RICE UNIVERSITY

A Framework for Building Pedagogic Java Programming Environments

by

Brian R. Stoler

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Approved, Thesis Committee:

Robert Cartwright, Chair
Professor of Computer Science

Keith Cooper
Professor of Computer Science

Dan Wallach
Assistant Professor of Computer Science

Houston, Texas

April, 2002

ABSTRACT

A Framework for Building Pedagogic Java Programming Environments

by

Brian R. Stoler

Java has become the dominant language for teaching introductory computer science at both the high school and college levels. Yet Java's development tools and syntax often distract beginning students from the programming concepts being taught. To combat this problem, we have implemented DrJava, a pedagogic programming environment, and devised a sequence of *language levels* for Java, which partition the language into pieces that can be more easily taught at one time.

DrJava is a lightweight, yet powerful, Java development environment suitable both for beginners and more advanced developers. The environment provides a simple interface based on a "read-eval-print loop" that enables a programmer to develop, test, and debug Java programs interactively and incrementally. DrJava is freely available under an open source license.

This thesis describes DrJava and an implementation framework and conceptual design for language levels in Java.

Acknowledgments

I would like to thank my adviser, Professor Robert "Corky" Cartwright, who knew how to strike a balance between giving astute advice and guidance and allowing me autonomy.

I also want to acknowledge the substantial contributions to this work made by others. Eric Allen co-managed the development of DrJava with me, and I am also grateful to Eric for introducing me to Extreme Programming and for his frequent discussions about the design of language levels. Also contributing substantially to DrJava were Charles Reis, Jonathan Bannet, Ben Vernot, and the students of COMP312: Principles of Program Engineering (spring 2002).

And finally, I would like to thank Elizabeth, for her encouragement and her helpful feedback.

Contents

Chapter 1
Introduction

As the Java programming language has rapidly gained in popularity in the marketplace, it has also become popular as a teaching language in colleges and high schools. In fact, most introductory computer science classes are now taught in Java [1].* Furthermore, starting in the 2003-2004 school year, the College Board Advanced Placement Examination in Computer Science will be in Java [2]. The reasons given by the College Board for changing to Java (from C++) are its safety, its relative simplicity, and its object orientation [1].

While these reasons make Java a better language for beginners than other mainstream languages, teaching object-oriented programming in Java to beginning students is still difficult. Not only is an instructor faced with the task of distilling challenging programming concepts, he also must explain the mechanics involved in writing, testing, and debugging Java programs. The complexity of Java's syntax and inadequate development tools create unnecessary burdens on beginning students. Since students can not learn everything at once, the more time they spend wrestling with the mechanics, the less time they spend learning the concepts. In Stephen Bloch's paper "Scheme and Java in the First Year," he documents his attempt to switch to Java for the first two semesters of computer science.

> ... After six weeks the majority of students felt that they were way behind the class, with no chance to catch up. On closer investigation, I realized

* I am not aware of a more recent systematic survey on the languages used in introductory programming courses.

> that many had never mastered the basic development-environment skills necessary to create, compile, and run the simplest program. I postponed my schedule of lecture topics for a week to review how to use the development environment, then for another week to review the most basic Java syntax (which they should have been practicing a month before). The year went on that way: every time I wanted to introduce an interesting new principle of programming, I had to spend precious lecture time on language syntax, deciphering cryptic error messages, and using the development environment. Students were dropping the course rapidly, and my own enthusiasm was waning. [3]

The basic problem Bloch identifies — which is the primary motivation for this work — is a lack of a suitable development environment for beginners, one that is not too complex, but one that is powerful enough to give students a useful tool to interact with their programs. To address this issue, we* have implemented DrJava, a development environment that gently introduces students to the mechanics of writing Java programs and leverages the student's understanding of the language itself to provide powerful support for developing programs. DrJava's most important contribution is its "interactions pane," which allows students to interactively evaluate arbitrary Java expressions and statements, with full access to the Java classes they have written.

DrJava addresses the complexity of tools for Java through use of the interactions pane and a simple, yet powerful editor. I will describe DrJava in detail in this thesis. But developing better tools does not address the issue of Java's syntactic complexity. To this end, I have designed a hierarchy of sublanguages of Java that can be added to DrJava to tame the complexity of the language. I have also implemented a Java lan-

* I have served as the lead developer on the DrJava project and have co-managed its implementation along with fellow graduate student Eric Allen. A number of other students contributed to DrJava's development. See the acknowledgements page for details.

guage processing framework, which will facilitate the implementation of the language levels design and of other Java language research projects.

Chapter 2 explains the philosophy that underlies this work. Chapter 3 puts this work in context by outlining related work and exploring the connections between DrJava and other similar projects. Chapter 4 documents DrJava, outlining its features and implementation. Chapter 5 details the Java language processing framework that was created to support language levels, and it also presents a language levels design. Finally, Chapter 6 discusses potential extensions of DrJava and language levels.

Chapter 2
Rationale and philosophy

The primary goal of this work is to provide tools to better support teaching beginners to program using Java. That goal reduces to three key priorities in supporting a programming interface: simplicity, interactivity, and a constant focus on the language. This chapter describes these priorities (and other secondary ones) and explains how they have impacted the design of DrJava.

2.1 Simplicity

The most important property of any tool for beginners is the ease with which a new user can learn to use it. The longer a student spends learning how to use the tool, the longer he is taken away from his true task of learning the material. (And if the tool frustrates him too much, he may simply give up on the material entirely!) One way to make a tool easier to use is to limit its functionality to the tasks the user actually needs — in other words, to make it as simple as possible while still fulfilling its function. This desire for simplicity has been the primary goal in the creation of DrJava, a goal achieved by a user-friendly interface, easy setup, and few program crashes and other bugs.

The focus on simplicity forced an important constraint on the development of DrJava: we could not build DrJava as a plug-in to an existing development environment.* To do this would be to cede control of the overall interface to some more

*There would have been a number of possible environments to plug in to, had this been an option. See Chapter 3 for more details.

general-purpose program, which would necessarily make it more complicated and less directly focused on the beginning Java programmer. Instead, we designed a simple, new interface that uses only one window with two panes, the primary of which is the student's source code. (See Figure 4.1 on page 21 for a screen capture.) This interface was inspired by the similar interface of DrScheme [4].

Ease of installation is important because many students will want to do class work on their personal computers. This is an idea many would give up if they were forced to download a very large file and then go through a potentially buggy and complex installation routine. (These are properties of the installation procedudes for many Java development environments.) To achieve ease of setup, we decided to distribute DrJava as a single Java archive (`.jar`) file, which can be double-clicked (or run via `java -jar drjava.jar`) with no additional setup. Due to DrJava's overall simplicity, as of March 2002 the current version is just over 800 kilobytes.

A related notion to simplicity is predictability; if a program does not work predictably, it is effectively much more complex since the user must try to second-guess the program, discerning when it is working correctly! In addition to striving toward general stability and correctness (as promoted by the aggressive use of automated unit tests), one detail on which we placed great importance was the absolute correctness of the syntax highlighting in the editor. While many source editors highlight various syntactic forms differently (like comments, quotations, and keywords), many of them do not ensure that these annotations are updated correctly as each key is pressed. One common case, for example, is when inserting `/*` at the beginning of

a document, some editors don't recognize that text that was off the screen at the time may need to be updated. This kind of inaccuracy, we felt, was unacceptable, so we set a requirement that DrJava's source editor would efficiently and correctly highlight source text. To ensure correctness while still remaining efficient, we devised a novel representation called the "reduced model," which is explained in more detail in Section 4.2.1 on page 31.

2.2 Interactivity

Any interesting program requires some form of input and output capabilities. Since even beginners prefer to write interesting programs, the I/O question comes up when teaching introductory programming. The problem is that I/O is usually not the conceptual point being taught, yet students want a way to give input and see output from their programs. We feel that teaching students some explicit form of I/O (like console I/O or graphical user interface construction) at the introductory level is not a good idea, because it does not fit in with the conceptual development of the material.

Our solution to this problem is to use a "read-eval-print-loop" (REPL) [5], which allows students to interactively evaluate Java expressions and statements (in a context including the classes they have written) and see the results. Input and output are therefore implicit instead of explicit. A REPL is a natural interface for students, since it uses the same language they are using to write their programs. The REPL concept, which has been present in languages like Lisp, Scheme, and ML for decades, has not previously been integrated in Java development environments. Java is the

first mainstream language that can easily support a REPL though its reflection interface and dynamic class loading, which allow the invocation of existing code and the ability to load new code at runtime. (Without these features, an entirely separate runtime would have to be used for the REPL, which would be questionable in terms of compatibility and performance.)

A REPL provides an easy-to-use, but effective framework for interacting with program components as they are being developed. Such an interface enables the programmer to quickly access the various components of a program without recompiling it or otherwise modifying the program text. It is particularly helpful in introductory programming classes because it enables students to conduct simple experiments to determine how language constructs behave and to determine the results of subcomputations within a program. A REPL interface also serves as a flexible tool for debugging and testing programs and experimenting with new libraries.

2.3 Focus on the language

A conscious choice in the development of DrJava was to provide an interface that focuses almost exclusively on the source language. This choice stands in opposition to some other environments, which focus on generating code from UML diagrams or graphical "wizards." We focus on the source language because students absolutely must understand this material, whereas other tools and representations are distractions.

We are also wary of providing students program templates containing boilerplate

code they cannot modify and may not understand, although we agree with the general idea of trying not to overwhelm the students with too many language details at once. Our solution to this problem is the definition of a series of "language levels," which are an incremental series of sublanguages of Java, akin to language levels implemented in DrScheme [4].

Language levels shield students from the full complexity of the language but still allow them to focus on learning to write programs. At each level, a reasonable number of new concepts is introduced. This gradual development of the language gives instructors the freedom to not even discuss the syntax for constructs they have not yet taught. One example is visibility; even if visibility is not to be introduced at first, it must be referenced in at least a passing way since methods need to be made `public` to be called from other classes.

Language levels are discussed in further detail in Chapter 5.

2.4 Secondary goals

Scalability

Though DrJava is aimed predominantly at neophytes, we haved designed it to scale to other types of users as well. (It would certainly be nice if students did not have to abandon the tool they used in their first class the moment they finished it.) Many advanced students and other developers use no IDE at all because most are too heavyweight and because they do not have enough added value to justify the effort of learning and using them. The combination of DrJava's simplicity and its REPL

make it attractive to many developers.

Available without cost

In order to make DrJava available to the widest possible audience, we have developed DrJava for free distribution. We feel that, when possible, the fruits of academic research should be made available to the public for free.

Open source

Although we always planned to distribute DrJava in binary form freely to the public, we were initially unsure of whether to distribute the source code under an open-source license. As the project evolved, we decided that the General Public License [6] was the best license for DrJava. It makes the source code available and usable by all, but it prevents others from taking our work, changing it or integrating it into another product, and redistributing it (selling it) without also distributing the changes to the source code. The decision to open-source DrJava was based on these factors:

Services provided by `sourceforge.net` SourceForge.net provides an outstanding array of services for hosting software projects — including hosting a CVS tree, Web-based project management tools, Web site hosting, and file download hosting — all at no cost. These services are provided, however, only to projects distributed under an open-source license. (The software behind SourceForge.net is also sold commercially.) The features of SourceForge.net did in fact help

motivate our decision to open-source DrJava, since setting up and maintaining comparable tools would require substantial additional work on our part.

Credible use of undergraduates We wanted to use DrJava as a case study for a software engineering class (COMP312: Program Construction), having students work directly on the project itself. This would be ethically dubious if there were any profit motive on the part of the project, since students would be working on something that will bring profit to others. Open source simply fits better with the academic environment.

Possibility of help from the community By making the source code available to all, sufficiently advanced users are able to contribute to the project. They can look at the code to try to debug problems they find. They can even submit patches to the code to fix problems or implement new features. We hope that this venue for additional developer involvement in DrJava will help the project continue.

A desire to give back to the open-source community We have all benefitted tremendously from the use of open-source software, both in developing DrJava and in our other work. In developing DrJava, we have used CVS, Ant, JUnit, DynamicJava, and Linux, among other things. We felt as though it would be good to contribute back to the open-source software ecosystem.

2.5 Development process

The development process used for this work was a meta-project itself. We were interested in exploring ways to develop high-quality, quickly evolving software as a part of the research process. We decided to try to use Extreme Programming [7] for DrJava's development, adapting it where needed to fit our environment.

Following the XP philosophy, we put a strong emphasis on constructing good unit tests over our code. We use the JUnit framework [8] for writing and executing our unit tests. Unit testing has been a great help to the project, both because it encourages better design (if it'd hard to test, it needs to be redesigned) and because it enforces rigor in program correctness. Also, having a good unit testing framework helped when we changed implementations of subsystems (since we could ensure that behavior was preserved) and kept us from inadvertently re-introducing previously fixed bugs. Unit tests also act as a form of documentation, which is important for a project like DrJava that has a number of people moving in to and out of the project.

We have also used pair programming when it was feasible. Most of this pairing was conducted during the summer of 2001 (when we had 5 people working more or less full-time) and as part of the spring 2002 incarnation a software projects course at Rice (COMP312). We also used pair programming to solve particularly difficult problems at other times. We would have liked to have made more use of pairing, but during the academic year, the difficulty of scheduling time together often made it more efficient to work separately.

We have also tried to adhere to the XP philosophies of incremental specification and development and a short release cycle. The rate of releases of DrJava depends on the number of people actively working on the project, but it has been as high as three per day! We believe this continuous integration has helped us keep the project in a consistent state, requiring less effort to reconcile different groups' changes.

Chapter 3
Related Work

3.1 Pedagogic development environments

In developing DrJava, we were influenced by two related streams of work on pedagogic programming environments: DrScheme and BlueJ. More recently, we have also encountered two other student-focused development environments, JJ and Ready to Program with Java Technology.

DrScheme

DrScheme [4] is an integrated development environment for Scheme with a transparent programming interface similar to DrJava's. It includes a REPL, multiple language levels, a source editor, a static analyzer for Scheme programs, and a "stepper" that shows the incremental steps involved in evaluating an expression. DrScheme has been used in the beginning programming courses at Rice University since 1996 and has served as a model for DrJava.

BlueJ

BlueJ [9] is a research project at Monash University that provides an integrated Java environment for teaching beginning students. BlueJ is written in Java, and executable versions of the program are freely available from `http://bluej.org` for Windows, MacOS X, and UNIX platforms.

BlueJ supports the interactive development of programs using UML diagrams. BlueJ also provides a GUI interface for creating instances of classes and calling meth-

ods on them. To use BlueJ, a student must learn both Java and the protocols for using the BlueJ graphical programming interface. Furthermore, since developing programs in the BlueJ environment does not scale to large systems, students eventually must abandon BlueJ and learn how to manipulate Java program text. BlueJ's editor does not provide brace matching, and it does not consistently update the highlighting of comments and quotations.

Ready to Program with Java Technology

Ready to Program [10] is a simple Java development environment designed for education. It is a commercial product sold by Holt Software primarily through site licenses and by being included in introductory Java textbooks like [11]. Ready to Program is written in C++ and is currently available only on Windows platforms. (A Linux version is in beta testing, according to Holt, and a MacOS version is not yet planned.)

Ready To Program has a simple user interface, with a source editor and intergrated Jikes compiler. However, it has little support for students interacting with their programs. That support is confined to a "run" button, which invokes the `main` method the student must have written, and dialog boxes that allow access to `System.out` and `System.in`.

JJ

JJ [12] is a development environment for introductory programming classes that works entirely within a Web browser. JJ can be used in three syntactic forms, as

explained in [13]: Jr, a syntax that uses words instead of punctuation, somewhat like Visual Basic; APS, a subset of Java closely corresponding to the AP Java Subset [14]; and standard Java. Jr is translated to Java by a simple line-by-line translation process [15]. As APS follows the AP Subset fairly closely, it has no support for inner classes. The APS and Jr syntactic forms of JJ have built-in support for design-by-contract and console I/O. They also support procedural programs with no explicit class definitions, in addition to supporting Java's object-oriented programming paradigm.

Students install no software (other than the browser) on their computers to use JJ; all compilation, file management, etc. occurs on the server. Running inside a Web browser severely limits JJ's user interface. There are many panes and buttons, and often it can become unclear what one can and should do next. JJ supports standard IDE features like source editing and compilation. Because all students' work is hosted on the server, JJ can support a mechanism that allows students to submit their completed work electronically.

The JJ environment uses a form of console I/O to allow students to interact with the programs they write. In Jr and APS, there are special I/O primitives (`System.out` is also supported) that read from/write to a window inside the JJ interface. For example, APS defines the method `JJS.readInt()` for reading an integer from the input window. The contents of the I/O window can be saved and later submitted to the teacher, as a way of demonstrating the test data sets on which the student ran his program.

The JJ project began at the California Institute of Technology and the California

State University-Northridge and now being supported by the company publicstaticvoidmain.com. Caltech hosts a JJ server that classes at other educational institutions can use at `http://jj.caltech.edu`.

3.2 Other Java development environments

Extensible editors: Emacs, JEdit, Jext

Emacs (with the Java Development Environment for Emacs [16]), JEdit [17], and Jext [18] are all general-purpose text editors that have integrated support for Java development. All are available on most platforms (JEdit and Jext are written in Java) and freely available under the GPL. All of these editors are very flexible, supporting plug-ins that can extend their feature set. (Emacs, of course, has the integrated Emacs Lisp system.)

Commercial IDEs: JBuilder, JDeveloper, Forte, etc.

Borland's JBuilder [19], Oracle's JDeveloper [20] and Sun Microsystems's Forte [21] (and many others) are commercial Java IDEs with very large feature sets. They are geared towards professional developers, not beginning students.

Open-source "professional" IDEs: Netbeans, Eclipse

Netbeans [22], supported by Sun Microsystems, and Eclipse [23], supported by IBM, are both large, extensible IDEs that are available under open-source licenses. They are similar in scope to the commercial IDEs (and Forte is actually an extended version of Netbeans), but they are freely available under open-source licenses.

3.3 Other REPLs that can interact with Java

Although none of the available IDEs prior to DrJava integrate a REPL, there are standalone tools that support read-eval-print-loops that interact with Java. Some use a version of Java as a source language, while others are based on other source languages.

DynamicJava

DynamicJava [24] is a Java source interpreter, written in Java, that is freely available under an open-source license. An extension of DynamicJava forms the core of DrJava's interactions pane, as detailed in Section 4.1.3. DynamicJava supports the entirety of the Java language (with some extensions to ease interactive use), including the definition of new anonymous inner classes. By using Java's reflection capabilities, DynamicJava can make calls to standard, compiled code in user or system classes. Although the project has been dormant since March 2001, DynamicJava is quite stable and effective.

BeanShell

BeanShell [25] is another Java source interpreter, supporting a very similar feature set to that of DynamicJava. BeanShell is available as a plug-in for various development environments, including Forte and JEdit.

MiniJava

MiniJava is a REPL for a Java-like language proposed in [26]. Like DrJava, it was developed as a pedagogic tool for beginners. However, from the details in the paper it is not clear exactly what MiniJava is. For example, it is not clear whether MiniJava runs on a Java virtual machine and calls into compiled Java class files via reflection, or whether it works as an entirely separate interpreter. MiniJava does not seem to be an active project, and it does appear to be available for download in any form.

REPLs with other source languages: Jython, Jacl, Kawa, Rhino

A number of other scripting-type languages have been implemented on top of the JVM, and these languages all support a REPL mode.

Jython [27] is an implementation of Python that runs on top of the Java virtual machine. Jython compiles Python programs to JVM bytecode and provides extensive support for interoperability between Jython and Java classes. Jython has a large user community, and it is robust and stable.

Jacl [28] is an implementation of Tcl for the JVM. It works as a Tcl interpreter, written in Java, with an extended Tcl syntax that provides access to Java classes and methods.

Kawa [29] is an implementation of most of standard Scheme for the JVM. It provides access to reference and instantiate Java classes and to call methods on objects.

Rhino [30] is an implementation of JavaScript for the JVM. Through the LiveConnect API, JavaScript code in Rhino can call out to Java code.

Chapter 4
DrJava

DrJava is a simple, yet powerful, Java development environment. It is written in Java, and runs on any Java 2 version 1.3-compatible virtual machine. (DrJava has been tested on Windows XP, Windows 2000, Windows 98, Linux, Solaris and MacOS X.) It is available for free, and it is distributed under the open source General Public License [6]. Links to download DrJava and its source code are available from its Web page, `http://drjava.sourceforge.net`.

DrJava as a tool supports any pedagogy. Though at Rice we teach an object-oriented pedagogy that incorporates functional programming concepts, the environment would work equally well for other pedagogies.

DrJava supports a transparent programming interface designed to minimize the amount of time that beginners must spend learning how to use the tool itself. The program leverages the students' understanding of the language, embracing the code rather than hiding it as some other environments do. The transparent interface consists of a window with two panes, as shown in Figure 4.1:

1. An *interactions pane*, where the student can input Java expressions and statements and immediately see their results. This pane is also used to display compiler error messages.

2. A *definitions pane*, where the student can enter and edit class definitions with support for brace matching, syntax highlighting, and automatic indenting.

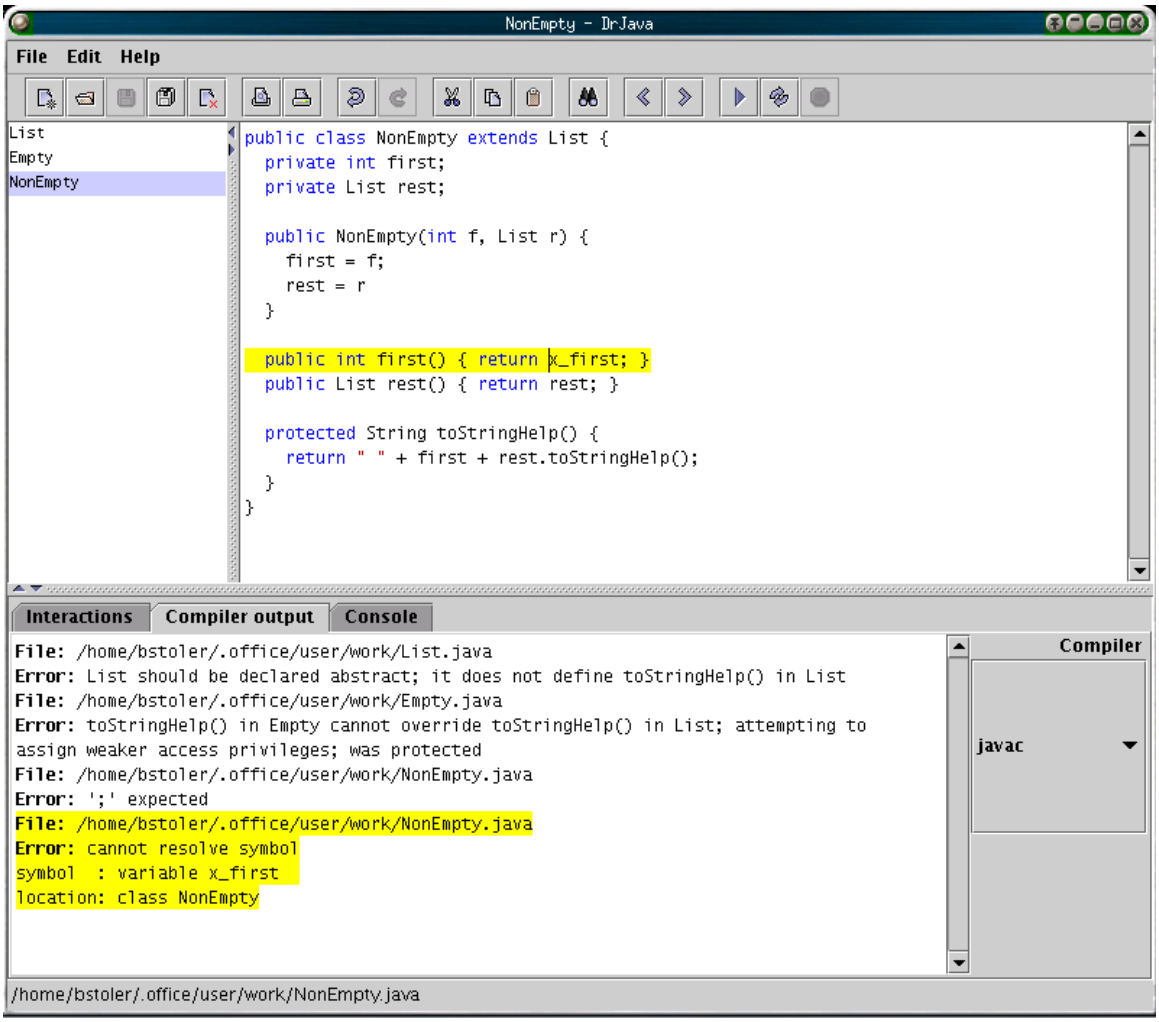

Figure 4.1 The interface of DrJava. This image shows DrJava displaying a number of compiler error messages.

The two panes are linked by an integrated compiler that compiles the classes in the definitions pane for use in the interactions pane. Using DrJava, beginning programmers can write Java programs without confronting issues such as text I/O, a command line interface, environment variables like `CLASSPATH`, or the complexities of the projects interface supported by a commercial Java development environment.

4.1 The Interactions Pane

The interactions pane provides a "read-eval-print loop" that enables the student to evaluate Java expressions and statements including statements that define new variables. The REPL is useful for testing and debugging programs, and also for experimenting with other libraries interactively.

4.1.1 Testing and debugging

The interactions window of DrJava provides an effective vehicle for testing and debugging programs. Using the REPL, a programmer can individually test program methods by embedding test methods in the program and invoking them from the REPL as alternate entry points. This approach to testing is far more flexible than the usual practice of including a `main` method in each class.

When testing reveals a bug, a REPL is often a better tool for program debugging than a conventional debugger. Although conventional debuggers allow a programmer to add breakpoints to a program and to step through its execution, they do not allow the programmer to interactively start execution with any method invocation. In the conventional batch programming model, selecting a new entry point for program execution is a cumbersome process that involves (i) modifying the `main` method of the program (or class in the case of Java), (ii) recompiling the program, (iii) re-executing the program from the command line, and (iv) restoring the main method of the program back to its "normal" configuration. This task is sufficiently awkward and time consuming that programmers avoid doing it, even when it may be the quickest

way to diagnose a particular bug! With the REPL, a programmer can start execution with any method call he likes, without recompilation.

A REPL is a particularly good debugging tool for beginning programmers because it does not require them to learn the mechanics of using a debugger such as as how to set and unset breakpoints, how to dump the stack, and how to query the value of variables. With a REPL, the same interface is used to run, test, and debug programs.

For more advanced programmers, debuggers are useful tools that complement the capabilities of the REPL. For this reason, we are implementing a conventional debugger for DrJava, as explained in more detail later.

4.1.2 Library exploration

The interactions window also provides an efficient means for exploring new API's. Students can learn the essential features of complex libraries much more quickly if they can conveniently conduct simple experiments. This mode of learning is particularly effective for graphical libraries like Swing. With the read-eval-print loop, students are able to interactively create new JFrames and JPanels, display them, and watch their content change as they add new components. This immediate feedback can help students learn how to construct and lay out GUI components much more efficiently.

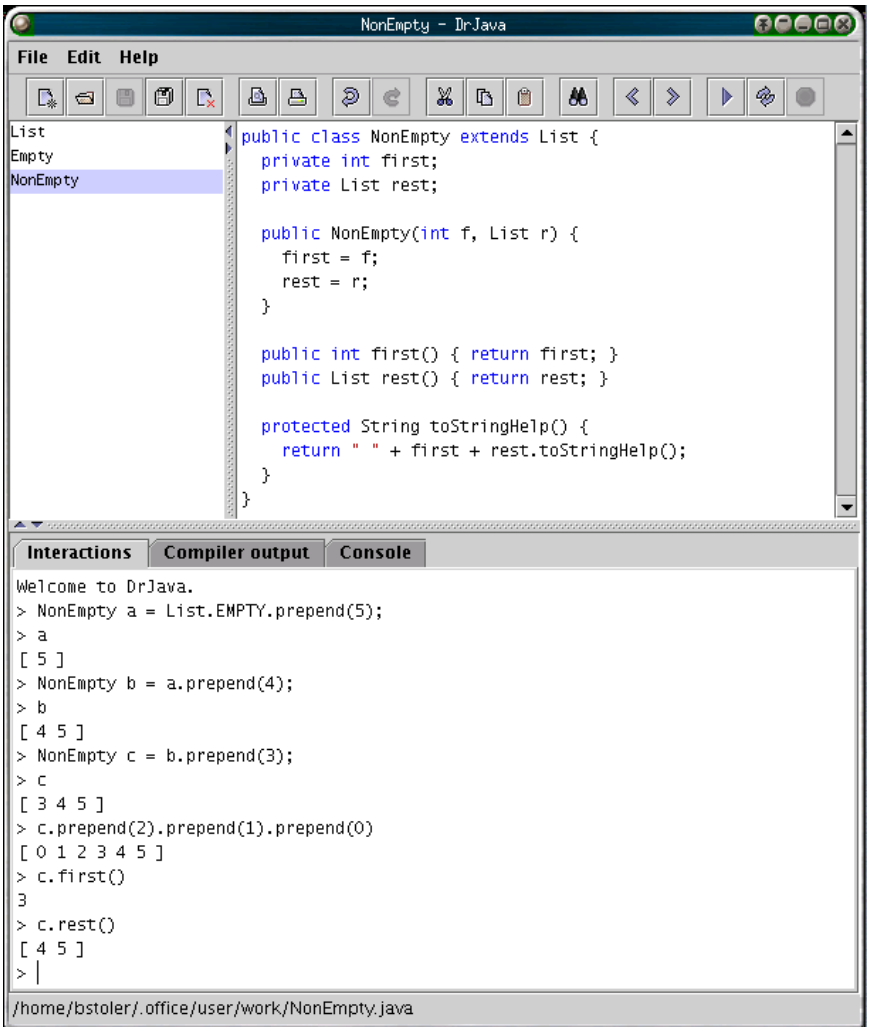

Figure 4.2 A screenshot of DrJava, with the REPL being used to inspect the results of calling some methods from the code in the definitions pane.

4.1.3 REPL Implementation

An early prototype implementation* of DrJava's REPL relied on a Java compiler to evaluate interactive transactions. In this implementation, the REPL takes the transaction input, builds a stub Java class around it, and compiles it. This approach manages to support the entire Java language (by virtue of being implemented through

* Two early DrJava prototypes were created by Paul Graunke, then a graduate student at Rice University studying under Professor Robert Cartwright. The first was based on the MrEd framework underlying DrScheme. The second, discussed here, was written in Java.

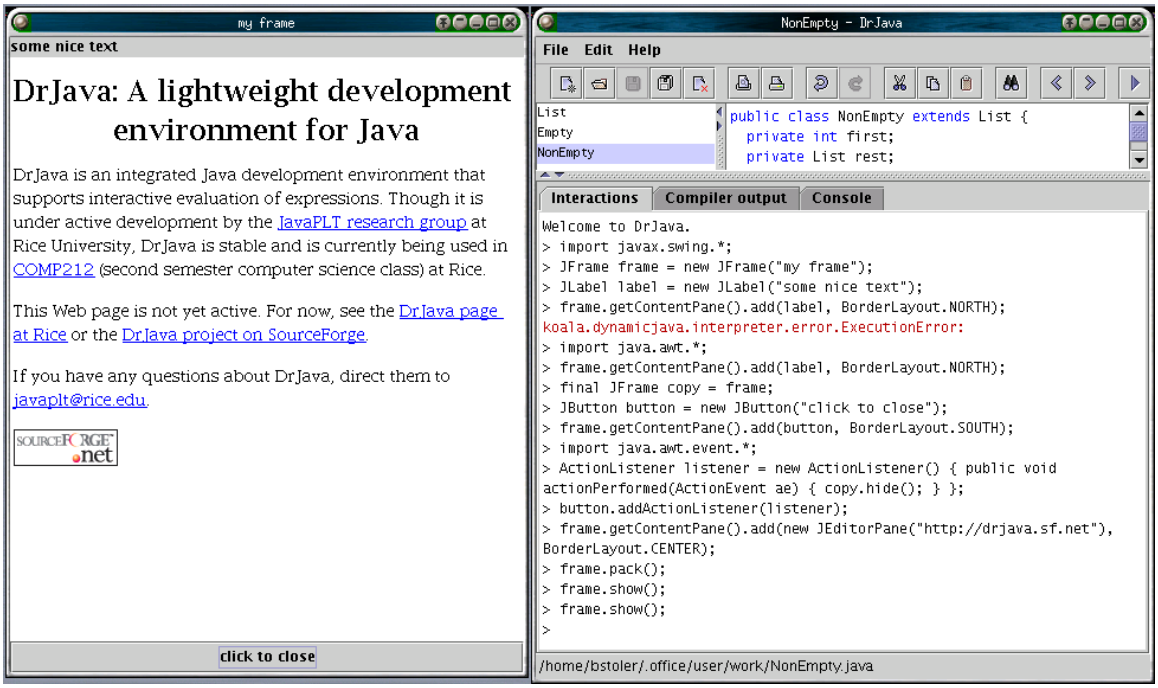

Figure 4.3 This screenshot shows that a GUI can be interactively created in the REPL and shown immediately.

a standard Java compiler) yet it is relatively simple to implement. But it is slow — very slow. Even with the optimization of writing the compiled classfile directly to memory (avoiding disk I/O for every evaluation), this incarnation of the REPL takes an unacceptably long time to interpret even the simplest inputs (like `5`). On a Pentium II/350, a trivial interpretation takes three seconds. The overhead of compiling a Java class for every interaction is too high to provide crisp performance.

For DrJava's REPL to be useful, we knew it had to respond almost instantaneously to simple transactions. Seeing no reasonable way to preserve this architecture, we decided on a new course: writing a Java interpreter to function as a REPL. The interpreter could use Java's dynamic class loading and reflection capabilities to call into pre-compiled code, meaning that the interpreter would only have to interpret

the text literally inputted into the REPL. Luckily, however, much duplicated effort was prevented when we discovered an open-source implementation of exactly this feature set. DynamicJava [24] is a Java source interpreter that allows the interactive evaluation of *any* Java language construct. We used this interpreter as the basis of our REPL. DrJava's REPL is implemented as an extension of DynamicJava, created by subclassing some of its components. To date, we have not modified any of the DynamicJava code base itself.

Our first extension to DynamicJava was to make it automatically include the classes defined in the definitions pane in its class path. Since DynamicJava uses its own custom class loader with support for adding items to its search path, this was fairly easy to do. Specifically, after each successful compilation of a source file, the REPL is reset and the set of all *source roots* of all open source files is added to the REPL's class search path. The source root of a file is the directory that forms the base of the package hierarchy relative to that source file. For example, if class `edu.rice.cs.drjava.DrJava` is in the directory `/home/bstoler/sf/src/edu/rice/cs/drjava`, the source root is `/home/bstoler/sf/src`. Implementing this feature allowed the REPL to automatically have access to classes compiled in the definitions pane, and, since the REPL's class loader is reset after each compilation, changes in classes were immediately reflected in the REPL — *except* when the class was on the system class path.

The reason for this problem is based in Java's `ClassLoader` architecture. In the standard delegation model used by `ClassLoader`, custom class loaders are asked to

load a class *only* if the default system class loader can not find the class. In other words, the system class loader always takes priority if it is capable of loading the class file. As a result, classes loaded by the system loader can never be unloaded. Hence, if a class compiled in DrJava is loaded by the system loader, future changes after recompilation will not be reflected in that DrJava session!

To get around this restriction, we implemented a custom class loader that does not follow the standard delegation model for class loaders. Instead, the custom class loader tries to load every class itself and only allows the system loader to load classes that only it can load (the standard Java libraries). The custom loader typically delegates the task of fetching the bytecode for a given class to the system class loader, but the custom loader always tries to call `ClassLoader.defineClass()` itself. This protocol is important because the loader that calls `defineClass` is the one associated with the loaded `Class` object. When that loader is garbage collected, the class itself can also be unloaded. In addition, because this class loader is associated with the class, the other classes whose loading is transitively forced are loaded with the custom loader. For this reason, we called this custom class loader a "sticky" class loader. Creating this custom class loader gave us the desired behavior of being able to dynamically reload any non-system class.

After we resolved the class loading problem, we set out to clean up the output returned from DynamicJava a little, this time by subclassing its interpreter. We cleaned up many small items (like the formatting of exceptional returns and of some error messages), but the most important change we made was the treatment of statements

that return no value. DynamicJava has no special value to represent an interpretation that returns no value at all; instead, it internally uses `null`, which is ambiguous. This led to the very confusing behavior of the REPL printing `null` as the result of statements that had no return value at all! To correct this problem, in our interpreter subclass, we override all the AST cases that return no value, forcing them to return a sentinal object so that the REPL could know to not print any result.

With these extensions, the REPL worked as a useful, integrated part of DrJava. But one important obstacle remained: Because the REPL ran in the same JVM as DrJava, there was no protection against errant or malicious code in the interactions pane taking down the entire DrJava process. The potential problems posed by the REPL interfering with the main DrJava program include:

1. An interaction calling `System.exit`;
2. an interaction entering an infinite loop; and
3. an interaction opening a modal dialog with no way to close it, effectively locking the entire user interface.

The first problem can be prevented by judicious use of a custom `SecurityManager` to prevent unauthorized calls to exit the JVM. The second problem, however, is more difficult to work around since the Java virtual machine has no inherent notion of independent processes that can safely be terminated. Although the Java `Thread` class originally supported a `stop()` method, this method is deprecated because it

is unsafe, potentially leaving the virtual machine in an inconsistent state [31]. To support safely killing an errant thread in the REPL, we considered two options: using a soft termination system as implemented by Rudys and Wallach [32], or running the REPL in a separate JVM.

However, for the third problem, soft termination is not sufficient. In fact, there is no general-case way to resolve this particular problem short of preventing the REPL from opening modal dialogs, since Java's AWT framework does not support multiple separate "modality groupings."* In addition, the soft termination system requires patching the standard class library to be able to interrupt blocking I/O, which would unacceptably complicate the installation procedure for DrJava.

Hence, we concluded that running the REPL in a separate JVM was the only viable option. We implemented this strategy by constructing a class, `ExecJVM`, which can invoke a new copy of the running JVM, and then by implementing a communications framework for the two JVMs based on Java's Remote Method Invocation facility.

The two JVM system works well, as it allows safely terminating the REPL without affecting DrJava itself, and it also gives the REPL its own graphics world to work in. The performance penalty imposed by the inter-JVM communication is negligible in practice, since the size of the input and output of the REPL is usually not large. The only price we pay for this system is in memory usage. Due to a documented deficiency

*A similar bytecode rewriting scheme could be used to simply eliminate modality of dialogs opened from the REPL, but this would change the semantics of these dialog boxes.

in Sun's JVM implementations (at least through version 1.4), multiple JVMs running on the same computer do not effectively share memory, leading to more memory being used that should be [33]. This makes DrJava consume more memory, and if memory becomes exhausted, run slowly due to excessive swapping.

4.2 The Editor

Since beginning students make many syntactic mistakes and have trouble diagnosing them, DrJava is designed to accurately detect basic syntactic errors as soon as possible. Like many other development environments, we support automatic indentation and keyword highlighting. In addition, we support *fast*, yet *fully correct* matching parenthesis and brace highlighting and the coloring of comments and quotations. Correctness is essential to our goal of providing functionality with minimal complexity, since incorrect behavior forces the user to focus on the idiosyncrasies of the tool, learning precisely when it cannot be trusted. Unfortunately, fully correct parenthesis/brace matching and comment/quotation highlighting is not supported in most development environments for Java.

DrJava's highlighting and parenthesis matching is correct in all cases, even in the presence of comments or quotations containing special characters. (Many environments fail to take into account the fact that a brace inside a comment does not in fact function as a brace.) DrJava immediately updates (at the granularity of each keystroke) the highlighting of commented and quoted sections of the code, shielding students from subsequent surprises when they compile their code. To efficiently

support these features, we created a data structure called the "reduced model."

4.2.1 Implementation via "reduced model"

The reduced model is a collapsed representation of the document's contents that omits all of the characters in the document that are not relevant to the "nesting" structure of the source file. This model is updated with every change to the document. These frequent updates are efficient because the size of reduced model is much smaller than the complete source file. The main cost of the reduced model is added code complexity. It is a complex mutable data structure, requiring careful attention to detail to maintain its consistency.

The reduced model maintains a linked list of `ReducedToken`s. Each reduced token is either a `Brace`, a character or group of characters contributing to the structural representation of the source code like `/*`, `"`, { or `(`, or a `Gap`, which represents a collection of characters that do not contribute to the structure of the code. Because most of the code (keywords, identifiers, contents of quotations and string literals, etc.) are inside `Gap`s (with multiple consecutive gap characters coalesced into one `Gap`), the number of tokens in the reduced model is much smaller than the number of characters in the document. Experimental evidence suggests that the number of `ReducedToken`s is about 14 percent of the number of characters in the document.

Each reduced token keeps track of its state, which encodes the syntactic form that it is a part of: "inside single quotes," "inside double quotes," "inside a line comment," "inside a block comment," or "free," meaning it's inside none of the above. The state

of all affected tokens is updated on every change to the document's text. While this takes time linear in the number of reduced tokens in the worst case (like inserting a `/*` at the front of a document containing no comments already), in most cases each modification only requires updating a few tokens. In fact, the most common case, simple increasing or decreasing the size of a `Gap`, requires no further updating at all. In practice, updating the reduced model is not slow enough to cause a noticable performance problem in DrJava.

In the presence of the reduced model, brace matching is implemented simply by linearly searching from one brace to find the opposite brace type (ignoring braces that are not "free"). Again, due to the small number of tokens, this linear search is inexpensive in practice. The reduced model also helps implement indentation, providing information about the location of the most recent open brace and the quoting/commenting status that is needed by the indenting algorithm. Without the reduced model, this information would have to be entirely recomputed for every indentation operation.

4.3 The Integrated Compiler

A Java compiler is integrated with the source editor, allowing the student to see the locations of compiler errors in their source simply by clicking on them. This integration prevents the student from having to interact with the compiler as a separate entity and manually move to locations of compiler errors in the source code.

4.3.1 Implementation

DrJava's facility for supporting compiler integration was designed to address two goals: (i) the ability to support multiple different compilers, and (ii) the ability to run the compiler in the same JVM as DrJava itself. To easily support multiple different compilers, we made the compilers work as plug-ins to DrJava via an interface, `CompilerInterface`. These compilers are then managed by the class `CompilerRegistry`, which loads and keeps track of compiler instances. Compilers are instantiated using reflection, which allows DrJava to support compilers that do not need to be present at all times. (If all of the supported compilers were explicitly referenced by class name, it would be a link error to run DrJava without the presence of all supported compilers.) This plug-in architecture also leaves open the possibility of allowing users to define their own compiler interfaces, to support compilers not yet natively supported by DrJava

We want to run the compiler in the same JVM as DrJava for effeciency reasons; invoking a new JVM for each compilation would significantly slow down compilation, and for short compilations it would dominate the running time. However, this goal is complicated by the fact that standard Java compilers were not designed to be able to run multiple times — in particular, `javac` 1.3 and 1.4 can only run one compile session per instantiation.

Our first attempt at getting around this problem was to simply recreate the compiler object for each compilation. While this worked, we discovered it was the source

of an enormous memory leak: with each compile, DrJava's heap usage increased permanently! The cause of this leak was that `javac`, which was not designed to be used multiple times in one JVM, stores data in static members that are never cleaned up. To resolve this problem, we needed a way to be able to unload the compiler classes after every invocation, which would clean up all the static data associated with them. Once again, a custom `ClassLoader` comes to the rescue here. We use a custom loader for each compiler, so that they can each be reloaded by simply removing all links to the class loader and then creating a new one.

4.4 Other implementation notes

This section discusses other interesting aspects of DrJava's implementation that have not been previously mentioned.

4.4.1 Global model, and designing for testability

As DrJava began to grow, we realized that we were falling short of our goal of complete coverage of the code with unit tests. Though low-level components were well-tested, a substantial amount of logic — the integration between different components — was buried inside the GUI code (the view in the model/view/controller design pattern), preventing it from being effectively tested using the JUnit harness. Our desire to test everything exposed a flaw in our design, namely that we had coupled the program's logic too tightly to the user interface. We solved the problem by moving the logic that glues together the different components from the user interface to one class, the global model.

DrJava already separated the model and view for its components. The `GlobalModel` then simply became the container of all of the models, providing methods corresponding to the high-level actions that would be visible in the user interface and therefore hiding the low-level models (following the Mediator pattern). Since the global model implements all of the logic integrating multiple components, it can be easily tested (since it has no GUI code at all). Then, the GUI code becomes dramatically simpler as well, since it basically just delgates all calls to the global model. However, to move all of the logic to the global model, it had to be able to asynchronously fire events back to the view; to do this, we created an interface `GlobalModelListener` and made the global model notify all registered listeners of the events. With this framework, we could also register listeners inside test cases, ensuring that the events that were expected were in fact fired.

To demonstrate the role of the global model, consider this example. There is a document open, which has never been saved, but it contains the text to define a class. The user then clicks "compile". The sequence of operations that occur (slightly simplified) is below:

1. User clicks "compile" button, firing the button's action listener.

2. Button's action listener calls `GlobalModel.startCompile`.

3. The global model checks if the file to be compiled is saved, and sees that it is not. A rule imposed by the global model is that an unsaved file can not be compiled, so the model sends an event to its listeners, `saveBeforeProceeding`,

notifying all listeners that the given file must be saved before the compile can proceed.

   (a) The UI's listener handles the `saveBeforeProceeding` event and prompts the user, asking whether he wants to save and continue or abort. If he says "yes" to saving, the UI calls `GlobalModel.save()`.

   (b) The global model delegates to the underlying model for the document, `DefinitionsDocument` and asks it to save the file.

   (c) The global model fires the event, `fileSaved` to its listeners. The UI listener responds to this event by updating the UI to remove the "modified since save" cue from the file name.

4. The global model once again checks if the file to be compiled has been saved. If it still has not been saved (meaning none of the listeners acted on the `saveBeforeProceeding` message), the compilation aborts.

5. Global model notifies listeners of event `compileStarted`. The UI listener reacts to the event by changing the cursor to the hourglass, wait cursor.

6. The global model gets the active compiler from the compiler registry, and calls the compile method on the active compiler. It stores the resulted list of errors.

7. The global model notifies listeners of the event `compileEnded`. The UI listener responds to this by updating the error list pane, showing the new list of errors, and by changing the cursor back to normal.

8. If there were no errors, the global model:

   (a) Clears the console pane's model.

   (b) Fires the event `consoleReset`.

   (c) Resets the interactions pane's model.

   (d) Fires the event `interactionsReset`.

Note that for testing purposes, the actions performed by the UI is in the above example would be performed by the test harness.

4.4.2 Tools: JSR-14, CVS, Ant, JUnit

We have taken advantage of a number of tools to aid in the development of DrJava. JSR-14 is a proposal to add support for generic types to Java. Because our research group has substantial experience and interest in using generics, we have used them in the development of DrJava, using the prototype implementation released by Sun [34].

Because DrJava was developed by a team (with up to 15 people working on it at once), we needed to use tools to manage the process and the code. We used the Concurrent Version System (CVS) [35] to safely store the code. To implement the unit testing part of our XP process, we used JUnit. To automate the compilation, testing, version control, and distribution creation for DrJava, we use Apache Ant [36], a make-like tool for Java. One important aspect of our Ant build script is that it does not allow any changes to be committed to the CVS repository unless all unit

tests pass. This simple restriction goes a long way towards maintaining stability of a fairly large code base even in the presence of many people simultaneously being able to modify the entire thing.

4.4.3 SourceForge

Because DrJava is distributed under the GPL, we are able to use SourceForge.net to host the project. We use SourceForge to host our CVS tree, our bug/feature request databases, our release distributions, and our Web site (`http://drjava.sourceforge.net`). In essence, all of our project management capabilities are handled by SourceForge. Besides providing greater functionality than we would otherwise be able to put together on our own, it's also very easy to give access to new project members, even if they are not from our university — this would be hard or impossible otherwise.

Chapter 5
Language Levels for Java

The complexity of Java's syntax can be a major stubling block for beginning Java programmers. To mitigate this complexity, we have developed (but not yet implemented) a series of progressively more advanced sublanguages of Java, with syntactic restrictions and automatic augmentations at each level. These "language levels" have been strongly influenced by the language level design in the DrScheme environment.

The goals of language levels for Java are:

- To allow instructors to incrementally teach language concepts, avoiding the need to tell students to mindlessly copy-and-paste code snippets they haven't been taught to understand;
- to support better error diagnostics in the cases of invalid syntax or other compile errors;
- to help teach students good habits by restricting them to a "tasteful" subset of the language at first;
- and to enable the development of better static analysis tools, like null-pointer analysis and static verification of casting operations, over restricted forms of the language.

To support this, we plan to implement four language levels other than full Java. Each language level consists of three components:

- The imposition of restrictions on the language constructs usable at a level (like no `static` methods, for example);
- automatic additions to the student-written code (like automatically making all variables be `final`); and
- customized error messages based on the level the student is at, so that the error messages don't refer to language constructs the student has not yet learned.

The rest of this chapter explains language levels in more detail, first outlining the implementation framework we have created and then giving a sketch of a design for the language levels themselves.

5.1 Java language processing tools framework

The architecture that we plan to use for supporting language levels for Java is source-to-source translation mapping "language level Java" source to standard Java code. This architecture is designed to minimize the number of components we have to write — namely, to keep from writing a complete source-to-bytecode compiler for each level.

The core of such an implementation of the language levels themselves would be the series of checks and elaborations that map the input to standard Java. However, there are a number of ancillary components that are required to allow the development of this core. There must be a consistent intermediate representation of Java code that supports language levels, a framework for developing a parser for each language level, and a mechanism for translating the intermediate representation back to source. These substantial components had to be developed before language level development could proceed.

This section outlines the development of the tools I developed to support Java language processing.

5.1.1 Java AST

The heart of programming language implementations is the intermediate representation of the code. These data structures form the interface used between all different components of language processing systems. That said, we decided that we wanted to invest some effort to create a nice, reusable abstract syntax for Java (including

support for GJ-style parametric types), one that we'd be happy to use for language levels and other future projects. (We would have been happy to use a pre-existing AST framework for Java, but we found none that fit our criteria and were available under acceptable licensing terms.)

Our previous experience working on Java language projects (NextGen [37], which is based on the Generic Java [38] code base) suggested three criteria of an easy-to-use Java AST framework: (i) immutable objects, (ii) no usage of `null` in ASTs, and (iii) strong static typing of nodes (having each concrete class only represent one form of syntax). Also, since we expect there to be many different functions to be applied to the AST, we wanted to use the Visitor pattern to allow these functions to be developed independently of the AST classes themselves. (In fact, we decided against putting *any* behavior in the AST classes at all, other than equality testing, hash code creation, and String generation for debugging.)

To create such a framework would require building scores of classes, all of which were basically just data structures organized in a complicated inheritance hierarchy, and all needing common utility methods (`.equals`, `.hashCode` and `.toString`). These classes would have substantial redundancy (just to rename a class would affect all classes that contained any references to it), making maintenance a nightmare.

Because each class was to have so little behavior, and because the behavior it was to have could be easily algorithmically determined, we decided the best way to proceed was to develop a specification language for AST-like structures and then to write a tool to generate the Java source from such a specification. This resulted in

the tool ASTGen (documented in the next section). With this written, creating our Java AST classes was simply a matter of creating the input specification to feed to the tool. This specification (which also serves as good overview documentation for the class hierarchy) is presented in Appendix A.

5.1.2 ASTGen

ASTGen* is a tool I wrote to generate code for a Composite pattern hierarchy of immutable† data objects. The need to generate such a hierarchy comes up in many circumstances, particularly when creating compilers and interpreters. The purpose of this tool is to allow the maintenance of the tree structure in a simple, high level description, and then to automatically generate the code to implement it. Note that its purpose is that the output of the tool is never edited; any changes are instead made in the input file. Thus, this input becomes the single point of control for the entire hierarchy, vastly simplifying maintenance. Because the data classes have no explicit behavior, they can be acted upon by visitors.

ASTGen is distributed under the General Public License [6] and is available from `http://astgen.sourceforge.net`.

Input file syntax

The input syntax for ASTGen is terse and simple. It is designed so that the input syntax can be easily read by a human as well as the tool, so that it can serve as simple

*__A__bstract __S__yntax __T__ree __Gen__erator, although it can be used to generate other hierarchies of immutable data objects as well.

†The tool could fairly trivially be extended to support mutation, but thus far we have had no need for this, as we prefer to keep things immutable when possible.

documentation for the class structure. An example of an input file to define a simple expression structure is shown in Figure 5.1.

```
// Example ASTGen input, for a simple expression structure
visitmethod apply;
visitormethodprefix for;
tabsize 2;
outputdir expr;
allownulls no;

package expr;

begin ast;
interface Expression(int lineNumber);
  // line number is the line the expression was on in when it was parsed
  // It is ignored for equality because it doesn't change the value of
  // the expression
  abstract AbstractExpression(ignoreForEquals int lineNumber);
    abstract BinaryOpExpression(Expression left, Expression right);
      AddExpression();
      MultiplyExpression();

    IntegerConstant(int value);
    FloatConstant(float value);
end;
```

Figure 5.1 An example ASTGen input, defining a simple expression abstract syntax.

Below is an informal description of the input syntax:

- Java-style line comments (beginning with `//`) and block comments (between `/*` and `*/`) are allowed anywhere, and their contents are ignored. Blank lines are also allowed anywhere.

- The first section of the `.ast` file is a list of unary options. (All options must be at the top of the file, before any class declarations.) Each option is set by

putting the option name, a space, and then the value, and then it is terminated by a semicolon. All option statements may be left unspecified, as each option has a default value. The set of valid options is declared below:

visitmethod This is the name of the "visit" or "accept" method that all concrete classes will implement. This is the hook used to run a visitor on a class. The default value is *visit*.

visitormethodprefix This is the prefix for all of the visitors' "case" methods. Each visitor has one such method to handle each concrete class in the hierarchy. The method name is of the form [prefix][class], where [prefix] is the value set by this option and [class] is the name of the class the method handles. The default prefix is *for*.

tabsize The number of spaces to be used to represent an indentation level. (ASTGen does not support using actual tab characters.) This value is used when generating the Java source files, and it's also used when parsing the ASTGen input file itself, for the part that defines the classes in the hierarchy. The default value is *2*.

outputdir This specifies the directory, relative to the location of the input file, where the generated output will be put. The default is ".". *Note: It is recommended to keep the automatically generated files in a different package, so that it can easily be removed when the tool is going to regenerate them.*

allownulls This boolean-valued parameter (yes/no/true/false) specifies whether fields in the generated classes can be assigned the value of `null`. If this is set to no/false, each class's constructor will throw an `IllegalArgumentException` if it is ever given a value of `null` for a parameter. The default value is *no*.

- After the list of options, optionally a package and list of import statements can be declared. These statements are simply copied verbatim into each of the generated Java source files. So, if the generated files are to be in package *edu.rice.cs.javaast.tree*, there needs to be a line `package edu.rice.cs.javaast.tree;` in the input file. Also, any references to other classes that are not in the same package must be explicitly imported via an import statements. *Note: You must ensure that the package statement is consistent with the outputdir.*

- The rest of the file will declare all of the classes and interfaces in the hierarchy. To begin this section, there must be a line of the form `begin ast;`, and there must be a line of the form `end;` at the end. Each line in this section declares one class or interface in the hierarchy. The first class or interface, designated the *root class*, is declared on the line immediately after the `begin ast` statement. It must have no spaces preceding its definition. All succeeding classes are indented multiples of *tabsize* spaces; indentation is used to signify subtyping relationships. The class/interface declared on a line is a subtype of the class/interface declared on the closest preceding line at one lower level of indentation. It is an error for an interface to attempt to be the subtype of a

class.

- Each class/interface is declared by a single line, of a form that is very similar to the declaration of a constructor. The general form of a class/interface declaration line is: `["abstract" | "interface"]` *name* `"("` *paramlist* `")"` `["implements"` *implementslist*`]`. The `abstract` or `interface` keywords signify that the line declares an abstract class or an interface, respectively. *name* is the name of the class or interface being declared. *paramlist* is a comma-separated list of fields declared in this class, or if it is an interface, the fields that all implementing classes must provide getters for.

  Each parameter is of the form `["ignoreForEquals"]` *type* *paramname*, where *type* is the type of the field and *paramname* is its name. `ignoreForEquals` signifies that this field should be ignored for the purposes of determining equality in the automatically generated `.equals` method. In class declarations, a parameter defines a field of that class. In interface declarations, a parameter simply requires that implementing classes have a field by that name, with a standard-named accessor of the form `get`*Field*, where *Field* is the name of the field with the first letter of its name capitalized.

  implementslist is a comma-separated list of additional interfaces that this class/interface should implement/extend, beyond those implied by this class/interface's position in the indentation hierarchy.

Generated output

From a single input file, ASTGen generates many files in order to create a composite hierarchy as defined and to create visitors over it. There is one source file created for each defined class/interface in the input, plus four visitor classes and one standard helper class, `TabPrintWriter`. An example of the top-level interface and some of the visitors generated by ASTGen (based on the input shown in Figure 5.1) is shown in Figure 5.2.

Defined classes For each class line in the input file, a Java class is generated. This class has a `private final` field for each field declared in the class definition, in addition to inheriting all fields from its superclass. A constructor is generated automatically, and the constructor takes as arguments all of the fields, in order, contained in the class and all of its ancestors. (The fields are in declaration order, so those declared at the root of the ancestry tree come first.) The constructor enforces the restriction (if turned on in the options) that no field can have a null value. Also, to allow comparison on `String` instances with `==` and to possibly save memory, all `String`s are automatically interned when they are stored.

To make the data members accessible, a getter method is automatically generated, with the name get*Field*, where *Field* is the name of the field, with the first letter capitalized. (Since the classes are all immutable, there are no setter methods.) Also automatically generated are `.equals` and `.hashCode`, which implement member-wise equality, and a `.toString` method, which prints the values of all the fields so as to

form a nice tree.

Each class also has two "accept" methods to use with visitors, one for visitors that return a value and another for visitors that do not. The value-returning visitor is parametric (using Generic Java-style type parameters), so the accept method is of the signature: `<T> public T accept(`*RootClass*`Visitor<T> v)`. The other accept method is of the signature: `public void accept(`*RootClass*`Visitor_void)`.

Defined interfaces Interfaces defined in the input file have a getter method for each "field" that is declared for it. Also, the interfaces mandate the presence of the accept methods that all classes generated by ASTGen will have, so that visitors can be dispatched on variables with a static type of an interface.

Visitor interfaces/classes Two visitor interfaces are created for each ASTGen input file: *rootClass*`Visitor<T>`, a Generic Java interface supporting a visitor that returns an instance of type parameter `T`, and *rootClass*`Visitor_void`, an interface supporting a visitor that returns no value. Each of these visitor interfaces has one method for each concrete class in the defined composite hierarchy, named [*visitorMethodPrefix*][*className*].

Also automatically generated are two abstract visitor classes that simplify creating visitors that process data in a depth-first fashion: *rootClass*`DepthFirstVisitor<T>` and *rootClass*`DepthFirstVisitor_void`. These visitors, patterned off the depth-first visitors generated by JTB [39], allow subclasses to implement only the part of each

case method that actually processes the subcomputed results and returns a value, without having to explicitly include the recursive calls. They also simplify visitor implementation by having each case's visit method default to calling the superclass's visit methods. (See the example output in Figure 5.2 and the simple evaluation visitor in Figure 5.3 for better illustration.)

TabPrintWriter This helper class is generated, verbatim, in every ASTGen output. It is used in the implementation of the `.toString` methods in the generated classes, to simplify the output of nicely indented trees.

5.1.3 A first attempt at a parser

With an abstract syntax defined, the next step needed to effectively process Java source is a parser that produces these ASTs. Our first inclination was to use a parser generator, particularly one that already had a defined Java grammar. Since a student in our research group had already adapted the JavaCC grammar for Java to support GJ-style generics, we decided to start there, adapting the grammar to produce our new AST. (It previously generated the AST that was produced by JTB [39].)

This effort produced a working parser for Generic Java, written in Java, that produced our abstract syntax. The parser is available under the LGPL [40] from `http://javalangtools.sourceforge.net`.

This parser, while perfectly useful for many language projects, proved difficult to extend towards our goals for language levels. Error handling is an important part of the language levels implementation; at each language level, the error messages must fit

the context of the language definition at that point. Further, since the language itself is changing at every level — sometimes entirely new syntactic forms are introduced, like inner classes — the parser would have to have many parameters to accommodate all of the different language levels. Doing this in one parser, we realized, would be very difficult to develop and test, and maintaining and extending it would be daunting.

5.1.4 JExpressions: A more flexible parser framework

Instead, we decided to add a third stage to the processing process, between tokenizing and the actual parsing. This stage parses the tokens into an abstract syntax called JExpressions that represents the lexical nesting structure of the code, but little else. Because this representation is more general than all language levels, each language level parser is implemented as a processor that walks this representation.

JExpressions are the Java analogue of S-expressions [41] in Lisp. Because Lisp has a simple, regular syntax directly tied to S-expressions, S-expressions are much simpler than JExpressions. JExpressions model the gross structure of a Java-like language without making any additional syntactic committments. They account for the code's lexical structure — braced, bracketed and parenthesized blocks; comments; and strings — but do not assign any specific meaning from the words themselves.

The abstract syntax to represent JExpressions was created using ASTGen, and its source is presented in Appendix B. Also, the original GJ parser written in JavaCC was adapted to create a JExpression parser. The parser and AST sources are available under the LGPL license from `http://javalangtools.sourceforge.net`.

With this framework in place, the development of individual parsers for each language level is substantially simplified. Since they can now each be a visitor over a JExpression tree, they can share code through inheritance without having to all work as one parser. Also, because JExpressions encompass the overall structure of the code, this complexity need not be handled by the parsers themselves.

JExpressions could be useful in other contexts as well. Since they make few commitments to the syntax, JExpressions could be used as the basis for parsers for other extended forms of Java or any language with a Java-like language.* A mutable extension of JExpressions would also be suitable for a shallow representation of syntax that must be dynamically updated. In particular, this could function as a replacement for DrJava's reduced model implementation, since JExpressions provide the requisite information on the block structure (for brace matching, etc.) without requiring the time to do a full parse. Also, for an editor, it is useful to be able to extract some information from the code even when it is not entirely syntacticly valid; since JExpressions encompass so little of the actual Java syntax, they would be good for this purpose.

5.1.5 Mechanism to convert AST representation back to source

Because language levels will be implemented as source-to-source translators, the final framework piece we needed was a way to convert abstract syntax back to (equivalent) concrete Java syntax. I implemented this as visitor over JavaAST in the class

*C++ and JavaScript, for example, qualify here, since they have the same commenting, quoting and block-structure constructs.

`CanonicalSourceVisitor`, which is distributed with JavaAST itself. It is called a "canonical source" visitor because it ignores spaces and comments from the original source, such that it could be used to canonicalize differently formatted sources. (I anticipate that someone in the future will implement a JavaAST visitor to produce source code that is formatted more similarly to the original source.)

The `CanonicalSourceVisitor` is a depth-first visitor over `JavaAST`. For each piece of AST, it returns a Command-pattern object, a `Piece`, that knows how to print the source for that piece. With this framework, it was possible to write the AST-to-source translation totally independently for each AST form. (Note that this implementation detail is hidden from clients, who can interact with the visitor by just calling its `.generateSource` static method.)

```
public interface Expression {
  public int getLineNumber();
  public <T> T visit(ExpressionVisitor<T> visitor);
}

/** An interface for visitors over Expression that return a value. */
public interface ExpressionVisitor<T> {
  public T forAddExpression(AddExpression that);
  public T forMultiplyExpression(MultiplyExpression that);
  public T forIntegerConstant(IntegerConstant that);
  public T forFloatConstant(FloatConstant that);
}

public abstract class ExpressionDepthFirstVisitor<T>
  implements ExpressionVisitor<T>
{
  // This method is called automatically by any forCASEOnly that
  // is not implemented explicitly.
  protected abstract T defaultCase(Expression that);

  public T forAbstractExpressionOnly(AbstractExpression that) {
    return defaultCase(that); }

  public T forBinaryOpExpressionOnly(BinaryOpExpression that, T left, T right){
    return forAbstractExpressionOnly(that); }

  public T forAddExpressionOnly(AddExpression that, T left, T right){
    return forBinaryOpExpressionOnly(that, left, right); }

  public T forMultiplyExpressionOnly(MultiplyExpression that, T left, T right){
    return forBinaryOpExpressionOnly(that, left, right); }

  public T forIntegerConstantOnly(IntegerConstant that){
    return forAbstractExpressionOnly(that); }

  public T forFloatConstantOnly(FloatConstant that){
    return forAbstractExpressionOnly(that); }

  // Implementation of ExpressionVisitor methods to
  // implement depth-first traversal omitted.
}
```

Figure 5.2 An abbreviated version of the `Expression` interface, the visitor interface, and the depth-first visitor implementation generated from the input given in Figure 5.1.

```
import expr.*;

public class EvalVisitor extends ExpressionDepthFirstVisitor<Double> {
  protected Double[] makeArrayOfRetType(int i) { return new Double[i]; };

  protected Double defaultCase(Expression that) {
    throw new RuntimeException("never called");
  }

  public Double forAddExpressionOnly(AddExpression that,
                                     Double left, Double right) {
    return new Double(left.doubleValue() + right.doubleValue()); }

  public Double forMultiplyExpressionOnly(MultiplyExpression that,
                                          Double left, Double right) {
    return new Double(left.doubleValue() * right.doubleValue()); }

  public Double forIntegerConstantOnly(IntegerConstant that) {
    return new Double(that.getValue()); }

  public Double forFloatConstantOnly(FloatConstant that) {
    return new Double(that.getValue()); }
}
```

Figure 5.3 A simple visitor to evaluate an expression, demonstrating how to use the ASTGen depth-first visitor.

5.2 A conceptual design of language levels

At Rice University, the introductory computer science class, COMP210, is taught using the Scheme language and the DrScheme programming environment. The material covered in this class has been codified into the book "How to Design Programs" [42]. Our design for language levels in Java is based on this heritage, with a similar conceptual progression and fairly analogous language level definitions. It is important to point out that many other language level progressions for Java could be implemented using the framework outlined in Section 5.1.

We have divided the language into five stages, with the final stage encompassing the entire language Java language. Our intention is that a one-semester first course taught in Java could reach Level 4 by its end (and possibly go beyond it). The idea is that each stage introduces a small number of concepts, which should be mastered by the students before continuing on to the next stage. Despite the fact that the first level omits a large portion of the Java language, it is still quite expressive, and students can certainly write real programs in this subset.

The sublanguages introduce automatic augmentations to the code written by students — namely, some methods and constructors are provided automatically, and also some field and methods modifiers are automatically introduced. An overview of these augmentations is provided in Table 5.1 on page 63. Each language level accepts only a subset of the full Java language. These subsets are defined by a set of restrictions on the language. These restrictions are detailed in Table 5.2 on page 64. Note

that though the language levels are presented below as whole entities, they may be customized to turn on or off specific restrictions or augmentations.

The conceptual progression from Level 1 to full Java is outlined in the sections below.

5.2.1 Level 1: Simple Functional Java

Level 1 is a purely functional subset of Java that allows students to be gently introduced to object-oriented program construction involving "algebraeic types." It supports the core object-oriented constructs of Java — abstract and concrete classes, interfaces, inheritance, dynamic methods and fields — but it simplifies many things.

For beginners, it's easier to not have to worry about multiple source files. Therefore, in Level 1 all classes for one project are in one source file, and all classes are therefore in the default package. This means the `package` statement is unneeded. Also, we also do away with explicit `import` statements, allowing us to entirely ignore the issue of packaging for the time being. If an instructor wants to provide support classes for students to use, these classes must also be in the default package.

As implied by the description of this level as "functional," there is no mutation of field or variable values; they are automatically declared `final`. Thus, each field must be given a value exactly once. We exploit this requirement to introduce another simplification to reduce the size of our initial language: There need be no explicit constructors, since the form of every constructor is to simply take as parameters the values for all fields. Students are taught that to construct an instance of class `C`,

one uses the expression `new C(`p_1`, `p_2` ...`p_n`)`, where p_i is the value of the ith field defined in class `C`.* In addition, following directly from the lack of mutation, arrays and loops are not supported in Level 1 Java.

Another substanatial simpliciation made at Level 1 is that the `null` keyword is not supported, which allows us to eliminate the possibility of null-pointer exceptions.*

We further simplify the way objects are used by making object identity non-observable. That is, we make it appear that two objects created by identical constructor calls are equal. We do this by disallowing the use `==` and `!=` on reference types and by automatically implementing a `.equals` method for all student-written classes. This `.equals` does a member-wise equality test (as opposed to the default version which does an instance identity test). We also implement a consistent `.hashCode` method, although this is not likely to be needed by beginners, in order to not violate the requirement that the two methods are consistent. Finally, we automatically implement `.toString` methods for student classes to return a String that corresponds precisely to the constructor call that would create an equivalent object.

Level 1 also hides the true complexity of Java's type system by making it appear that primitive types are a part of the object hierarchy. To do this, primitive values are boxed and unboxed as needed.

Another Java complexity that is avoided for the moment is `static` fields and methods. Because DrJava's REPL obviates the need to define `public static void`

*This is analogous the way Scheme's `define-struct` automatically creates constructors.

*Note that instructors must take care to not introduce students to library functions that can possibly return null.

`main(String[])` all over the place, `static` methods are not needed. And though `static` fields are useful for implementing the Singleton design pattern, they do not need to be in the initial core language.

Other highlights of Level 1 Java include:

- Polymorphism is limited to the declaration of abstract methods and to final implementations of these methods. That is, there is no overriding of concrete methods. This simplification is to prevent students from inadvertently overriding methods.

- There are no explicit visibility specifiers for classes, methods and fields. Everything is implicitly `public`, except classes which are in the default package and package visible. Although at first glance it violates the notion of encapsulation to make fields public, this is not as bad as it seems because there is no mutation. So, clients of a class can *see* its fields, but since they can't modify them, they can't do any harm. The benefit here — avoiding a discussion of visibility, which is hard to motivate early on when programs are small — outweighs this cost. Also, making everything public now allows for a good explanation (at Level 3) of the importance of making data private once mutation has been introduced.

- All classes must have explicit `extends` clause, even if they directly subclass `Object`. This simple restriction makes it clear to students that all classes (except the primordial `Object`) have superclasses. It's more important to be consistent than terse for beginners.

- The only primitive types available are `int`, `double` and `boolean`. The others are not important for beginners and simply get in the way.

- There are no inner classes of any kind. This is prevented in the parser, which allows error messages to tell the user when a class is missing a close brace instead of claiming that there was some kind of error in an inner class definition.

- There are no `void` methods, since they would be of limited value without mutation.

5.2.2 Level 2: Functional Java Part 2

Level 2 retains the immutability of data but introduces a number of the complexities hidden from the Level 1 student. Constructors must be explicitly defined, and each constructor must give a value to every field. Also, the Java package system is introduced, including the `package` and `import` keywords. Now students can build projects using multiple source files. Java's exception system and the `try/catch/finally` construct is also introduced.

Two design patterns, Singleton and Command, are intended to be teachable at this stage, so the requisite language constructs must also be opened up. To enable the use singletons, `static` fields and methods are introduced. (But even `static` fields are automatically `final`!) This also enables the definition of class-wide constants. The Command pattern is enabled through the introduction of dynamic inner classes.*

*Anonymous inner classes may be made available as well, but it is an open debate whether they are more or less confusing to beginners.

One other new feature introduced in Level 2 is downcasting reference types. One reason for introducing this is to allow the use of generic data structures.† Also available at Level 2 is the complementary operation, `instanceof`, although its use is discouraged except for cases where it is truly needed. (Often, uses of `instanceof` could be better replaced by applications of dynamic dispatch.)

5.2.3 Level 3: Mutation introduced

With Level 3 comes the mutation of fields and variables, as well as the various other language features that naturally follow from mutability. Fields and local variables are now not implicitly `final`, and the `final` keyword is introduced to allow for self-imposed immutability. However, fields still must be explicitly given a value at their declaration site or in every constructor.

With mutability comes a discussion of visibility. The instructor can explain that there is a need to hide the internal components of an object, primarily to ensure that the appropriate invariants can not be violated. Therefore, no longer are all methods and fields automatically made `public`.

Also in Level 3 the introduction of object identity. The `==` and `!=` operators can now be used to check object identity, and the `null` keyword is introduced. In addition, no longer is an `.equals` method automatically provided to implement value-wise equality. Students are taught how to correctly implement `.equals` when they

†I think that an early introduction of parametric types might be a better answer — I think they can be fairly natural to beginners. But I don't think that instructors would be interested in discussing parametric types, at least not until they are in standard Java. But if we introduced parametric types, we could avoid casts for a while longer, and possibly until full Java.

need it. Note that `==` must be handled specially to work in the presence of auto-boxed primitives. In cases where `==` is applied to two boxed primitives, the language level translator must convert this into a call to `.equals` to implement value-wise equality.

Other concepts introduced at this level are `while` loops for use with the Iterator pattern, the `switch` statement, `void` methods, and `static` inner classes.

5.2.4 Level 4: Almost complete Java

Level 4 rounds out the beginner's knowledge of Java with other concepts they will need to be able to read standard Java code. As a part of this, all automatic augmentation of the code end, meaning that students have to define their own `.toString` methods. Still, a number of features are omitted, as they can be learned when the time comes.

Level 4 brings arrays, and with arrays come `for` loops. Along with these loops, the ability to alter control of the iteration through the `break` and `continue` statements is shown.

Other simplifying assumptions carried through previous levels are removed as well. Concrete methods can be overridden, and static method overloading can be used to define multiple methods with the same name.

	L1	L2	L3	L4
Automatic constructor generation	✓			
Automatic generation of `void error(String)` in each class	✓			
Automatic `import`ation of support code	✓			
Fields are automatically `public final`	✓	✓		
Methods are automatically `public`	✓	✓		
Automatic generation of `equals` and `hashCode`	✓	✓		
Non-`abstract` methods are automatically `final`	✓	✓	✓	
Automatic `.toString` that looks like constructor call	✓	✓	✓	
Automatic boxing and unboxing of primitive types, allowing primitives to appear to be in the object hierarchy	✓	✓	✓	

Table 5.1 Chart of automatic augmentations applied to the source code at each language level.

	L1	L2	L3	L4	Full
Classes (concrete and abstract)	✓	✓	✓	✓	✓
Interfaces	✓	✓	✓	✓	✓
Dynamic methods (concrete and abstract)	✓	✓	✓	✓	✓
Fields	✓	✓	✓	✓	✓
`int`, `boolean` and `double` primitive types	✓	✓	✓	✓	✓
Standard arithmatic operators	✓	✓	✓	✓	✓
`if` statement	✓	✓	✓	✓	✓
`+-*/%`, relational and boolean logical operators	✓	✓	✓	✓	✓
Explicit constructors		✓	✓	✓	✓
`package` and `import` statements		✓	✓	✓	✓
Static methods and fields		✓	✓	✓	✓
Dynamic inner classes		✓	✓	✓	✓
Exceptions		✓	✓	✓	✓
Casts and `instanceof`		✓	✓	✓	✓
Fields can have initializers at declaration site		✓	✓	✓	✓
Fields and local variables are mutable			✓	✓	✓
Explicit use of `final` modifier			✓	✓	✓
Visibility can be explicitly specified			✓	✓	✓
`null` value			✓	✓	✓
`while` loops			✓	✓	✓
Superclass can be left unspecified			✓	✓	✓
`switch` statement			✓	✓	✓
`void` methods			✓	✓	✓
`==` and `!=` operators to check object identity			✓	✓	✓
Static inner classes			✓	✓	✓
`for` and `do` loops				✓	✓
Arrays				✓	✓
`if`/`switch`/loop conditions can have side effects				✓	✓
`break` and `continue`				✓	✓
Method name overloading				✓	✓
Ability to override concrete methods				✓	✓

Table 5.2 Summary chart of language features available at each language level, excluding features available only in full Java. Those features are detailed in the next table.

	L1	L2	L3	L4	Full
`float`, `long` and `char` data types					✓
Static or instance initialization blocks					✓
`native` methods					✓
Threading primitives, including `synchronized`, `volatile`, `Thread`* classes					✓
Assignment can be done in an expression					✓
Bitwise operators					✓
Labeled statements					✓
Can define more than one variable/field in a statement					✓
Field names can be shadowed					✓
A class can contain a field and a method of the same name					✓
Implicit field initialization allowed					✓
Catch block can be empty					✓
`switch` fallthrough from case that had code					✓
Static inner classes					✓

Table 5.3 Summary chart of language features available only in full Java.

Chapter 6
Future Work

6.1 DrJava extensions

Although we never want to lose sight of DrJava's focus on simplicity, there are a number of promising extensions to DrJava that could be developed. Some of these are currently in development as student projects, The most up-to-date information on DrJava is available from its Web site, `http://drjava.sourceforge.net`.

Configurability

Many parameters inside DrJava should be made configurable, with a graphical user interface to change parameters and a way to store them permanently on disk. This project is currently underway in COMP312.

Debugger

Although the REPL is very effective for debugging programs, in some situations a lower-level debugger would be helpful. This would allow the user to set breakpoints and to single-step through the code during execution. Students in COMP312 are currently investigating integrating an existing GPL-licensed debugger into DrJava.

JUnit integration

DrJava could better support Extreme Programming (and other test-oriented development philosophies) by integrating support for running unit tests. Since JUnit [8] is the de facto standard for unit test frameworks for Java, it would be helpful to have

an integrated way to pick and run JUnit tests from within DrJava, and then to be able to click on test failures to jump to their source locations. This feature is currently under development.

Access to API documentation

It would be very helpful to be able to access the javadoc documentation for the standard libraries, and also for other libraries and even the current project. A first implementation could basically just include a way to view the HTML-formatted javadoc from within DrJava. A better integration would allow DrJava to have a custom documentation-browsing interface, with the ability to more easily search the documentation and the ability to have the documentation for a method come up automatically when the cursor is over the method call. This could be implemented on top of the existing javadoc tool or through a separate parser and interpreter.

Ability to view project in UML form

Although a complete UML editor is outside the scope of DrJava (and contrary to our philosophy of focusing on the code in teaching programming to beginners), in some cases it is helpful to be able to view a project overview in UML form. This would automatically generate a UML diagram for a project (or a selected subsection), automatically arranging the components if possible, to keep the user interaction to a minimum. Since we want to focus on the source code as the single point of control, the diagram should not allow editing; it should simply be another view over the contents of the source code.

Instrument DrJava to collect data on students' errors

Though not a feature for DrJava's users per se, I have thought that DrJava could be an excellent tool for collecting data on the problems that students face when learning to program, and how the program (particularly error messages) could be improved. To do this, DrJava could have the ability to log the errors a student encountered, along with the relevant part of the source code, and either save them for later retrieval or automatically send them to a central server over the network. (This would, of course, be configurable and protected for privacy.) Then the data could be collated and analyzed to provide insight into the kinds of errors students tend to make and how best to help them.

Features for more advanced users

Though DrJava should remain accessible to novices, it could also be a good environment for more advanced users as well. There are a number of features that could be added to DrJava to support more advanced users.

Ant integration

Ant [36] is a build process automation tool, similar to Make, that is focused on Java. Many larger projects use Ant to control compilation, testing, and source repository access. A good integration of Ant with DrJava would be more than just running Ant targets and putting their output into a text pane. Instead, each command should be supported natively. For example, when compiling through the Ant script, the compiler errors should be shown in the same user interface as errors are shown

when compiling in DrJava without Ant. Similarly, other commonly used tasks (JUnit and CVS, for example) should have direct support in the interface.

"Code completion"

Many commercial IDEs support the ability to complete keywords, identifiers and method calls automatically or when requested by the user. This would be a useful feature to support. Doing it well would require DrJava to parse the source code, at least to some degree, to discern the meaning of various parts of the program text to discover how to complete them. Though this is a useful feature for advanced users, it might confuse beginners, who might rely too heavily on the automatic completion, even when it is not inserting the code that is correct for the situation.

Refactoring tools

DrJava would be an excellent framework for implementing some tools to automate various refactorings of source code. Such an implementation could be based on the parser and abstract syntax framework outlined in Section 5.1. Refactorings could include all those proposed in [43] and also some additional ones to support generic types, like "Introduce Type Parameter."

6.2 Language levels

Implement the language levels design

The design presented in Section 5.2 should be implemented using the framework outlined in Section 5.1. This work will consist primarily of writing the JExpression-to-AST parse visitors for each language level and the visitors to implement the re-

strictions and augmentations needed at each level. (A restriction is a visitor mapping AST to `boolean`, while an augmentation maps AST to AST.)

Design a language level progression that corresponds to the AP subset

The language level concept, and the framework presented in this thesis, can be used to develop multiple different actual language levels. As the high school Advanced Placement exam is moving to Java starting in 2003-2004, it might be useful to implement a different progression of language levels more precisely aimed at the AP Java subset outlined in [14]. While this set of levels would have much in common with the design presented in this thesis (and could share much of the implementation), some high school teachers might be more comfortable with a set of levels that were more directly focused on the AP.

Appendix A
Java AST source

The ASTGen source for the Java AST framework is presented below. The most current version of this source is available from `http://javalangtools.sourceforge.net`.

```
// Source for AST classes for Java to be run through ASTGen
// $Id: appendix1.tex,v 1.4 2002/04/21 05:45:40 bstoler Exp $
visitmethod visit;
visitormethodprefix for;
tabsize 2;
outputdir tree;
allownulls no;

package edu.rice.cs.javaast.tree;
import edu.rice.cs.javaast.Visibility;
import edu.rice.cs.javaast.ClassModifier;
import edu.rice.cs.javaast.SourceInfo;

begin ast;
interface JavaAST(SourceInfo sourceInfo);
  interface BlockStatementI();
  interface AllocationQualifierI();
  interface ResultTypeI();
  interface VariableInitializerI();
  interface ForInitI();
  interface ForUpdateI();
  interface ForConditionI();

  // base type for all things that define a new class,
  // incl local class, anon inner class, etc.
  interface ClassDefBaseI(ReferenceType superclass, MethodDef[] methods,
                          FieldDef[] fields, Initializer[] initializers,
                          InnerDefI[] inners);

  interface TypeDefI(String name, Visibility visibility,
                     TypeParameter[] typeParameters, boolean strictfp);
    interface InnerDefI();
```

```
    interface DynamicInnerDefI();
    interface StaticInnerDefI();

abstract JavaASTBase(ignoreForEquals SourceInfo sourceInfo);
  CompilationUnit(PackageDeclaration package, ImportDeclaration[] imports,
                  TypeDefI[] classes);
  PackageDeclaration(String name);

  abstract ImportDeclaration();
    ClassImportDeclaration(String className);
    PackageImportDeclaration(String packageName);

  EmptyForCondition() implements ForConditionI;
  EmptyForInit() implements ForInitI;
  EmptyForUpdate() implements ForUpdateI;

  ArrayInitializer(VariableInitializerI[] items)
    implements VariableInitializerI;
  NoVariableInitializer() implements VariableInitializerI;

  NoAllocationQualifier() implements AllocationQualifierI;

  TypeParameter(TypeVariable variable, ReferenceType bound);

  FormalParameter(String name, Type type);
    FinalFormalParameter();

  abstract Type() implements ResultTypeI;
    PrimitiveType(String name); // should there be subtype per type?
    ArrayType(Type elementType);
    abstract ReferenceType();
      MemberType(ReferenceType left, ReferenceType right);
      ClassOrInterfaceType(String name, Type[] typeArguments);
      TypeVariable(String name);

  VoidResult() implements ResultTypeI;

  abstract SwitchCase(BlockStatementI[] code);
    LabeledCase(Expression label);
    DefaultCase();

  StatementExpressionList(StatementExpression[] statements)
    implements ForInitI, ForUpdateI;
```

```
LocalVariableDeclarationList(LocalVariableDeclaration[] declarations)
  implements ForInitI;

CatchBlock(FormalParameter exception, Block block);

abstract TypeDefBase(String name, Visibility visibility,
                     TypeParameter[] typeParameters,
                     boolean strictfp) implements TypeDefI;
  ClassDef(ClassModifier modifier, ReferenceType superclass,
           ReferenceType[] interfaces, ConstructorDef[] constructors,
           MethodDef[] methods, FieldDef[] fields,
           Initializer[] initializers,
           InnerDefI[] inners) implements ClassDefBaseI;

    abstract InnerClassDef() implements InnerDefI;
      DynamicInnerClassDef() implements DynamicInnerDefI;
      StaticInnerClassDef() implements StaticInnerDefI;

  InterfaceDef(ReferenceType[] superinterfaces,
               AbstractMethodDef[] methods,
               FinalStaticFieldDef[] fields,
               StaticInnerDefI[] inners);
    InnerInterfaceDef() implements StaticInnerDefI;

abstract FieldDef(String name, Visibility visibility, Type type,
                  VariableInitializerI initializer, boolean volatile,
                  boolean transient);
  InstanceFieldDef();
    FinalInstanceFieldDef();
  StaticFieldDef();
    FinalStaticFieldDef();

abstract MethodDef(String name, Visibility visibility,
                   ResultTypeI returnType, FormalParameter[] parameters,
                   TypeParameter[] typeParameters, ReferenceType[] throws,
                   boolean synchronized);
  AbstractMethodDef(boolean strictfp);
  abstract ConcreteMethodDef(boolean final);
    NativeMethodDef(boolean static);
    abstract NonNativeConcreteMethodDef(boolean strictfp, Block code);
      InstanceMethodDef();
      StaticMethodDef();
```

```
ConstructorDef(String name, Visibility visibility,
               FormalParameter[] parameters, ReferenceType[] throws,
               ConstructorBody body);

ConstructorBody(BlockStatementI[] statements);
  ConstructorBodyWithExplicitConstructorInvocation(ConstructorInvocation
      invocation);

abstract ConstructorInvocation(Expression[] arguments);
  ThisConstructorInvocation();
  UnqualifiedSuperConstructorInvocation();
  QualifiedSuperConstructorInvocation(Expression qualifier);

abstract Initializer(Block code);
  InstanceInitializer();
  StaticInitializer();

LocalVariableDeclaration(Type type, String name,
                         VariableInitializerI initializer)
                       implements BlockStatementI;
  FinalLocalVariableDeclaration();

LocalClassDef(String name, ClassModifier modifier,
              TypeParameter[] typeParameters, boolean strictfp,
              ReferenceType superclass, ReferenceType[] interfaces,
              ConstructorDef[] constructors, MethodDef[] methods,
              FieldDef[] fields, Initializer[] initializers,
              InnerDefI[] inners) implements BlockStatementI,
                                             ClassDefBaseI;

abstract Statement() implements BlockStatementI;
  LabeledStatement(String label, Statement statement);
  Block(BlockStatementI[] statements);
  EmptyStatement();
  StatementExpression(Expression expression);
  SwitchStatement(Expression test, SwitchCase[] cases);
  IfThenStatement(Expression test, Statement conseq);
    IfThenElseStatement(Statement alt);
  WhileStatement(Expression condition, Statement code);
  DoStatement(Statement code, Expression condition);
  ForStatement(ForInitI init, ForConditionI condition, ForUpdateI update,
               Statement code);
  BreakStatement();
```

```
    LabeledBreakStatement(String label);
  ContinueStatement();
    LabeledContinueStatement(String label);
  abstract ReturnStatement();
    VoidReturnStatement();
    ValueReturnStatement(Expression value);
  ThrowStatement(Expression thrown);
  SynchronizedStatement(Expression lockExpr, Block block);
  TryCatchStatement(Block tryBlock, CatchBlock[] catchBlocks);
    TryCatchFinallyStatement(Block finallyBlock);

abstract Expression() extends VariableInitializerI, AllocationQualifierI,
                              ForConditionI;
  ConditionalExpression(Expression test,Expression conseq, Expression alt);
  InstanceOfExpression(Expression left, Type type);
  CastExpression(Expression value, Type type);

  abstract BinaryOpExpression(Expression left, Expression right);
    OrExpression();
    AndExpression();
    BitOrExpression();
    BitXorExpression();
    BitAndExpression();
    EqualsExpression();
    NotEqualsExpression();
    LessThanExpression();
    LessThanOrEqualExpression();
    GreaterThanExpression();
    GreaterThanOrEqualExpression();
    LeftShiftExpression();
    RightShiftExpression();
    UnsignedRightShiftExpression();
    AddExpression();
    SubtractExpression();
    MultiplyExpression();
    DivideExpression();
    RemainderExpression();

    abstract Assignment();
      NormalAssignment();
      MultiplyAssignment();
      DivideAssignment();
      RemainderAssignment();
```

```
    AddAssignment();
    SubtractAssignment();
    LeftShiftAssignment();
    RightShiftAssignment();
    UnsignedRightShiftAssignment();
    BitAndAssignment();
    BitOrAssignment();
    BitXorAssignment();

abstract UnaryOpExpression(Expression operand);
  UnaryPlusExpression();
  UnaryMinusExpression();
  UnaryNotExpression();
  UnaryBitNotExpression();
  PreIncrementExpression();
  PreDecrementExpression();
  PostIncrementExpression();
  PostDecrementExpression();

abstract Literal();
  StringLiteral(String value);
  IntegerLiteral(int value);
  LongLiteral(long value);
  FloatLiteral(float value);
  DoubleLiteral(double value);
  CharLiteral(char value);
  BooleanLiteral(boolean value);
  NullLiteral();
  ThisLiteral();

abstract AllocationExpression();
  ArrayAllocation(ArrayType type, Expression[] dimensions,
                  VariableInitializerI initializer);

  // qualifier is for weird a.new Foo() syntax.
  InstanceAllocation(ReferenceType type, AllocationQualifierI qualifier,
                     Expression[] arguments);

  AnonymousInnerClass(ReferenceType superclass,
                      AllocationQualifierI qualifier,
                      Expression[] superArguments, MethodDef[] methods,
                      FieldDef[] fields, Initializer[] initializers,
                      InnerDefI[] inners) implements ClassDefBaseI;
```

```
        Name(String value);

        DotThis(Expression left);
        DotClass(ResultTypeI type);
        ArrayAccess(Expression left, Expression index);
        MemberAccess(Expression left, Name right);
        SuperMemberAccess(Name right);
        MethodInvocation(Expression left, Expression[] arguments);

end;
```

Appendix B
JExpression AST source

The ASTGen source for the JExpression framework is presented below. The most current version of this source is available from `http://javalangtools.sourceforge.net`.

```
// Source for AST classes for JExpressions
// $Id: appendix2.tex,v 1.3 2002/04/25 19:33:21 bstoler Exp $
visitmethod apply;
visitormethodprefix for;
tabsize 2;
outputdir tree;
allownulls no;

package edu.rice.cs.jexpression.tree;
import edu.rice.cs.jexpression.*;

begin ast;
abstract JExpression();
  JExpressionList(JExpressionAtom[] items);

  abstract JExpressionAtom(SourceInfo sourceInfo);
    Word(String name);
    Numbers(String chars);
    PunctuationMark(char char);

    SquareBracketed(JExpressionList list);
    Braced(JExpressionList list);
    Parenthesized(JExpressionList list);
    DoubleQuoted(String value);
    SingleQuoted(String value);

    abstract CommentOrWhitespace();
      Whitespace(String text);
      SingleLineComment(String text);
      MultiLineComment(String text);
      JavadocComment(String text);
end;
```

References

1. Recommendations of the AP Computer Science Ad Hoc Committee. At `http://cbweb2s.collegeboard.org/ap/pdf/adhoc_report_surveys.pdf`.

2. Computer
   Science A & AB: Introduction of Java in 2003-2004. At `http://apcentral.collegeboard.com/members/article/1,1282,151-165-0-4614,00.html`

3. S. Bloch. Scheme and Java in the first year. In *The Journal of Computing in Small Colleges* 15 (5). May 2000, 157-165.

4. R. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In International Symposium on Programming Languages: Implementations, Logics, and Programs, 1997, 369-388.

5. E. Sandewall. Programming in an interactive environment: the "Lisp" experience. In *Computing Surveys*, 10(1), March 1978, 35-71.

6. Free Software Foundation. The General Public License. At `http://www.gnu.org/licenses/gpl.html`.

7. K. Beck. *Extreme Programming Explained*, Addison-Wesley, 2000.

8. E. Gamma and K. Beck. JUnit. At `http://www.junit.org`.

9. J. Rosenberg and M. Kölling. BlueJ. At `http://www.bluej.org`.

10. Holt Software. Ready to Program with Java^{TM}Technology Home Page. At `http://www.holtsoft.com/ready`.

11. J.N.Patterson Hume and Christine Stephenson. *Introduction to Programming with Java*. Holt Software Associates Inc., 2000.

12. The JJ Home Page. At `http://www.publicstaticvoidmain.com`.

13. Description of JR, JJ languages. At `http://www.publicstaticvoidmain.com/jj_jjBNF_JJandJr.htm`

14. Computer Science AP (AB) Java subset `http://apcentral.collegeboard.com/members/article/1,1282,151-165-0-8390,00.html`

15. Jr to JJ (Java) Translation Definition. At `http://www.publicstaticvoidmain.com/jj_jjBNF_JJandJr.htm`.

16. Java Development Environment for Emacs. At `http://jdee.sunsite.dk`.

17. JEdit. At http://www.jedit.org.

18. Jext. At http://www.jext.org.

19. JBuilder. At http://www.borland.com/jbuilder.

20. JDeveloper. At http://www.oracle.com/tools/jdeveloper.

21. Forte for Java. At http://www.sun.com/forte/ffj.

22. Netbeans. At http://www.netbeans.org.

23. Eclipse. At http://www.eclipse.org.

24. S. Hillion. DynamicJava. At http://koala.ilog.fr/djava.

25. BeanShell. At http://www.beanshell.org.

26. E. Roberts. An Overview of MiniJava. In Technical Symposium on Computer Science Education, 2001, 1-5.

27. Jython. At http://www.jython.org.

28. Jacl. At http://tcl.activestate.com/software/java.

29. Kawa. At http://www.gnu.org/software/kawa

30. Rhino. At http://www.mozilla.org/rhino

31. Java Thread Primitive Deprecation. At http://java.sun.com/products/jdk/1.2/docs/guide/misc/threadPrimitiveDeprecation.html.

32. A. Rudys and D. Wallach. Termination in Language-based Systems. In *ACM Transactions on Information and System Security* 5 (2), May 2002, *to appear.*

33. Java Developer Connection Bug Database. Bug ID: 4416624 multiple JVM runtimes do not share memory between themselves. At http://developer.java.sun.com/developer/bugParade/bugs/4416624.html.

34. Prototype for JSR014: Adding Generics to the Java^{TM} Programming Language v. 1.2. At http://developer.java.sun.com/developer/earlyAccess/adding_generics.

35. Concurrent Versions System. At http://www.cvshome.org.

36. Ant. At http://jakarta.apache.org/ant.

37. E. Allen. Efficient Implementation of Run-time Generic Types for Java. Master's thesis, May 2002.

38. G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *OOPSLA Proceedings*, October 1998.

39. K. Tao and W. Wang. Java Tree Builder Documentation. At `http://www.cs.purdue.edu/jtb/docs.html`

40. Free Software Foundation. The Lesser General Public License. At `http://www.gnu.org/licenses/lgpl.html`.

41. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, p. 184-195, April 1960.

42. M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Computing and Programming*. MIT Press, 2001. Available online at `http://www.htdp.org`.

43. M. Fowler. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.