that our distribution of EQ values is normal. Similarly, using a common, non-parametric test, we might employ the Shapiro-Wilk test; this test is useful as it has both non-parametric and parametric components, whereas the Pearson test explicitly assumes that the underlying distribution is Gaussian. As we expect (as our intuition is that this distribution is fairly normal), the Shapiro-Wilk test reports $W = 0.99$, and a $p = 0.77$; again, we accept the null hypothesis, and believe that we have a fundamentally Gaussian distribution of EQ values in our population.

4.3.4 Neville, Ron, Harry, and Hermione revisited

In Table 4.3, we presented the error quotient for one session from Neville, Ron, Harry, and Hermione; in each case, the students were working on the Notebook project. In Figure 4.10, we

[2]`pearson.test()` as defined in the **R** library `nortest`.

have plotted the average error rates of Neville, Ron, Harry, and Hermione against the distribution of the rest of their peers. Hermione has the lowest error rate of the four, Harry is on the low side of the curve, Ron is just below the mean of the of the distribution, and Neville has an above-average error-rate for the population. We must be cautious, however; the standard deviation for each students' EQ is quite high; both Neville and Ron and Harry and Hermione have EQ averages with high enough standard deviations that they fall within one standard deviation of each-other. In calculating the average EQ for each of these four students, we can look at what the variance in their sessions is; this is reported in Table 4.4. While Neville may have an average EQ of 0.59, and Ron has an average EQ of 0.37, these individual sessions that make up these values are actually highly variable ($\sigma = 0.17$ and $\sigma = 0.15$, respectively). While the means of each of these distributions is distinct, they overlap significantly, which is a situation we must be wary of when trying to draw strong conclusions about one student being substantially different than another. Figure 4.11 shows the distribution of EQ values, per session, for both Ron and Neville, to illustrate this condition.

Table 4.4: Average error quotient for Neville, Ron, Harry, and Hermione

Student	**EQ**	σ	**# sessions**
Neville	0.59	0.17	17
Ron	0.37	0.15	15
Harry	0.22	0.11	20
Hermione	0.11	0.09	7

The point we are trying to make is that we have a relatively small amount of data for some of our students, and that data tends to be quite variable. Therefore, making distinctions like "an EQ of 0.11 is far better than an EQ of 0.18" is not, we believe, possible at this time. This does not mean that we cannot employ the error quotient in our analyses; quite the opposite, we still believe the number is potentially of great value to researchers. However, we do need to be aware of the variability in our dataset, and keep that in mind when drawing conclusions about our students and their compilation behaviour.

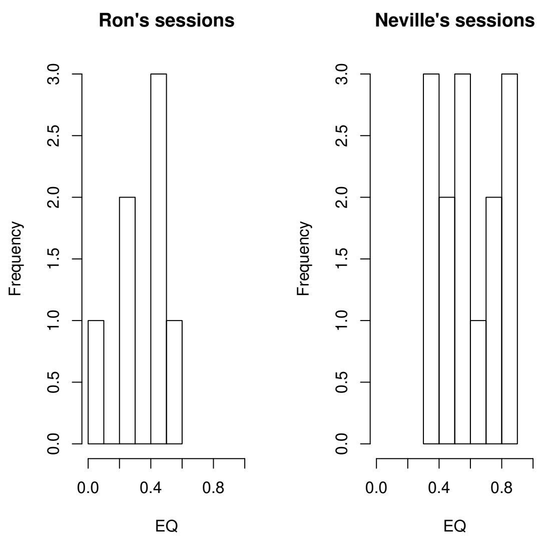


Figure 4.11: Neville, Ron's EQ session scores.

4.4 Exploring error rates

The error quotient is a simple metric that tells us how often our students compile code that has a syntax error in it, and how those errors persist over subsequent compilations. While it is clear that for small numbers of compilations, this value can be highly variable, it would appear in the first instance to (informally) correlate with our ongoing qualitative analysis of novice compilation behaviour.

If we take the current population of 96 students who generated "enough" data to be considered in this analysis, we might divide them up into three crude categories. First, those students who seem to deal effectively with syntax errors; second, students who deal very poorly with syntax errors, and then those students who fall in-between. Our document analysis would support the notion that we might use the mean and standard deviation from our EQ distribution to divide up our students. In the first quartile are both Hermione and Harry. This lowest quartile includes students who tend to deal most effectively with syntax errors. The fourth quartile includes Neville and those students who clearly struggle with syntax errors. In the middle two quartiles students like Ron whom we have a harder time categorizing; sometimes they manage errors quickly and effectively, and sometimes they don't.

In breaking up our population into three pools of students, we can now "dip our toes in" to each of those pools, and see what their behaviour looks like. Whereas our in-depth qualitative work was a fairly linear (or unguided) process, we can now use the EQ as a way of picking out what might be interesting or troubling sessions. We start with students who deal with syntactic errors most easily (the "best" students), and work our way through to students like Neville, who struggle with syntax in a very fundamental way.

4.4.1 "The best of the best"

Hermione was our exemplar for students with an EQ of less than 0.33. With an average EQ of 0.11, she demonstrated mastery of the language and the ability to not only correct syntax errors quickly, but also make aesthetic edits as well. There are twenty-four students who have average error rates less than 0.33 (Table 4.5).

In several cases, we have included a summary table that is representative of a session we captured by these students. In the Vignettes chapter, we went in-depth into Hermione's Notebook code; in that session, we saw that she had no significant difficulties dealing with the syntax of the language, and even focused some of her editing on (what appeared to be) strictly aesthetic concerns. In reading through the code for other students in this bracket, we find that they tend to exhibit many of the same characteristics—syntax, in short, does not seem to be an issue for them.

Gloria's programming demonstrates the syntactic efficiency we might expect from students with a low average EQ value. The session we highlighted (in which she is working on the "Horse Race" assignment) is particularly long, so we will only highlight one sequence of edits. The first sequence of errors that Gloria encounters comes from the addition of a new method; she begins by leaving out brackets around the body of the method—a problem that Neville attempted to deal with as well, but we might argue that he fared poorly (by comparison) when faced with the same syntax error.[3]

[3]We should also note that when authoring this method, Gloria also authored a correct JavaDoc comment—a commenting practice we rarely see in student code.

Table 4.5: The twenty-four students with the lowest average EQs.

Student	**Avg. EQ**	σ	# Sessions	**Rep. Sess.**
Hermione	0.11	0.09	7	A.11
Martin	0.19	0.17	6	
Cynthia	0.21	0.20	17	
Gloria	0.21	0.17	46	A.15
Harry	0.22	0.11	11	
Lee	0.23	0.19	8	
Julie	0.24	0.14	13	
Aaron	0.25	0.19	4	
Kevin	0.25	0.16	6	
Antonio	0.26	0.14	7	
Jacob	0.26	0.21	16	
Juan	0.26	0.17	10	
Ryan	0.28	0.18	5	
Francis	0.28	0.20	11	
Bryan	0.28	0.11	6	A.14
Martin	0.28	0.10	4	
Jesus	0.28	0.22	21	
Jonathan	0.30	0.18	3	
Johnny	0.30	0.17	12	
Marie	0.30	0.26	39	
Jimmy	0.31	0.19	14	
Aaron	0.32	0.21	15	
Danny	0.32	0.18	7	
Jeffery	0.32	0.09	3	

Before

```
087   public Horse getFirst()
088   for (int i = 0; i < numHorses - 1; i )
089   lanes[i].isFurther(lanes[i + 1]);
```

After

```
087   public Horse getFirst()
088   {
089   for (int i = 0; i < numHorses - 1; i )
090   lanes[i].isFurther(lanes[i + 1]);
091   }
```

After adding the method, the compiler reports `';' expected` on line 88; the actual error is that Gloria neglected to open a bracket for the start of the method body; she remedies this problem in fifteen seconds. In doing so, the compiler then reports that the method is missing a return statement (line 91).

Before

```
public Horse getFirst()
{
for (int i = 0; i < numHorses - 1; i )
lanes[i].isFurther(lanes[i + 1]);
}
```

After

```
public Horse getFirst()
{
  Horse h;
  for (int i = 0; i < numHorses - 1; i )
  {
    h = lanes[i].isFurther(lanes[i + 1]);
  }
  return h;
}
```

Gloria does not just add a `return` statement to her code; instead, she realised that she needed to iterate through an entire array for the method to perform the correct operation. She sets up a temporary variable (of the correct type), and returns the contents of that variable at the end of the method, after the loop. The compiler then complains that the variable `h` might not be initialized; so, she initialized the variable to the first member of the array.

Before

```
087  public Horse getFirst()
088  {
089    Horse h;
090    for (int i = 0; i < numHorses - 1; i )
```

After

```
087  public Horse getFirst()
088  {
089    Horse h = lane[1];
090    for (int i = 0; i < numHorses - 1; i )
```

Note again the correction; we assume that Gloria was attempting to initialize the variable `h` on line 89 to the first element of the array; instead, she initialized it to 1, or the *second* element of the array. This is a minor error—and, in fact, one that does not matter in the least.[4] As the subsequent `for`-loop visits every pair of `Horses` in the array, the initial value of `h` probably does not matter, as its initial value will likely never be passed back as a result of invoking this method.

While we may be picking a bit regarding the initial value of a temporary variable, we believe what is clear from both our earlier exploration of Hermione's code, as well as this short sequence of edits belonging to Gloria is that these students are writing code that is largely syntax-error free. Their edits typically deal with semantic issues in their code, not syntactic issues. While it can be seen from the standard deviation for the error quotient of each of these students that they must certainly have their "off" days, we would posit that these students are not *distracted* by the syntax they are working in, but instead dealing with the semantic content of the problem set before them.

4.4.2 "Middle of the road"

The students with the lowest EQs appear to deal quickly and efficiently with syntax errors in their programs. They focus their edits on semantic, not syntactic issues, and appear to make

[4]Except, possibly, in a one-horse race.

Table 4.6: Students with an average error rate between 0.33 and 0.48, inclusive

Student	Avg. EQ	σ	# Sess	Student	Avg. EQ	σ	# Sess
Jeremy	0.33	0.31	3	Victor	0.4	0.15	3
Kathleen	0.33	0.25	22	Jeff	0.4	0.25	44
Luis	0.33	0.21	10	Mark	0.41	0.22	3
Mike	0.33	0.12	8	Phillip	0.41	0.15	8
Craig	0.33	0.18	5	Justin	0.41	0.26	3
Chad	0.33	0.13	8	Glenn	0.41	0.25	45
Diane	0.34	0.2	5	Roy	0.42	0.22	21
Rebecca	0.34	0.14	13	Fred	0.42	0.19	4
Ann	0.35	0.18	26	Kyle	0.42	0.23	6
Curtis	0.35	0.14	3	Roger	0.43	0.05	3
Heather	0.35	0.13	5	Terry	0.43	0.35	6
Rodney	0.35	0.18	17	Melissa	0.44	0.08	11
Arthur	0.35	0.2	10	Anna	0.44	0.23	4
Philip	0.36	0.18	10	Bradley	0.44	0.18	9
Christine	0.36	0.27	8	Alice	0.45	0.17	4
Ruth	0.36	0.22	4	Melissa	0.45	0.19	5
Ron	0.37	0.15	7	Debra	0.45	0.21	11
Carolyn	0.37	0.16	6	Norman	0.47	0.19	39
Allen	0.38	0.24	28	Samuel	0.47	0.22	3
Manuel	0.38	0.2	12	Louis	0.47	0.14	3
Deborah	0.38	0.33	8	Shawn	0.47	0.24	18
Pamela	0.38	0.04	4	Billy	0.47	0.17	13
Linda	0.38	0.33	3	Walter	0.47	0.2	3
Janet	0.39	0.24	6	Jesse	0.48	0.18	5
Marvin	0.39	0.2	7	Chris	0.48	0.17	8
Phillip	0.39	0.2	5	Nancy	0.48	0.21	5
Melvin	0.4	0.27	17				

progress toward their goals—a subjective claim based on our knowledge of *Objects First with Java* and the assignments issued to students at the University of Kent.

What about the students with "average" error rates—students, for example, like Ron? These students make up the majority of our sample population. Fifty-three students have average error rates that fall between 0.33 and 0.48, inclusive; Table 4.6 lists the students in this category, as well as their average error quotient and the standard deviation associated with that average. What do sessions look like from students who are more prone to make syntax errors and for whom those errors persist?

Mysterious semantics

In the session that we've highlighted next, Linda makes error free additions to her code consistently, but makes a rather obvious semantic error that will cause problems before she is done. In adding some output to her `listAllNotes()` method, Linda introduces a variable `noteNumber`, which is not defined; this probably comes about due to a copy/paste of code she had written earlier in the class she is working on.

	Before		*After*
074	`public void listNotes()`	073	`public void listNotes()`
	. . .		. . .
078	`System.out.println(notes.get(index));`	077	`System.out.println(noteNumber + ":" + notes.get(index));`

To fix this problem, she adds the variable to the formal parameters of the `listAllNotes()` method—which changes the signature of the method. Admittedly, this fix makes the syntax error go away, but it does not make the program "more correct." She also could have introduced and initialized a variable local to the method, which would have also made the error go away. In this case, she has made her program "less correct;" by changing the method header, she will probably break other code she is expected to write for this particular assignment.

	Before		*After*
073	`public void listNotes()`	073	`public void listNotes(int noteNumber)`

Not only did Linda introduce a fix to her syntax error that potentially broke the interactions of this method with others, but it doesn't even do what she probably wants it to do. When executed, this method will now print the same `noteNumber` on every line, which is not the behaviour she was looking for. Instead, Linda probably want to print the `index` (or something based thereon) on each line of output.

While our analyses of novice programming errors have focused on the syntactic, we cannot deny that the syntactic and semantic are intimately intertwined. As students struggle to learn to program, that struggle necessarily takes place as students wrestle to map their understanding of a problem and their desired solution onto the syntax and semantics of a programming language. Despite many studies regarding programming comprehension and mental models

([MMNS83, GG84, GO86]), we do not feel that this research provides us with a model comprehensive enough to explain either Linda's behaviour or Catherine's[5].

In both Linda's and Catherine's edits (following), changes were made that fixed syntax errors, but in doing so the semantics of the programs were altered in apparently significant ways. These semantic alterations do not appear to be intentional in that the program is not made "more correct" by their edits. So while we might like to say that these students are dealing quickly and effectively with their syntax errors, we have to question whether the fix is "effective" if it significantly alters the semantics of the program in ways that do not move the student closer to a correct solution.[6]

As an example of a syntactic fix that seems to break the desired semantics of the program, Catherine wrestles for a stretch with bracketing and a return statement. From this code, it is not immediately clear how she ultimately derived the default return value for the `compareTo` method.

[5]There are only two sessions belonging to Catherine that meet our criteria for inclusion in our aggregate analysis; we include her work here as it was flagged as interesting during our document analysis, and is relevant to the discussion.

[6]Given that we are familiar with the exercises involved, we can say that these semantic changes are not likely part of any correct solution the student might devise.

Before

```
public int compareTo(Object other)
{


  ContactDetails otherDetails = (ContactDetails) other;
  if (otherDetails != null) {
  int comparison = name.compareTo(otherDetails.getName());
  if(comparison != 0){
    return comparison;
  }
  comparison = phone.compareTo(otherDetails.getPhone());
  if(comparison != 0){
    return comparison;
  }
  return address.compareTo(otherDetails.getAddress());

rn (0);

```

After

```
public int compareTo(Object other)
{

  if (other instanceof ContactDetails){
  ContactDetails otherDetails = (ContactDetails) other;
  if (otherDetails != null) {
  int comparison = name.compareTo(otherDetails.getName());
  if(comparison != 0){
    return comparison;
  }
  comparison = phone.compareTo(otherDetails.getPhone());
  if(comparison != 0){
    return comparison;
  }
  return address.compareTo(otherDetails.getAddress());

rn (0);

```

After adding in an `if` statement near the top of the method, Catherine's bracketing becomes confused, causing the `return` statement to fall outside of the brackets defining the extent of the method. She addresses this quickly, adjusting the brackets appropriately at the end of the method, despite the rather awful indentation she is employing. This is perhaps a sign of her comfort with the syntax, but not mastery—we have often seen students with high error quotients fight with bracketing, and become lost in their own poor indentation (so we assume). While we think this is less of a problem for students like Linda and Catherine, they clearly have not mastered the language both syntactically and semantically—that, or they're just very careless.

It is the last edit of Catherine's session that leaves us scratching our heads, however; she changes the `return` value of `0` to a return value of `7`. This is where the nature of our data collection and subsequent analysis breaks down—we cannot read our students' minds. We assume that Catherine had a good reason for this change, but there is nothing in our experience with this project that otherwise seems to suggest that they should return `7` instead of `0` as the "default" return value from `compareTo`. Furthermore, this is the last compilation of the session, and the last compilation that we record for this particular piece of code; as a result, we cannot explore Catherine's edits on this program further in an attempt to divine what is significant about a return value of "7".

	Before		*After*
084	`public int compareTo(Object other)`	084	`public int compareTo(Object other)`
085	`{`	085	`{`
086		086	
087	`if (other instanceof ContactDetails){`	087	`if (other instanceof ContactDetails){`
088	`ContactDetails otherDetails = (ContactDetails) other;`	088	`ContactDetails otherDetails = (ContactDetails) other;`
089	`if (otherDetails != null) {`	089	`if (otherDetails != null) {`
090	`int comparison = name.compareTo(otherDetails.getName());`	090	`int comparison = name.compareTo(otherDetails.getName());`
091	`if(comparison != 0){`	091	`if(comparison != 0){`
092	`return comparison;`	092	`return comparison;`
093	`}`	093	`}`
094	`comparison = phone.compareTo(otherDetails.getPhone());`	094	`comparison = phone.compareTo(otherDetails.getPhone());`
095	`if(comparison != 0){`	095	`if(comparison != 0){`
096	`return comparison;`	096	`return comparison;`
097	`}`	097	`}`
098	`return address.compareTo(otherDetails.getAddress());`	098	`return address.compareTo(otherDetails.getAddress());`
099		099	
100		100	`rn (7);`
101		101	
102	`rn (0);`	102	
103		103	

4.4.3 "On the high side"

The last group of students we wish to focus on—those whose average error quotient is in the fourth quartile—are those students who wrestle excessively with syntax (Table 4.7). We posit that their compilation behaviour is, generally speaking, similar to that of Neville. These students have a hard time writing syntactically correct statements in the language they are using, and struggle with the errors reported by the compiler when the syntax of their code is incorrect. Put simply, they cannot write code.

Table 4.7: Students with an average EQ greater than 0.48

Student	Avg. EQ	σ	# Sess	Rep. Sess.
Randy	0.50	0.18	7	
Peter	0.51	0.27	5	
Amy	0.51	0.19	7	
Dale	0.52	0.16	3	
Travis	0.53	0.3	7	
Sean	0.53	0.14	7	
Virginia	0.53	0.1	3	
Doris	0.54	0.13	3	
Paul	0.55	0.25	3	
Amanda	0.55	0.16	6	
Herbert	0.55	0.25	19	
Joyce	0.56	0.21	5	
Louis	0.56	0.13	4	
Neville	0.57	0.17	14	
Terry	0.60	0.08	5	A.21
Stanley	0.62	0.23	15	
Roger	0.63	0.2	3	A.22
Stephanie	0.65	0.23	10	A.26
Deborah	0.67	0.1	5	
Karen	0.69	0.05	3	A.25, A.24

While this may sound harsh, it is as concise a statement as can be made. These students wrestle greatly with syntax, and (in our observations) rarely write programs that compile. We will look in-depth at one of these students, and in passing at a second.

Roger: how not to invoke a method

Roger's session involves two files, and he focuses his efforts on one of them (Table A.22). The code he is working on in this session is part of the "Game of Zuul" project from *Objects First with Java*. In this project, the students are expected to fill out a text adventure game in the style of Zork and other Infocom games[7]. They create objects to represent rooms in their world, handle input from the user, and generally are expected to create a small game with some creative elements of their own choosing. Roger never seems to get that far.

The session begins with the `traceMoves` method commented out (on a line-by-line basis); it could be that Roger was working on this at home, and had commented out this method until

[7]http://en.wikipedia.org/wiki/Zork, http://www.infocom-if.org/

he could devote more time on it. After making one or two small edits elsewhere, he then brings his attention back to the `traceMoves` method.

The compiler proceeds to report a misuse of the method `Command: cannot resolve symbol - method Command (java.lang.String,java.lang.String)`. Over the next dozen compilation events, Roger focuses his efforts on lines 63 and 64; we will likewise focus our view on these two lines only. First, he removes his attempted object creation statement on line 63.

	Before
063	`Command com = Command.Command("go", direction);`
064	`goRoom(com);`
	After
063	`goRoom(Command("go", direction));`
064	

As the compiler is still complaining about an unknown method Command (it is a class, not a method, and should be instantiated with `new`), Roger reintroduces the use of the class `Command` (incorrectly) as an argument to the `goRoom` method.

	Before		*After*
063	`goRoom(Command("go", direction));`	063	`goRoom(player.Command("go", direction));`

Roger then deletes the word `Command` completely, but then introduces a new syntax error. The compiler, after this edit, reports a `')' expected` error. This is because a `,` is expected between `"go"` and `direction`.

	Before		*After*
063	`goRoom(player.Command("go", direction));`	063	`goRoom("go" direction));`

This error repeats on this line three times before he re-expands this one line of code into two.

	Before		*After*
063	`goRoom("go" direction);`	063	`Command com = getCommandWord, getSecondWord`
064		064	`goRoom("go" direction);`

A `';' expected` error is, in some ways, correctly reported for line 63, although the line itself is far from semantically meaningful Java. Once the semicolon is added, the compiler once again reports `')' expected` on line 64; Roger resolves this by replacing line 64 entirely.

	Before
063	`Command com = getCommandWord, getSecondWord`
064	`goRoom("go" direction);`
	After
063	`Command com = getCommandWord, getSecondWord;`
064	`goRoom(com);`

Now, the semantically meaningless line 63 is the cause of new errors: `cannot resolve symbol - variable getCommandWord`. If the variable isn't able to be resolved, then perhaps it is a method? He adds a pair of `()` to the end of line 63, to which the compiler replies `';' expected`.

	Before
063	`Command com = getCommandWord, getSecondWord;`
064	`goRoom(com);`
	After
063	`Command com = getCommandWord(), getSecondWord();`
064	`goRoom(com);`

We could go on, but we will not; Roger eventually solves this problem by removing these two lines completely; we strongly suspect that his program fails to do what it is expected to do at this point.

4.4.4 Karen: no peace unto the desperate

We have included two of Karen's sessions simply because they represent what is almost typical behaviour for students with error quotients of 0.50 and higher. We will not explore these sessions in depth, as their story is immediately obvious simply by looking at our summary tables.

The first session we've highlighted (Table A.24) comes from early in the semester—it is the Notebook project, which we looked at in-depth with Neville, Ron, Harry, and Hermione. In this session, Karen spends roughly 20 minutes working on her program, and it compiles rarely. She deals with a wide variety of errors (`';' expected`, `illegal start of expression`, `'.class' expected`, `missing return statement`, `'(' expected`, `cannot resolve symbol - variable`, and $<$ *variable* $>$ `has private access in` $<$ *class* $>$), and at no point does she get her code to compile. In fact, she doesn't even get her code close to something that is syntactically correct:

```
public String listNotes()
{
  int i=0;
  while(i
    System.out.println(notes.get(i));
    i++;
  }
}
```

This must be a frustrating experience for the student—how much of Karen's programming time is spent failing to achieve anything? From our snapshots, it would appear that the majority of her time is spent making small changes to her programs, getting errors of one sort or another from the compiler, and generally failing to write a program that runs.

The second session we've highlighted (Table A.25) is seven minutes long—a short session, generally speaking. It would appear that she either no longer programs in the public labs (save when she must, in-class), or Karen has completely given up on the prospect of learning to program; we cannot truly say, given the data we have to hand. We expect that most of Karen's sessions look the same: she explores at least one error type per minute, and never manages to get her code to compile. The code she is working on is no better than the snip we presented above; her behaviour has all the appearance of random guessing, perhaps with the vain hope that she will magically type the correct thing that will make her program work, complete the assignment, and let her get on with her day, week, and life. Or, perhaps she doesn't care, and is just filling time in-class, hoping she won't be asked any questions by her lab supervisor.

We've gone a bit far in attributing emotion and intent to Karen, but it is hard not to—we can tell from the overviews and the source associated with them that she has no clear idea how to program in Java, taken alone or in comparison with many of her peers at the same time of the semester. We assume she is frustrated, or perhaps bored—but we are certain she doesn't have a clue when it comes to writing a syntactically correct Java program.

4.5 The EQ and grades

The error quotient gives us a quantitative view into a large, complex, qualitative dataset. As we saw in the previous section, it allows us to classify students along a single, gross metric, based almost entirely on the repetition of syntax errors. Students with low EQs typically spend their time working with syntactically correct code; when they have to deal with a syntax error, it is a quick and efficient process. On the other hand, students with high EQs can spend ten, twenty, thirty, or more minutes wrestling with one part of their code, and sometimes with the same syntax error.

It is interesting to us that the error quotient appears to correlate inversely with grades. This means that, according to our data, students who have low EQ scores tend to have higher marks on their assignments and exams than students with high EQ scores. At a glance, many of our colleagues say "well, that makes sense," as it seems unlikely that a student who routinely spends 30 minutes wrestling with a `';' expected` error will do well on their homeworks and exams, where correct code with good semantics is expected.

4.5.1 About the population

While much of our research has taken our entire population of students from the 2003-2004 and 2004-2005 academic years into consideration, our claim regarding the connection between the EQ and course marks are based entirely on data from the 2004-2005 academic year. We have focused on this population for two reasons. First, we have more subjects in the 2004-2005 academic year (68), whereas only 31 students took part in our study through the entirety of the 2003-2004 academic year. Second, we have more sessions on record for each of the students in the 2004-2005 year. On average, we have more sessions per student recorded for the 2004-2005 academic year than the previous, fractured year.

Of the 68 students in our study, 56 of them took the final examination. This does not mean that 14 of our students failed out of the introductory course in object-oriented programming; instead, it means 14 of the students in our study were only required to take the first semester of the two course sequence, CO320/520. For purposes of this analysis, we have simply left the students who only took one semester of the sequence out of our calculations. This leaves us with a sample population of 56 students; of these, nine students only generated 2 sessions. So, we are left with 47 students in our sample population.

At a glance, 47 of 213 students may seem like a rather small sample of the total population; we do not believe this is the case. During the 2004-2005 academic year, 213 students started the introductory course; of those 213, only 118 went on to take the second-semester of the course. This is not uncommon, as the students who drop the course at the midterm are joint-majors or students from other departments for whom the second semester is not mandatory. Of the 118 who took the second semester, 56 were in our study, leaving 62 students who were not; 47% of the students who took the final exam, therefore, were subjects in our study through the entire 2004-2005 academic year. Although we eliminate a few of these from our population due to a lack of data, our claim is that we have a significant, representative sample to work with.

4.5.2 Grades: assignments and examinations

We have two kinds of grades for the subjects in our study. The first are the marks they received on their assignments; these assignments are graded by postgraduate students and members of the academic faculty according to a reasonably well-defined rubric. Students sometimes come back to argue they should receive more marks than they did, and sometimes those marks are granted. The process, we are trying to say, is somewhat fluid.

We have two examination grades for our students. Students who take only CO320 have one examination at the end of the year; students who go on to take CO520 take one for each semester of the course. This does mean that students who only take the first semester see several months pass by where they might not be doing any programming before they are tested on it—certainly, not an ideal situation.

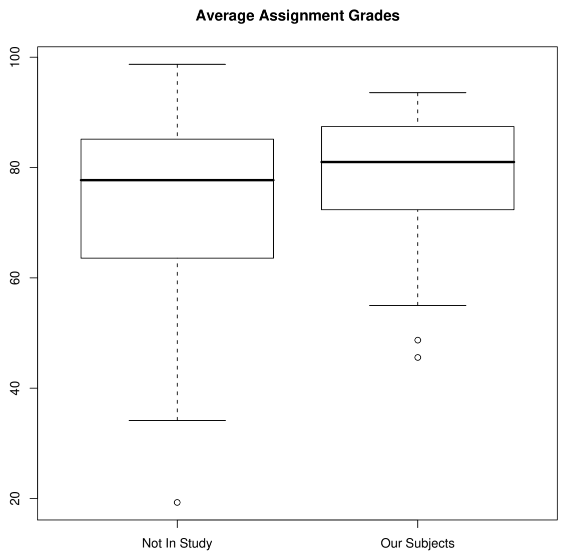
It should be noted that the exams are quite different from the assignments the students work on during the semester. During the term, students work through a series of assignments that expose them to different syntactic elements in the language, as well as different concepts fundamental to object-oriented programming in Java. These assignments are programming-heavy, in that the majority of the points a student can earn on the assignment comes from extending code they are given to do one or more tasks; while working on their code, they can (and do) use many resources—their book, their peers, and their instructor, to name just a few. By comparison, when taking the final exam, they are in a large hall with many other students. They are allowed a pencil, and are able to request paper for scratch work—this must be submitted with the exam, as they are not allowed to leave the hall with anything more than they entered with. While they are expected to write some code on the exam, it is by hand, without the support (or distraction, as the case may be) of a compiler. In short, the way students interact with their code under exam conditions is radically different from what they are accustomed to from their experiences doing coursework during the semester or year.

Assignment grades

Figure 4.12 summarises assignment grades for the 2004-2005 academic year. The boxplots depict the median, first and third quartile, as well as 1.5 times the inter-quartile distance (dashed lines) and outliers (marked individually) for the students in our study as well as those students who are not. In all cases these are students who completed the entire course.

At a glance, it would appear that the students in our study did better on their assignments overall than the students in the rest of the population. We can test that intuition statistically; a two-sample t-test is one mechanism that can be used to tell if two distributions are actually derived from the same parent distribution[Kan99]. In this case, the t-value reported is -1.48, with a p of 0.14. The t-value tells us that the mean of our second distribution is greater than the first, and the p value tells us that we should probably accept the null hypothesis—these two distributions are derived from the same parent distribution. In short, the students in our study tended to do better on their assignments (to some degree) than those who did not take part; given the amount of overlap, however, we would claim that this is not a very powerful claim.

It is possible that the t-test is not entirely appropriate in this case; it is typically only applied when the two distributions in question are independent, and when the distributions being com-



	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Not in Study	19.29	63.57	77.71	75.02	85.14	98.71
	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Our Subjects	45.57	72.35	81.00	78.76	87.43	93.57

Figure 4.12: Distribution of assignment marks for the 2004-2005 academic year.

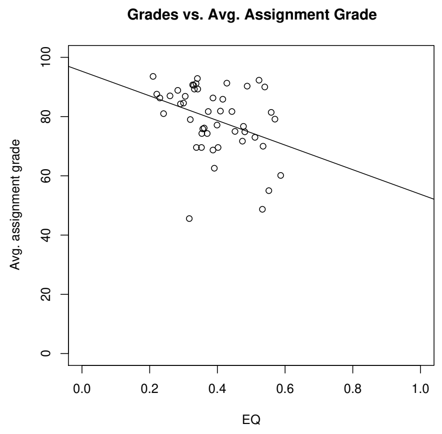
pared are Gaussian. While our two distributions may not be entirely normal, our purpose here is simply to quickly characterise the similarities and differences between these two populations. Applying a non-parametric test, like a two-sample Kolmogorov-Smirnov test[8], we come to the same conclusion as we did with the t-test: these two distributions are similar (D = 0.24, p-value = 0.19). Thus, our intuition that the students taking part in the study had slightly better assignment grades than those who did not take part in the study appears to be reasonable.

We are curious about how similar the grades are for these two populations because we want to know if the students in our study are, generally speaking, *representative* of the population as a whole. It would appear that they did not perform hugely better or worse on the assignments throughout the term than their peers, which gives us some measure of confidence that they are probably, at least on this one metric, typical of a first-year student learning to program.

Figure 4.13 is a plot of the EQ and average assignment mark for each student in our study. Using a simple linear regression, there is a significant inverse relationship between average assignment mark and the EQ (at the p = .01 level). However, an adjusted R-squared value of .11 is poor; an R-squared value of 1.0 in this case would indicate a perfect linear fit.

As we can see, there is a great deal of noise in the assignment data; while there is a possible trend, we can also see a number of students who look, at best, like "outliers." Given the poor confidence value ($p = 0.01$) and $R^2 = 0.11$, indicating a very poor fit of the trend line to the data, we're inclined to think that there isn't any clear connection between students' EQ and their performance on coursework. If we were to remove some of the students from this sample—those students at the extremes—we might find that a reasonable trend might begin to develop. At either end of the coursework-mark spectrum, we could make a reasonable case for eliminating them from this distribution. Students with very poor coursework marks may not have completed the assignments; given that the coursework only accounts for a small portion of their overall grade, students will often "economise" their assignments, focusing on work that is more important, or worth a higher percentage of their overall grades. At the other end of the spectrum, students with high marks and low EQ scores—students who we suspect have a great deal of trouble with syntax—may be cheating. We just might be observing the effect of plagiarism. While we like to believe that all of our students do their own work, we know that

[8]http://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm



Residual standard error: 10.79 on 45 degrees of freedom
Multiple R-Squared: 0.1322, Adjusted R-squared: 0.113
F-statistic: 6.858 on 1 and 45 DF, p-value: 0.01198

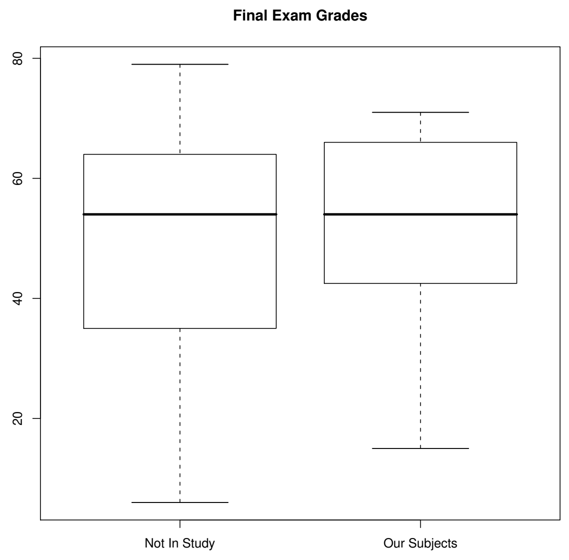
Figure 4.13: EQ vs. average assignment grades for the 2004-2005 academic year.

plagiarism is common in introductory programming courses[LC01]. If 10-15% of our sample population were to plagiarize or cheat consistently, submitting another student's work as their own, they would contribute greatly to the amount of noise in this plot.

Final exam grades

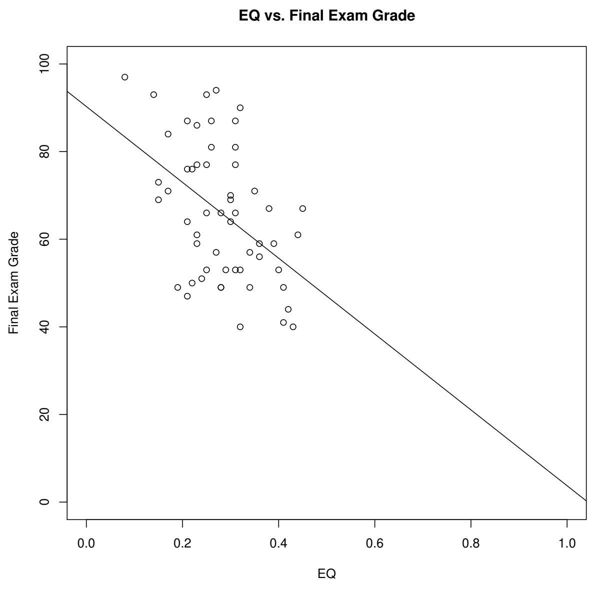
The process by which final exams are graded is much more controlled; the rubrics are clearer regarding what constitutes a correct answer, what is worthy of partial credit, etc., and multiple graders must come to a consensus on the marking of each exam. It is not surprising, therefore, that the relationship between EQ and final exam grades is statistically more significant.

Unlike our assignment grades, the final exam marks for the students in our study (n = 56) and who were not in our study (n = 62) are quite similar (Figure 4.14). Statistically, we believe they are from the same parent distribution. A two-sample t-test (t = -0.944, df = 38.6, p = 0.35) strongly implies they are from the same parent distribution; likewise, a Kolmogorov-Smirnov test (D = 0.18, p = 0.58) also would imply that these two distributions are statistically equivalent. So, we feel comfortable claiming that the subjects in our study did not perform significantly better or worse on their final examination than the students who did not take part in our study.



	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Not in Study	23.00	50.00	70.00	65.48	81.00	96.00
	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
Our Subjects	40.00	53.00	66.00	65.78	77.00	97.00

Figure 4.14: Distribution of final exam grades for the 2004-2005 academic year.



Residual standard error: 11.86 on 45 degrees of freedom
Multiple R-Squared: 0.2671, Adjusted R-squared: 0.2508
F-statistic: 16.4 on 1 and 45 DF, p-value: 2e-04

Figure 4.15: EQ vs. final exam grades for the 2004-2005 academic year.

The correlation between EQ and final exam grade ($p = 2 \times 10^{-4}$) is far more significant than that between EQ and assignment grades, and the quality of the linear fit (as measured by the Pearson coefficient, R^2 = 0.25) is twice as good, but still not excellent by any stretch. We can see that there is significant noise in the data, but by inspection also believe the significance of the trend line: clearly, the two values are related.

4.5.3 EQ: correlation and prediction

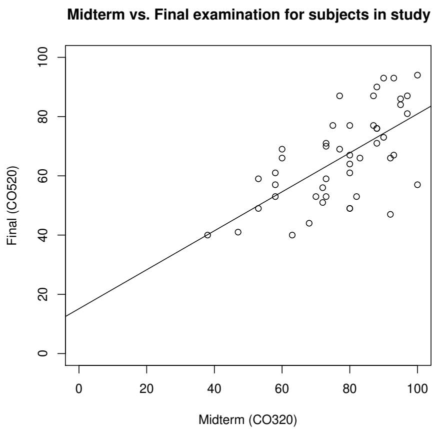
Novice compilation behaviour is a complex thing; the error quotient is just one way of distilling several observable quantities down to a single, non-dimensional number that reflects how a novice programmer deals with syntax errors. It would appear that novice compilation behaviour, as measured by their EQ, correlates with their performance on programming assignments and end-of-year examinations. This implies, to some degree, that their compilation behaviour relates to their performance as measured by traditional, summative measures. However, can we use a student's EQ to *predict* their behaviour on examinations?

As our sample group appears to be representative of the population as a whole, the data that follows will only reflect those students who took part in our study.

CO320 exam vs. CO520 exam

The CO320 examination is a good predictor of final exam performance. Students who do well on the exam covering material from the first term of the course do well on the final (Figure 4.16).

The fit is significant ($p = 3 \times 10^{-6}$), and the quality of the fit is better than any of the correlations we've considered so far ($R^2 = 0.32$). We would hope, in some ways, that the student's performance on material from the first term would reflect their final exam score; it would indicate some kind of consistency in our examination process, and that a student who "knows their stuff" is not just memorizing material by rote, but might have some broader understanding of the material.



Residual standard error: 11.88 on 45 degrees of freedom
Multiple R-Squared: 0.4112, Adjusted R-squared: 0.3981
F-statistic: 31.42 on 1 and 45 DF, p-value: 1.198e-06

Figure 4.16: Midterm vs. final examination for subjects in our study, 04/05.

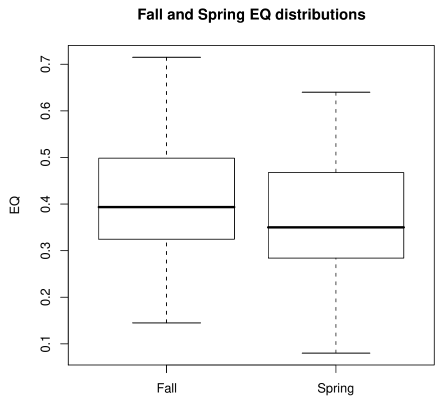
Fall and Spring EQ

The error quotient for our sample population dropped, on average, over the course of the 2004-2005 academic year (Figure 4.17). This is, of course, a good thing; it implies that the population, taken as a whole, learned to deal with syntax errors more quickly and with fewer repetitions of the same error over the course of the year. Of course, some students had lower error quotient scores in the fall semester (implying their compilation behaviour was better earlier in the term), but this kind of effect may be an artefact of the limited data we have for some subjects in our study. We are pleased that, in the aggregate, it would appear that our students showed some improvement in their average error quotient scores.

Given that (it appears that) their compilation behaviour improved over the course of the year, is it in any way predictive of their performance on their midterm and final examinations?

Fall EQ vs. CO320 examination

It would appear that there is some significance to the correlation between the our students' EQ (averaged over the Fall 2004 semester only) and their CO320 examination scores (R^2 = .22, p = .0004, Figure 4.18). Our previous comparisons of course marks and a student's EQ assumed



Fall 2004 EQ

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.15	0.32	0.39	0.42	0.50	0.72

Spring 2005 EQ

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
0.08	0.29	0.35	0.37	0.46	0.64

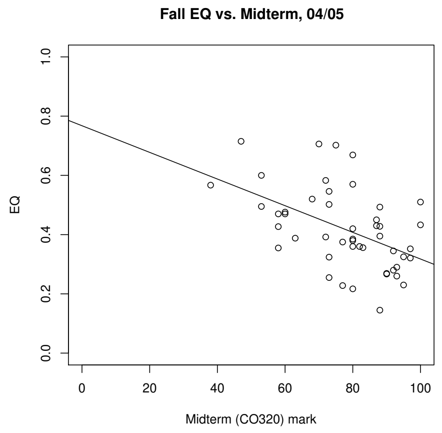
Figure 4.17: Fall and Spring EQ for subjects in our study, 04/05.

that the error quotient was the *dependent* variable; here, we are assuming that a student's performance on an examination is actually dependent on their EQ score.

It would appear, based on our data from the Fall of 2004, that a student's EQ may have some predictive power. While the correlation is not good enough to consider dropping examination completely in favour of EQ observation alone, it is interesting to see that, for our Fall data at least, our students' behaviour may possibly be an indicator of exam performance.

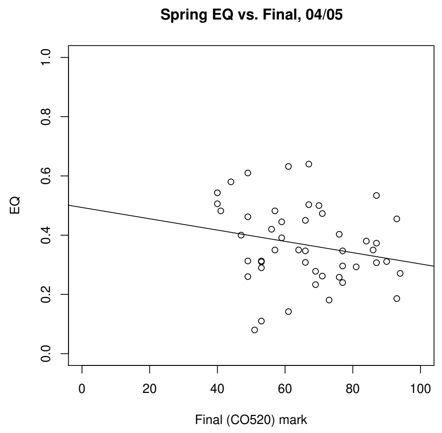
Spring, Fall EQ vs. final examination

Figure 4.19 plots a student's EQ from the Spring 2005 semester vs. their final exam scores. As can be seen, the fit has almost no predictive power (R^2 = 0.03), and no statistical significance (p = 0.13). We already know that the students' EQ, taken over the entire year, does correlate with their final exam marks (Figure 4.15). Given the variability in our data at the level of the individual session, and the heavy filtering we have already carried out to restrict the 2004-2005 population, it is difficult to explain why there is significance in the data taken over the entire year, and that which comes only from the second half of the year.



Residual standard error: 13.01 on 46 degrees of freedom
Multiple R-Squared: 0.2413, Adjusted R-squared: 0.2248
F-statistic: 14.63 on 1 and 46 DF, p-value: 0.0003919

Figure 4.18: Fall 2004 EQ vs. midterm exam scores for subjects in our study.



Residual standard error: 15.14 on 46 degrees of freedom
Multiple R-Squared: 0.048, Adjusted R-squared: 0.028
F-statistic: 2.357 on 1 and 46 DF, p-value: 0.1316

Figure 4.19: Spring 2005 EQ vs. final exam scores for subjects in our study.

Our suspicion is that we have less quality data (sessions of significant length, enough sessions per student) to work with in the Spring term. This could be due to students doing more work off-campus as they become more familiar and comfortable with the programming process. With a more thorough/invasive approach to data collection, we might develop a more complete picture of novice programming behaviour not just late in the second semester, but throughout the entire year.

4.6 Conclusions: putting the EQ to use

Our research involved the analysis of the "invisible", or heretofore unobserved interactions that take place between a student and their compiler. From these observations, we believe a number of conclusions can be drawn that have immediate value to practitioners teaching today. These observations also suggest as many questions as they answer; there is ample work to be done.

4.7 Qualitative assessment

A strong case can be made for applying our rich visualisations of novice programmer behaviour and their subsequent distillation into a single error quotient value in the classroom context immediately. Our current mechanisms for assessing the progress of our students are crude and sparse at best—assignments are too few and far between, typically involving little or no interaction between the student and their instructor. Likewise, examinations are even more infrequent, and students' fear the negative repercussions of doing poorly on these one-off exams. In all cases, it is rare that students are given the opportunity to submit work for evaluation, obtain feedback, and evolve their work based on this feedback.

Put another way, there is little room in today's cost-conscious university for models of instruction and learning that approach the apprenticeship model, and therefore we must look for ways to approach that as cost-effectively as possible. We believe that automated tools based on the notion of novice programming behaviour can provide an instructor with rich, yet succinct information about their students—information that they can use to improve their instruction, and their students opportunities to learn, in measurable ways.

4.7.1 Summative vs. formative assessment

Our correlation between novice programming behaviour (as measured by a student's EQ) and their performance on assignments and exams is not unreasonable: they are both a form of assessment. Assignments and examinations are both examples of **summative assessment**. Summative assessment tends to be comprehensive in nature, and is often used to test student understanding or learning at the end of a section, term, or module.[AC93] In this regard, both the assignments given to students throughout the year, as well as the final examination, were summative measures of student performance.

While it is typical for assignments to serve as a kind of **formative** assessment—delivered as part of the learning process—they are often marked too slowly to be of particular use to the student. Furthermore, assignments currently mark the end of the learning process: students are not encouraged to correct and resubmit assignments to improve their marks (and possibly their understanding) of the material being explored by the assignment in question. As a result, students progress through the semester with little or no interactive, formative assessment.

Using our tabular visualisations of novice programming sessions and the error quotients for those sessions, an instructor can easily tell whether a student tends to get "bogged down" in their programming (like Neville), or if they tend to quickly manage syntax errors in their program and focus more of their time on the semantics of their code (like Hermione). This information is (we believe) inappropriate for "evaluating" or otherwise "grading" a student; however, they are an excellent window into their process. As a tool for instructors, they can be used as a basis for suggesting any number of interventions:

Tutoring Students could be encouraged to come in for one-on-one tutoring sessions with their instructor or the class teaching assistant. Or, the instructor could set up a network of "peer tutors," where students who are struggling with syntax can sit down and work with peers who are more confident in their programming abilities. Many departments already have structures like this in place for helping students with all kinds of programming difficulties; however, the observation of students' programming behaviour provides the opportunity to offer help in a timely and targeted manner.

Peer-groups Similar to tutoring, the instructor might recommend several students who are struggling with syntax work together on the next assignment. Sometimes formalised as "pair programming," the instructor could place weak students together, so they might struggle together with the concept of syntax.[WU01, Van04] Or, perhaps the instructor might pair stronger students with weaker students, but insist that the weaker student be responsible for the actual typing when developing code, thus encouraging communication between the students about the work they are doing.

Passive tutorials We have found that many of the errors students struggle with can be turned into excellent documentation for future students. Providing students with a step-by-step walk-through of erroneous code, and discussing strategies for fixing it, may provide insights that students can then bring to their own programming practice. That said, it might be that students simply spend too much time attempting to write programs, and not enough time reading and understanding programs. The readability of code has most often been discussed in terms of whether or not students can comprehend the meaning of that code.[LAF$^+$04, Nor82] The readability of visual languages has been explored[GPB91], as well as the role of typography in code presentation[Bae88]. We are not aware of any work exploring the connection between the reading of code and student comprehension of a language's syntax.

Interactive tutorials Meta-content for students to read is useful, but may be difficult for some students to apply to their own learning. As viewed by many novices, programming is more of a kinetic than design activity, and as a result tutorial content that is not active in nature may do little to aid troubled students.

BlueJ ships with a complete embedding of Javascript; this extension is generally referred to as the "BlueJscript" engine. Using BlueJscript, it is possible to create interactive tutorials that take students through implementing programs in very small steps. For example, we have experimented with walking students through the definition of classes and methods one line at a time, and verifying the correctness of their input at each step[9]. Additional tutorials based on existing novice errors could be developed—"fix-me" challenges—allowing students to explore the mistakes of others in an active, tutorial-based environment embed-

[9]http://trails.cs-ed.org/

> ded within the BlueJ IDE.
>
> Having developed several of these tutorials, it is unclear what value they have for the student. While they appreciate the step-wise nature of the tutorial, it is not clear they are able to apply what they learn in the tutorial to the larger task of writing and editing programs. At the same time, the notion of being able to provide a drill-and-practice environment for practising writing basic syntactic elements in the language has been well received by secondary and tertiary instructors who have seen the system in action; certainly, the utility of such a tutorial system will be dependent on how it is employed in the classroom.

As can be seen, a great deal can be done on the instructor's part when they have even limited insight into the beginner's actual programming behaviour. This kind of formative assessment—not used to evaluate success, but instead continuously measure progress—is currently missing from most introductory programming courses.

4.7.2 EQ as a real-time tool

While it is possible for instructors to use the EQ as a formative evaluation mechanism, it is also possible that we could put the tool in the students' hands. There is nothing that prevents us from running the error quotient algorithm over as few as three events, and displaying either the numeric EQ, or some simplified iconic representation, directly in the editor window. This way, students can get immediate feedback as to how well or how poorly they are managing the syntax of the language and the errors they are (or are not) getting from the compiler.

At the same time as it is possible, and provides immediate feedback to the student, it may become part of the information overload that a struggling student is already dealing with. Does it help a student who cannot fix a `';' expected` error to know that their EQ is 0.83? Probably not. The instructor is likely better equipped to be able to look at the data from a given programming session and make recommendations to the student about how they might improve their programming practice or solve similar problems in the future.

That said, if the algorithm is running in real-time, it is possible to *automatically* suggest to the student that they seek help. If, after a number of compilations, they continue to struggle with one type of error, or with one region of their code, we could prompt them to ask for guidance from a peer or their instructor. "It looks like you're having a hard time with the `illegal start of expression` error; is there someone you can ask for help?" This kind of prompting can go any number of ways; it may become another dialog box that the student clicks simply to make it go away. Then again, having read hundreds of sessions where students spend 15, 30, and 45 or more minutes on one line of code, thrashing through one syntax error after another, we want to believe that *anything* that prompts them to get out of their rut may be of value. A balance would need to be struck; we don't want to encourage "stopper" behaviour (in Perkins's terminology), but we certainly want to encourage students to recognize when they need to seek help.

4.7.3 Application vs. evaluation

We have heard our colleagues say that "every class is an experiment." That is to say, they constantly try new things on their students, and see what happens. Unlike a real experiment, we find that many of our colleagues do not keep notebooks or other data regarding their teaching practice; they rely on their perception of how their "experiment" was received, and may (or may not) remember to apply what they observed in future iterations of their course.

What we have suggested here are a number of ways we might apply our observations regarding novice programming behaviour to the classroom context. Each of these might be considered an "experiment" in its own right. If we were to make use of a student's EQ value for recommending further tutoring, it would be interesting to see how the end-of-year exam marks did (or did not) change from previous years (assuming some consistent trend from year to year). Or, we might make more instructional material available to students based on this research—but we would want to know how the students made use of that material in their studies, and how often. In short, for it to be an *experiment*, we would want to have some notional hypothesis, as well as some mechanism for collecting data to verify or negate that hypothesis.

For the practitioner, our qualitative work can provide critical insight into student behaviour. For the researcher, we believe it begs many questions, and provides a fertile starting point for future work.

Chapter 5

Looking Forward

It is possible that a student's programming behaviour (or EQ) is a predictor for their academic performance on traditional assignments and exams. From our quantitative analysis of students' EQ and their relation to exam marks, we are inclined to believe there is a correlation between these two figures. Given that examinations of novice programmers tend to have syntactic and semantic elements, we intuitively believe that the student who cannot master syntax will likely fail to master the semantic challenges of programming. For example, a student who never masters the syntax for declaring methods of a class may never develop a complete understanding of how objects are created and interact with each other—this is implied, in part, by our apparent correlation between a student's EQ and their performance on end-of-year examinations.

Unlike our qualitative work, a single instructor collecting data on a single classroom of students will not provide an adequate pool of subjects for further exploring our larger questions regarding the EQ as a predictor of student performance. Our research hints that there probably is a correlation, but further work will require both a refinement of method and an expanded pool of subjects for further analysis.

5.1 Analytical refinement

Our algorithm for calculating the error quotient is simple, yet effective (Figure 4.4). However, as can be seen from our analysis, it is not a terribly powerful discriminator. While the students who manage syntax errors best are well separated from those who cannot manage syntax errors at all, the students whose EQ sits within one standard deviation of the mean are too varied. Upon closer inspection, some students in this middle-range seem to make progress, while others seem to struggle mightily.

The algorithm we employ for calculating a student's error quotient evolved out of our qualitative analysis of novice programmer behaviour. Having read through thousands of edits made by dozens of students, our algorithm reflects a theory of novice programming behaviour that evolved throughout the process of our research. The criteria we found most useful for characterising a student's programming session developed slowly over time; it was then codified and refined in the form of the algorithm presented in Chapter 4. That said, there are easily observable quantities that do not factor into our analysis; *time* is the most notable of these quantities. Does it matter, for example, if a student makes three edits in rapid succession that all end in errors vs. making three edits with a minute or more in-between that are error-free? In the same vein, do the number of characters added or removed matter? Or do we need to consider much higher-level behavioural data? In each case, our ultimate goal is to better infer whether the student is struggling or making process in their battle with the compiler.

Other observable quantities include the absolute time of day the work is being carried out (morning vs. late at night), whether a deadline is imminent, and how long the session has carried on. For example, if a student has been working for two hours without an apparent break, will their ability to focus on syntactic detail be diminished? All of these and other observable quantities may aid us in further discriminating between those students who are productively employing their time programming and making progress vs. those who are struggling and in need of assistance.

5.2 Expanding the methodological sphere

Our research to date has been based extensively on the manual and automatic analysis of documents produced as a side-effect of our students' programming process: we have read (and processed) a lot of code. While this is a fascinating view of what takes place while the students are developing their code, it is only part of a much larger context.

When working in a classroom, students have many more modes of interaction available to them than just the compiler. Students can raise their hand and flag down the instructor—or, perhaps, the instructor will offer help unrequested. Depending on the situation, a student might ask a friend for help, or perhaps they have been explicitly requested to work in pairs. If we are interested in these human interactions that take place during the programming process, we will need to sit in classrooms and observe what takes place. And even then, we must decide what we are looking for: are we counting the number of times students ask for help? Are we more generally interested in the kinds and quality of interactions that take place between people in an introductory programming laboratory? Would we later wish to correlate this information with an automated collection of the compilation data?

Simply counting events and correlating them with our existing data could be very interesting; there are many times when we wonder how a student "broke out of" a string of bad compilations; we might see four, five, or more failed compilations, all ending with the same error in the same part of the code. Then, after a reasonably long delay (2 or more minutes), we see that their code improves markedly. Simply knowing that during those two minutes the student asked for some kind of help would allow us to begin correlating the ends of error sequences with interactions with the instructor or the student's peers.

However, the quality of the interactions may be more important than simply knowing when they took place. As Barker, Garvin-Doxas, and Jackson reported in "Defensive Climate in the Computer Science Classroom," computing classrooms are not typically the most supportive and encouraging places for students to be.[BGDJ02] A laboratory setting where the instructor is always ready to gush over a particularly "elegant" or "concise" piece of code, but rarely encouraging or supportive of smaller victories might discourage students from asking "stupid questions," or otherwise draw attention to themselves. With additional pressures regarding plagiarism and copying, students are often left wondering what they can and cannot talk to their peers about. This tends to yield a very closed and distant environment where ideas are not freely discussed and engaged—two processes we believe are critical to learning programming.

So while simple observational techniques might provide more (automatically) processable data, more in-depth observations (and perhaps focused discussion groups or interviews) are necessary to develop a complete picture regarding the kinds and quality of interactions that take place between students, their peers, and their instructors.

5.3 Small population, more data

We suspect that many of our students did a great deal more programming than our automatic data collection suggests; in fact, given the demographic information we have regarding computing students attending Kent over the past several years, we know for a fact that 95% or more of our students have access to computers off-campus. Therefore, to come to a better understanding of these students' compilation behaviour, we must observe their programming behaviour outside of the classroom.

Capturing programming behaviour data regardless of where the student is presents both technical and ethical issues. Technically, we can no longer assume that a globally unique identifier (a username) will be provided by the operating system, nor can we assume a high-availability network; while DSL and similar technologies are becoming more common, there are still many students who work with a dial-up connection that is infrequently connected to the Internet. Likewise, many of our students make use of laptops that are infrequently connected either to dial-up or a high-speed LAN. These are difficult issues to deal with transparently (from the student's perspective) and robustly (from the researcher's perspective).

Some, but not all, of the technical concerns could probably be overcome. Data could be stored local to the student, and infrequently shipped to a server, either on physical media or over a network connection. Matching a delayed datastore to a student is trickier: we no longer have a guaranteed student ID. We could provide custom BlueJ extensions to each student, which could provide some level of assurance that extension build #2933 (which was given to, say, Hermione) collected all the data in a particular data store. However, the process is significantly more invasive: whereas students in our study were likely oblivious to much of the data collection going on after a short time, students in an extended, and more personal, study would necessarily be very much aware of the data collection that was taking place every time they compiled their code.

Technical issues aside, there would certainly be ethical issues to consider. Could we provide assurance that such data would not be used to blame, or otherwise prosecute, a student who was suspected of cheating? Once we are observing them "everywhere," so to speak, it becomes very tempting to look and see just how much time a student spent on a particular assignment. If the first time a program appears it is 90% complete, clearly something suspicious is going on... and that is not the purpose of this kind of research. How these and other ethical concerns would be handled are questions that would need to be addressed before a more invasive/personal data collection effort could be undertaken.

5.4 Error quotients everywhere

Without looking deeper into our own classroom environments, and without peering deeper into the programming lives of our students, we could simply expand the scope of our own data collection. Replicating our (relatively) low-fidelity data collection at other institutions is a possibility. For example, if we can do a similar, low-cost, low-impact collection of novice compilation data at another institution, we might find that the error quotient is a more general phenomenon than we thought.

With the start of the new millennium we have seen a growing trend amongst computer science education researchers to develop larger and more inclusive studies of novice programmers. McCracken et al's multi-national, multi-institutional study of assessment of programming skills of first-year computer science students provided one of the first broadly applied characterisations of novice programmers at the university level[MAD$^+$01]. Their study, carried out at eight institutions in five countries (Australia, Israel, Poland, UK, USA) with a total of 217 participants, challenged students to write a series of short programs that would evaluate arithmetic expressions:

> The exercises focused on arithmetic expression evaluation. The easiest of the three exercises (P1) required a computer program to evaluate a postfix expression. The second exercise (P2) required a computer program to evaluate an infix expression with no operator precedence (the operations were to be performed strictly left to right, with no parentheses present). The last exercise (P3) required a computer program to evaluate an infix expression with parenthesis precedence (operations were

to be performed left to right, with parentheses forcing sub-expressions to be evaluated first).

These exercises were then marked according to a common rubric, scored, and the data compiled and evaluated. One of the results reported by McCracken et al. resonates with our own experiences observing data regarding novice programmers:

> The first and most significant result was that the students did much more poorly than we expected. There are many possible causes: Our expectations may have been too high, the problems may have been too hard or a poor fit to the students' background and interests, there may not have been enough time given, and so on.

We observed this same phenomenon in our own research: after collecting a large amount of data together in one place regarding novice programmers, the inconsistencies between individuals begins to wash out, and you are struck with, in some cases, how poorly students seem to fare with the task of writing programs. This tells us something important: we, as educators, sometimes do not realize how little our students really know or are capable of.

The McCracken study was interesting, and demonstrated that we could, without too much difficulty, look outside our own walls as computer science education researchers, and attempt to develop a kind of "bigger picture" regarding our own experiences and the data we might collect in our own institutions. This process has been formalized, and exported, through the successful Bootstrapping Research in Computer Science Education and Scaffolding Research in Computer Science Education workshops run by Sally Fincher, Marion Petre, and Joshua Tennenberg[PFea03, SFB$^+$05, FT05]. Each of these workshops brought together 20-30 practitioners in computer science education, and over the course of several days, provided a "crash-course" in research practice and methodology. Over the course of the next year, the participants put into practice a research "kit," collecting data regarding students in their own university. At the end of the year, this data is brought together, cleaned, and the now aspiring researchers dive into an analysis and report their findings. This process has been well documented at this point, and instituted four times on three continents, with over 120 researchers participating[FLC$^+$05].

These workshops not only provide a model for distributing research, but additionally provide a context in which researchers new to the field are introduced to the process of carrying out a rigorous computer science education research study. If we are confident that novice compilation behaviour is a fruitful area for continued exploration, a distributed, multinational study could provide an invaluable, empirical base for further research and collaboration.

5.5 Thousands of languages, thousands of students

There are many other programming languages used to teach novices how to program. Scheme, C, Smalltalk, Pascal, Ada... all of these languages and more appear in the first course on programming *somewhere*. In many cases, there are pedagogic environments not entirely unlike BlueJ that are used for introducing students to programming. And in the case of every language, it is not entirely unlikely that students struggle with the syntax of the language they are being taught.

We would like to think that the error quotient is a language-neutral measure; any language and/or environment with an edit-compile cycle is probably attacked by students in a manner very similar to the way our students wrote programs in BlueJ. Currently, we only have our own experience to indicate that this would seem to be the case. However, we have to wonder—is it worse for novice Java programmers than novice Scheme programmers? These kinds of questions would be difficult to answer, but without data of some sort, we'll never know.

5.6 Future work: a three-year plan

There are many ways to take this research forward; we have a rich set of data and preliminary analysis that inspires more questions than it answers. At one level, it would be fascinating to instrument DrScheme, Helium, Squeak, and Visual Basic, and look for similarities between the data we collected in BlueJ and similar data from these other environments. However, such a comparison would be riddled with questions of comparability and significance—it would be fun, and interesting, but it would represent a substantial leap of faith from where we are now.

Our research helped us develop a more complete picture of novice compilation behaviour, but the fundamental question—"What are they doing?"—remains unanswered. Having developed tools for quickly distilling complex data resulting from the program development process into simple tables and quantitative measures, we are in a position to use these tools as a starting point for exploring more interesting questions regarding the programming process of novice programmers in the wild.

5.6.1 A three year plan

We propose a three year plan to further develop the depth and breadth of our understanding of novice compilation behaviour (Figure 5.1). It is not far reaching in scope; our desire is to make incremental progress in our exploration of novice programmers, and develop robust tools and techniques for exploring their behaviour. Small steps, not big leaps, are called for.

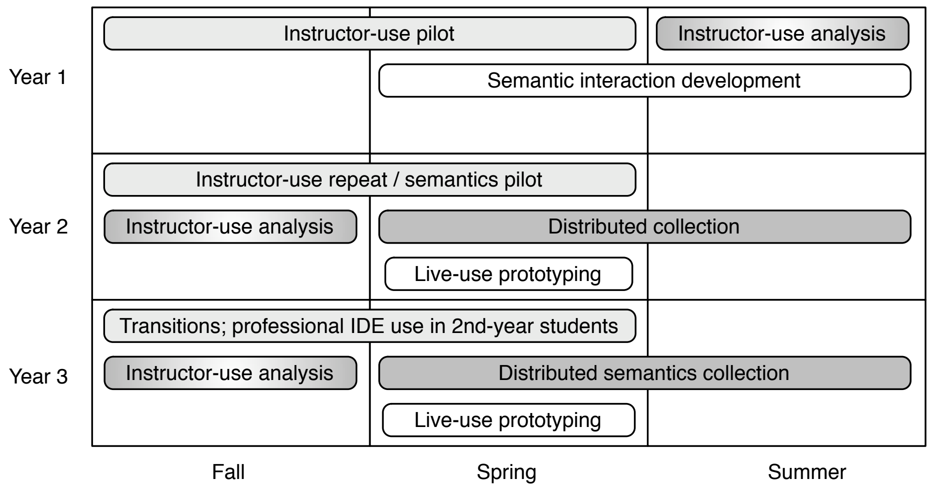


Figure 5.1: A three-year plan for further research into compilation behaviour.

In our first year, we aim to fill in details about novice compilation behaviour we are missing now. In the second, we would grow our scope of data collection to include not just a student's syntactic interactions with their IDE, but also their semantic actions—while also repeating some of our previous collection, as repetition is one of the cornerstones of a good experimental method. In this vein, we have colleagues who are interested in exploring this kind of behaviour in their own classrooms—distributing our data collection framework and analytical tools to other institutions is important to us, both for building a broader base of data, and for growing the computer science education research community. We then propose to expand our sphere of collection and analysis into the second year of a computer science degree, and look at the compilation and interaction behaviour of second-year students who are making the transition from their initial learning environment into a more powerful, or “professional” integrated development environment.

5.6.2 Year 1: putting the EQ to use

The error quotient and session summary tables provide instructors the means to quickly assess whether or not their students are mastering Java’s syntax when working in BlueJ. Our first goal is to explore whether or not instructors find these tools to be useful, and explore how rapid, formative feedback regarding novice programming behaviour can be put to use in the classroom.

We are interested, first and foremost, as to how this impacts the *instructor*: in what ways might programming instruction change when real-time, formative feedback is available? We have focused our research to date on the previously invisible interactions between a student and the compiler. There is a growing body of research regarding the applicability of pair programming and its value in both the classroom and the workplace[WU01, Van04]. Now, we need to better understand how to use information regarding our students’ compilation behaviour in the classroom. We believe that this is an important step towards developing a coherent theory of instruction regarding the first teaching of programming.

Reigeluth defines an instructional-design theory as “a theory that offers explicit guidance on how to better help people learn and develop”[Rei99]. Currently, we have few tools, as programming instructors, for 1) assessing our students in a formative manner, and 2) appropriate interventions and instructional tools we might apply when we think an intervention might be called for. Generally, strategies range from “have students read more code” to “have students write more code.” Our goal in focusing our first year of study on the instructor is to better understand how they make use of tools like the EQ in their instruction. Currently, we can only guess at the utility of our visualisations and the EQ in a classroom context; a concerted exploration will allow us to begin developing a theory of instruction that is evidence-driven.

To explore how instructors employ these tools, we would collect data as we have so far;

no changes to our existing infrastructure would be necessary. However, new tools would be needed to allow instructors to quickly and easily assess the compilation behaviour of their students. At the least, this presents a number of visualisation challenges[Tuf01], but those are best tackled through an iterative design process involving the primary consumers of the information—the instructor involved in the pilot. Our larger goal in focusing on the instructor is to develop an empirically-driven theory of novice programming instruction.

Our first year of research will be largely qualitative in nature, as we carry out an extended observation of the instructor in action, both in lecture and the laboratory. In addition, we would propose a bi-weekly series of interviews with the instructor, focusing on how they were responding to—and making use of—their ability to "peer into" the compilation behaviour of their students. Lastly, we would engage in several small-group interviews with students, exploring their perceptions of the instructor, with a particular focus on the interactions they have between the compiler, their peers, and the instructor. In short, we wish to begin "filling in the gaps" in our current research.

Semantic interactions

In instrumenting BlueJ for data collection, we have focused entirely on the students' interactions with the compiler. However, environments like BlueJ often have tools to support the novice in exploring the run-time behaviour of their programs; in the case of BlueJ, it provides students with the ability to directly interact with objects through the *object bench*. This allows them to instantiate classes, invoke methods on objects, and generally play the part of the `main` method in their Java programs. In addition, BlueJ provides integrated support for writing and running unit tests, as well as a simplified debugger. There are many interesting kinds of semantic interactions students can have with their programs, and we have no data regarding what they do in the time after their program compiles successfully.

While carrying out our extended observations of an instructor putting our tools into practice, we would begin developing a second-generation data collection framework to provide insights into students' interactions with their programs *after* they are syntactically correct. This supports our larger goal of providing more information for instructors to make data-driven, reflective choices about their instructional strategies in the classroom.

5.6.3 Year 2: distributing collection

Year two looks very similar to year one, with an expanded pilot to include student run-time interactions with their programs. We consider this a pilot study much like our first year, as it will represent the first time we have a window into the our students' semantic interactions with their programs. How we analyse this kind of data, and how it might guide instructors in providing a better learning environment for their students is the open question which we would be exploring.

Leading into the second semester, we would encourage our contacts in the southern hemisphere (Australia, New Zealand) to employ the refinement (from year one) of our observational tools in their own classrooms. Our goal in this distribution is two-fold: first, we hope to provide more tools for instructors to make good decisions about their instruction; second, we wish to develop a larger pool of data regarding novice programmers working in Java from which we can make statistical generalisations. Through distribution of the data collection framework, we bring other practitioners into the research process, and develop a wider base of student behaviours to reason about. This distribution would be coupled with a scaled-down version of the qualitative work we carried out in our first year; with on-site visits, we would hope to gain insight into the environments in which the tools are being employed, how they are being used to guide instruction, and provide a vehicle for disseminating best practices between the sites involved in the study.

Lastly, given time, we would attempt to pilot the "live-use" of error quotients in the classroom. Can we automate the intervention process we have been developing, and encourage students to seek help or access resources without an instructor reviewing their compilation behaviour? Such a tool could only be developed in light of how a real instructor made use of compilation behaviour data (gathered in year one), and how students respond to those interventions (year one and two). We do not have any great hopes for such an "expert system," but feel we would be remiss if we did not attempt to encode and encapsulate some of the knowledge gathered to that point.

5.6.4 Year 3: repetition

Projecting three years into the future, with an unknown research agenda, is a challenge at best. Were our initial pilot experiments successful, we would look to the Bootstrapping and Scaffolding projects as a model for distributing research broadly across researchers and institutions[SFB+05, FLC+05]. Assuming we are collecting good data that has use for instructors, developing a broader usage base, as well as continuing to reflect on the impact of that data's use in practice, is a good step. Assuming, for example, that we as instructors can gain greater insight into how our students are learning to program using *both* syntactic and semantic interaction data, distributing our refined tools and instructional recommendations to a larger community is a critical form of dissemination.

Predicting where we might expand into is difficult; however, there are some natural "next steps" that may nicely bridge our work regarding novice programmers with existing research regarding experts. We know from experience and conversation with other educators that the transition from BlueJ to larger, professional integrated development environments is often challenging for their students. This "phase change" is necessarily a fascinating milestone in a student's evolution as a programmer: they are transitioning from the tools of a novice to the tools of an expert.

Exploring students' compilation behaviour and (if possible) semantic interactions with their programs in a professional IDE at the time of transition may yield interesting insights and connections that instructors can continue to exploit in the classroom.

5.7 In conclusion

With this work, we have just begun to realise that novice compilation behaviour is a non-trivial area of study unto itself. In the course of our work, we developed powerful, automated tools for analysing novice behaviour, we can only begin to scratch the surface of what students are doing when they are writing their first programs.

Further work will require us to ask additional questions, and attempt to answer them in appropriate ways. We have already begun preparing to continue our data collection on a new cohort of novice programmers, and to expand the data we collect to include basic student interactions with their code *after* it is syntactically correct. With this new data, we hope to address some simple questions regarding whether students stop and test their code when it is syntactically correct, or if they simply use the compiler as their one and only metric for correctness.

The exploration we are most excited to begin is the utility of our visualisations and analyses as tools for supporting the instructor. If we are able to develop simple, usable tools for programming teachers to carry out ongoing, formative evaluations of their students, then we believe our initial explorations into the area of novice programming behaviour will have achieved more than we could have hoped for at the outset. While evalutating the applicability and utility of visualisations and representations of programming behaviour is a non-trivial task unto itself, it is one we are ready and eager to explore.

Appendix A

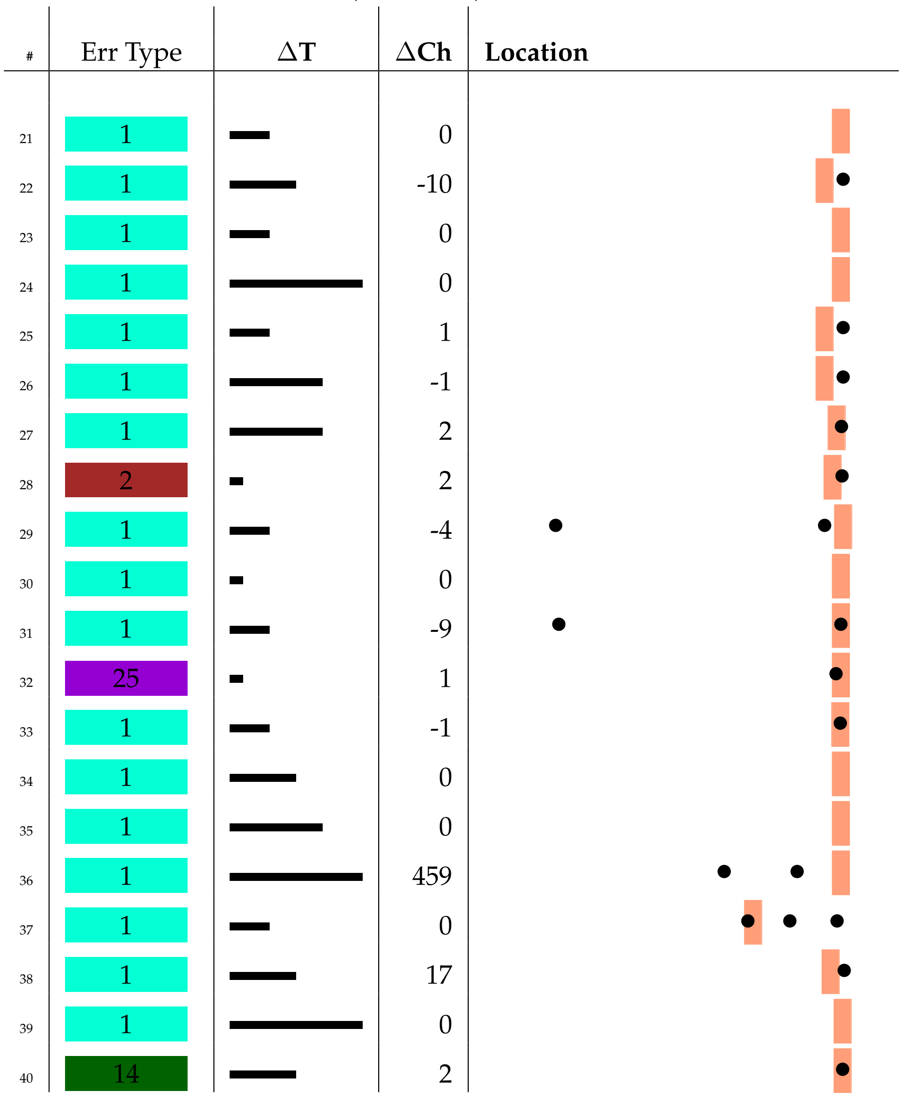
Traces

Table A.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	

Neville, Session 1, Continued...

#	Err Type	ΔT	ΔCh	**Location**
41	14	▬▬▬▬▬	-2	▮ ●
42	14	▬▬▬▬▬	-94	▮● ● ●
43	14	▬▬▬	0	▮
44	⋆	▬▬▬	2	▮●
45	3	▬▬▬▬▬	102	● ● ●
46	⋆	▬	0	▮● ●
47	⋆	▬▬▬▬▬▬▬	-18	● ● ●
48	13	▬▬▬▬▬▬▬	0	● ● ●
49	⋆	▬▬▬▬▬	-60	▮● ● ●
50	⋆	▬▬▬▬▬▬▬	4	●

Table A.2: Ron, Session 1

Project:	notebook1	**File:**	Notebook.java
Duration:	25m39s	**EQ:**	0.39
Date:	Tuesday, November 2nd, 2004 3:03pm		

#	Err Type	ΔT	ΔCh	**Location**
1	1	▬▬▬▬▬▬▬▬▬▬	49	●
2	2	▬	1	▨●
3	2	▬	0	▨
4	2	▬▬▬	0	▨
5	8	▬▬▬▬▬▬▬▬▬▬	9	●▨
6	⋆	▬	-1	▨●
7	1	▬▬▬▬▬▬▬▬▬▬	62	●
8	1	▬▬▬▬▬▬▬	39	▨●
9	4	▬	2	▨●
10	4	▬	0	▨
11	4	▬	3	▨●
12	3	▬	-1	▨●
13	2	▬	1	▨●
14	24	▬▬▬	2	▨●
15	⋆	▬	-1	▨●
16	3	▬▬▬▬▬▬▬▬▬▬	85	●
17	2	▬	2	▨●
18	21	▬	2	▨●
19	8	▬▬▬▬▬	-88	▨●
20	⋆	▬	-1	▨●

Table A.3: Harry, Session 1

Project:	notebook1	**File:**	Notebook.java
Duration:	24m34s	**EQ:**	0.21
Date:	Thursday, November 4th, 2004 10:13am		

#	Err Type	ΔT	ΔCh	Location
1	3	▬▬▬▬	99	●
2	3	▬	1	▨●
3	2	▬	-2	▨●
4	⋆	▪	1	▨●
5	1	▬▬▬▬	248	●
6	1	▪	1	▨●
7	2	▪	3	▨●
8	21	▬	2	▨●
9	18	▬▬▬	-19	▨●
10	⋆	▬	4	●▨
11	⋆	▬▬	-1	●
12	⋆	▬	1	●
13	⋆	▬▬▬▬	0	
14	⋆	▬▬▬▬	88	●
15	3	▬▬▬▬	115	●
16	⋆	▬▬	-2	▨●
17	⋆	▬▬▬	46	●

Table A.4: Hermione, Session 3

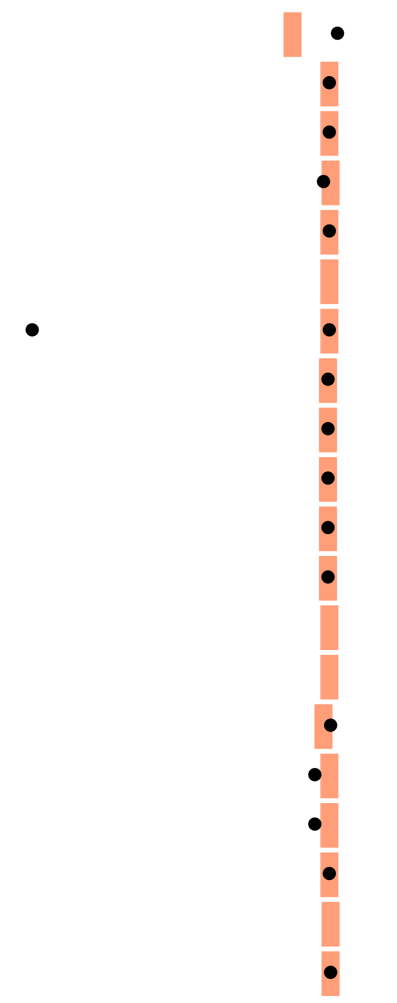
Project:	notebook1	**File:**	Notebook.java
Duration:	18m11s	**EQ:**	0.06
Date:	Tuesday, October 28th, 2003 3:18pm		

#	Err Type	ΔT	ΔCh	**Location**
1	⋆	▬	2	🟧●
2	⋆	▬▬	5	●
3	⋆	▬▬	-1	●
4	⋆	▬	2	●
5	4	▬▬▬	-29	●
6	3	▬	10	🟧●
7	⋆	▬	2	🟧●
8	1	▬▬▬	78	●
9	2	▬▬	10	● ● 🟧●
10	⋆	▬	4	🟧● ●
11	⋆	▪	8	●
12	⋆	▬▬▬	6	●
13	⋆	▬▬▬	4	● ●

Table A.5: Terry, Session 19

Project:	Assignment3	**File:**	CourseMarks.java
Duration:	37m08s	**EQ:**	0.71
Date:	Friday, November 7th, 2003 11:35am		

#	Err Type	ΔT	ΔCh	**Location**
1	4	▬▬▬▬	139	
2	9	▬▬▬▬	18	
3	1	▬	5	
4	4	▬▬▬▬	2	
5	4	▪	-1	
6	4	▪	0	
7	4	▬▬▬▬	-4	
8	4	▬	1	
9	4	▬	6	
10	4	▪	2	
11	10	▬▬▬▬	13	
12	1	▬▬▬	-27	
13	1	▬	0	
14	1	▬▬	0	
15	1	▬▬	-2	
16	1	▬	2	
17	1	▪	-2	
18	1	▬	0	
19	1	▬▬	0	
20	8	▬	-2	



Terry, Session 19, Continued...

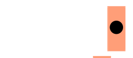
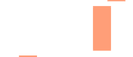
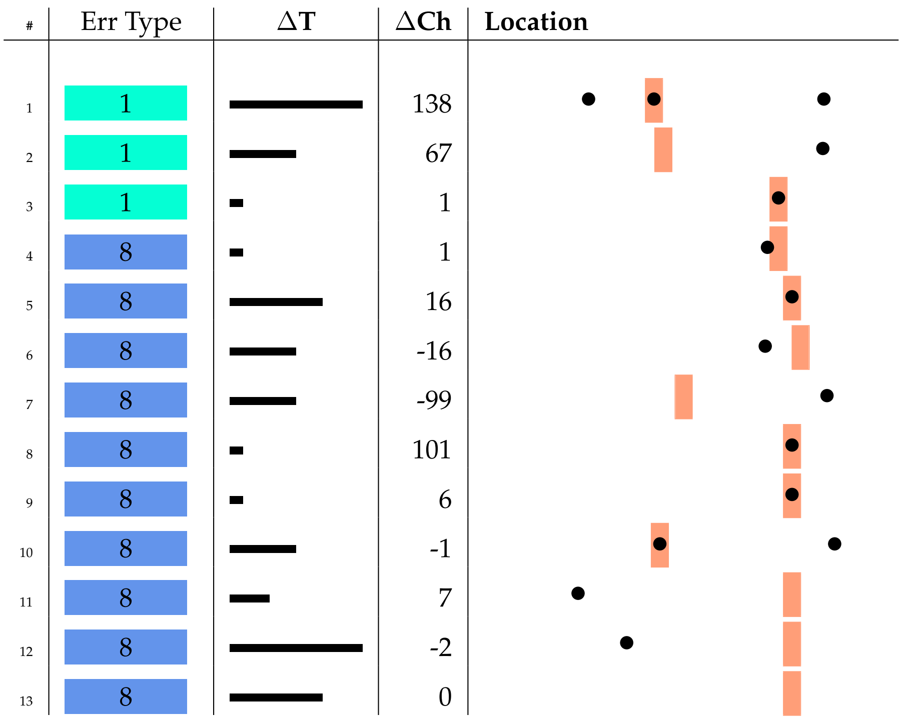
#	Err Type	ΔT	ΔCh	Location
21	12	▬	-1	
22	12	▬▬▬▬▬	0	
23	⋆	▬▬▬▬	-29	
24	⋆	▬▬▬▬▬▬▬	-3	

Table A.6: Deborah, Session 1

Project:	book-exercise	**File:**	Book.java
Duration:	11m19s	**EQ:**	0.79
Date:	Wednesday, October 22nd, 2003 11:08am		



#	Err Type	ΔT	ΔCh	**Location**
1	1		138	
2	1		67	
3	1		1	
4	8		1	
5	8		16	
6	8		-16	
7	8		-99	
8	8		101	
9	8		6	
10	8		-1	
11	8		7	
12	8		-2	
13	8		0	

Table A.7: Deborah, Session 25

Project:	CourceMarks	**File:**	Student.java
Duration:	30m35s	**EQ:**	0.64
Date:	Monday, November 10th, 2003 11:13am		

#	Err Type	ΔT	ΔCh	**Location**
1	14	▬▬▬▬▬	1	
2	6	▬▬▬	2	
3	6	▬▬▬▬▬	-9	
4	6	▬▬▬	0	
5	99	▬▬▬▬▬	13	
6	99	▬▬▬▬▬▬▬▬▬▬	-15	
7	6	▬	1	
8	6	▬▬▬▬▬▬▬	0	
9	6	▬	0	
10	6	▬▬▬▬▬▬▬▬▬▬	6	
11	6	▬▬▬	10	
12	6	▬▬▬▬▬	-10	
13	⋆	▬▬▬▬▬▬▬▬▬▬	-164	
14	7	▬▬▬▬▬▬▬▬▬▬	-19	

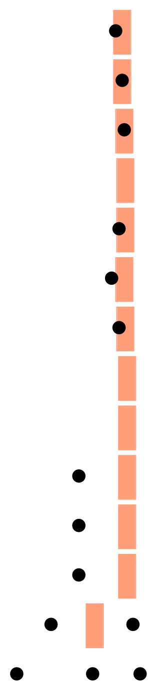


Table A.8: Catherine, Session 11

Project:	address-book-v1t	**File:**	ContactDetails.java
Duration:	17m40s	**EQ:**	0.03
Date:	Thursday, March 3rd, 2005 10:49am		

#	Err Type	ΔT	ΔCh	**Location**
1	🟧 12	▬	0	🟧●
2	🟦 8	▬▬▬▬	17	● 🟧
3	⬜ 13	▬▬	-1	● 🟧
4	⋆	▬	-8	🟧●
5	🟩 4	▬▬	26	●
6	⋆	▬▬	2	🟧●
7	⋆	▬▬▬	28	●
8	🟧 12	▬	-8	● ●
9	⋆	▬▬▬▬	11	🟧●
10	⋆	▬	-56	●
11	🟧 12	▬▬▬▬	38	● ●
12	⋆	▬	0	🟧●
13	⋆	▬▬▬▬	-1	●

Table A.9: Gregory, Session 2

Project:	notebook1	**File:**	Notebook.java
Duration:	23m26s	**EQ:**	0.02
Date:	Wednesday, October 29th, 2003 11:24am		

#	Err Type	ΔT	ΔCh	**Location**
1	4	▬▬▬▬	140	●
2	2	▬	-8	🟧●
3	⋆	▬	2	🟧●
4	8	▬▬▬▬	18	●
5	⋆	▬	2	●🟧
6	⋆	▬▬	-19	●
7	⋆	▬▬▬▬	-17	●
8	⋆	▬▬▬	11	●
9	24	▬▬▬▬	67	●

Table A.10: Jack, Session 3

Project:	assign6	**File:**	Game.java
Duration:	22m52s	**EQ:**	0.0
Date:	Tuesday, January 20th, 2004 3:25pm		

#	Err Type	ΔT	ΔCh	**Location**
1	⋆	▬▬▬▬▬▬	68	●
2	⋆	▬▬▬	15	●
3	⋆	▬▬▬▬▬▬	6	● ●
4	⋆	▬▬▬▬▬▬	22	●
5	⋆	▬▬▬▬▬▬	28	●
6	15	▬▬▬▬▬▬	63	●
7	⋆	▬▬	7	●
8	⋆	▬▬▬▬▬▬	7	●

Table A.11: Hermione, Session 10

Project:	calculator-gui	**File:**	CalcEngine.java
Duration:	26m44s	**EQ:**	0.04
Date:	Tuesday, December 9th, 2003 3:18pm		

#	Err Type	ΔT	ΔCh	**Location**
1	⋆	▪	1	🟧● ●
2	⋆	▬▬▬▬	345	● ● ● ● ●
3	⋆	▬▬▬	0	
4	⋆	▬▬	20	● ● ●
5	⋆	▬▬▬▬	-88	● ●
6	⋆	▬▬	0	
7	⋆	▬▬▬	76	● ● ● ● ●
8	⋆	▬▬	16	●
9	⋆	▬▬▬▬	63	● ● ● ● ● ●
10	⋆	▬▬▬	0	
11	🟩 5	▬▬▬	65	● ● ● ● ●
12	⋆	▪	3	🟧● ●
13	⋆	▬▬▬	0	
14	⋆	▬	48	●
15	🟥 2	▬▬	4	● ● ●
16	🟥 2	▪	0	🟧
17	⋆	▪	-16	🟧● ●
18	⋆	▪	0	
19	⋆	▬▬	-4	● ●
20	⋆	▪	0	

Hermione, Session 10, Continued...

#	Err Type	ΔT	ΔCh	**Location**
21	⋆	▬▬▬	15	●

Table A.12: Patricia, Session 1

Project:	notebook1	**File:**	Notebook.java
Duration:	35m39s	**EQ:**	0.22
Date:	Tuesday, October 28th, 2003 1:11pm		

#	Err Type	ΔT	ΔCh	**Location**
1	1	▬▬▬▬▬▬	252	●
2	⋆	▬▬	-1	●
3	1	▬▬▬▬▬▬	187	●
4	1	▬	-1	●
5	3	▬	2	●
6	2	▬▬	1	●
7	14	▬▬▬▬	-10	●
8	2	▬▬▬▬	2	●
9	⋆	▬▬	2	●
10	⋆	▬▬▬▬	-1	●
11	3	▬▬▬▬▬▬	45	● ●
12	1	▬▬	0	●
13	23	▬	1	●
14	⋆	▬▬▬▬	2	● ●
15	⋆	▬▬	48	●
16	⋆	▬▬	96	●
17	3	▬▬▬▬	18	●
18	3	▬▬	1	●
19	⋆	▬▬	2	●
20	⋆	▬▬▬▬▬▬	16	●

Patricia, Session 1, Continued...

#	Err Type	ΔT	ΔCh	**Location**
21	⋆	▬▬▬▬▬▬	63	● ● ●
22	⋆	▬▬▬▬	2	●

Table A.13: Carlos, Session 2

Project:	auction	**File:**	Auction.java
Duration:	24m43s	**EQ:**	0.17
Date:	Tuesday, October 28th, 2003 11:26am		

#	Err Type	ΔT	ΔCh	Location
1	⋆	▬▬▬▬▬▬▬	5	
2	⋆	▬▬▬▬▬▬▬▬▬▬	470	
3	3	▬▬▬▬▬▬▬▬▬▬	290	
4	1	▬▬▬	2	
5	3	▬	1	
6	12	▬▬▬	1	
7	⋆	▬▬▬	14	
8	2	▬▬▬▬▬▬▬▬▬▬	19	
9	2	▬▬▬	2	
10	⋆	▬	2	

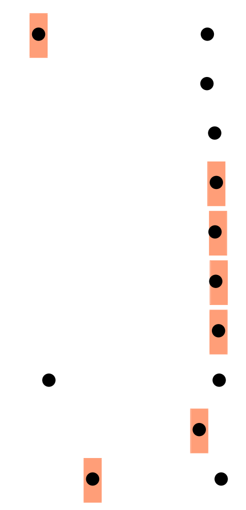
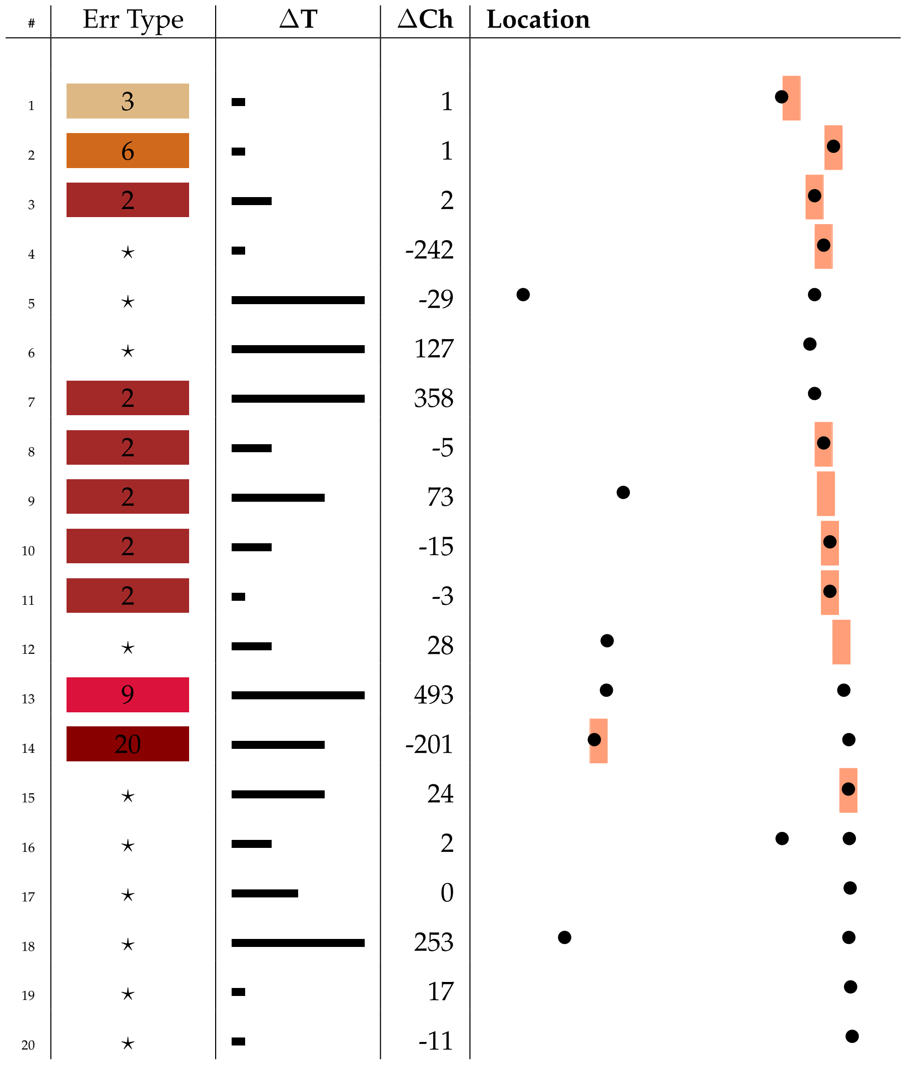


Table A.14: Bryan, Session 9

Project:	Coursework2	**File:**	Gui.java
Duration:	43m20s	**EQ:**	0.2
Date:	Thursday, February 17th, 2005 11:13am		



#	Err Type	ΔT	ΔCh	**Location**
1	3	▬	1	●
2	6	▬	1	●
3	2	▬▬▬	2	●
4	⋆	▬	-242	●
5	⋆	▬▬▬▬▬▬▬▬▬▬	-29	● ●
6	⋆	▬▬▬▬▬▬▬▬▬▬	127	●
7	2	▬▬▬▬▬▬▬▬▬▬	358	●
8	2	▬▬▬	-5	●
9	2	▬▬▬▬▬▬▬	73	●
10	2	▬▬▬	-15	●
11	2	▬	-3	●
12	⋆	▬▬▬	28	●
13	9	▬▬▬▬▬▬▬▬▬▬	493	● ●
14	20	▬▬▬▬▬▬▬	-201	● ●
15	⋆	▬▬▬▬▬▬▬	24	●
16	⋆	▬▬▬	2	● ●
17	⋆	▬▬▬▬▬	0	●
18	⋆	▬▬▬▬▬▬▬▬▬▬	253	● ●
19	⋆	▬	17	●
20	⋆	▬	-11	●

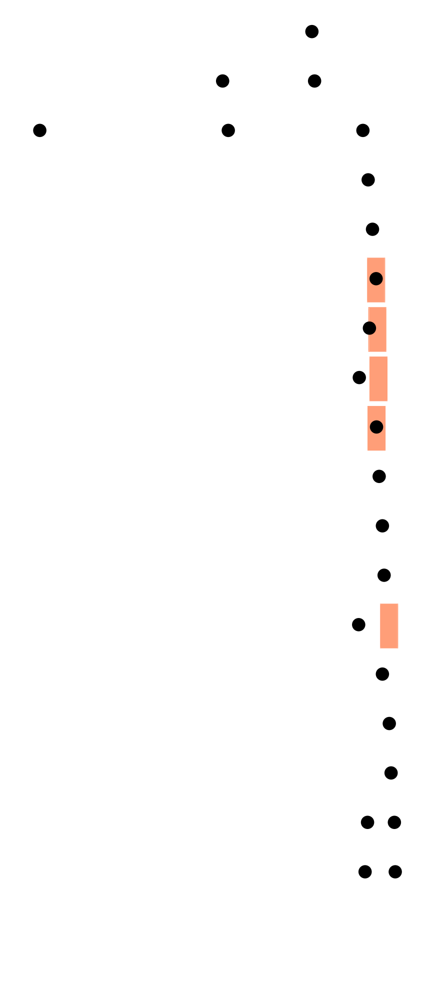
Bryan, Session 9, Continued...

#	Err Type	ΔT	ΔCh	Location
21	🟥 20	▬▬	1	●
22	★	▬▬	13	● ● 🟧
23	★	▬▬	-20	●
24	🟥 20	▬▬	-5	● ● ●
25	🟥 20	▪	0	🟧
26	★	▪	5	🟧●
27	★	▪	0	
28	★	▬▬▬	299	●
29	🟪 16	▬▬▬▬	-306	● ●
30	🟪 16	▪	0	🟧●
31	🟧 3	▬▬▬	7	🟧●
32	🟪 16	▪	-8	🟧●
33	★	▪	2	🟧●

Table A.15: Gloria, Session 16

Project:	HorseAssessment	**File:**	Race.java
Duration:	1h23m31s	**EQ:**	0.02
Date:	Wednesday, December 8th, 2004 12:28pm		

#	Err Type	ΔT	ΔCh	Location
1	⋆	▬▬▬▬▬▬▬	27	
2	⋆	▬▬▬▬▬▬▬▬▬▬	58	
3	⋆	▬▬▬▬▬▬▬▬▬▬	457	
4	⋆	▬▬▬▬▬▬▬▬▬▬	49	
5	1	▬▬▬▬▬▬▬▬▬▬	324	
6	12	▬▬▬	2	
7	18	▬▬▬▬▬▬▬	48	
8	2	▬▬▬	10	
9	⋆	▬	1	
10	⋆	▬▬▬▬▬▬▬▬▬▬	347	
11	⋆	▬▬▬▬▬▬▬▬▬▬	67	
12	12	▬▬▬▬▬▬▬▬▬▬	496	
13	⋆	▬▬▬▬▬▬▬	19	
14	⋆	▬▬▬▬▬▬▬▬▬▬	6	
15	⋆	▬▬▬▬▬▬▬▬▬▬	163	
16	⋆	▬▬▬▬▬▬▬▬▬▬	270	
17	⋆	▬▬▬▬▬▬▬	171	
18	⋆	▬▬▬▬▬▬▬▬▬▬	206	
19	⋆	▬▬▬▬▬▬▬▬▬▬	0	
20	⋆	▬▬▬▬▬	0	

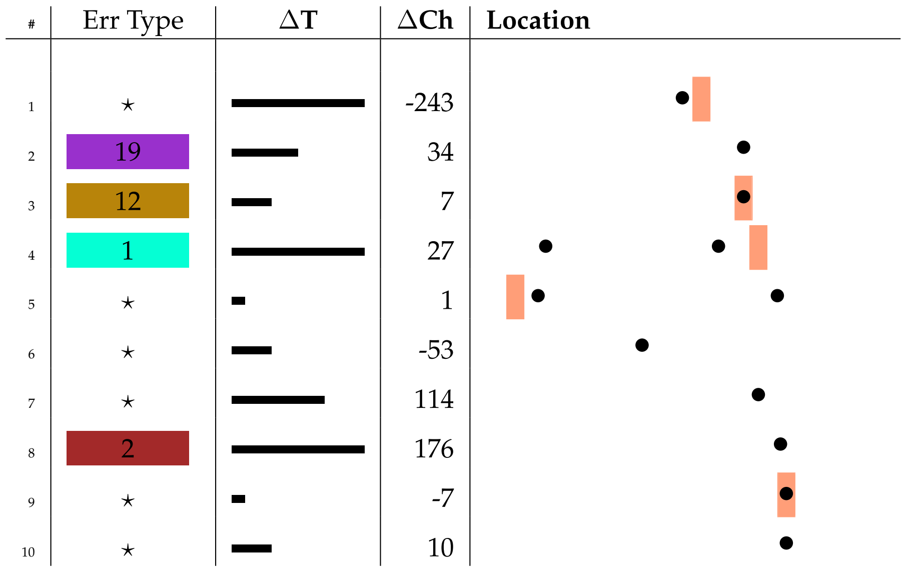


Gloria, Session 16, Continued...

#	Err Type	ΔT	ΔCh	**Location**
21	⋆	▬▬▬▬▬▬▬	0	
22	⋆	▬▬▬▬▬▬▬▬▬▬	-8	●
23	⋆	▬▬▬	-8	●
24	2	▬▬▬▬▬▬▬▬▬▬	1106	●
25	⋆	▬▬▬	2	●
26	3	▬▬▬▬▬▬▬▬▬▬	143	● ●
27	⋆	▬	1	●
28	⋆	▬▬▬▬▬▬▬	8	●
29	⋆	▬▬▬▬▬▬▬▬▬▬	436	● ●
30	⋆	▬▬▬▬▬▬▬▬▬▬	180	●

Table A.16: Victor, Session 3

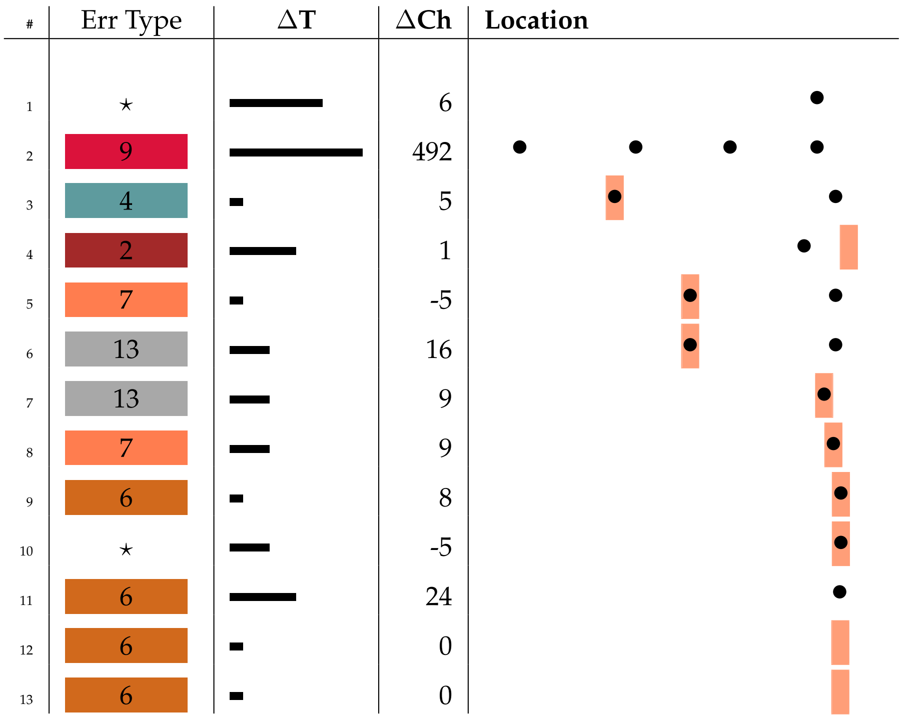
Project:	week10	**File:**	TokenizerTest.java
Duration:	12m59s	**EQ:**	0.06
Date:	Tuesday, November 25th, 2003 1:03pm		



#	Err Type	ΔT	ΔCh	**Location**
1	⋆		-243	
2	19		34	
3	12		7	
4	1		27	
5	⋆		1	
6	⋆		-53	
7	⋆		114	
8	2		176	
9	⋆		-7	
10	⋆		10	

Table A.17: Joe, Session 2

Project:	assign8-option1	**File:**	LineFill.java
Duration:	25m48s	**EQ:**	0.37
Date:	Tuesday, March 16th, 2004 2:36pm		



#	Err Type	ΔT	ΔCh	**Location**
1	⋆		6	
2	9		492	
3	4		5	
4	2		1	
5	7		-5	
6	13		16	
7	13		9	
8	7		9	
9	6		8	
10	⋆		-5	
11	6		24	
12	6		0	
13	6		0	

Table A.18: Joe, Session 5

Project:	assign8-option1	**File:**	LineFill.java
Duration:	30m43s	**EQ:**	0.24
Date:	Friday, March 19th, 2004 12:52pm		

#	Err Type	ΔT	ΔCh	**Location**
1	4	▪	-1	🟧● ● ●
2	4	▬▬▬	-3	🟧● ●
3	4	▬	4	🟧●
4	4	▪	5	🟧●
5	4	▬▬	-3	🟧● ●
6	6	▬▬	6	🟧● ●
7	5	▬▬	21	● 🟧 ●
8	6	▬	-1	🟧 ●
9	⋆	▬▬	-5	🟧● ● ● ●
10	⋆	▬▬▬▬	-183	● ● ● ●
11	8	▬▬▬	1333	● ● ● ●
12	⋆	▬	-1149	🟧 ●
13	⋆	▪	0	
14	2	▬▬▬	-91	●
15	⋆	▪	-36	🟧● ●
16	⋆	▬▬▬▬	0	
17	⋆	▬▬	26	●
18	⋆	▬▬▬	4	● ●
19	⋆	▬▬▬▬	0	
20	⋆	▬▬▬▬	-85	● ● ● ● ●

Table A.19: Eugene, Session 2

Project:	notebook1	**File:**	Notebook.java
Duration:	23m53s	**EQ:**	0.57
Date:	Tuesday, October 28th, 2003 8:28am		

#	Err Type	ΔT	ΔCh	**Location**
1	1	▬▬▬▬	8	●
2	⋆	▬	1	● ●
3	1	▬▬▬▬	117	●
4	10	▬	1	●
5	10	▪	1	●
6	10	▬	4	●
7	4	▬▬▬▬	-3	●
8	4	▪	-1	●
9	4	▪	1	●
10	1	▪	-1	●
11	1	▬	0	●
12	1	▪	0	●
13	10	▬	1	●
14	1	▬	-4	●
15	3	▬	1	●
16	2	▬	1	●
17	2	▬	-1	●
18	2	▬▬	2	●
19	⋆	▪	1	● ●

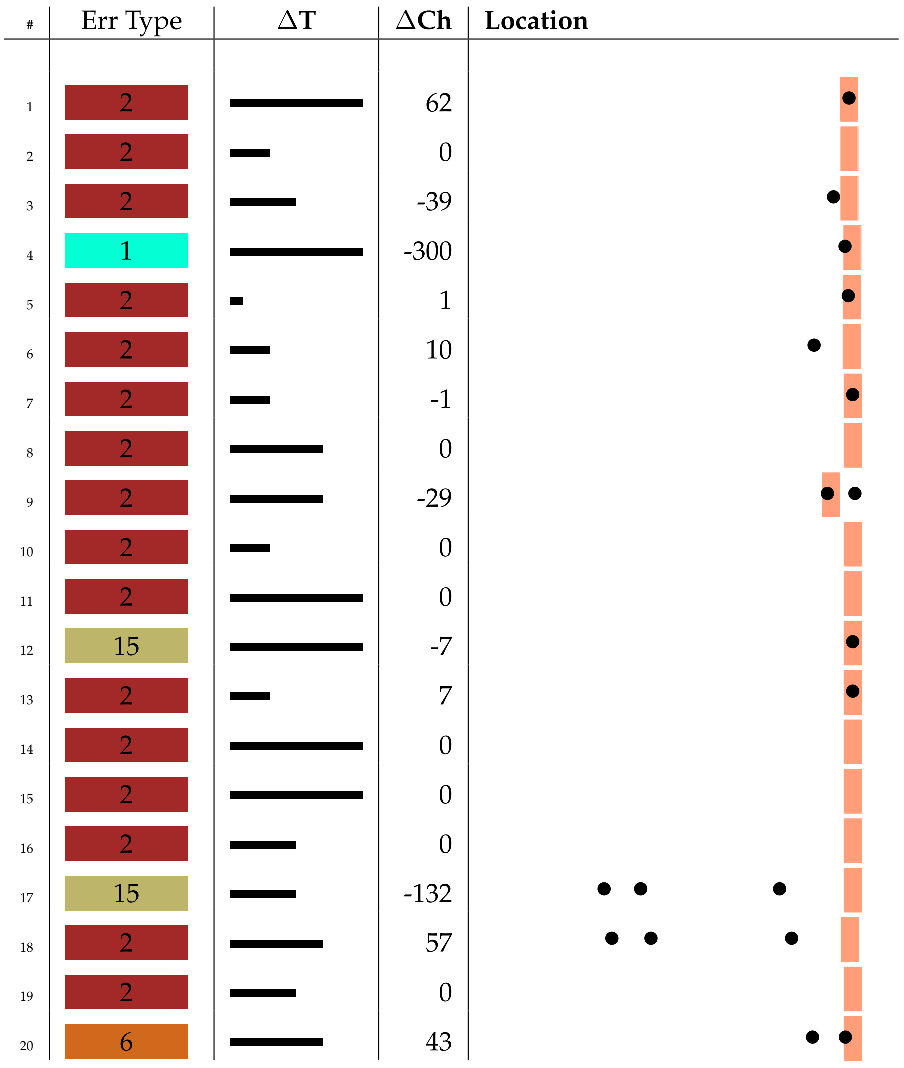
Table A.20: Justin, Session 5

Project:	address-book-v3t	**File:**	AddressBook.java
Duration:	15m15s	**EQ:**	0.68
Date:	Wednesday, February 11th, 2004 6:27am		

#	Err Type	ΔT	ΔCh	**Location**
1	3	▬▬▬▬▬▬▬▬▬▬	-6	●
2	3	▬▬▬▬▬▬▬	1	●
3	9	▬▬▬▬▬▬▬	14	●
4	9	▬▬▬▬▬▬▬	2	●
5	9	▬▬▬	-2	●
6	5	▬	2	●
7	5	▬▬▬	-1	●
8	59	▬▬▬	1	●
9	⋆	▬▬▬▬▬▬▬▬▬▬	69	●

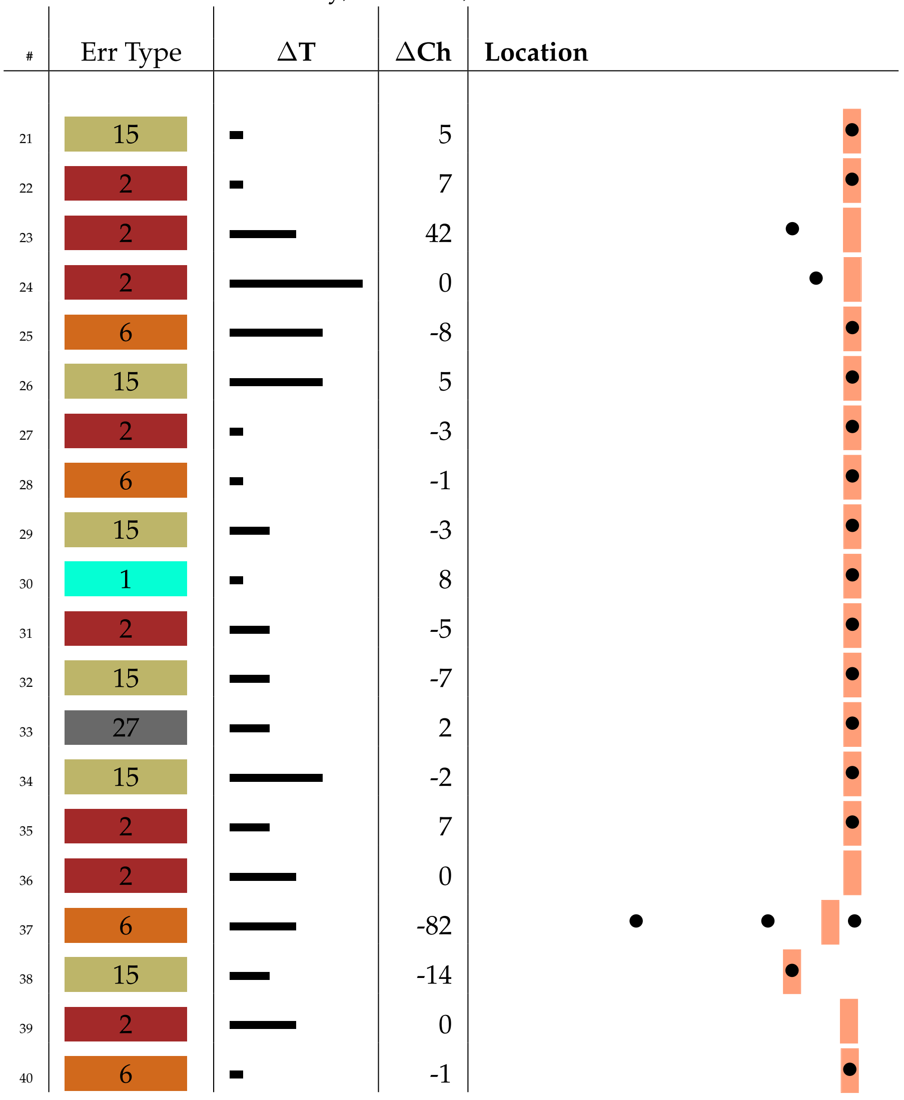
Table A.21: Terry, Session 3

Project:	assign6	**File:**	Game.java
Duration:	1h29m41s	**EQ:**	0.51
Date:	Monday, January 19th, 2004 6:01pm		



#	Err Type	ΔT	ΔCh	**Location**
1	2		62	
2	2		0	
3	2		-39	
4	1		-300	
5	2		1	
6	2		10	
7	2		-1	
8	2		0	
9	2		-29	
10	2		0	
11	2		0	
12	15		-7	
13	2		7	
14	2		0	
15	2		0	
16	2		0	
17	15		-132	
18	2		57	
19	2		0	
20	6		43	

Terry, Session 3, Continued...



#	Err Type	ΔT	ΔCh	**Location**
21	15		5	
22	2		7	
23	2		42	
24	2		0	
25	6		-8	
26	15		5	
27	2		-3	
28	6		-1	
29	15		-3	
30	1		8	
31	2		-5	
32	15		-7	
33	27		2	
34	15		-2	
35	2		7	
36	2		0	
37	6		-82	
38	15		-14	
39	2		0	
40	6		-1	

Terry, Session 3, Continued...

#	Err Type	ΔT	ΔCh	**Location**
41	🟩 14	▬	-30	🟧●
42	27	▬	0	🟧
43	🟧 6	▬	7	🟧●
44	⋆	▬	-33	🟧●
45	⋆	▬▬▬	0	
46	⋆	▬▬▬	0	
47	⋆	▬▬	0	
48	⋆	▬▬▬▬	0	
49	⋆	▬▬▬▬	31	●
50	⋆	▬▬▬▬	9	●
51	🟧 6	▬▬▬▬	9	●

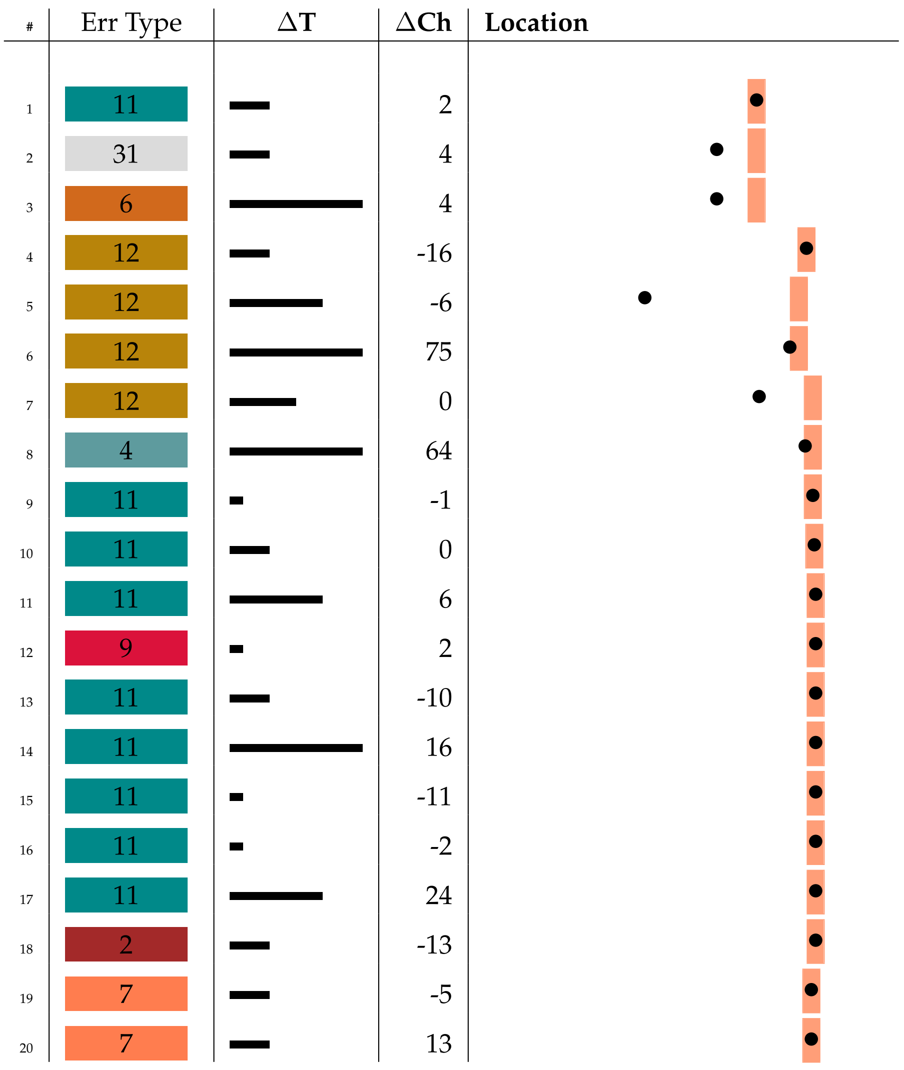
Table A.22: Roger, Session 2

Project:	Assesment 5	**File:**	Game.java
Duration:	3m11s	**EQ:**	0.23
Date:	Tuesday, January 20th, 2004 10:13am		

#	Err Type	ΔT	ΔCh	**Location**
1	6	▪	0	▨
2	2	▬▬▬▬	7	●
3	6	▬▬	19	●
4	6	▬▬	7	●
5	2	▪	7	●
6	6	▬▬	19	●
7	6	▪	32	●
8	⋆	▬▬	26	●
9	⋆	▬▬	-10	●
10	6	▬▬▬▬	-7	●
11	⋆	▪	7	●

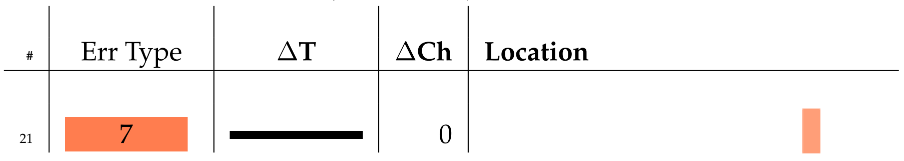
Table A.23: Cain, Session 12

Project:	Classmarks	**File:**	CourseMarks.java
Duration:	29m56s	**EQ:**	0.72
Date:	Wednesday, November 5th, 2003 4:25am		



#	Err Type	ΔT	ΔCh	Location
1	11		2	
2	31		4	
3	6		4	
4	12		-16	
5	12		-6	
6	12		75	
7	12		0	
8	4		64	
9	11		-1	
10	11		0	
11	11		6	
12	9		2	
13	11		-10	
14	11		16	
15	11		-11	
16	11		-2	
17	11		24	
18	2		-13	
19	7		-5	
20	7		13	

Cain, Session 12, Continued...



#	Err Type	ΔT	ΔCh	Location
21	7		0	

Table A.24: Karen, Session 2

Project:	notebook1	**File:**	Notebook.java
Duration:	22m33s	**EQ:**	0.6
Date:	Wednesday, October 29th, 2003 4:25am		

#	Err Type	ΔT	ΔCh	Location
1	1	▬▬▬▬▬▬▬	54	●
2	1	▬	1	●
3	4	▬▬▬▬▬	27	●
4	1	▬	0	●
5	1	▬▬▬▬▬▬▬▬▬▬	53	●
6	4	▬▬▬	17	●
7	1	▬▬▬▬▬▬▬	3	●
8	10	▬▬▬	2	●
9	10	▬	9	●
10	10	▬▬▬	-10	●
11	10	▬▬▬	13	●
12	10	▬▬▬▬▬	-12	●
13	10	▬	-1	●
14	12	▬▬▬▬▬	-5	●
15	12	▬▬▬	0	
16	3	▬▬▬	5	●
17	⋆	▬▬▬	-12	● ●
18	1	▬▬▬▬▬▬▬	-20	●
19	1	▬▬▬	1	●
20	1	▬▬▬	0	●

Karen, Session 2, Continued...

#	Err Type	ΔT	ΔCh	**Location**
21	1	▬	1	▮●
22	1	▬▬▬	0	▮
23	1	▬	0	▮
24	⋆	▬▬▬▬▬▬▬	-91	▮●
25	2	▬▬▬▬▬▬▬▬▬▬	90	●
26	14	▬▬▬	5	●▮
27	12	▬▬▬▬▬▬▬▬▬▬	1	●▮
28	⋆	▬▬▬	-5	● ▮ ●

Table A.25: Karen, Session 13

Project:	tokenizertest	**File:**	StringTokenizer.java
Duration:	7m08s	**EQ:**	0.46
Date:	Wednesday, November 26th, 2003 4:25am		

#	Err Type	ΔT	ΔCh	**Location**
1	19	▬▬▬▬▬	0	
2	2	▬▬▬	10	
3	3	▬▬▬	-6	
4	2	▬	2	
5	2	▬▬▬	-3	
6	5	▬▬▬▬▬▬▬	84	
7	5	▬▬▬	-4	
8	4	▬▬▬	-23	
9	16	▬▬▬▬▬	-93	
10	6	▬▬▬▬▬▬▬	8	
11	6	▬▬▬▬▬	0	
12	6	▬▬▬▬▬▬▬	-207	

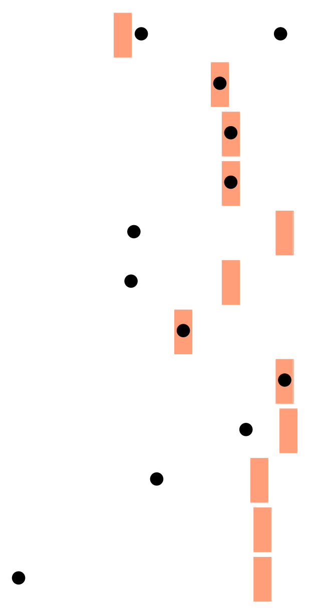
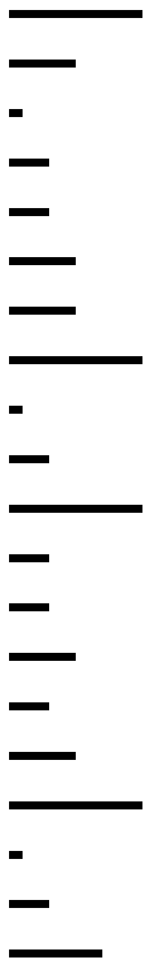
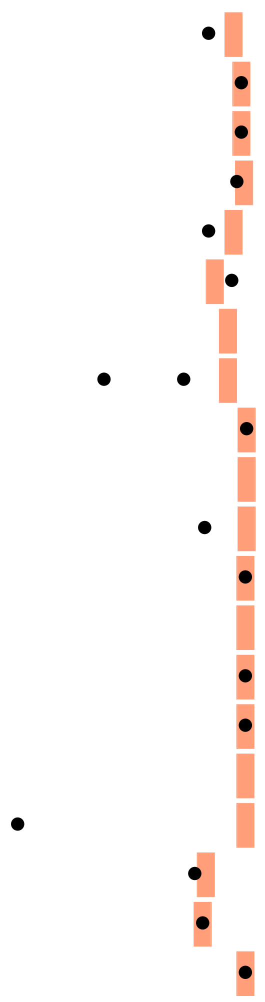


Table A.26: Stephanie, Session 4

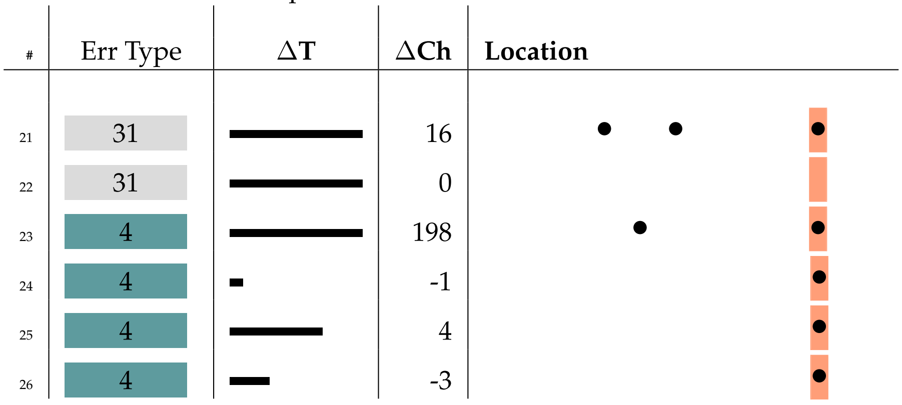
Project:	Hotels	**File:**	Hotel.java
Duration:	1h18m05s	**EQ:**	0.61
Date:	Friday, November 26th, 2004 6:31am		

#	Err Type	ΔT	ΔCh	**Location**
1	10		24	
2	3		6	
3	3		1	
4	7		1	
5	5		3	
6	5		21	
7	5		0	
8	31		-82	
9	10		-7	
10	10		0	
11	3		86	
12	31		1	
13	31		0	
14	1		1	
15	31		-1	
16	31		0	
17	1		23	
18	7		1	
19	31		7	
20	4		-13	





Stephanie, Session 4, Continued...



#	Err Type	ΔT	ΔCh	Location
21	31	▬▬▬▬▬▬▬▬▬▬	16	● ● ●
22	31	▬▬▬▬▬▬▬▬▬▬	0	
23	4	▬▬▬▬▬▬▬▬▬▬	198	● ●
24	4	▬	-1	●
25	4	▬▬▬▬▬▬▬	4	●
26	4	▬▬▬	-3	●

Table A.27: Eric, Session 3

Project:	randomness	**File:**	RandomTester.java
Duration:	8m05s	**EQ:**	0.66
Date:	Tuesday, November 11th, 2003 6:49am		

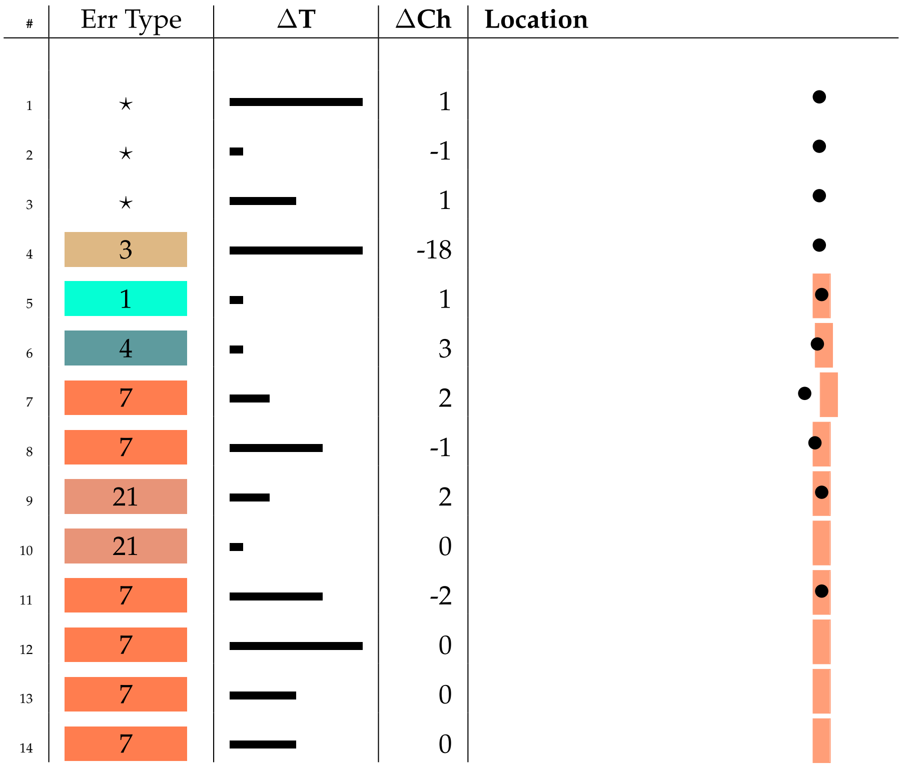
#	Err Type	ΔT	ΔCh	Location
1	1	▬▬▬▬	0	● ▮
2	1	▬	1	● ▮
3	10	▬▬	1	● ▮
4	10	▬	-7	●
5	10	▬▬▬▬	3	●
6	5	▬▬	-25	●
7	25	▬▬▬	7	●
8	1	▬	-1	●
9	25	▬	1	● ▮
10	1	▬▬▬▬	-1	●
11	19	▬	-1	● ▮
12	1	▬	-1	● ▮
13	1	▬▬▬	7	● ▮
14	25	▬▬	1	● ▮
15	19	▬▬▬	-5	●
16	19	▬	-1	●
17	1	▬	-2	●
18	1	▬	2	● ▮
19	25	▬▬	1	● ▮
20	6	▬▬	-2	●

Eric, Session 3, Continued...

#	Err Type	ΔT	ΔCh	Location
21	10	▬▬	-11	●
22	10	▬	7	●

Table A.28: Alfred, Session 1

Project:	weblog-analyzer	**File:**	LogAnalyzer.java
Duration:	33m53s	**EQ:**	0.47
Date:	Tuesday, November 16th, 2004 6:12am		

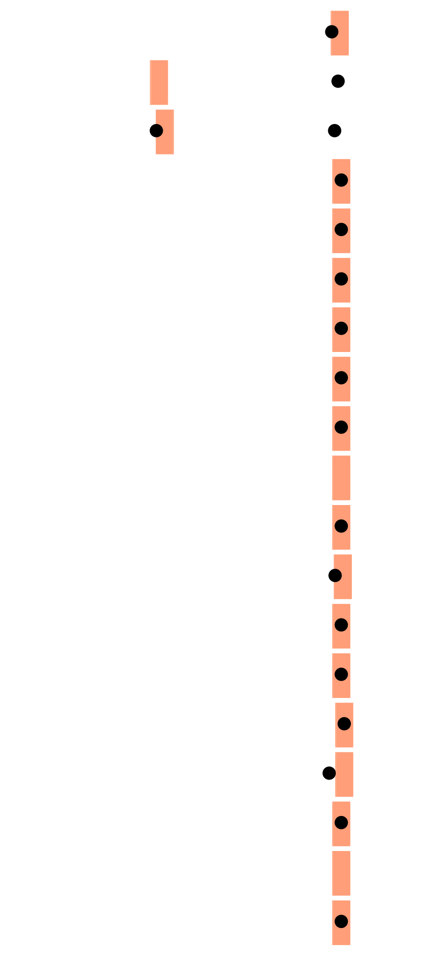


#	Err Type	ΔT	ΔCh	**Location**
1	⋆	▬▬▬▬	1	●
2	⋆	▪	-1	●
3	⋆	▬▬	1	●
4	3	▬▬▬▬	-18	●
5	1	▪	1	●
6	4	▪	3	●
7	7	▬	2	●
8	7	▬▬▬	-1	●
9	21	▬	2	●
10	21	▪	0	
11	7	▬▬▬	-2	●
12	7	▬▬▬▬	0	
13	7	▬▬	0	
14	7	▬▬	0	

Table A.29: Abel, Session 2

Project:	class-wk15	**File:**	PayInterestAccount.java
Duration:	20m24s	**EQ:**	0.58
Date:	Wednesday, January 21st, 2004 4:23am		

#	Err Type	ΔT	ΔCh	**Location**
1	15	▪	1	
2	15	▬▬▬▬	14	
3	11	▬▬▬▬▬▬	14	
4	1	▪	-8	
5	14	▬▬	-2	
6	47	▬▬▬▬▬▬	14	
7	14	▬▬▬	-14	
8	3	▬▬▬▬	23	
9	6	▬▬	1	
10	21	▬▬	0	
11	15	▬▬▬▬	-19	
12	15	▬▬▬▬	1	
13	11	▬▬▬▬	2	
14	1	▬▬	-8	
15	15	▬▬	-1	
16	1	▬▬▬▬	8	
17	31	▬▬	0	
18	31	▬▬▬	0	
19	⋆	▬▬▬▬	7	



Appendix B

Syntax error types

We recognise 85 syntax errors in our study; the majority of these exist in the "long tail" of the errors encountered by our students. Many of our students only encounter a few different *types* of errors during their explorations of the Java programming language.

In this appendix we have provided a complete listing of all the error types we recognise, the number of those errors we encountered in our study, and the percentage of erroneous compilations they represent. This information is presented in two tables; following those tables is the Scheme code that was used to categorise syntax errors; the function defined here consumed the error message generated by the compiler, and would categorise the error accordingly. Our students were working with the Sun Java compiler distributed with version 1.4.2 of the SDK for the duration of our study.

Table B.1: Error types recognised in our study.

Index	Abbrev.	#	%
1	semicolon	1973	10
2	unknown-variable	3297	16.7
3	bracket-expected	2034	10.3
4	illegal-start-of-expression	997	5.0
5	unknown-class	922	4.6
6	unknown-method	2005	10.1
7	incompatible-types	803	4.0
8	class-or-interface-expected	454	2.3
9	identifier-expected	548	2.7
10	.class-expected	349	1.7
11	not-a-statement	469	2.3
12	missing-return	659	3.3
13	op-application-error	175	0.8
14	private-access-violation	689	3.5
15	method-application-error	692	3.5
16	unknown-constructor	370	1.8
17	illegal-start-of-type	134	0.6
18	possible-uninit-variable	131	0.6
19	return-type-required	146	0.7
20	previously-defined-variable	186	0.9
21	unexpected-type	166	0.8
22	unreachable-statement	92	0.4
23	else-without-if	129	0.6
24	package-not-exist	116	0.5
25	missing-body-or-abstract	78	0.3
26	unclosed-comment	16	0.0
27	method-reference-from-static-context	211	1.0
28	file-io	56	0.2
29	no-return-void-method	64	0.3
30	dereferencing-error	146	0.7
31	loss-of-precision	145	0.7
32	empty-character-literal	4	0.0
33	unclosed-character-literal	18	0.0
34	inconvertible-types	96	0.4
35	illegal-escape-character	15	0.0
36	protected-access-violation	4	0.0
37	type-mismatch	54	0.2
38	assign-to-final	38	0.1
39	class-public-in-file	24	0.1
40	bad-modifier-combination	10	0.0
41	illegal-character	71	0.3
42	array-dim-missing	16	0.0
43	cannot-access	46	0.2

Table B.2: Error types recognised in our study, continued

Index	**Abbrev.**	#	%
44	void-type	30	0.1
45	incomparable-types	23	0.1
46	modifier-static	9	0.9
47	super-first-statement	16	0.0
48	dot-expected	28	0.1
49	non-static-in-static-context	43	0.2
50	weaker-privs	80	0.4
51	abstract-no-instantiate	16	0.0
52	abstract-no-override	131	0.6
53	cannot-override-ret-type	41	0.2
54	unknown-symbol	87	0.4
55	declare-var-final	66	0.3
56	catch-without-try	11	0.0
57	try-without-catch	14	0.0
58	unreported-exception	207	1.0
59	unthrown-exception	50	0.2
60	break-outside	4	0.0
61	while-expected	12	0.0
62	class-expected	38	0.1
63	missing-ret-val	18	0.0
64	unclosed-string	14	0.0
65	not-enc-class	5	0.0
66	colon-expected	1	0.0
67	orphaned-case	4	0.0
68	duplicate-default-lab	2	0.0
69	ret-outside-method	11	0.0
70	repeated-modifier	1	0.0
71	ref-before-supertype	10	0.0
72	modifier-not-here	11	0.0
73	eq-expected	2	0.0
74	no-interface-expected	8	0.0
75	interface-expected	1	0.0
76	access-outside-pkg	3	0.0
77	illeg-fwd-reference	1	0.0
78	abstract-no-body	5	0.0
79	exception-already-caught	1	0.0
80	interface-no-body	1	0.0
81	semicolon-expected	1	0.0
82	unreachable-statement	0.0	0.0
83	abstract-not-reachable	1	0.0
84	illegal-initializer	1	0.0
85	int-too-large	4	0.0

B.1 Categorising errors

To categorise Java syntax errors into the distinct types listed here, we employed a series of regular expressions. These regular expressions would be run over the syntax error reported by the `javac` compiler, and ultimately reported to the student through the BlueJ interface. Using the extensions API, we were able to capture a copy of these messages, and store them in our database along with the source that generated the error.

This function requires PLT Scheme version 300 or greater, and assumes the `pregexp.ss` library is installed; this ships as a standard library with the PLT Scheme distribution.

```
(define get-error-type
    (lambda (str)
      (let ([check? (lambda (s)
                      (equal? s str))])
        (cond
          ;;Simple (atomic) matches

          ;;Expectations
          [(check? "';' expected") 'semicolon]

          [(check? "'class' or 'interface' expected")
           'class-or-interface-expected]
          [(check? "invalid method declaration; return type required")
           'return-type-required]
          [(check? "unexpected type") 'unexpected-type]
          [(check? "'.class' expected") '.class-expected]
          [(check? "empty character literal") 'empty-character-literal]

          ;;Illegal syntax
          [(check? "illegal start of expression")
           'illegal-start-of-expression]
```

```
[(check? "illegal start of type")
 'illegal-start-of-type]
[(check? "illegal escape character")
 'illegal-escape-character]
[(check? "unclosed character literal")
 'unclosed-character-literal]

;;Statements
[(check? "not a statement") 'not-a-statement]
[(check? "unreachable statement") 'unreachable-statement]
[(check? "missing return statement") 'missing-return]
[(check? "'else' without 'if'") 'else-without-if]
[(check?
  (string-append
   "cannot return a value from "
   "method whose result type is void"))
 'no-return-void-method]

;;Missing
[(check? "array dimension missing") 'array-dim-missing]
[(check? "unclosed comment") 'unclosed-comment]
[(check? "missing method body, or declare abstract")
 'missing-body-or-abstract]

;;types?
[(check? "inconvertible types") 'inconvertible-types]
[(check? "possible loss of precision") 'loss-of-precision]

;;Regular expression matches
[(ormap (lambda (sym)
          (pregexp-match
           (format "'~a' expected" sym) str))
         '("\\(" "\\)" "\\{" "\\}" "\\[" "\\]")) 'bracket-expected]

;;package resolution
[(pregexp-match "cannot resolve symbol - (.*?) (.*)" str)
 =>
 (lambda (m)
   (let ([type (cadr m)]
         [name (caddr m)])
     (cond
       [(equal? type "variable") 'unknown-variable]
       [(equal? type "method") 'unknown-method]
       [(equal? type "class") 'unknown-class]
       [(equal? type "constructor") 'unknown-constructor]
       [else
        (fprintf (current-error-port)
                 "~a" str)
        'unknown-unknown])))]

[(pregexp-match "cannot resolve symbol" str)
 'unknown-symbol]

[(pregexp-match "package (.*) does not exist" str)
 =>
 (lambda (m)
   (let ([package (cadr m)])
     'package-not-exist))]

;;typing
```

```
[(pregexp-match
  "incompatible types - found (.*) but expected (.*)" str)
 =>
 (lambda (m)
   (let ([found (cadr m)]
         [expected (caddr m)])
     'incompatible-types))]

[(pregexp-match
  "(.*) required, but (.*) found" str)
 =>
 (lambda (m)
   (let ([type-needed (cadr m)]
         [type-found (caddr m)])
     'type-mismatch))]

[(pregexp-match "illegal character.*" str)
 'illegal-character]

;;application
[(pregexp-match
  "operator (.*) cannot be applied to (.*)" str)
 =>
 (lambda (m)
   (let ([op (cadr m)]
         [rand (caddr m)])
     'op-application-error))]

[(pregexp-match
  "(.) in (.*) cannot be applied to (.*)" str)
 => (lambda (m)
      (let ([method (cadr m)]
            [class (caddr m)]
            [type (cadddr m)])
        'method-application-error))]

;;referencing/dereferencing?
;; method invocation?
[(pregexp-match "(.*) cannot be dereferenced" str)
 => (lambda (m)
      (let ([type (cadr m)])
        'dereferencing-error))]

;;possible uninit variable
[(pregexp-match
  "variable (.*) might not have been initialized" str)
 =>
 (lambda (m)
   (let ([var (cadr m)])
     'possible-uninit-variable))]

;;order of ... ? variable errors?
[(pregexp-match "(.*) is already defined in (.*)" str)
 => (lambda (m)
      (let ([var (cadr m)]
            [method (caddr m)])
        'previously-defined-variable))]

[(pregexp-match
  "cannot assign a value to final variable (.*)" str)
 => (lambda (m)
      (let ([var (cadr m)])
        'assign-to-final))]

;;Access permissions
[(pregexp-match "(.*) has (.*) access in (.*)" str)
 =>
 (lambda (m)
   (let ([method-or-var (cadr m)]
         [level (caddr m)]
         [class (cadddr m)])
     (string->symbol
      (format "~a-access-violation" level))))]

[(pregexp-match
  "non-static method (.*) cannot be.*" str)
 ;;; rest is " referenced from a static context"
 => (lambda (m)
      (let ([method (cadr m)])
        'method-reference-from-static-context))]

[(pregexp-match
  "illegal combination of modifiers: (.*) and (.*)" str)
 => (lambda (m)
      (let ([first (cadr m)]
            [second (caddr m)])
        'bad-modifier-combination))]

[(pregexp-match
  (string-append
    "class (.*) is public, should be "
    "declared in a file named (.*)") str)
  => (lambda (m)
       (let ([class-name (cadr m)]
             [file-name (caddr m)])
         'class-public-in-file))]

;;file i/o
[(pregexp-match "error while writing.*" str) 'file-io]

[(pregexp-match "cannot access (.*)" str) 'cannot-access]

[(pregexp-match "'void' type not allowed here" str)
 'void-type]

[(pregexp-match
  "incomparable types: (.*?) and (.*?)" str)
 'incomparable-types]

[(pregexp-match
  "modifier static not allowed here" str)
 'modifier-static]

[(pregexp-match
  (string-append
   "call to super must be first "
   "statement in constructor") str)
 'super-first-statement]

[(pregexp-match "'\\.' expected" str)
```

```
 'dot-expected]

[(pregexp-match
  (string-append
   "non-static variable (.*?) cannot be "
   "referenced from a static context") str)
 'non-static-in-static-context]

[(pregexp-match
  (string-append
   "(.*?) in (.*?) cannot (.*?) (.*?) in (.*?); "
   "attempting to assign weaker access "
   "privileges; was (.*?)") str)
 'weaker-privs]

[(pregexp-match
  "(.*?) is abstract; cannot be instantiated" str)
 'abstract-no-instantiate]

[(pregexp-match
  (string-append
   "(.*?) is not abstract and does not "
   "override abstract method (.*?) in (.*?)") str)
 'abstract-no-override]

[(pregexp-match
  (string-append
   "(.*?) in (.*?) cannot (.*?) (.*?) in (.*?); "
   "attempting to use incompatible return type") str)
 'cannot-override-ret-type]
```

```
[(pregexp-match
  (string-append
   "local variable (.*?) is accessed from within "
   "inner class; needs to be declared final") str)
 'declare-var-final]


[(pregexp-match "'catch' without 'try'" str)
 'catch-without-try]

[(pregexp-match "'try' without 'catch' or 'finally'" str)
 'try-without-catch]

[(pregexp-match
  (string-append
   "unreported exception (.*?) must be "
   "caught or declared to be thrown") str)
 'unreported-exception]

[(pregexp-match
  (string-append
   "exception (.*?) is never thrown in body "
   "of corresponding try statement") str)
 'unthrown-exception]

[(pregexp-match "break outside switch or loop" str)
 'break-outside]

[(pregexp-match "while expected" str)
```

```
 'while-expected]

[(pregexp-match "class expected" str)
 'class-expected]

[(pregexp-match "missing return value" str)
 'missing-ret-val]



[(pregexp-match "unclosed string literal" str)
 'unclosed-string]

[(pregexp-match "not an enclosing class: (.*?)" str)
 'not-enc-class]

[(pregexp-match ": expected" str)
 'colon-expected]

[(pregexp-match "orphaned case" str)
 'orphaned-case]

[(pregexp-match "duplicate default label" str)
 'duplicate-default-lab]


[(pregexp-match "return outside method" str)
 'ret-outside-method]

[(pregexp-match "repeated modifier" str)
```

```
 'repeated-modifier]

[(pregexp-match
  (string-append
   "cannot reference (.*?) before supertype "
   "constructor has been called") str)
 'ref-before-supertype]

[(pregexp-match "modifier (.*?) not allowed here" str)
 'modifier-not-here]

[(pregexp-match "= expected" str)
 'eq-expected]



[(pregexp-match "no interface expected here" str)
 'no-interface-expected]

[(pregexp-match "interface expected here" str)
 'interface-expected]

[(pregexp-match
  (string-append
   "(.*?) is not (.*?) in (.*?); cannot be "
   "accessed from outside package") str)
 'access-outside-pkg]



[(pregexp-match "illegal forward reference" str)
 'illeg-fwd-reference]
```

```
[(pregexp-match
  "abstract methods cannot have a body" str)
 'abstract-no-body]

[(pregexp-match "interface methods cannot have body" str)
 'interface-no-body]

[(pregexp-match "exception (.*?) has already been caught" str)
 'exception-already-caught]

[(pregexp-match "identifier(.*)expected" str)
 'identifier-expected]

[(pregexp-match "';' expected" str)
 'semicolon-expected]

[(pregexp-match "unreachable statement.*" str)
 'unreachable-statement]

[(pregexp-match
  (string-append
   "abstract method (.*?) "
   "cannot be accessed directly") str)
 'abstract-not-reachable]

[(pregexp-match "illegal initializer for .*?)" str)
 'illegal-initializer]

[(pregexp-match "integer number too large" str)
```

```
 'int-too-large]


[else
 (fprintf (current-error-port) "~a~n" str)
 'unknown]
))))
```

Appendix C

Calculating the error quotient

The error quotient algorithm is presented in Figure 4.4. While a good deal more code is involved in calculating the error quotient than presented here—code for extracting data from the database, organizing it, etc.—the `calc-score` function performs the critical calculations on a session. Given the line numbers where errors occurred, the lines that were touched, the errors that occurred, and the STRING-EDIT values for each of those lines, it calculates the error quotient score for a given session.

As described previously, we began with one set of parameters for this algorithm, and used a semi-exhaustive search to find a better set of parameters in the surrounding space. Those are now the default parameter set employed by the `calc-score` function.

```
  (define TOUCHED-MULTIPLIER 1.0)
  (define ELINE-RANGE '(-3 -2 -1 0 1 2 3))
  (define ELINE-PENALTY 0)
  (define ETYPE-SAME-PENALTY 11)
  (define ETYPE-DIFF-PENALTY 8)
```

```
(define (calc-score elines toucheds errors string-edits)
  (let ([score 0]
        [scores '()])
    (define (average lon)
      (/ (apply + lon) (length lon)))

    (define (std-dev lon)
      (let ([avg (average lon)])
        (expt (/ (apply + (map
                           (lambda (n) (expt (- n avg) 2))
                           lon))
                 (length lon)) .5)))

    (foreach ([i (map sub1 (iota (length elines)))]
              [eline elines]
              [touched toucheds]
              [errorp errors]
              [se string-edits])
      ;; Don't run a comparison on the first element
      ;; in these lists; no sense in doing that.
      (unless (zero? i)
        (let ([prev-eline (list-ref elines (sub1 i))]
              [prev-touched (list-ref toucheds (sub1 i))]
              [prev-errorp (list-ref errors (sub1 i))])
          ;; Don't do anything if they managed to
          ;; eliminate syntax errors.
          (unless (or
                   (zero? (car errorp))
                   (zero? (cdr errorp)))
            (let ([looked-up
                   (lookup-score
                    (cons eline prev-eline)
                    (cons touched prev-touched)
                    (cons errorp prev-errorp)
```

```
                          se )])
              (set! score (+ score looked-up))
              (gather scores score))

            ))))

    (let* ([SCORE-CONSTANT
            (* (sub1 (length elines))
               (+ TOUCHED-MULTIPLIER ELINE-PENALTY
                  (max ETYPE-SAME-PENALTY ETYPE-DIFF-PENALTY)))]
           [final-score
            (round-to (exact->inexact
                       (/ score SCORE-CONSTANT)) 2)])
      final-score
    )))
```

Appendix D

Compilation chains

Students who make many successive syntax errors have high error quotient scores. Effectively, this means we are interested in the sequence of compilations, the kind of error that results (if any) from the compilation, and how often it repeats. This led us to be interested in a number of questions regarding the sequencing of correct and erroneous compilations in student sesions. We ended up exploring two specific questions: what typically happened after a given compilation—did the same error type repeat, or did the students move on in some way? And second, we were curious which errors tended to repeat the most. For example, we suspected that `';' expected` errors were handled more quickly than bracketing errors—but our qualitative analysis was not appropriate for this kind of question.

We address both of these questions, briefly, here.

D.1 What came next?

First, we were curious: when a student has a correct compilation, what comes next? Or, for that matter, when they are sitting on a missing semicolon error, do they typically see another missing semicolon error? The analyses in this section look at the error that most commonly follows a given compilation result for syntactically correct compilations and the five most prolific error types observed during the 2004-2005 academic year.

We believe that these tables might provide a starting point for an interesting analysis of specific error types within our data, and how they are delt with. This might be a productive strategy for developing a complete catalogue of how students deal with different types of syntax error within the Java programming language.

D.1.1 Syntactically correct compilations (Table D.1.1)

There were 7429 syntactically correct compilations recorded in the 2004-2005 academic year from students in our population. 64% of those compilations were followed by another syntactically correct compilation; the remaining 46% were followed by one or more erroneous compilations. The most common types that students encountered after successfully compiling (and then editing) their code included `unknown symbol - variable` errors and `';' expected`.

Table D.1: Syntactically correct code and what follows it.

Next error	# Occurrences	%
0	4817	64
2	487	6
1	359	4
3	348	4
6	213	2
4	150	2
7	103	1
5	88	1
14	79	1
12	78	1

D.1.2 Undefined symbol - variable (Table D.1.2)

When students received this error, 31% of the time it would be followed by another error of the same type. Another 31% of the time, they would manage to correct. Beyond that, this error class was typically followed by one or more errors of a variety of types.

D.1.3 Bracket expected (Table D.1.3)

Three kinds of bracketing errrors—those involving parentheses, square brackets, and squiggly brackets (sometimes called "braces" or "curley brackets") are rolled into this error type. This error repeated itself 35% of the time it was encountered, and students managed to correct it 19%

Table D.2: Unknown symbol - variable

Next error	# Occurrences	%
2	692	31
0	690	31
6	155	7
1	74	3
7	61	2
12	57	2
14	54	2
3	47	2
5	32	1
27	29	1
4	26	1
9	24	1

of the time they came upon it. The rest of the time it was encountered, this led to one or more other erroneous compilations.

Table D.3: Bracket expected

Next error	# Occurrences	%
3	454	35
0	250	19
2	107	8
1	63	4
6	60	4
5	56	4
4	37	2
12	36	2
7	28	2
11	21	1
14	19	1
15	15	1
8	14	1

D.1.4 ';' expected (Table D.1.4)

This error repeated 24% of the time it was encountered, and was fixed 23% of the time it was encountered. By some measure, this means the error is harder to fix than `unknown symbol - variable`, but easier to fix than most bracketing errors.

Table D.4: ';' expected

Next error	# Occurrences	%
1	307	24
0	296	23
2	148	11
3	87	7
6	52	4
5	45	3
7	44	3
4	34	2
11	25	2
14	21	1
12	14	1

D.1.5 Unknown symbol - method (Table D.1.5)

Like other "unknown symbol" errors, these are often due to typos and misspellings. Method symbol errors repeat with roughly the same frequency as their variable counterparts, but are corrected less frequently, perhaps implying that the errors are more often entwined with code that is significantly more "broken" in some way.

Table D.5: Unknown symbol - method

Next error	# Occurrences	%
6	363	32
0	319	28
2	116	1
1	42	3
7	37	3
3	25	2
12	22	1
15	21	1
5	21	1
11	13	1
14	12	1

D.2 How many repetitions?

When a student encountered a bracketing error, did they fix it on their first attempt? Second? Eigth? Were some errors more "persistent" than others? The answer to this small question is *yes*: not all kinds of error are created equal in the fingers of a desperate novice programmer.

D.2.1 Syntactically correct chains (Table D.2.1)

It was four times more common for students to have one or two successful compilations in a row than for them to immediately revert back to a syntax error. Sequences of three or four successful compilations in a row were almost as common as solitary, successful compilations. We are curious if this correlates with the size of the edits made during those successful sequences: did students edit a small amount of code? Were they cautious? Or, were they just good?

Table D.6: Syntactically correct chains

Chain length	# Occurrences
1	342
2	255
0	141
3	139
4	113
5	73
6	53
7	49
8	30
9	24
10	19

D.2.2 ';' expected (Table D.2.2)

It is five times more likely that a semicolon error will be transient (fixed or a new error generated) than repeat. We suspect this error is one students quickly learn to correct. However, it might be that the quick heuristic of simply adding a semicolon (as the compiler claims) fails students who have not yet mastered the language. As a result, they end up adding semicolons (as directed) when more fundamental problems actually exist in their code.

D.2.3 Unknown symbol - variable (Table D.2.3)

When students are faced with an unknown symbol error, it rarely repeats (we saw this previously). It is roughly four times more common that this error is transient (corrected or otherwise)

Table D.7: Semicolon error chains

Chain length	# Occurrences
0	679
1	103
2	25
4	10
3	9

than for students to wrestle with it over successive compilations. Like a missing semicolon error, we believe students quickly adapt to this error, and learn what to look for.

Table D.8: Unknown symbol chains

Chain length	# Occurrences
0	966
1	235
2	87
3	29
4	19
5	6
6	4

D.2.4 The trend

The trend for other error types continues; the "persistence ratio" for almost all error types (for which we have sufficient data) hovers around 3.5. This means that it is 3.5 times more likely that the error type will be transient—either be fixed, or the student will generate a different error type—than repeat; the previous section of this appendix details repetition rates for some error types to depth one. When errors do repeat, there does not appear to be any one error that is significantly more difficult to deal with than any other, based solely on how many times it repeats.

Certainly, more exploration might be waranted in this area. Are these factors a side-effect of the structure of Java's grammar? Or, are they useful tools that we might employ in characterising a student's behaviour? Perhaps more likely, these might be metrics we could employ in guiding an in-depth analysis of a student's session, or perhaps might be tools an instructor could use when surveying a student's session behavior.

Bibliography

[AC93] T. A. Angelo and K. P. Cross. *Classroom Assessment Techniques: A Handbook for College Teachers*. Jossey-Bass, 1993.

[Bae88] R. Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 356–366, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[BGDJ02] Lecia Jane Barker, Kathy Garvin-Doxas, and Michele Jackson. Defensive climate in the computer science classroom. In *SIGCSE '02: Proceedings of the 33rd SIGCSE technical symposium on Computer science education*, pages 43–47, New York, NY, USA, 2002. ACM Press.

[BM83] Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of basic programming statements. *Commun. ACM*, 26(9):677–679, 1983.

[Bro83] P. J. Brown. Error messages: the neglected area of the man/machine interface. *Commun. ACM*, 26(4):246–249, 1983.

[BS83] Jeffrey Bonar and Elliot Soloway. Uncovering principles of novice programming. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 10–13, New York, NY, USA, 1983. ACM Press.

[BU] Damiano Bolla and Ian Utting. The bluej extensions framework.

[BV80] J. S. Brown and K. VanLehn. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 4:379–426, 1980.

[CDGW76] D. D. Cowan, P. H. Dirksen, J. W. Graham, and J. W. Welch. Development of educational software using the dec pdp-11. In *SIGMINI '76: Proceedings of the ACM SIGMINI/SIGPLAN interface meeting on Programming systems in the small processor environment*, pages 109–112, New York, NY, USA, 1976. ACM Press.

[Cla01] Steven Clarke. Evaluating a new programming language. *Proceedings of the Psychology of Programming Interest Group 13*, pages 275–289, 2001.

[Dal02] Peter Dalgaard. *Introductory Statistics with R*. Springer, New York, NY, 2002.

[DL05] N. Denzin and Y. Lincoln, editors. *The SAGE Handbook of Qualitative Research*. SAGE Publications, April 2005.

[DLPQ04] Peter DePasquale, John A. N. Lee, and Manuel A. P&#233;rez-Qui&#241;ones. 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.

[Fla02] M. Flatt. Composable and compilable macros: You want it when, 2002.

[Fla05] Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR05-1-v300, PLT Scheme Inc., 2005. `http://www.plt-scheme.org/techreports/`.

[FLC$^+$05] 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.

[FT05] Sally Fincher and Josh Tenenberg. Making sense of card sorting data. *Expert Systems*, 22(3), July 2005.

[GF03] Kathryn E. Gray and Matthew Flatt. Professorj: a gradual introduction to java through language levels. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177, New York, NY, USA, 2003. ACM Press.

[GG84] D J. Gilmore and T R.G. Green. Comprehension and recall of miniature programs. *Int. J. Man-Mach. Stud.*, 21(1):31–48, 1984.

[GMD$^+$04] P. Gray, I. McLeod, S. Draper, M. Crease, and R. Thomas. A distributed usage monitoring system. In *Proc. CADUI 2004*, pages 121–132. CADUI, 2004.

[GO86] Leo Gugerty and Gary M. Olson. Comprehension differences in debugging by skilled and novice programmers. In *Empirical Studies of Programmers*, pages 13–27. 1986.

[GP96] T. R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[GPB91] T. R. G. Green, M. Petre, and R. K. E. Bellamy. Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. In *Empirical Studies of Programmers: Fourth Workshop*, Papers, pages 121–146, 1991.

[Gre89a] T. Green. Cognitive dimensions of notations. In A. Sutcliffe and L. Macaulay, editors, *People and Computers V, Proceedings of HCI 89*. CUP, 1989.

[Gre89b] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press.

[Gro] The PLT Group. Drscheme.

[GS67] Barney G. Glaser and Anselm L. Strauss. *Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine, June 1967.

[IBNS83] Barbara S. Isa, James M. Boyle, Alan S. Neal, and Roger M. Simons. A methodology for objectively evaluating error messages. In *CHI '83: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 68–71, New York, NY, USA, 1983. ACM Press.

[Jad] Matthew C. Jadud. Vincent. `http://www.cs.indiana.edu/vincent/4.0/`.

[Jad05] Matthew C. Jadud. A first look at novice compilation behavior. *Computer Science Education*, 15(1):25–40, 2005.

[KAM05] 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.

[Kan99] Gopal K. Kanji. *100 Statistical Tests*. Sage Publications, London, UK, 1999.

[KB05] Michael Kölling and David J. Barnes. *Objects first with Java: A practical introduction using BlueJ*. Prentice Hall, 2nd edition, 2005.

[KQPR03] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Journal of Computer Science Education*, 13(4), 2003.

[LAF$^+$04] 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.

[LC01] Thomas Lancaster and Fintan Culwin. Towards an error free plagarism 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.

[LD76] Charles R. Litecky and Gordon B. Davis. A study of errors, error-proneness, and error diagnosis in cobol. *Commun. ACM*, 19(1):33–38, 1976.

[Lyn67] W. C. Lynch. Description of a high capacity fast turnaround university computing center. In *Proceedings of the 1967 22nd national conference*, pages 273–288, New York, NY, USA, 1967. ACM Press.

[MAD$^+$01] 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.

[May81] Richard E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, 1981.

[McI01] Linda McIver. *Syntactic and Semantic Issues in Introductory Programming Education*. PhD thesis, Monash University, 2001.

[MDV86] Richard E. Mayer, Jennifer L. Dyck, and William Vilberg. Learning to program and learning to think: what's the connection? *Commun. ACM*, 29(7):605–610, 1986.

[MM67] P. G. Moulton and M. E. Muller. Ditran–a compiler emphasizing diagnostics. *Commun. ACM*, 10(1):45–52, 1967.

[MMNS83] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Commun. ACM*, 26(11):861–867, 1983.

[Mor70] Howard L. Morgan. Spelling correction in systems programs. *Commun. ACM*, 13(2):90–94, 1970.

[Mun69] Robert G. Munck. Meeting the computational requirements of the university - the brown university interactive language. In *Proceedings of the 1969 24th national conference*, pages 665–673, New York, NY, USA, 1969. ACM Press.

[Nor82] A. F. Norcio. Indentation, documentation and programmer comprehension. In *Proceedings of the 1982 conference on Human factors in computing systems*, pages 118–120, New York, NY, USA, 1982. ACM Press.

[PFea03] Marian Petre, Sally Fincher, and Josh Tenenberg et al. ”My Criterion is: Is it a Boolean?”: A card-sort elicitation of students' knowledge of programming constructs. Technical Report 6-03, Computing Laboratory, University of Kent, Canterbury, Kent, UK, June 2003.

[PHH$^+$89] D.N. Perkins, C. Hancock, R. Hobbs, F. Martin, and R. Simmons. Conditions of learning in novice programmers. *Studying the Novice Programmer*, 1988.

[PM86] 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.

[PM96] John F. Pane and Brad A. Myers. Usability issues in the design of novice programming systems. Technical Report CMU-CS-96-132, Carnegie Mellon University, 1996.

[PTG04] 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.

[Rei99] C. M. Reigeluth. *Instructional-Design Theores and Models: A New Paradigm of Instructional Theory*. Instructional Design Theories and Models. Lawrence Erlbaum Associates, Hillsdale, NJ, 1999.

[Ris86] R. Rist. Plans in programming: Definition, demonstration, and development. *Studying the Novice Programmer*, 1988.

[Rob93] Eric S. Roberts. Using c in cs1: evaluating the stanford experience. In *SIGCSE '93: Proceedings of the twenty-fourth SIGCSE technical symposium on Computer science education*, pages 117–121, New York, NY, USA, 1993. ACM Press.

[RW97] Vennila Ramalingam and Susan Wiedenbeck. An empirical study of novice program comprehension in the imperative and object-oriented styles. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 124–139, New York, NY, USA, 1997. ACM Press.

[SBE83] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: an empirical study. *Commun. ACM*, 26(11):853–860, 1983.

[SC90] A. Strauss and J. Corbin. *Basics of qualitative research: grounded theory procedures and techniques*. Sage Publications, Newbury Park, Calif., 1990.

[SFB$^+$05] 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.

[SGM$^+$67] 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.

[SJ81] O. Stromfors and L. Jones. The implementation and experiences of a structure-oriented text editor. In *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*, pages 22–27, 1981.

[SS86a] James C. Spohrer and Elliot Soloway. Analyzing the high frequency bugs in novice programs. In *Empirical Studies of Programmers*, pages 230–251. ACM, 1986.

[SS86b] James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, 1986.

[SS86c] 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.

[SS88] E. Soloway and James C. Spohrer. *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA, 1988.

[TR81] Tim Teitelbaum and Thomas Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, 1981.

[Tuf01] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, May 2001.

[Van04] 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.

[VJ93] K. VanLehn and R. M. Jones. *Foundations of knowledge acquisition: Cognitive models of complex learning*, chapter Learning by explaining examples to oneself: A computational model, pages 25– 82. Kluwer, Boston, 1993.

[WU01] 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.

[YBDZ97] Sherry Yang, Margaret M. Burnett, Elyon DeKoven, and Moshé M. Zloof. Representation design benchmarks: A design-time aid for vpl navigable static representations. *J. Vis. Lang. Comput.*, 8(5-6):563–599, 1997.

[Yin03] Robert K. Yin. Case study research. In *Design and Methods*, volume 5 of *Applied Social Research Methods*. SAGE, third edition, 2003.