RICE UNIVERSITY

A Pedagogic Programming Environment for Java that Scales to Production Programming

by

Charles S. Reis

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

Master of Science

Houston, Texas

April, 2003

A Pedagogic Programming Environment for Java that Scales to Production Programming

Charles S. Reis

Abstract

This thesis describes extensions to the DrJava development environment that make it suitable for production programming. DrJava is an effective tool for teaching introductory programming skills in Java, and its simplicity is a desirable characteristic for projects of any size. To better support the development of large projects in DrJava, a carefully selected suite of features has been added to the environment. To facilitate interoperation with professional development environments, a plug-in supporting the DrJava interface has been written for the Eclipse environment. As a result of this work, DrJava has become an appropriate tool for teaching production programming skills in an academic environment, without sacrificing its original goals.

Acknowledgments

I would like to thank my adviser, Robert “Corky” Cartwright, whose passionate views on software development have greatly influenced me as a programmer and as a project manager. His leadership and feedback have been essential to DrJava’s success.

I am very fortunate to have worked with Eric Allen, for his insight, discipline, and continuous encouragement. I would also like to thank Brian Stoler, whose lead I have followed and whose guidance and continued contributions have been immensely beneficial.

I would like to express my appreciation for the advice and feedback provided by Walid Taha and Peter Druschel, as well as the enthusiastic support of Zung Nguyen. Finally, I would like to thank the numerous undergraduate and graduate students at Rice whose substantial contributions to DrJava have been invaluable for this work.

Contents

Illustrations

Tables

Chapter 1

Introduction

Integrated development environments (IDEs) for Java are designed to simplify the task of writing Java programs. IDEs provide a graphical interface to a suite of development tools, including source code editors and compilers. Some Java IDEs are targeted at supporting programming instruction; others are directed at professional development. Pedagogic IDEs play a critical role in instruction because they reduce the clerical burden on the student and reinforce good program design.

Pedagogic IDEs for Java typically have simple, intuitive interfaces that are easy for novice programmers to learn. However, existing pedagogic Java IDEs are limited in scope to introductory programming courses because they do not scale to the larger projects assigned in later courses. This limitation stems partly from a lack of advanced features, such as testing and debugging tools, which are important for effective development of large Java projects. Another limiting factor is the choice of program representation in many pedagogic IDEs. For example, in the BlueJ environment [26], users are encouraged to write and interact with programs by manipulating UML diagrams [27]. While such graphical representations can help students visualize object oriented programming concepts, they are ultimately restrictive and unable to effectively express many programming abstractions, such as the *Command* design pattern [14]. As the size and complexity of projects increase, programmers should have access to any well-known abstraction mechanisms, not simply those with convenient graphical representations. BlueJ's "object bench" for interacting with objects is also impractical for experimenting with large programs, as it requires menu selections and graphical dialog completions for each object instantiation and method call.

Professional Java IDEs, on the other hand, are designed for developing large projects. These environments often contain an overwhelming number of advanced features, from incremental compilation and debuggers to refactoring tools, graphical user interface (GUI) builders, and various code generation wizards. The quantity of advanced features in these environments tends to be valued higher than the simplicity or accessibility of the interface. This results in a steep learning curve for new users, rendering most professional IDEs inappropriate for classroom use. In the classroom, instructors should not be expected to spend valuable lecture or lab time teaching students how to use the tools for the course. Indeed, even many professional developers avoid professional IDEs in favor of lightweight solutions, including extensible editors such as Emacs and compilation at a command line [17].

Despite the limitations of pedagogic IDEs and the complexity of professional IDEs, it is still desirable to leverage integrated development environments in Java programming courses and on large projects through the level of production development. To address the needs of course instructors and the desires of many professional developers, it is important to create a lightweight and simple IDE with sufficient strength for production programming.

A promising approach to create such an environment is through the extension of an existing pedagogic Java IDE. The DrJava environment [1] was selected for this task for several reasons. First, its interface is designed to focus on the source language itself rather than graphical program representations [34]. This ensures that programmers using DrJava are not limited in their expressiveness by the constraints of alternative representations, allowing DrJava to scale beyond the introductory level. Second, DrJava has been adopted in many introductory courses at institutions around the world (see Appendix A), largely because of its simple interface and its Interactions Pane, which provides a "read-eval-print loop" (REPL) for interactively evaluating Java statements and expressions.[1] Instructors and students in these courses actively

[1]See Chapter 2 for descriptions and a screenshot of DrJava's features.

provide feedback to DrJava developers to improve the interface of the IDE. Third, DrJava is freely available under the GNU GPL open source license [12], making it easily accessible to instructors, students, and developers at any level. Finally, the design principles and practices employed in the development of DrJava make it feasible to rapidly and safely extend the environment while maintaining its high code quality. Each of these characteristics makes DrJava a good candidate to extend to meet the needs of production programmers.

To achieve this goal, a set of recent extensions to DrJava have been implemented to make it an appropriate environment for writing programs up to the level of production development. Both these extensions and the process used for their implementation have also enabled the use of DrJava for teaching production programming skills in a classroom. Specifically, several carefully selected features have been added to DrJava to help programmers effectively develop large projects. Meanwhile, modifications to IBM's Eclipse IDE [18] have provided an easier transition from DrJava to a professional IDE to help developers leverage additional features when necessary, without facing a prohibitive learning curve in the new environment. Finally, DrJava has been used as both an IDE and a course project for teaching students production programming skills in an undergraduate course.

The remainder of this thesis is organized as follows. Chapter 2 introduces the DrJava IDE and its development process, and Chapter 3 surveys related IDEs. Chapter 4 describes the implementation of new features for DrJava to support the development of large projects. Chapter 5 covers a plug-in for the Eclipse IDE to ease the transition from DrJava, and Chapter 6 discusses the use of DrJava to teach production programming skills. Chapter 7 outlines future enhancements to DrJava, and Chapter 8 concludes.

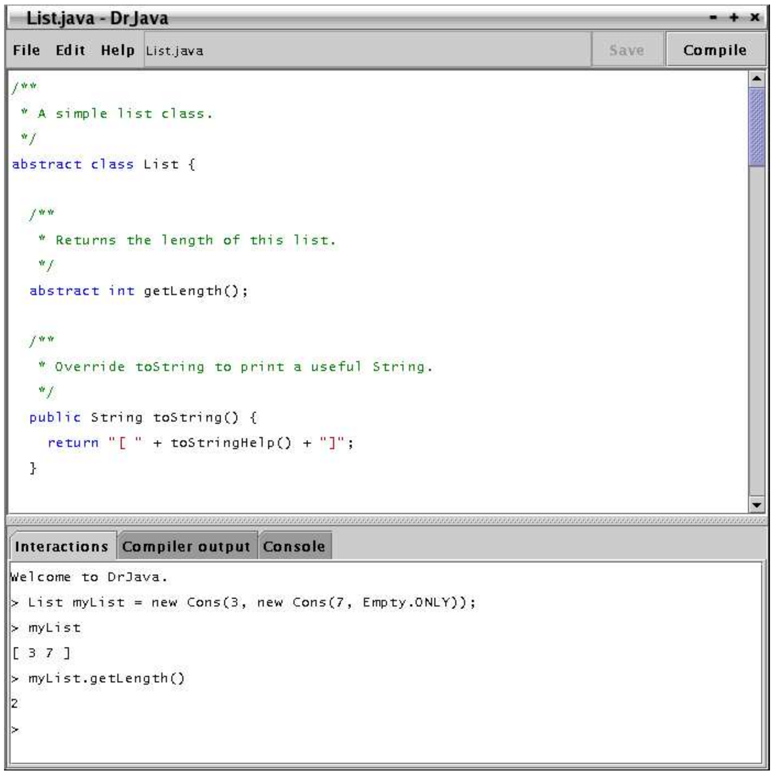
Chapter 2

DrJava Background

DrJava is a pedagogic Java programming environment developed and maintained by students at Rice University. It is designed to support introductory programming courses by providing a simple and interactive interface accessible to novices. To manage the evolution of the DrJava code base, the development team uses high quality design principles, Extreme Programming practices, and open source software. These principles and practices are consistent with extending DrJava to support production programming.

2.1 Goals of the IDE

The DrJava interface is designed to satisfy three primary goals: simplicity, interactivity, and a focus on the source language [34]. First, the environment is designed to be as simple and intuitive as possible to make it easily accessible to students with no programming experience (see Figure 2.1 for a screenshot of DrJava). For this reason, it has a small set of features relative to professional IDEs, and an emphasis is placed on the reliability and predictability of the features which are included. Second, an interactive user interface is provided by a "read-eval-print loop" (REPL) in DrJava's Interactions Pane, in which users can incrementally experiment with programs or libraries. Third, DrJava maintains a focus on the source language itself by avoiding code generation tools and graphical program representations. This focus helps inexperienced students quickly become comfortable with writing code, and ensures that the expressiveness of the programmer is not limited by the IDE.



```
/**
 * A simple list class.
 */
abstract class List {

  /**
   * Returns the length of this list.
   */
  abstract int getLength();

  /**
   * Override toString to print a useful String.
   */
  public String toString() {
    return "[ " + toStringHelp() + "]";
  }
```

```
Welcome to DrJava.
> List myList = new Cons(3, new Cons(7, Empty.ONLY));
> myList
[ 3 7 ]
> myList.getLength()
2
>
```

Figure 2.1 : The DrJava Pedagogic IDE, as of October 24, 2001. The basic interface includes a Definitions Pane for writing programs (top) and an Interactions Pane for experimentation (bottom).

2.2 Design Principles

DrJava developers are expected to use object oriented (OO) design principles with extensive use of design patterns. To create a loosely coupled and extensible code base, the developers employ the *Model-View-Controller*, *Mediator*, *Singleton*, *Command*, *Visitor*, and *Adapter* design patterns [14], among many others, in the implementation of DrJava. The use of design patterns allows new developers with OO design backgrounds to quickly understand and extend the code base. Other good Java programming practices are also encouraged, including designing for testability, documentation using Javadoc comments, avoiding the use of `null` as a value, and avoiding mutable state whenever possible. To make the code easier to read and understand, a set of coding standards is enforced. Our experience suggests that these principles help keep the code robust, legible, modular, and extensible.

One notable aspect of DrJava's architecture is the use of two Java Virtual Machines (JVMs). Most of the IDE executes in one JVM as a traditional Java application, but the Java interpreter used in the Interactions Pane must execute in a separate JVM for safety reasons. This ensures that the IDE will remain responsive if the user enters a divergent computation (such as an infinite loop) into the Interactions Pane.

2.3 Development Process

Beyond the design principles and goals of DrJava, a set of effective development practices is necessary to make the project successful on a reasonable time scale. The DrJava team relies on the Extreme Programming [25] methodology for this purpose, because it supports rapid development of robust products and adapts well to an academic environment [2]. The use of an open source license and open source software have also been essential to the development of DrJava.

2.3.1 Extreme Programming

The DrJava development team adheres to the tenets of Extreme Programming whenever possible. Extreme Programming consists of a simple set of practices that emphasizes testing, communication, and customer feedback to safely and incrementally deliver value in a product. The most important practices for DrJava include unit testing, pair programming, the use of on-site customers, incremental development, and continual refactoring. Each of these practices delivers specific benefits to the DrJava project, as outlined below.

Unit Testing

A comprehensive suite of unit tests can provide an effective safeguard for the quality of an Extreme Programming product. DrJava developers write unit tests for every non-trivial method in the code base, creating executable guarantees of program behavior and expected invariants. Unit tests are also written to exhibit any bug that is discovered, in an effort to both provide assurance that the corresponding bug fix actually eliminates the problem and that a recurrence of the bug will be automatically detected. Additionally, unit tests serve as a form of up-to-date, executable documentation for the code base [23]. New developers read relevant unit tests to learn how the original developers intended their code to be used, helping to transfer knowledge and cope with high developer turnover.

To easily support the development of ubiquitous unit tests, programmers must design code with testability in mind and practice test-driven development [25], in which unit tests are written at the same time as the methods they protect. Also, DrJava's build script enforces that the entire suite of tests is run before each commit to the code repository, ensuring that the code in the repository is high quality at any point in time.

Pair Programming

Pair programming is a form of continual code review in which two developers work at the same computer on the same task [24]. Pair programming has been shown to be a remarkably effective technique for rapidly producing well-designed and well-tested code, while lowering the rate of defect injection [40]. Pair programming also serves as an effective mechanism for knowledge transfer, which is essential in the face of the high developer turnover common in an academic environment. Specifically, new team members can pair program with experienced members to quickly learn relevant portions of the code base. Because of these properties, pair programming is an important technique for quickly achieving the goals of the DrJava project, and it is used whenever possible.

On-Site Customer

The presence of an on-site customer for an Extreme Programming project allows developers to receive continuous feedback on the progress of the project and the relative priority of each task. Such feedback is crucial to delivering a valuable product to customers [24]. The DrJava developers themselves are actually on-site customers of the product, as they use DrJava almost exclusively to work on the code base. This allows the team to frequently discuss and refine specifications to improve the quality of the product. Feedback is also frequently received from students and instructors of introductory courses, both within Rice University and from around the world. This feedback helps prevent DrJava from straying from its pedagogic goals.

Incremental Development

Each new feature for DrJava is broken into small tasks, each of which should take no longer than a week of pair programming time. The time required for smaller tasks is easier to estimate, and small tasks can add value to the product incrementally, allowing for frequent releases to customers [24]. By operating on such a schedule, the DrJava team produces incremental releases with new features every few weeks and milestone releases intended for widespread use approximately once a month. Each release is designed to add functionality requested by customers and fix bugs reported by customers. This quick response to customer feedback demonstrates to the community that DrJava is actively maintained and continually evolving to better meet its goals.

Continual Refactoring

Extreme Programming does not allocate a significant amount of initial time on a project for designing for extensibility or modularity. Instead, each task is implemented as simply as possible, since future requirements for extensibility are often unclear. As these requirements change and evolve, the code must be refactored to improve its design and adapt to the changes [13]. Extreme Programming suggests that this refactoring process should be continual, with developers improving existing code as new features are added and as requirements change [25]. The process is facilitated by the presence of ubiquitous unit tests, which ensure that refactoring one part of the code base does not break invariants expected in another [24]. Continual refactoring is actively employed by the DrJava team to improve modularity and eliminate poorly written code as the project grows.

2.3.2 Use of Open Source

DrJava benefits in several ways from the use of an open source license and open source software. Its free availability encourages its adoption by instructors and students. By also making the source code available under an open source license such as the GNU General Public License (GPL) [12], the pedagogic value of the project is raised. Students learning how to build graphical Java applications can easily download and inspect DrJava's source code as a case study. The open source license also encourages collaboration on the DrJava code base with developers outside the core team, either for the implementation of new features and bug fixes or for new projects built on top of DrJava. While other developers have the freedom to distribute modified versions of DrJava, the original development team retains control over the official version distributed to customers.

The DrJava development team itself also leverages open source software and other resources available to the open source community to maintain the high momentum of the project. Many professional quality open source tools are used in DrJava's development process, including Ant [3] for build scripting, CVS [9] for a source code repository, and JUnit [4] for a unit test framework. Other open source code is incorporated into the DrJava code base itself when it helps avoid the duplication of effort for a particular task. For example, DrJava's Interactions Pane uses an extension of DynamicJava [15], an open source Java interpreter. Many development and management resources are also available to the open source community, most notably including the SourceForge web site [30]. SourceForge is a collection of free hosting services for open source projects, including a remote CVS repository and a file distribution system. SourceForge also provides a suite of professional quality management tools for tracking feature requests, bug reports, and current tasks. The effort and resources that would be required for development and management of the DrJava project without SourceForge would have been largely cost-prohibitive. Thus, the availability of this free service to open source projects like DrJava has been immensely beneficial.

Chapter 3

Related Work

3.1 Pedagogic Programming Environments

DrScheme

DrScheme [11] is a freely available pedagogic IDE for the Scheme programming language that has served as a model for DrJava's interface. DrScheme provides a source editor, an Interactions Pane with a Scheme REPL, and support for a set of pedagogic language levels.

BlueJ

BlueJ [26] is a widely used pedagogic Java IDE intended for teaching object oriented programming concepts. Users construct classes graphically with UML diagrams [27], with the ability to interact with instantiated objects through a graphical "object bench." BlueJ is maintained by Monash University and is freely available.

JJ

JJ [32] is a commercial, online development environment developed by PublicStaticVoidMain.com. Students interact with JJ solely through a web browser, with all program code stored on a server accessible by instructors. JJ's interface is relatively cluttered, with several poorly labeled text panes and numerous buttons. Some of JJ's useful features include easy administration for instructors and support for pedagogic subsets of the Java language.

Ready To Program

Ready To Program with Java Technology [16] is a commercial pedagogic Java IDE developed by Holt Software. It is written in C++ and is only available for the Windows platform. Ready To Program uses Java 1.1.8 and IBM's Jikes compiler, and its simple interface supports compilation and execution but not program debugging.

3.2 Professional Java IDEs

A large number of both commercial and open source Java IDEs are available for professional developers, each of which contains a significant number of advanced features. Because of their complexity and steep learning curves, these tools are inappropriate for a pedagogic audience. JBuilder [7] and IntelliJ IDEA [22] are commercial IDEs for writing enterprise applications. CodeGuide [29] is a commercial Java IDE with support for GJ-style generic types [8]. NetBeans [39] is an open source IDE with a modular framework for building large desktop applications. Eclipse is an open source platform written by IBM for building IDEs through a plug-in architecture [18]. Eclipse is bundled with a set of Java development tools by default, while a large number of additional plug-ins are available online, both from IBM and third parties.

3.3 Java Debuggers

Sun distributes the Java Platform Debugger Architecture (JPDA) [36] as a framework to support common debugging functionality on Java programs. There are many debugging tools that leverage JPDA to suspend and step through program execution while querying or modifying values.

JDB

JDB [35] is a text-based, standalone debugging tool distributed with JPDA, providing a command-driven interface to most of the JPDA functionality. It is intended primarily as a demonstration of Java debugging functionality, and it requires users to learn its syntax and manually connect to a Java Virtual Machine. In addition to controlling program execution and inspecting values, JDB can evaluate some Java expressions in the context of a suspended execution.

JSwat

JSwat [6] is a standalone, open source Java debugger. JSwat has a graphical user interface and supports most common debugging functionality, but it does not contain editing facilities or other typical IDE features.

BlueJ

BlueJ [26] provides a traditional debugger integrated with its "object bench." Unlike most Java debuggers, BlueJ allows arbitrary points of entry to the debugger, since users can manually invoke any method from the "object bench" (not just `main`) to reach breakpoints of interest. However, no other interactions can be performed while a method call is suspended.

Professional IDEs

Almost all professional development environments provide integrated debugging features through graphical panes in the application. These interfaces often have tables or trees of information and buttons to control execution. A program must be started from its `main` method to be debugged by these tools. CodeGuide [29] supports execution control and value inspection, but not value modification or expression evaluation. NetBeans, JBuilder, and Eclipse all support execution control, value inspection and modification, and limited expression evaluation through sophisticated user interfaces [39, 7, 18].

Chapter 4

Features for Production Programming

The task of scaling DrJava to production programming primarily consists of adding new features to the IDE to increase its utility for large projects. Early versions of DrJava included the Interactions Pane and a source editor for editing a single file at a time, along with integrated compiler support. These versions adapted well to the needs of introductory courses, since small programs can be easily written in a single source file and tested manually in DrJava's Interactions Pane. Assignments in later courses, however, tend to require multiple source files and have greater complexity. To support the development of such projects in DrJava, it is necessary to evaluate which features can be most effective without sacrificing the philosophy of DrJava.

Beyond the simplest programs, almost all Java projects contain multiple source files, since each public class in Java must be defined in its own file. Because programmers tend to work on several related classes at a time, it is important to have the ability to open multiple files concurrently. Dealing with the increased complexity of larger programs and production level projects is another concern, which can be addressed through new testing and debugging features. Each of these features can be designed not to detract from DrJava's simple interface, while providing important benefits to programmers at almost any level.

4.1 Editing Multiple Files

Supporting multiple open documents is a fundamental requirement for effective development of large projects in DrJava. It is a prerequisite for advanced features such as the debugger, and it allows programmers to more naturally modify programs without being restricted by the user interface.

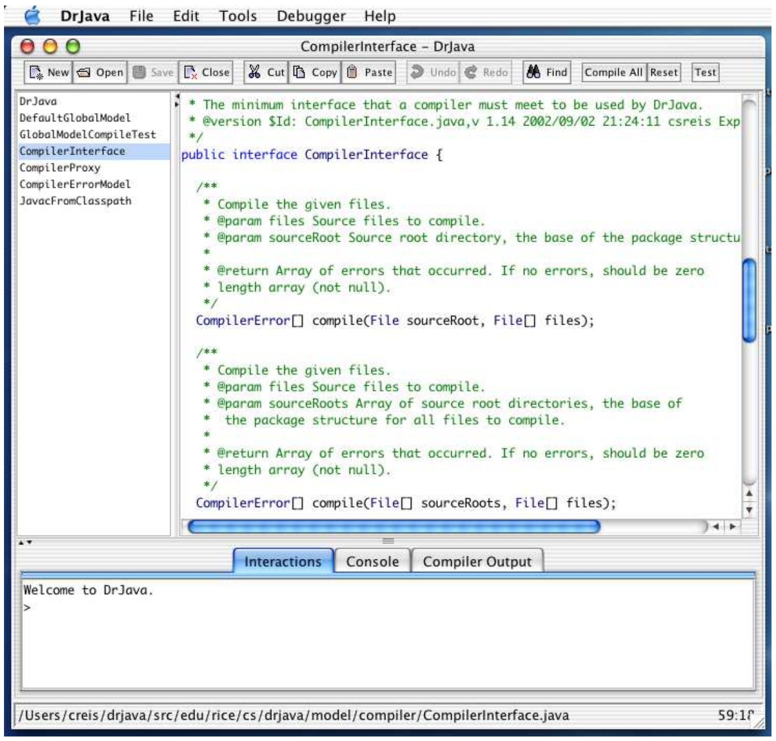
4.1.1 Selecting a File

To preserve a simple and clean user interface, only a single file is displayed in DrJava at any time, but with multiple file support, the user can easily switch between open files through a graphical pane (see Figure 4.1). While many professional IDEs display a tree of all of the files in a project, such a tree can require significant screen real estate for files deeply nested in a package hierarchy. To reduce the amount of visual space required and to simplify the appearance of the graphical pane, only a simple list of open file names is displayed in DrJava. The full path to the "active" file is always displayed in the status bar at the bottom of the window, to differentiate between files of the same name in different directories. With this interface, the user can easily select a file by name, only having to choose between files that are actually open.

4.1.2 Implementation

Prior to support for multiple open files, the logic for manipulating a file in DrJava was located in the `DefaultGlobalModel` class, which provides access to all of the components of the model as part of both the *Mediator* and *Model-View-Controller* design patterns [14]. Adding support for multiple files involved moving file-related behavior, such as saving, compiling, or closing a file, to a new `OpenDefinitionsDocument` interface. An inner class of `DefaultGlobalModel` implements this interface, while `DefaultGlobalModel` itself maintains a list of the open files.

For extensibility purposes, the `DefaultGlobalModel` class is unaware that only a single file is selected in the user interface at any time. Thus, although the current implementation of the view component can only display a single file, the model supports alternative views, which could display multiple files side by side. To maintain the invariant that exactly one document is "active" in DrJava's default view, the relevant logic and state are included in a `SingleDisplayModel` class, a subclass of



```
 * The minimum interface that a compiler must meet to be used by DrJava.
 * @version $Id: CompilerInterface.java,v 1.14 2002/09/02 21:24:11 csreis Exp
 */
public interface CompilerInterface {

  /**
   * Compile the given files.
   * @param files Source files to compile.
   * @param sourceRoot Source root directory, the base of the package structu
   *
   * @return Array of errors that occurred. If no errors, should be zero
   * length array (not null).
   */
  CompilerError[] compile(File sourceRoot, File[] files);

  /**
   * Compile the given files.
   * @param files Source files to compile.
   * @param sourceRoots Array of source root directories, the base of
   *  the package structure for all files to compile.
   *
   * @return Array of errors that occurred. If no errors, should be zero
   * length array (not null).
   */
  CompilerError[] compile(File[] sourceRoots, File[] files);
```

```
Welcome to DrJava.
>
```

Figure 4.1 : Support for multiple open files in DrJava.

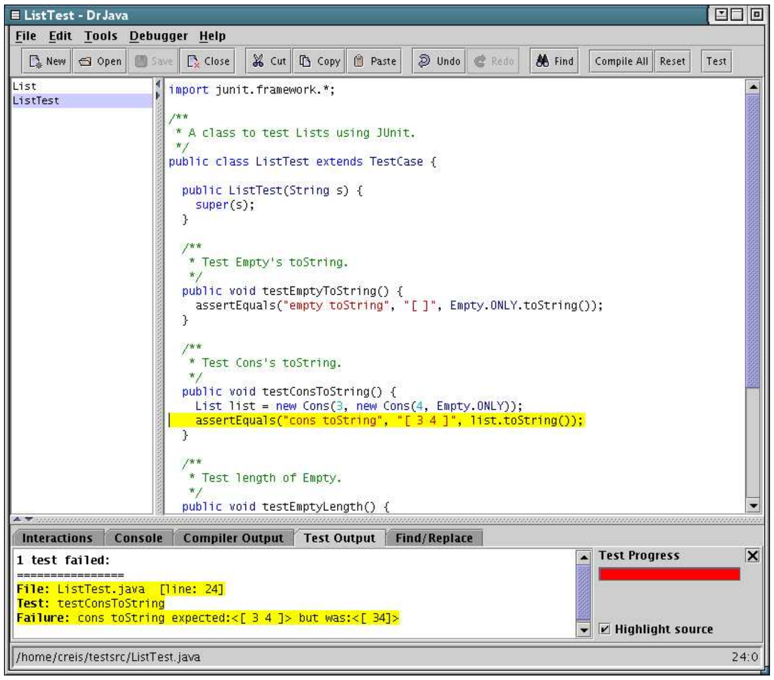
`DefaultGlobalModel`. The view component listens for changes to the active document in `SingleDisplayModel` and updates the user interface accordingly.

4.2 Improved Testing Support

At any skill level, testing programs is essential to understand a program's behavior and to avoid mistakes. DrJava's Interactions Pane can be used to quickly test aspects of a program by invoking methods and observing their behavior, but as programs grow in complexity, it becomes important to make use of a more structured test framework. Specifically, it should be easy to frequently repeat an entire suite of tests as a program is modified, to ensure that old invariants are not broken. It is also important for the test framework to be automated, without requiring the user to manually inspect test results for their correctness.

4.2.1 Unit Test Support

The unit test methodology proposed by Extreme Programming is an effective technique for testing programs of any size, since it focuses on testing every non-trivial method as it is written [24]. Thus, the test suite evolves as the program is written and does not need to be addressed in a separate phase of development. For Java, JUnit [4] is a widely used open source framework that enables developers to easily write and run unit tests for their programs. Developers simply write test methods in subclasses of JUnit's `TestCase` class, and they can then use graphical or command line tools to run the tests, being notified of any errors that occur. With integrated JUnit support in DrJava, users can easily write and run unit tests within the IDE to effectively test program components.



```
import junit.framework.*;

/**
 * A class to test Lists using JUnit.
 */
public class ListTest extends TestCase {

  public ListTest(String s) {
    super(s);
  }

  /**
   * Test Empty's toString.
   */
  public void testEmptyToString() {
    assertEquals("empty toString", "[ ]", Empty.ONLY.toString());
  }

  /**
   * Test Cons's toString.
   */
  public void testConsToString() {
    List list = new Cons(3, new Cons(4, Empty.ONLY));
    assertEquals("cons toString", "[ 3 4 ]", list.toString());
  }

  /**
   * Test length of Empty.
   */
  public void testEmptyLength() {
```

```
1 test failed:
================
File: ListTest.java  [line: 24]
Test: testConsToString
Failure: cons toString expected:<[ 3 4 ]> but was:<[ 34]>
```

Figure 4.2 : Integrated support for unit testing in DrJava. Test failures are highlighted in the source code.

4.2.2 Implementation

Since the JUnit framework is open source, its classes can be bundled within DrJava itself. In this way, the JUnit classes are always available when compiling test classes, since they are automatically included on the classpath of DrJava without any intervention by the user.

To run a set of unit tests, the user can click the “Test” button on DrJava’s toolbar. If this button is clicked while a JUnit test class is selected, DrJava invokes JUnit’s text-based `TestRunner` class to run the tests. Any failures or errors returned by the `TestRunner` are parsed and displayed in DrJava, in an interface similar to compiler errors (see Figure 4.2). The user can click on an error in the “Test Output” tab, and the corresponding line in the test class that caused the error is highlighted in yellow.

In the first iteration of JUnit integration in DrJava, JUnit tests were executed in the same Java Virtual Machine (JVM) as the IDE. However, this architecture was unsafe, since divergent code executed in a test case could cause the IDE to become unresponsive. Tests are now executed in the same JVM as the Java interpreter used by the Interactions Pane, which can be safely reset if a test stops responding.

4.3 An Interactive Debugger

For students with little programming experience, a REPL can be a useful debugging tool with minimal conceptual overhead. The user can instantiate objects and call methods to observe the behavior of a program, with the ability to conduct multiple experiments without needing to recompile the program. This significantly reduces the overhead of conventional experimentation, where developers must edit the `main` method of a class and recompile the class between each experiment [34]. Many problems in simple programs can be isolated and understood solely through experimentation in a REPL.

However, as programs grow more complex, a traditional debugger can be an important tool for locating and understanding defects in a program. Many such debuggers exist for Java, either as standalone products or integrated into larger IDEs (see Chapter 3). These debuggers provide a relatively common set of features presented in a graphical user interface. They are implemented using Sun's Java Platform Debugger Architecture (JPDA) [36], allowing them to gain control over a program running in a separate JVM. Common debugging features include the ability to:

- set "breakpoints" in source code, such that a program's execution will be suspended if the flow of control reaches a line of code with a breakpoint,
- inspect the values of variables and fields that are in scope in the context of a suspended program, and
- modify the values of variables and fields that are in scope before resuming execution, in an effort to dynamically correct erroneous values.

To make DrJava more effective for locating and correcting defects in complex programs, a debugger with such features has been added. Through the integration of this debugger with the REPL in DrJava's Interactions Pane, these debugging features have been made both simpler to use and more powerful than their counterparts in traditional Java debuggers. Specifically, this integration allows arbitrary points of entry for debugging, as well as the ability to query and modify the state of a program using the source language itself, rather than through a large collection of graphical panes.

4.3.1 Debugger Startup

One of the more restrictive and complicated aspects of traditional Java debuggers is the process of starting a debugging session and conducting multiple related experiments. DrJava provides a more intuitive technique for debugging programs through conjunction with the Interactions Pane.

Starting Traditional Debuggers

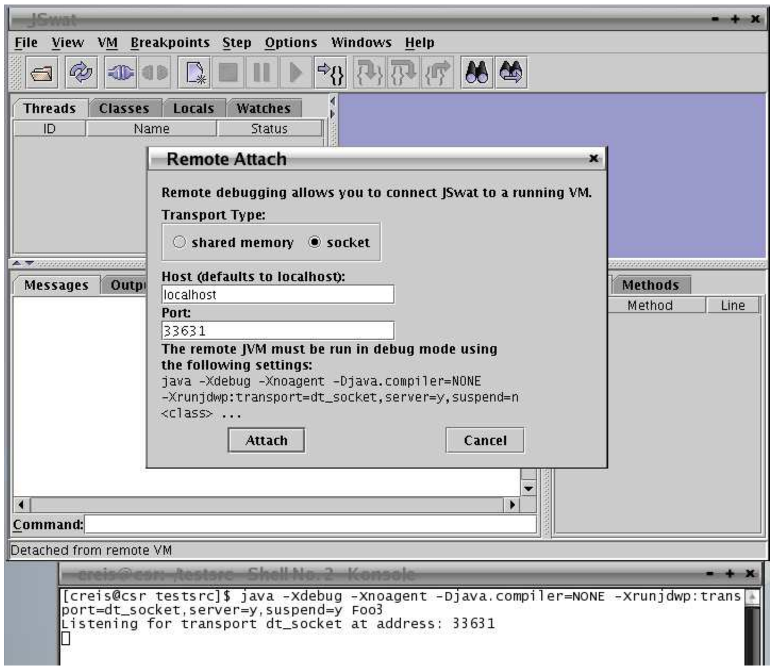
In traditional Java debuggers, the user must select the `main` method of a particular class to start a debugging session. This method is used to start a new JVM which the debugger can control using JPDA. Unfortunately, this approach limits the programmer to debugging the parts of a program which are accessible from the usual control flow from the `main` method. To conduct more specific experiments on particular methods, the programmer must write and compile a custom `main` method solely for debugging. This method must be updated and recompiled for each new experiment, imposing a significant overhead for debugging.

Most traditional debuggers also provide an option to "attach" to an existing JVM running either locally or on a remote host, but this can be a remarkably complicated process for novice programmers. Specifically, this involves specifying a series of cryptic command line arguments to the external JVM on startup, including an available port number for the debugger to use (see Figure 4.3). While this can be a powerful technique for advanced users, it is inappropriate for students first learning to use a debugger, and it also fails to solve the dependence on a compiled `main` method to initiate debugging.

Starting DrJava's Interactive Debugger

In comparison to traditional debuggers, starting a debugging session in DrJava is notably easier and more flexible. Once the user enables "Debug Mode" through a command in the Debug Menu, DrJava's debugger transparently attaches to the JVM used by the Interactions Pane (see Chapter 2). The programmer can then use any method invocation in the Interactions Pane as an arbitrary point of entry to the debugger, rather than being limited to the `main` method of a class.

Additionally, the user is not restricted to single invocation of a method during a debugging session. Instead, the user can continue to invoke methods and build up state, reaching additional breakpoints or modifying and repeating particular ex-


JSwat
File View VM Breakpoints Step Options Windows Help
Threads
Classes
Locals
Watches
ID
Name
Status
Messages
Outp
Methods
Method
Line
Command:
Attach
Cancel
creis@csr: ~/testsrc - Shell No. 2 - Konsole

Remote Attach

Remote debugging allows you to connect JSwat to a running VM.

Transport Type: ( ) shared memory [✓ socket]

Host (defaults to localhost): localhost

Port: 33631

The remote JVM must be run in debug mode using the following settings:

```
java -Xdebug -Xnoagent -Djava.compiler=NONE
-Xrunjdwp:transport=dt_socket,server=y,suspend=n
<class> ...
```

Detached from remote VM

```
[creis@csr testsrc]$ java -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:trans
port=dt_socket,server=y,suspend=y Foo3
Listening for transport dt_socket at address: 33631
```

Figure 4.3 : Attaching to a remote JVM with the JSwat debugger. Requires several command line arguments to the JVM.

periments. Notably, multiple related experiments can be conducted without needing to modify a `main` method or recompile any classes. Indeed, recompilation is not necessary until the original program code itself is modified.

4.3.2 Value Inspection and Modification

In addition to suspending and stepping through the execution of a program to inspect control flow, most Java debuggers allow users to inspect and modify the value of any variable or field that is in scope when execution is suspended. DrJava's Interactive Debugger provides this functionality through the familiar interface of the Interactions Pane.

Inspection and Modification in Existing Debuggers

Almost all traditional Java debuggers provide value inspection features through a graphical interface with a “control panel”-like metaphor. These interfaces consist of a collection of tables or trees displaying information and buttons to control the flow of execution. The user can create a list of watched variables and fields, which is updated with corresponding values each time the program execution is suspended. Often, a separate tree is also presented with the values of all variables that are in scope. Nodes in such a tree whose values are objects (and not primitive values) can be visually expanded to view the enclosed fields. Many debuggers allow the user to modify primitive values through the same graphical interface, simply by editing the displayed values.

The debugger in the Eclipse IDE [18] also provides the ability to evaluate some Java expressions in the context of a suspended execution. Unfortunately, this feature is also provided in the “control panel” metaphor of the user interface, making it less intuitive and more complicated than expression evaluation in a REPL. Specifically, the user can enter a block of statements or expressions into a dialog box, which is evaluated each time execution is suspended. Multiple code blocks can be stored in a table displaying the final result of each block, but each code block must be independent and cannot be evaluated incrementally. Thus, the user interface prevents the user from incrementally building up state without repeatedly modifying existing expressions. Eclipse also does not support some language constructs in this expression evaluation feature, such as defining anonymous subclasses or interface implementations. This limits the expressiveness of the user to a subset of the language. Perhaps most significantly, Eclipse’s error handling for user-defined expressions is remarkably poor, merely indicating that an error has occurred without providing any diagnostic messages. This is not only an unacceptable interface for inexperienced programmers who are still learning the language, but also makes it difficult for experienced programmers to determine which language constructs are actually available in the feature.

Inspection and Modification in DrJava's Interactive Debugger

Consistent with DrJava's focus on the source language, DrJava's debugger is designed to allow the user to interact with program state as much as possible using Java itself, rather than having a total dependence on a "control panel" of graphical components. Some graphical components are still provided for the sake of convenience, including tables displaying the current stack frames, a list of active threads, and even a table of watched variables and fields. However, most value inspection and modification is intended to be performed using source code directly typed into the Interactions Pane.

Just as the Interactions Pane can be used to invoke methods to encounter breakpoints, it can also be used to interact with the state of a program while a thread of execution has been suspended. When a breakpoint is encountered, a new prompt is presented to the user in the Interactions Pane. Any interactions entered at this prompt are evaluated in the context of the program suspension, as if the user was incrementally running new lines of code in the suspended method. All variables that are in scope at the suspension are available in the interpreter, including the "`this`" object, which can provide access to any visible fields or methods. Thus, the user can query the value of any variable or field by typing its name into the Interactions Pane, or he can modify the value of any variable or field through a simple assignment statement.

In addition to simple value queries and assignments, the Interactions Pane provides a very intuitive interface for evaluating any arbitrary expressions and statements within the context of a suspended execution. The user can easily instantiate new objects and invoke methods, either to observe their behavior in a particular context or to achieve the side effects of a particular method call. The REPL in the Interactions Pane is a much more convenient interface for incrementally building up state than the table of code blocks provided in Eclipse, since intermediate values can be queried as they are defined and later expressions can use results defined by early expressions. DrJava's Interactions Pane also supports the full range of Java language constructs, including anonymous subclasses or interface implementations, which are frequently useful when programming with the *Listener* or *Command* design patterns [14].

4.3.3 Implementation

To provide traditional debugging functionality, DrJava leverages the JPDA framework for controlling a separate JVM. JPDA is distributed with Sun's Java Software Development Kit (SDK), which is already a requirement for running DrJava because it also includes the Java compiler. Using methods in JPDA classes, DrJava's debugger is able to set breakpoints and resume or step through the execution of program threads, as well as query or modify visible variables and fields in the context of a suspension.

Integrating the Debugger and Interactions Pane

A significant part of the necessary infrastructure for DrJava's debugger was already in place when the debugger project began, since the interpretations performed in the Interactions Pane (via DynamicJava [15]) already ran in a separate JVM. While this decision was originally made to allow the Java interpreter to execute potentially unsafe or divergent code without adversely affecting DrJava's own virtual machine [34], it also provided a convenient architecture to debug the Interactions Pane from the primary JVM.

For the purposes of simplicity, it is important that the debugger can attach to the interpreter's JVM transparently, without exposing low level details of JPDA to the user. Thus, each time the interpreter's JVM is started or reset, the requisite command line arguments are automatically provided to the JVM to enable debugging, including a unique port number for JPDA to use. To avoid the overhead imposed by the JPDA classes when the debugger is not in use, DrJava does not actually attach to the interpreter's JVM using JPDA until the user enables "Debug Mode" through a menu command.

Once Debug Mode is enabled, DrJava monitors any activity in the interpreter's JVM via the JPDA framework. Since the Interactions Pane can be used to invoke methods in compiled classes, the user can set breakpoints in any source file and reach them through methods invoked from the Interactions Pane. Thus, invoking a class's `main` method is not necessary to use DrJava's debugging features, although any `main` method can be conveniently invoked by hand using a shortcut provided by DrJava. (The user can enter "`java Foo arg1`", which is automatically translated to "`Foo.main(new String[]{"arg1"})`".)

Some debugging information is only available through JPDA if the classes being debugged are compiled with the "`-debug`" flag, so DrJava always enables this flag when compiling classes. This ensures that all debugging features can be used for all user-defined classes which have been compiled with DrJava.

Allowing Interactions at a Program Suspension

The first iteration of DrJava's debugger provided transparent startup and arbitrary points of entry using the Interactions Pane, but it did not allow users to enter interactions while a program was suspended at a breakpoint. Instead, an hourglass cursor was displayed in the Interactions Pane until the program was resumed, preventing the user from entering new input. Values could only be queried by typing the names of variables or fields into a table of watched values, similar to traditional debuggers.

To allow users to enter interactions in the context of a breakpoint, a new instance of the DynamicJava interpreter must be created in the original interpreter's JVM. To provide the correct contextual environment to the user, this new interpreter must be given definitions for every visible variable in the suspended stack frame, as well as a reference to the "`this`" object from the stack frame. The user can then interact with these variables in a new thread until the original thread is resumed. As soon as the thread is resumed, the values of any variables in the stack frame which have been modified must be copied back into the stack frame.

These actions are accomplished using both JPDA and Remote Method Invocation (RMI) [37]. When DrJava's debugger classes detect that a thread has been suspended, an RMI call is made to the interpreter's JVM to create a new interpreter. The debugger classes then obtain the names and values of all visible variables in the suspended stack frame from JPDA. To define these variables in the newly created interpreter, the debugger must obtain a JPDA `ObjectReference` to the Singleton instance of DrJava's `InterpreterJVM` class, which manages all the interpreters in use. The debugger can then invoke the `defineVariable` method on the new interpreter (via JPDA), for each variable and value in scope, as well as for the "`this`" object from the stack frame. Once these variables have been defined, the debugger uses RMI to instruct the `InterpreterJVM` to use this new interpreter for any subsequent interactions. The debugger also changes the visual prompt in the user interface to contain the name of the suspended thread, which is usually equal to the string being interpreted by the thread.

At this point, the user can interact with the new interpreter to query and modify values or invoke methods. Once the user steps or resumes the thread's execution, values for the variables in the stack frame are copied back from the interpreter, using a similar technique to the one outlined above. The new interpreter can then be disposed and the original interpreter and prompt are restored until the next program suspension.

Current Limitations

Conceptually, the user is able to enter interactions at a breakpoint as if he is writing new lines of code in the suspended method itself. However, this metaphor is not yet entirely supported, since the current implementation of the debugger only provides access to fields and methods if users explicitly include "`this`" in the name. For example, if the "`this`" object contains a field named "`x`", the user must refer to it as "`this.x`" and not merely "`x`." This is inconsistent with how fields can be referenced within the methods of a class, and it is therefore unintuitive to the user. This problem can be resolved either by aliasing field and method names in the interpreter with local variables and methods, or by providing a more comprehensive notion of context to DynamicJava.

Chapter 5

Transition to Professional IDEs

While some advanced features can be added to the DrJava IDE to better support the development of larger projects, it is neither feasible nor desirable for DrJava to gain the scores of features available in professional development environments. Indeed, the simplicity of DrJava is considered an explicit strength in both pedagogic and professional settings.

However, there are some cases when it is desirable to leverage certain features from professional IDEs on large projects. For example, many professional IDEs provide powerful refactoring tools which allow programmers to make sweeping changes to a project quickly and safely. Such features might only be useful occasionally, but they are unavailable in DrJava, and the transition from DrJava to professional IDEs can be quite difficult due to the complicated user interfaces of the latter environments. Users of DrJava may also find useful features from DrJava missing from such IDEs, such as the Interactions Pane and Interactive Debugger.

To allow users of DrJava to leverage features from professional IDEs with minimal effort, it is necessary to ease the transition to such environments. This can be accomplished through *plug-ins* to professional IDEs, which are pieces of code that add or modify the functionality of an IDE. A plug-in can be used to provide a user interface similar to DrJava within a professional IDE, complete with familiar features such as the Interactions Pane and Interactive Debugger. While the IDE would retain much of its original functionality and complexity, such a plug-in would make it easier for users familiar with DrJava to quickly learn the environment.

Many existing development environments support the development of plug-ins, including IBM's Eclipse [18] and the jEdit extensible editor [31]. The Eclipse platform was chosen as the best candidate for a DrJava plug-in because of the flexibility of its architecture, its wide variety of advanced features produced by both commercial third parties and the academic community, and its free availability under an open source license.

5.1 The Eclipse Architecture

Eclipse is an open source platform created by IBM for building extensible integrated development environments. The Eclipse platform contains a base set of functionality, with all useful features provided as plug-ins. In fact, even the Java development tools bundled with Eclipse are written as plug-ins to the base platform. This architecture is explicitly designed to allow plug-ins to build upon one another through published interfaces, allowing tool builders to leverage existing work. Plug-ins can be individual *views* of information or *editors* of resources, which are all commonly displayed as graphical panes in the main Eclipse window (known as the *workbench*). Plug-ins can also define *perspectives*, which specify a set of views and editors with a particular visual layout. For example, the bundled Debugger plug-in provides a Debug Perspective which includes several graphical panes, such as a tree of running and suspended threads, a list of watched variables, and a list of expressions to evaluate.

One of the key strengths of Eclipse is its active plug-in development community [19], as well as IBM's commitment to research, academics, and education through the Eclipse Technology Project [20]. In fact, the DrJava Plug-in for Eclipse itself was partly funded by an Eclipse Innovation Grant [21] provided by IBM.

5.2 DrJava Plug-in for Eclipse

The DrJava Plug-in for Eclipse is designed to provide several aspects of DrJava's user interface to the Eclipse platform, in an effort to support interoperability between DrJava and Eclipse. The aspects of the plug-in include a simplified user interface to make Eclipse's learning curve less steep, an Interactions Pane with the full range of functionality available in DrJava, and an Interactive Debugger similar to the one discussed in Chapter 4.

The plug-in project is scheduled to be completed at the end of 2003, and it is being developed incrementally. The first iteration contains a partly simplified user interface and an Interactions Pane, while further interface refinement and an Interactive Debugger will be pursued as future work.

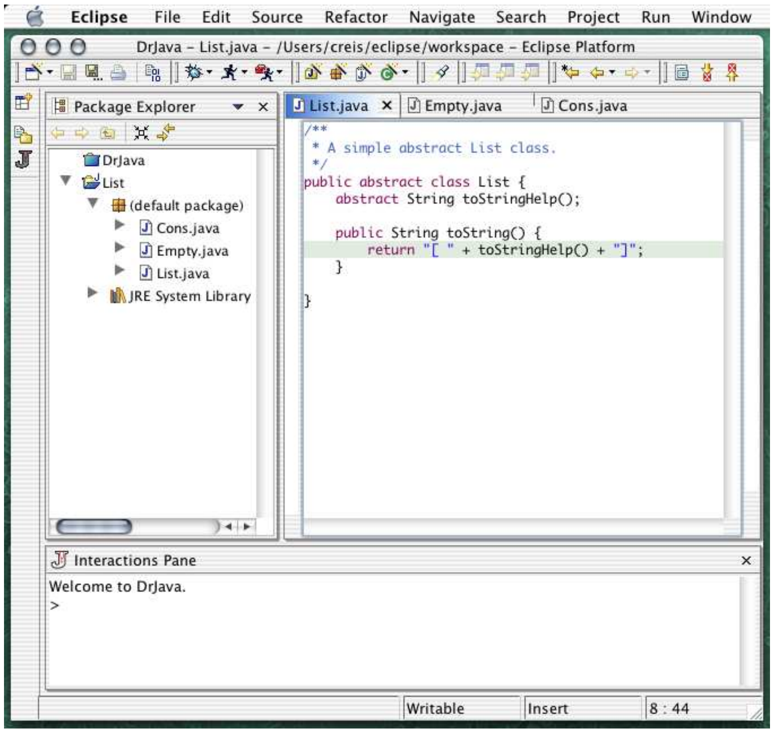
5.2.1 Design Goals

In accordance with the development practices used for DrJava itself, the DrJava Plug-in for Eclipse has been developed with Extreme Programming practices. Notably, this includes the use of pair programming, unit tests, and continual refactoring.

In addition, an important design decision was made to reuse as much code from the DrJava code base as possible, without duplicating code into a separate source tree. This ensures that any bug fixes or feature enhancements to the Interactions Pane in DrJava will be immediately available to the Eclipse plug-in without additional effort. This decision also minimizes the amount of new code to be written for the plug-in, as well as the opportunities for new defects to be introduced.

5.2.2 Simplified Interface

The first component of the plug-in provides a perspective to make Eclipse similar in appearance to DrJava. As in the DrJava IDE, the DrJava perspective for Eclipse includes an area for editing source files, with the new Interactions Pane displayed below it (see Figure 5.1 for a screenshot). To switch between open documents, Eclipse's Package Explorer view is displayed to the left of the source editor, while the Console and Search Results are displayed (when relevant) in a tabbed pane shared with the Interactions Pane. To preserve the simplicity of the interface, other graphical panes from Eclipse's Java Development perspective, including a structural outline of the selected class and a list of outstanding tasks, are hidden by default. Experienced users can display these extra panes with menu commands.



```
/**
 * A simple abstract List class.
 */
public abstract class List {
    abstract String toStringHelp();

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

}
```

```
Welcome to DrJava.
>
```

Figure 5.1 : The DrJava Plug-in for Eclipse.

5.2.3 Interactions Pane

To provide a familiar environment in Eclipse to DrJava users, the plug-in includes an Interactions Pane with equivalent functionality as its counterpart in DrJava. While Eclipse does provide general purpose expression evaluation through its Java Scrapbook feature, the interface is cumbersome and less intuitive than the REPL used in DrJava's Interactions Pane. In a Scrapbook page, users must type a block of code to be evaluated, select it with the mouse, right-click on it, and then choose an appropriate evaluation command from the resulting contextual menu. Any resulting error messages are inserted as text in the Scrapbook itself, requiring them to be manually deleted or carefully avoided when evaluating new expressions. In contrast, DrJava's Interactions Pane is designed with a console-style interface, popularized through a "read-eval-print loop" (REPL) for Lisp [33]. Input is accepted and evaluated each time the user hits `Enter`, allowing the user to easily build up state incrementally. Also, in the Interactions Pane, the values of any expressions ending without semicolons are displayed after each interpretation, making it easy for the user to view the values of any variables or method invocations without using graphical widgets or menus. Thus, adding the Interactions Pane to Eclipse not only provides a familiar tool to programmers making a transition from DrJava, but also makes a convenient Java REPL available to professional developers.

Implementation

To implement the Interactions Pane in Eclipse with as much code reuse as possible, a fairly significant refactoring effort was necessary in relevant parts of the DrJava code base. DrJava is designed with a *Model-View-Controller* architecture [14], so the logic for the Interactions Pane is separate from the user interface code. Thus, it would theoretically only be necessary to write a new view component to fit into Eclipse's Workbench, along with the controller code to link the view to DrJava's model. However, much of the logic in the model for the Interactions Pane was tightly coupled with DrJava's `DefaultGlobalModel` class, the *Mediator* class which provides access to all of the functionality in DrJava. Since the plug-in for Eclipse does not use most of DrJava's model (including the source editor, compiler integration, JUnit integration, and many other features), it was necessary to separate the logic for the Interactions Pane into its own independent module.

This particular refactoring task is an excellent example of the benefits of Extreme Programming practices. In Extreme Programming, an emphasis is placed on finding a simple solution for a task rather than initially designing an extensible architecture [25]. While the architecture of DrJava was already relatively modular and extensible, it was previously unnecessary for the Interactions Pane to stand apart from the rest of DrJava, so no time was spent creating a design to facilitate this. This only became a requirement once the plug-in was designed, so it was addressed as a refactoring effort at this time. Fortunately, other Extreme Programming practices ensured that this refactoring could be completed quickly and effectively. Specifically, the existing unit tests over the Interactions Pane logic ensured that any previous functionality was not broken as a result of the refactoring, while the importance of frequent releases ensured that the code base in the repository remained in a usable state at all times. The refactoring effort itself resulted in a more modular design for DrJava, which both allowed the Interactions Pane to run outside the context of the full DrJava IDE and also made many aspects of DrJava's logic easier to test. In fact, several previously undetected bugs were discovered and corrected in the process of this refactoring.

Beyond the creation of a more modular and independent `InteractionsModel` class, another large refactoring effort was necessary due to a notable difference between DrJava and Eclipse. DrJava, like most graphical Java applications, uses Sun's Swing architecture for its windowing toolkit. Swing provides all the classes necessary to implement general purpose, cross-platform graphical user interfaces in Java. Eclipse, on the other hand, uses IBM's SWT framework for its windowing toolkit. SWT is designed to be more closely integrated with each platform's native windowing toolkits than Swing, but it requires the user to download a platform-specific version of Eclipse with appropriate native code.[1] Ideally, the differences between Swing and SWT would be isolated in the view components of the DrJava IDE and the Eclipse plug-in. However, DrJava's model previously used a Swing `Document` class to store the contents of the Interactions Pane. This Swing `Document` could not be used effectively as the model for an SWT `StyledText` widget, which expected text to be stored in an equivalent SWT class. In the interest of avoiding duplicated state between Dr-

[1]Currently, Eclipse and SWT are available for Microsoft Windows, Linux (Motif or GTK 2), Solaris (Motif), QNX, AIX, HP-UX, and Mac OS X.

Java's model and the view component of the plug-in, the Swing `Document` class used by DrJava was replaced by a new toolkit-independent `DocumentAdapter` interface, using the *Adapter* design pattern [14]. This `DocumentAdapter` interface is used by DrJava's model without knowledge of which windowing toolkit is actually in use, and it is implemented using Swing for the DrJava IDE and using SWT for the Eclipse plug-in. With this `DocumentAdapter` in place, all technical hurdles to implementing the Interactions Pane in Eclipse had been overcome.

The Interactions Pane phase of the Eclipse plug-in was developed and released incrementally, in accordance with Extreme Programming. The schedule included two pre-release versions of the plug-in as it evolved. With a simplified perspective already in place, the first pre-release included an initial prototype of the Interactions Pane using the `DocumentAdapter` framework. Part of the `InteractionsModel` refactoring had been completed, but the code for communicating with the Java interpreter's JVM had not yet been refactored. Thus, for this pre-release, the Java interpreter actually ran in the same JVM as Eclipse itself. While this was unsafe for production use (since the user could render Eclipse unresponsive with divergent code), it provided a good proof of concept for using the Interactions Pane in a plug-in. The second pre-release included refactored RMI code in DrJava so that the Java interpreter used in the Eclipse plug-in could be run in a separate JVM. This pre-release was safe to use in a production setting. However, it was not integrated with the user's open projects in Eclipse, so user-defined classes were not placed on the classpath of the Interactions Pane. The first full release of the plug-in will add any classpath entries from the user's open projects to the Interactions Pane, and it will also include a few menu commands to reset the interpreter and load and save the history of commands.

5.2.4 Debugger Integration

The initial debugger integration in the DrJava Plug-in for Eclipse is relatively straightforward, leveraging the Debugger plug-in bundled with Eclipse. The RMI framework for running DrJava's Interactions Pane already allows the interpreter's JVM to be remotely debugged, since it passes the required command line arguments to the JVM on startup. A menu command can be added as an extension to Eclipse's Debugger plug-in, allowing the user to start a debugging session by simply attaching to the Interactions Pane rather than by invoking the `main` method of a particular class. This menu command attaches to the interpreter's JVM using the appropriate port, allowing the user to take advantage of all of Eclipse's debugger features with arbitrary points of program entry from Interactions Pane. This approach does not yet support the use of the Interactions Pane in the context of a suspended thread.

5.2.5 Plans for Future Integration

The remaining functionality for the DrJava Plug-in for Eclipse consists of allowing interactions in the context of a suspended thread. This will allow the benefits of DrJava's Interactive Debugger (outlined in Chapter 4) to be leveraged within Eclipse.

With the Interactive Debugger functionality in place in the DrJava IDE, the plug-in implementation of the debugger can proceed with similar goals and development practices as the Interactions Pane implementation. Specifically, a component of the plug-in will need to respond to events generated by Eclipse's debugger. This component will create appropriate responses in the Interactions Pane to provide a new prompt in the context of a breakpoint. The majority of the code can be reused from the debugger implementation in the DrJava IDE, with minimal code duplication.

After the debugger integration has been fully implemented, the remaining work on the plug-in can include further refinements to the plug-in's perspective to more closely mimic the user interface of DrJava.

Chapter 6

Teaching Production Programming with DrJava

Through the extensions discussed in Chapter 4, DrJava has become an effective tool for production programming. In fact, the DrJava developers use it almost exclusively to work on the DrJava code base, illustrating its ability to scale to the development of actively used products.

As a result, DrJava can be used as a pedagogic IDE to teach production programming skills. It remains an appropriate tool for introductory courses, but these recent extensions have allowed it to scale to more advanced courses as well. Interestingly, the development process used for DrJava actually makes it an appropriate project for production programming courses. Specifically, by extending the DrJava code base, students are exposed to production development in an academic setting.

6.1 A Pedagogic IDE for Production Programming

When designing a production programming course, instructors must consider which tools students will use for development. It is desirable for students to work in a common development environment to ease course management, but the complex user interfaces of most professional IDEs make them inappropriate for classroom use. Some such courses have employed text editors (such as Emacs) and command-line compilation, but instructors have lamented the lack of effective debugging tools with this setup [41].

To solve this problem, instructors can use the DrJava IDE in production programming courses. The simple and intuitive interface of DrJava makes it an easy tool to introduce to students, while it provides sufficient power to support production development. In fact, the task of introducing DrJava to students is entirely unnecessary if the students are already familiar with it from introductory courses. Thus, DrJava can be a very useful tool in production programming courses.

6.2 Students as DrJava Developers

Selecting a course project for a production programming course is another difficult task, since the project should simulate many aspects of actual production development, which can be difficult to reproduce in a classroom. For the past two years, the COMP 312 Program Engineering course at Rice University has used DrJava itself as a course project, putting students directly into development and management roles for DrJava. In addition to providing lectures on the design principles appropriate for software development in industry, the course teaches students Extreme Programming practices and expects students to use these practices to extend and improve the DrJava IDE. Through development on the DrJava code base, students learn effective techniques for joining existing projects, supporting customers, and adding features to a product in a safe and incremental manner. In fact, students in COMP 312 have written a significant amount of DrJava's core functionality, including the debugger, the JUnit integration, and even the logic for indenting source files. Students also gain experience with a suite of powerful development tools likely to be seen in industry, as well as a large number of advanced programming concepts and technologies.

Managing even a small group of students working on diverse tasks on a common code base can be formidable, so effective communication is critical. To make this task feasible, the course employs several teaching assistants to serve as project managers for the students. These teaching assistants must have prior experience developing DrJava, because they are expected to pair program with students early in the course on smaller projects to help transfer knowledge about the code base. Later in the course, the teaching assistants manage project groups with no more than three pairs of students, supervising each group's progress on their tasks.

6.2.1 Extreme Programming in the Classroom

The COMP 312 course is designed to expose undergraduate students to Extreme Programming as a set of effective development practices. All development on DrJava has leveraged Extreme Programming since the project's inception [34], making the DrJava team an appropriate model for teaching these practices. While the general benefits of Extreme Programming have been discussed in Chapter 2, some additional benefits and necessary adaptations arise in a classroom setting.

Unit Testing

A comprehensive suite of unit tests is essential in a production programming course, as it prevents new developers from inadvertently breaking functionality as changes are made to the code base. While test-driven development is a simple concept to teach, it can be difficult to enforce, as students often habitually leave testing until the last phase of development.

Several techniques have been used in COMP 312 to ensure that students write adequate unit tests for new code. First, lectures repeatedly emphasize the benefits and safeguards provided by unit tests. Second, the first course assignment involves writing unit tests for a small and partially incorrect code base, to give students experience on a small scale. Third, students are introduced to the DrJava code base by fixing small bugs, using unit tests to verify that a bug is fixed. Fourth, the teaching assistants for the course monitor the code committed by students to ensure that adequate tests have been provided.

It is worth noting that some tools, such as Clover [10], have been produced to quantify the unit test coverage of a project. Such tools could be quite valuable in a classroom setting to ensure students are writing new tests. Unfortunately, no code coverage tools which support GJ-style generic types [8] have been found. Since the DrJava code base uses generic types, the DrJava development team cannot yet use unit test coverage tools.[1]

Pair Programming

Pair programming has been shown to be remarkably effective in a classroom setting, helping students to rapidly produce high quality code while also transferring knowledge of the code base among students in the class [40]. However, unlike an industrial setting where developers have a common work schedule, undergraduate students have diverse schedules which can make finding time to pair program quite difficult.

The COMP 312 course schedule is structured to address this situation. While the course has three scheduled meetings per week, only the first two are used for lecture. The third meeting is reserved for a closed lab to provide time for pair programming. A single lab session of pair programming per week is insufficient for the course, however, so the students are allowed to select their own pairs for each project to maximize the amount of time they can spend working together.

On-site Customers

To ensure that the students address issues of high priority to the customers of DrJava, Extreme Programming requires the team to work with on-site customers who can provide regular feedback. While it may be infeasible for students to meet with customers from other institutions, the students themselves are required to use DrJava on their tasks, making them on-site customers of the product. To benefit from this, class discussions are frequently held to refine specifications and resolve ambiguities, with the students acting as both customers and developers. Students are also given the opportunity to respond to feedback from users at other institutions, providing another means for interaction with real customers.

[1]The Clover developers have expressed intentions to support generic types in future versions of Clover, so this tool could be useful in future iterations of COMP 312.

Incremental Development

In a semester long production programming course, it is very important to define tasks in small increments. This technique helps students refine their skills for estimating the time required for a task, and it allows teaching assistants to easily monitor the students' progress and revise specifications if necessary. In fact, no fixed deadlines are imposed on the tasks, since Extreme Programming places a greater emphasis on producing high quality code and refining schedules than on meeting deadlines. To ensure that students give appropriate attention to a course without project deadlines, the students are required to log ten hours of course work per week outside lecture.

Another benefit of incremental development in a classroom is that the product can be released frequently throughout the semester. In this way, students can see their coursework downloaded and used by customers around the world, providing more powerful feedback than a conventional grading system.

Continual Refactoring

Finally, instructing students to continually refactor existing code provides an opportunity for them to put design principles from earlier courses into practice. Even well-designed code often needs to be refactored as a project evolves and new goals are determined, and Extreme Programming makes this feasible through the safeguards provided by the unit tests. Giving students the freedom to frequently refactor instills a sense of pride and ownership in the project, keeping the quality of the code high.

6.2.2 Open Source Tools

The use of an open source license for DrJava has been quite beneficial for managing COMP 312. As an open source project, DrJava is freely hosted on the SourceForge development web site [30], which provides a powerful web interface for managing tasks and feedback (see Chapter 2). Even with several teaching assistants acting as project managers, the administration of COMP 312 would be remarkably more difficult without the tools freely available through the SourceForge website.

The DrJava team also uses a large number of open source development tools which have pedagogic value in COMP 312. Students learn valuable software engineering concepts with these tools, such as software versioning with CVS [9], build scripting with Ant [3], and unit testing with JUnit [4]. Each of these tools is freely available for students to use on their own projects, providing an additional advantage over cost-prohibitive commercial tools.

6.2.3 Advanced Concepts and Technologies

By extending DrJava in COMP 312, students are also introduced to a large number of advanced programming concepts and technologies which are not covered in most introductory courses. Most notably, DrJava involves a significant amount of concurrency, exposing students to synchronization issues and possible race conditions. Many tasks on DrJava can thus be useful for teaching students how to be aware of the complications of concurrent programming.

Students also have the opportunity to learn several advanced frameworks and technologies through work on the DrJava code base. DrJava not only has multiple threads of execution, but actually runs in two separate JVMs: one for the IDE and one for the Java interpreter used by the Interactions Pane. These two JVMs communicate using Sun's Remote Method Invocation (RMI) framework, a technology frequently used in networked or distributed applications. Meanwhile, DrJava's Interactive Debugger extensively uses Sun's Java Platform Debugger Architecture (JPDA) to provide debugging functionality.

It is also notable that DrJava is written using GJ-style generic types [8] to produce better type-safe designs. Currently, this requires DrJava to be compiled using Sun's JSR-14 prototype compiler [38], which is still under active development. Through the use of JSR-14, students learn the safety and syntactic advantages of generic types, while also learning how to deal with experimental and potentially buggy technologies.

Chapter 7

Future Work

7.1 Additional Features

A small number of additional advanced features can be added to DrJava to better support the development of large projects, while still maintaining DrJava's simple interface for inexperienced users.

Project Support via Ant

All program manipulation in DrJava is currently handled at the granularity of individual source files, without any representations of projects. While this interface is appropriate for the small programs written by novice students, it can often be convenient to maintain project-level information for larger and more complex programs. DrJava could be modified to provide support for projects, to allow users to save configuration options specific to a project and compile or test an entire project at once. To prevent tying users to a single development environment through a proprietary project file, DrJava could save project related information in the form of Ant build scripts [3]. Ant scripts are commonly used with large projects, and many professional IDEs provide integrated support for Ant.

Javadoc Integration

Integrated support for Sun's Javadoc API documentation tool is currently in progress. Once completed, DrJava will be able to generate HTML documentation files for a program, via Javadoc. A graphical window for viewing the HTML files will be provided within DrJava.

7.2 Collaboration with External Developers

As an open source project, it is possible for developers outside Rice to contribute to the DrJava project or leverage the code base for new products. These opportunities can be leveraged to increase the pedagogic value of DrJava.

Software Engineering Courses

With published results on the use of DrJava development to teach Extreme Programming [2], instructors from other institutions have expressed interest in forming similar courses. These courses can use a similar technique with a different project, but it would also be possible to allow students at other institutions to extend DrJava in their own production environments. Control over DrJava is maintained by developers at Rice University, but students in other courses could submit patches with new features or bug fixes to be approved by the developers at Rice. This presents an opportunity to teach distributed development practices which may be encountered in industry.

DrScala

The DrJava code base can also be used as a foundation for other projects to build simple, lightweight, and stable IDEs. For example, the DrScala project involves modifying DrJava to support the Scala programming language [28]. Close communication with such teams can be valuable for both DrJava and the resulting products, as DrJava can be refactored for greater extensibility as these products are developed.

Chapter 8

Conclusion

The DrJava pedagogic IDE has become effective for production programming, both as a simple but powerful environment for developing large projects and as a tool to teach production programming skills. This has been accomplished through the addition of a small set of features which make editing, testing, and debugging programs easier, without detracting from the simplicity which makes DrJava valuable for novice programmers. An easier transition to professional IDEs has also been provided in the form of an Eclipse plug-in, allowing DrJava users to take advantage of additional features in Eclipse while using a familiar interface. Through the development and use of these extensions, DrJava has been used effectively to teach production programming skills in a classroom setting. Indeed, while using DrJava as an IDE, students have added much of the core functionality of DrJava. As a result, it is clear that the benefits of a pedagogic IDE such as DrJava can indeed be applied at the level of production development.

Appendix A

Feedback from DrJava Users

The DrJava development team has received significant feedback on the IDE from customers around the world, including many instructors and students in introductory courses using DrJava. This feedback has been in the form of feature requests, bug reports, and support requests through DrJava's SourceForge project web site, as well as through direct email to the development team. A list of institutions which have provided feedback on DrJava is below.

Institution	Location
Bloomington High School	Bloomington, IL, USA
University of California, Irvine	Irvine, CA, USA
Colby College	Waterville, ME, USA
Cornell University	Ithaca, NY, USA
Dublin City University	Dublin, Ireland
Georgia Tech	Atlanta, GA, USA
University of Huddersfield	Queensgate, Huddersfield, UK
Indiana University	Bloomington, IN, USA
University of Maryland	College Park, MD, USA
University of Missouri, Columbia	Columbia, MO, USA
Universite de Nice Sophia-Antipolis	Nice, France
Rice University	Houston, TX, USA
Universidade de São Paulo	São Paulo, Brazil
Sheridan College	Brampton, Ontario, Canada
University of Toronto	Toronto, Ontario, Canada
University of Washington	Seattle, WA, USA
Westminster College	New Wilmington, PA, USA
Universidad de Sevilla	Seville, Spain

Table A.1 : Institutions providing feedback on DrJava.

Bibliography

[1] E. Allen, R. Cartwright, B. Stoler. *DrJava: A Lightweight Pedagogic Environment for Java*. *SIGCSE 2002*, March 2002.

[2] E. Allen, R. Cartwright, C. Reis. *Production Programming in the Classroom*. *SIGCSE 2003*, February 2003.

[3] The Apache Software Foundation. "The Apache Ant Project."
(URL: `http://ant.apache.org`)

[4] K. Beck, E. Gamma. "JUnit, Testing Resources for Extreme Programming."
(URL: `http://www.junit.org`)

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

[6] Blue Marsh Softworks. "JSwat - Graphical Java Debugger."
(URL: `http://www.bluemarsh.com/java/jswat/`)

[7] Borland. "JBuilder."
(URL: `http://www.borland.com/jbuilder/`)

[8] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler. *Making the future safe for the past: adding genericity to the Java programming language*. *OOPSLA '98*, October 1998.

[9] "Concurrent Versions System."
(URL: `http://www.cvshome.org`)

[10] Cortex eBusiness. "Clover, A Code Coverage Tool for Java."
(URL: `http://www.thecortex.net/clover/`)

[11] 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.

[12] Free Software Foundation. "The General Public License."
(URL: `http://www.gnu.org/licenses/gpl.html`)

[13] M. Fowler, K. Beck, J. Brant. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass. 1995.

[15] S. Hillion. "DynamicJava."
(URL: `http://koala.ilog.fr/djava/`)

[16] Holt Software. "Ready To Program with Java Technology."
(URL: `http://www.holtsoft.com/ready/`)

[17] A. Hunt, D. Thomas. *The Pragmatic Programmer*. Addison-Wesley, 2000.

[18] IBM. "eclipse.org."
(URL: `http://www.eclipse.org`)

[19] IBM. "Eclipse Community Projects and Plugins."
(URL: `http://www.eclipse.org/community/plugins.html`)

[20] IBM. "The Eclipse Technology Project."
(URL: `http://www.eclipse.org/technology/`)

[21] IBM Scholars Program. "eclipse Innovation Grants."
(URL: `http://ibm.com/university/eclipseinnovation/`)

[22] IntelliJ. "IntelliJ IDEA." (URL: `http://www.intellij.com/idea/`)

[23] R. Jeffries. "Essential XP: Documentation."
(URL: `http://www.xprogramming.com/xpmag/expDocumentationInXP.htm`)

[24] R. Jefferies, A. Anderson, C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2001.

[25] R. Jeffries. "What is Extreme Programming?"
(URL: `http://www.xprogramming.com/xpmag/whatisxp.htm`)

[26] M. Kölling, A. Patterson, B. Quig, J. Rosenberg. "BlueJ, The Interactive Java Environment."
(URL: `http://www.bluej.org`)

[27] Object Management Group. "Universal Modeling Language."
(URL: `http://www.uml.org`)

[28] M. Odersky. "Scala: A scalable language."
(URL: `http://lamp.epfl.ch/scala/`)

[29] Omnicore Software. “CodeGuide.”
(URL: `http://www.omnicore.com`)

[30] Open Source Developer Network. “SourceForge.”
(URL: `http://sourceforge.net`)

[31] S. Pestov. “jEdit - Open Source programmer’s text editor.”
(URL: `http://www.jedit.org`)

[32] PublicStaticVoidMain.com. “JJ Home Page.”
(URL: `http://www.publicstaticvoidmain.com`)

[33] E. Sandewall. *Programming in an interactive environment: the “Lisp” experience*. In *Computing Surveys*, 10(1), March 1978, 35-71.

[34] B. Stoler. *A Framework for Building Pedagogic Java Programming Environments*. Master’s thesis, April 2002.

[35] Sun Microsystems, Inc. “jdb - The Java Debugger.”
(URL: `http://java.sun.com/products/jpda/doc/soljdb.html`)

[36] Sun Microsystems, Inc. “Java Platform Debugger Architecture.”
(URL: `http://java.sun.com/products/jpda/`)

[37] Sun Microsystems, Inc. “Java Remote Method Invocation.”
(URL: `http://java.sun.com/products/jdk/rmi/`)

[38] Sun Microsystems, Inc. “JSR 14: Add Generic Types To The Java Programming Language.”
(URL: `http://www.jcp.org/jsr/detail/14.jsp`)

[39] Sun Microsystems, Inc. “NetBeans.”
(URL: `http://www.netbeans.org`)

[40] L. Williams, E. Wiebe, K. Yang, M. Ferzli, C. Miller. *In Support of Pair Programming in the Introductory Computer Science Course*. Computer Science Education, September 2002.

[41] C. Wege, F. Gerhardt. *Learn XP: Host a Bootcamp Extreme Programming Examined*. Addison-Wesley, 2001.

[42] Xinox Software. “JCreator.”
(URL: `http://www.jcreator.com`)