DrJava: A lightweight pedagogic environment for Java

Eric Allen, Robert Cartwright, and Brian Stoler
Rice University
6100 Main St., MS-132
Houston, TX 77005-1892
{eallen,cork,bstoler}@rice.edu

September 7, 2001

Abstract

DrJava is a pedagogic programming environment for Java that enables students to focus on designing programs, rather than learning how to use the environment. The environment provides a simple interface based on a "read-eval-print loop" that enables a programmer to develop, test, and debug Java programs in an interactive, incremental fashion. This paper gives an overview of DrJava including its pedagogic rationale, functionality, and implementation.

1 Introduction

Teaching object-oriented programming in Java to beginning students is hard. 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. Students can only learn so many things at a time, so the more time they spend wrestling with the mechanics, the less time they spend learning the concepts. To address this issue, we have implemented a development environment called DrJava that gently introduces students to the mechanics of writing Java programs and leverages the student's understanding of the language itself (which must be assimilated anyway) to provide powerful support for developing programs.

DrJava supports a transparent programming interface designed to minimize the "intimidation factor" that beginning students experience when confronted with the task of writing code. The transparent interface consists of a window with two panes:

1. An *interactions pane*, where the student can input Java expressions and statements and immediately see their results.

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

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.

DrJava is a standard Java 1.3 application (a **.jar** file) available on the web for downloading at **http://www.cs.rice.edu/~javaplt/drjava**. Although DrJava is a new system that is still under active development, it is already being used in introductory Java courses at several high schools and universities including our own institution.

2 Features that focus on the language

2.1 The Interactions Window

The interactions window provides a "read-eval-print loop"[8] that enables the student to evaluate Java expressions and statements including statements that define new variables. For several decades, most implementations of functional programming languages like Lisp, Scheme, and ML, have supported a read-eval-print loop (REPL) to facilitate incremental program development. A REPL provides a conceptually simple, yet powerful 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.

Java is the first mainstream programming language that is well-suited to supporting a REPL interface. The compilation model for most mainstream languages makes it difficult, if not impossible, to access elements of compiled code from an interactive interpreter. In such languages, supporting a REPL requires building a separate interpreter that is completely independent of the compiler. All computations performed using the REPL are executed entirely by the separate interpreter. Not only does this implementation scheme involve implementing the entire language twice–once with a compiler and once with an interpreter–it also introduces a troubling consistency problem. Does the interpreter implement the same semantics as the compiler? Can it interface to the same binary libraries as the compiler? Past experience with supplementary interpreters and fast "checkout" compilers for compiled languages strongly suggest the answer is "no" [1, 2, 9]. This problem is compounded in computing environments where multiple vendors provide compilers for the language because these compilers typically are not semantically equivalent.

Because Java dynamically loads program classes and provides reflection facilities for inspecting and accessing the loaded code base, it is possible to implement a read-eval-print loop for Java that dynamically loads and accesses compiled class files as needed. In this way, the consistency issues mentioned above are neatly avoided. Class files generated by any valid Java compiler can be loaded as necessary during interpretation.

In DrJava, we have incorporated an extension of DynamicJava[5], a freely available Java interpreter, to provide fully integrated access to a read-eval-print loop during program development. We extended DynamicJava's class loader (through subclassing) to allow the class definitions in the definitions pane of DrJava to be reloaded each time that they are compiled. The default class loader for Java only allows classes to be loaded once.

Testing and debugging The interactions window of DrJava also provides a simple, yet powerful 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.

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.

2.2 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.

As a demonstration of how DrJava performs brace matching, assume that definitions pane contains the following program text

```
public class C {
  ...
}
```

where the two braces shown match. If the user comments out the closing brace, DrJava immediately finds the new closing brace. Most development environments completely or partially ignore the impact of comment blocks on brace matching.

Next, consider a definitions pane containing the following program text:

```
/*
public class C {...}
*/
```

If the user deletes the first slash, DrJava immediately recognizes that the definition of class C is visible program text and highlights it accordingly. Again, most development environments will not change the highlighting of a line until that line is actually edited.

DrJava immediately updates (at the granularity of every keystroke) the highlighting of commented and quoted sections of the code, shielding students from subsequent surprises when they compile their code.

2.3 The Integrated Compiler

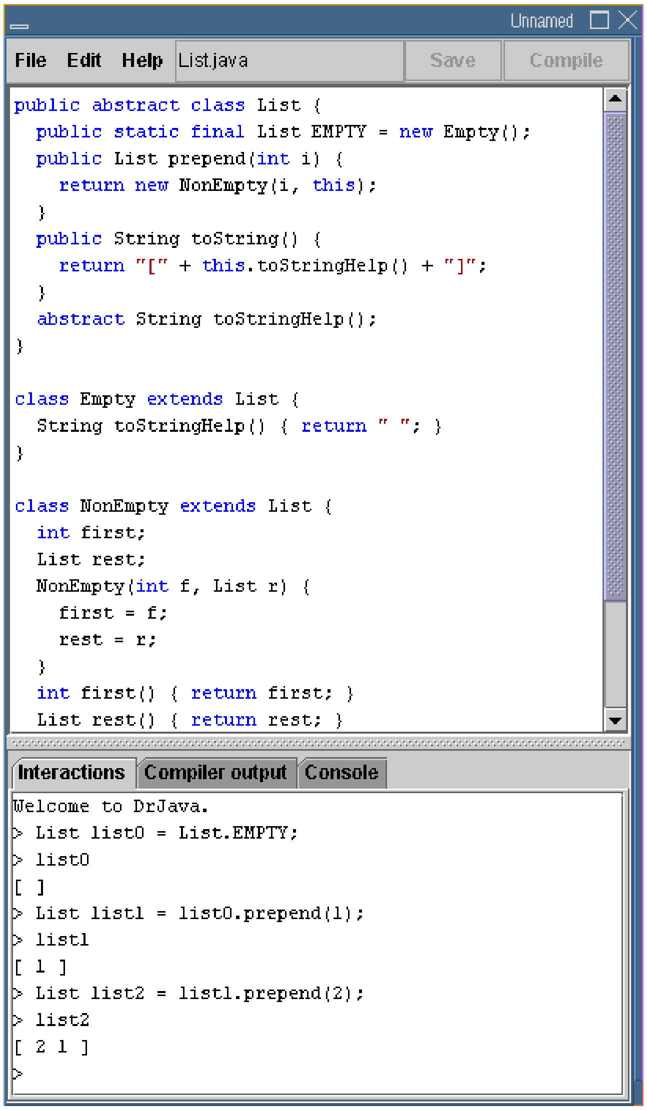
For the sake of simplicity, a Java compiler is bundled with DrJava. The compiler is integrated with the source editor, allowing the student to see the locations of compiler errors in their source simply by clicking on them. This prevents the student from having to interact with the compiler as a separate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

3 Using DrJava

To see how DrJava can be used to develop a new Java program, let's work through a simple example. Suppose we want to write a small class representing an immutable list of ints. We could do this by typing the following code into the definitions pane:

```
public abstract class List {

  public static final List EMPTY = new Empty();

  public List prepend(int i) {
    return new NonEmpty(i, this);
  }

  public String toString() {
    return "[" + this.toStringHelp() + "]";
  }

  abstract String toStringHelp();
}


class Empty extends List {

  String toStringHelp() { return " "; }

}


class NonEmpty extends List {

  int first;
  List rest;

  NonEmpty(int f, List r) {
    first = f;
    rest = r;
  }

  int first() { return first; }

  List rest() { return rest; }

  String toStringHelp() {
    return " " + first + rest.toStringHelp();
  }

}
```

After the user defines this program (using the compiler errors window to uncover any syntax or type errors along the way) he can use the interactions window to examine its functionality, as shown in the screen shot below:



In this way, the programmer can call various methods in the program with a variety of inputs, without ever having to recompile the code. This functionality has tremendous advantages for beginning students. Students can interact with their Java code without ever leaving the environment, allowing them to concentrate on program design instead of manipulating a variety of tools with inconsistent interfaces. In fact, through the use of the read-eval-print loop, student programs can accept input and display results without the students having to use explicit I/O or GUI facilities!

4 Implementing DrJava

Paul Graunke, a Rice graduate student, developed an early prototype of DrJava with a REPL based on incremental compilation. We abandoned that implementation because the response time of the REPL was too slow. During the summer of 2001, the authors wrote the current version of DrJava with the help of two undergraduates and one graduate of the class of 2000. The rapid development of a production quality system was made possible through the use of the Extreme Programming methodology. The bulk of the programming was done in pairs, and significant attention was paid to thorough unit testing of the code using the JUnit tool[4]. In fact, of the approximately 16,000 lines of code comprising the current release (excluding DynamicJava), 38% of the code consists of unit tests. We have found this base of unit tests extremely useful as a safety net in extending the existing code base. It has also served as a form of documentation: when reading over a block of code written by another developer, it is useful to refer to the unit tests he wrote for that code to see how he intends it to be used. The unit tests are also an integral part of our release process: No changes can be committed to the repository unless all unit tests pass.

In addition to the use of ubiquitous unit testing, we started using DrJava to develop DrJava as soon as the implementation reached the "alpha" test stage. As a result, we have been able to discover a few of the bugs that escaped past the unit tests. When a new bug is found in this way, a new unit test is added to check for the erroneous behavior, and the program is fixed to pass this new test.

5 Related Work

In developing DrJava, we were influenced by two related streams of work on pedagogic programming environments: DrScheme[3] and BlueJ[7].

5.1 DrScheme

DrScheme is an integrated development environment for Scheme with a transparent programming interface similar to DrJava. 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 and has served as a model for DrJava.

5.2 BlueJ

BlueJ is a research project at Monash University that provides an integrated Java environment for teaching beginning students[7]. BlueJ supports the interactive development of programs using UML diagrams. BlueJ also provides a GUI interface for creating instances of classes and calling methods on them. While DrJava and BlueJ both emphasize interactive software development, DrJava focuses on supporting a single medium for describing programs, namely program text. In contrast, BlueJ describes programs using both UML diagrams and text which makes the environment more complex.

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.

6 Directions for Future Extension

The authors, along with several undergraduate students, are actively working on extending DrJava (i) to provide even more assistance to beginning programmers and (ii) to provide the tools and facilities required to support programming in the large.

To help beginners, we plan to implement a hierarchy of sublanguages of Java (akin to DrScheme's language levels) to include progressively more features of the language. Language levels shield students from the full complexity of the language while still allowing them to focus on learning to write programs. This approach to reducing the complexity of introductory programming stands in sharp contrast to alternatives that rely on tools that hide program text behind graphical notation such as UML diagrams and produce code stubs that students cannot modify and may not understand. Language levels also may enable the development environment to perform more precise static checking. Within a sublanguage of Java (such as Java with immutable objects), important invariants may hold that fail in general for full Java. The environment can leverage these constraints to perform more helpful and descriptive syntax checking, as well as context-sensitive checking such as efficient null-pointer analysis and static verification of casting operations.

To support programming in the large, we have begun work on building a conventional debugger in DrJava, using the Java Platform Debugger Architecture, and integrating it with the environment. We also plan to integrate support for unit testing, allowing the user to pop to the locations of unit test failures in a manner similar to that currently provided for compilation errors.

7 Conclusion

By leveraging the student's understanding of the Java language, DrJava provides a simple yet powerful environment for developing Java programs. Since DrJava has been developed through the use of "extreme programming" methodology, it is unusually robust and flexible–particularly for a software system developed primarily by students. During the coming year, we anticipate building on this robust base to provide even more support for beginning programmers and to add the facilities required to support programming in the large.

8 Acknowledgements

The authors would like to thank undergraduate students Jonathan Bannet, Ben Vernot, and graduating senior Mike Yantosca for their help as summer interns on the DrJava project.

References

[1] R. Conway, T. Wilcox. Design and Implementation of a Diagnostic Compiler for PL/I. In *Communications of ACM*, 16 (3), March 1973, 169-179.

[2] P. Cress, P. Dirksen, J. Graham. *Fortran IV With WATFOR and WATFIV.* Prentice Hall, Englewood Cliffs, New Jersey, 1970.

[3] 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.

[4] E. Gamma and K. Beck. JUnit. At http://www.junit.org.

[5] S. Hillion. DynamicJava. At http://koala.ilog.fr/djava.

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

[7] J. Rosenberg and M. Kölling. BlueJ. At http://www.bluej.org.

[8] Erik Sandewall. Programming in an interactive environment: the "Lisp" experience. In *Computing Surveys*, 10(1), March 19781, 35-71.

[9] P. Shantz, R.German, J.Mitchell, R. Shirley, C. Zarnke. WATFOR–The University of Waterloo FORTRAN IV compiler. In *Communications of the ACM* 10 (1), January, 1967, 41-44.

[10] W. Teitelman, L. Masinter. The Interlisp programming environment. In *Computer* 14 (4), March 1981, 25-34.