Improving the Usability of the ESC/Java Static Analysis Tool

A Thesis
in TCC 402

Presented to

The Faculty of the
School of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Science

by

Michael Peck

March 25, 2004

____

Approved ______________________ (Technical Advisor)
Professor David Evans

Approved ______________________ (TCC Advisor)
Professor Claire Chantell

Preface

I'd like to thank my technical advisor, David Evans, for introducing me to the thesis topic, providing me with relevant comments and feedback, and supporting my work on this project during the summer of 2003. I'd also like to thank my TCC advisors, Bryan Pfaffenberger and Claire Chantell, for their assistance with preparing this document. This project would not have been possible without the support of Derek Leach of ITC Labs and Classrooms, who went above and beyond the call of duty by helping me install new versions of the ESC/Java plugin on the ITC lab computers well after the ITC deadline for software installations had past. Many revisions to the software were necessary throughout the course of the fall 2003 semester, and Derek was very helpful in pushing revised copies of the plugin to the lab machines as needed. Thanks also to Chris Frost for being the not-so-anonymous test subject for the plugin's usability evaluation. Last but not least, I'd like to thank the students and assistant coaches of CS 201J in fall 2002 and fall 2003 for their participation and feedback.

Table of Contents

Preface ii
Table of Contents iii
Abstract iv
Chapter 1: Introduction 1
  1.1 Statement of Thesis 1
  1.2 Problem Background and Rationale 1
    1.2.1 ESC/Java 1
    1.2.2 Software Engineering Education and CS 201J 3
    1.2.3 Eclipse 4
    1.2.4 ESC/Java Plugin for Eclipse 5
    1.2.5 Social and Ethical Contexts of the Project 6
  1.3 Overview of Technical Report 6
Chapter 2: Background and Motivation 7
Chapter 3: ESC/Java Plugin for Eclipse 10
Chapter 4: User Experience 16
Chapter 5: Conclusions 19
Bibliography 21

Abstract

The Extended Static Checker for Java (ESC/Java) static analysis tool examines Java source code looking for errors that could lead to runtime exceptions. ESC/Java also has the ability to check annotations placed in source code by programmers that express design decisions about how their code works. Although ESC/Java was developed for industry programmers, these features make ESC/Java a powerful tool for use to teach software engineering concepts. Unfortunately, as a command-line tool, ESC/Java is not tightly integrated with any integrated development environments (IDEs) that are regularly used by developers to write and test software. Also, ESC/Java's warning messages, which provide details about coding errors that it found, can be difficult for students (and even experienced software developers) to understand. Students may also be unsure of how to write annotations to describe their code. My project analyzes the above problems with ESC/Java and alleviates some of them by creating a plugin to run ESC/Java from within the Eclipse IDE.

Chapter 1: Introduction

1.1 Statement of Thesis

The Extended Static Checker for Java (ESC/Java) static analysis tool examines Java source code looking for implementation errors. It can be a useful tool for teaching software engineering principles to computer science students. As a command-line tool, ESC/Java can be difficult for students (and even experienced software developers) to use. Integrating ESC/Java with the Eclipse IDE solves some of ESC/Java's usability problems.

1.2 Problem Background and Rationale

1.2.1 ESC/Java

ESC/Java is a static analysis tool developed by Digital Equipment Corporation's Systems Research Center (now HP Research). It analyzes properties of software source code without actually executing the code. ESC/Java can discover common coding mistakes that could, under a certain flow of execution, cause a program to act unexpectedly (and likely terminate with a runtime exception). These coding mistakes are generally not found by compilers and without static analysis would potentially only be discovered at runtime. Some of the problems ESC/Java can discover include:

- Possible division by zero
- Possible negative array index
- Array index possibly too large
- Possible attempt to allocate array of negative length
- Possible null dereference

ESC/Java can also use programmer-supplied annotations inside source code documenting design and implementation decisions. Using these annotations, it attempts to determine if the source code complies with the intentions of the programmer. ESC/Java is a powerful tool but has seen most of its use inside research labs only. Usability problems could be what is preventing ESC/Java from being used in commercial software development. The use of ESC/Java allows software flaws to be detected in the implementation phase of software development, instead of during the testing phase, or even worse, in the maintenance phase (after the software has already been released). Flaws in software are much cheaper to correct the earlier they are detected in the development cycle. HP Research mostly halted its work on ESC/Java in 2001, but the University of Nijmegen in the Netherlands resumed its development in 2003.

One potential drawback with using static analysis tools is that they have theoretical limits on their abilities [8]. ESC/Java is both unsound and incomplete. Unsoundness means that ESC/Java cannot catch all programming errors. Incompleteness means that ESC/Java will sometimes mistakenly point out an error where none exists. Software developers cannot assume that their code is bug-free simply because ESC/Java found no mistakes. Developers must still consider using other verification techniques such as code inspections and unit testing. Furthermore, when a software developer encounters an ESC/Java warning message, the developer must consider the possibility that it is a spurious warning (false positive). Fortunately, continued development of static analysis tools will decrease the likelihood of false positives and false negatives.

1.2.2 Software Engineering Education and CS 201J

CS 201J, a software engineering course at the University of Virginia, was first taught in Fall 2002 and was taught again in Fall 2003. It is typically taken by second-year students. The class teaches software engineering with a strong emphasis on producing secure and reliable code by performing careful design, specification, implementation, and testing. Many of the problem sets involve using ESC/Java to discover and correct flaws in given software source code. Students also use ESC/Java annotations to carefully specify the expected behavior of their own code as they write it. Students then check their code with ESC/Java to make sure it meets the given specification.

For each problem found by ESC/Java, a warning message is outputted with a description of the problem and the location in the source code where ESC/Java believes the problem exists. Unfortunately, these warning messages can be difficult for programmers to understand. A warning could also involve several sections of code, all located in different files. When presented with an ESC/Java warning, a programmer may be unsure of the best way to fix the problem behind the warning. For example, some kind of modification to their source code could be necessary to fix the problem. Or, just adding an annotation might be sufficient.

ESC/Java is a command line tool. Most programmers are accustomed to working in graphical integrated development environments (IDEs), which attempt to make the process of writing, compiling, and testing source code as easy as possible. Expecting programmers to manually invoke ESC/Java from a command line every time they want to use it to analyze their source code is impractical, as it would make it difficult for programmers to integrate ESC/Java into their development process.

Learning how to write annotations for ESC/Java has the side benefit of encouraging programmers to reason about the specification of their software before actually implementing code. Computer science courses often mention the need to specify software before implementation to students. However, with the small programs that students write in these classes, they can easily fail to see the importance of specification [7]. By using ESC/Java, students can get instant feedback about whether or not their code meets the specification that they wrote.

In the real world, especially today when software is more likely to be Internet-accessible, accidental and even malicious inputs need to be expected and correctly handled. Information security experts claim that security must be a fundamental part of software design, rather than just an afterthought. Students often consider software to be finished when it produces the correct output for a few test cases. Unfortunately, much of today's software is buggy, unreliable, and insecure. Students need to be taught in software engineering courses to attempt to logically reason about the correctness of their software under many potential input cases. Software engineering courses must stress the importance of producing secure, reliable, and easily maintainable software.

1.2.3 Eclipse

Eclipse is an open source IDE primarily developed by Object Technology International (now owned by IBM). It is primarily used for developing Java software, but its architecture is designed to be flexible through the use of plugins, which can be written by any third party to extend the functionality of Eclipse. Plugins exist, for instance, to allow Eclipse to be used to develop software using various other programming languages besides Java. Most plugins are released as open source, meaning the source code to the plugin is made freely available to its users. Users can make changes and improvements to the plugin, or adapt its code for use in other projects. This ability to adapt the code of existing plugins greatly simplified the task of writing an ESC/Java plugin for Eclipse.

1.2.4 ESC/Java Plugin for Eclipse

For each potential problem found by ESC/Java, a warning message is outputted with a description of the problem and the location in the source code where ESC/Java believes the problem exists. Unfortunately, these warning messages can be difficult for programmers to understand. A warning could also involve several sections of code, all located in different files. When presented with an ESC/Java warning, a programmer could be unsure of the best way to fix the problem behind the warning. For example, some kind of modification to their source code could be necessary to fix the problem. Or, just adding an annotation might be sufficient. Also, if programmers are running ESC/Java from the command line, they need to examine its output, read the file names and line numbers where problems exist, and manually jump to the code inside their editor. An ESC/Java plugin for Eclipse should attempt to alleviate as many of these shortcomings as possible. Most software engineering lifecycles consist of five phases: requirements analysis, specification and design, implementation, testing, and maintenance. Most problems in software are introduced in the beginning steps of the lifecycle, but they are often not detected until the testing or maintenance phases. However, the problems become much more expensive to correct the later they are found, especially if they are not found until the maintenance phase, after the product has shipped. By using a tool like ESC/Java, problems can potentially be found and corrected during the implementation phase of software development. Implementing ESC/Java to run within Eclipse every time the compiler is invoked means that programmers will immediately see ESC/Java's warning messages and correct the problems that it finds.

1.2.5 Social and Ethical Contexts of the Project

Our lives are impacted every day by the effects of buggy software. For example, an Internet worm targeting a version of Windows forced Maryland to close all of its Department of Motor Vehicles offices for a day in August 2003 [1]. Another Internet worm targeting Windows machines caused a widespread Bank of America ATM failure [4]. In the past, the software industry has not considered writing secure software to be a priority, though this is beginning to change. If static analysis tools like ESC/Java are easier to use, they could potentially be integrated into the software development process followed by industry.

1.3 Overview of Technical Report

Chapter 2 reviews relevant literature related to static analysis tools, their usability, and the use of static analysis tools in computer science education. Chapter 3 discusses some of ESC/Java's shortcomings and how an ESC/Java plugin for Eclipse will alleviate these. The process used to develop the plugin will then be described, followed in chapter 4 by an evaluation of the plugin's usability and effectiveness. The report also describes why ESC/Java and similar tools are useful in software engineering education, using the University of Virginia's CS 201J course as an example. Chapter 5 concludes the report with recommendations and suggestions for future work.

Chapter 2: Background and Motivation

The literature contains many sources describing formal methods and static analysis. However, there is a very limited amount of literature specifically studying the usability of static analysis tools or about the use of static analysis tools in computer science education.

Jeanette Wing argues [14] that formal methods, defined by her as "the specification and verification of hardware and software systems", should be closely tied into undergraduate computer science courses. She states that her "ideal is to get to the point where computer scientists use formal methods without even thinking about it." She mentions the importance of tools, including ESC/Java, for formal methods instruction and makes the insight that while mathematicians will work with just paper and pencil, computer scientists are used to using software-based tools such as graphical user interfaces and compilers to help them work out problems.

In some ways, compilers can be used "without even thinking about it." Eclipse [12], for example, automatically recompiles source code whenever a source code file is changed and saved. Any problems found during compilation are displayed at the bottom of the screen in a "Problems View" for the programmer to examine and fix. Unlike other IDEs, Eclipse tends to stay out of the programmer's way, yet still offers timesaving features to make the programmer's job easier, such as keyword and syntax coloring, context specific coding assistance, and automatic code formatting. Eclipse was built from the ground up with support for plugins. Plugins are modules that programmers can develop relatively easily to extend the functionality of Eclipse. For example, the Checkclipse plugin [13] analyzes the style of Java code (as opposed to the code's functionality) and reports problems alongside the Java compiler's error and warning messages in the "Problems View" as described above.

Work was done at HP Research (known as DEC at the time) to improve the usefulness of ESC/Java's warning messages. ESC/Java utilizes a theorem prover, which produces a counterexample showing where possible mistakes in the code exist. Converting the theorem prover's output to an understandable message that programmers can use to correct their mistakes is a difficult problem. Todd Millstein [11] came up with methods to extract a code trace from the ESC/Java theorem prover's counterexample output. The code trace shows programmers the specific path through the code that causes a problem to occur, allowing programmers to more easily correct the problem.

JML [2], the Java Modeling Language, can be used by software developers to formally document design decisions they make while implementing Java classes. ESC/Java's annotation language resembles the language of JML in many ways, and ESC/Java is currently being updated to fully comply with JML. A verification tool called JACK (Java Applet Correctness Kit) [3], created by Gemplus, also uses annotations specified in JML to attempt to prove properties about Java code. The developers of JACK, understanding the importance of making their formal methods approach available to programmers of all levels of expertise and the importance of letting programmers work in a familiar environment, have integrated JACK into Eclipse by creating a plugin.

The Fluid Project at Carnegie Mellon University [9] has created a prototype tool that performs static analysis of source code to give its programmers some assurance about the software's properties. The project has focused on making its tools usable by common software developers with minimal effort. In order to pursue this goal, the project created a plugin for Eclipse. The plugin features a “Code Assurance Information” view inside Eclipse showing properties, such as potential race conditions, in the analyzed Java source code. Like ESC/Java, Fluid uses programmer-supplied annotations documenting design decisions.

ESC/Java has been used at a few universities for teaching purposes. At the University of Virginia, ESC/Java is used in CS 201J, a software engineering course taken mainly by second-year engineering students. Problem sets were designed to show students the value of static analysis in developing software. Several problem sets require students to use ESC/Java annotations to document design decisions made while coding. The students were asked to run ESC/Java against their code and fix any warnings or errors reported before submitting the code for grading. ESC/Java is also used for teaching a course on software specification at Kansas State University. An advanced programming course at Embry-Riddle Aeronautical University is using ESC/Java along with some of the University of Virginia course materials.

Chapter 3: ESC/Java Plugin for Eclipse

This chapter documents the development methodology followed in the creation of the ESC/Java plugin for Eclipse.

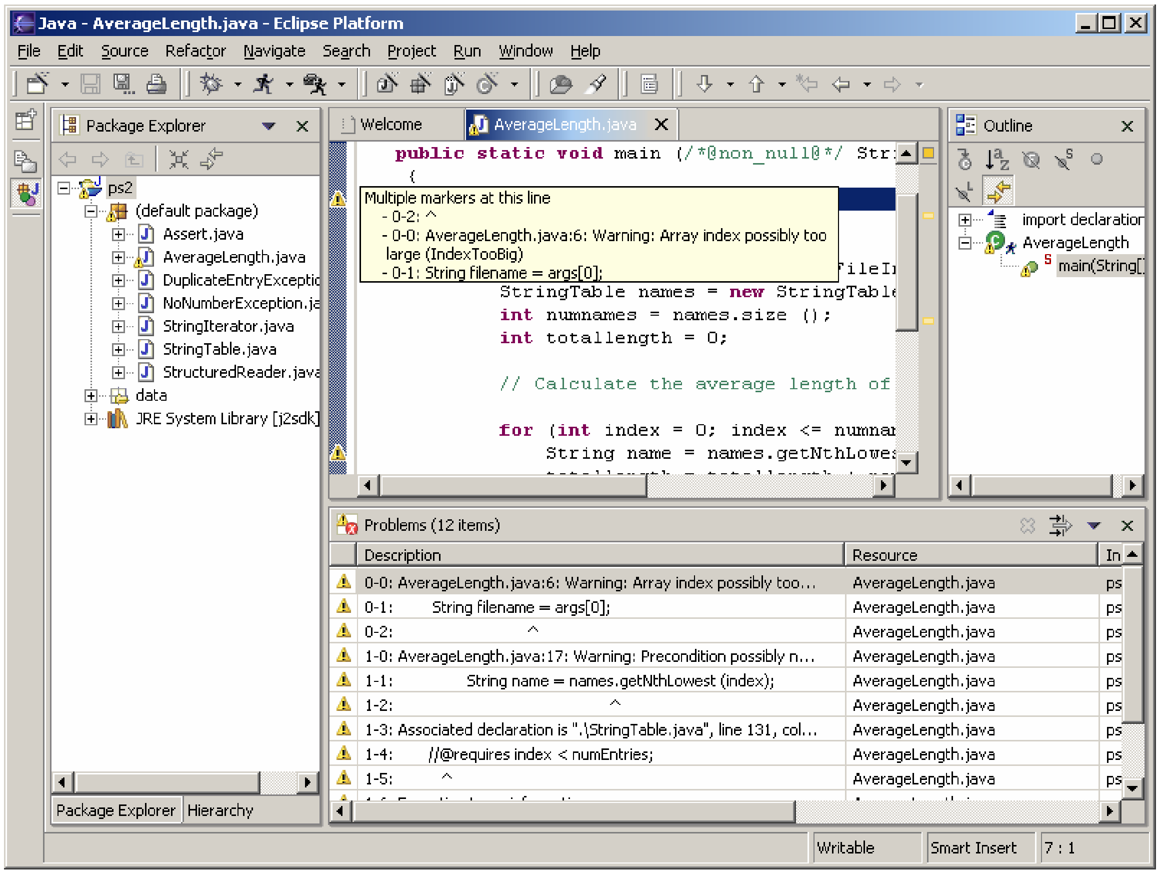
Students in the first offering of CS 201J in Fall 2002 developed Java code using the Windows command-line interface, running both ESC/Java and the Sun JDK from the command prompt. In Fall 2002, we were unable to find a good Java development environment that we could encourage students to use. For the Fall 2003 offering of the course, partly due to the struggles students had working from the command prompt, we decided to use a graphical user interface. We first evaluated the DrJava IDE from Rice University, which was developed with educational use in mind, and we even easily developed an ESC/Java plugin to run within DrJava. However, for various reasons we decided against DrJava and turned to the Eclipse project.

Eclipse is a much more powerful development environment than DrJava and would be more likely to be found in a commercial software development setting. It has many more features and also runs much quicker and more smoothly than DrJava. These benefits of Eclipse also added complexity when we decided to add support for ESC/Java to Eclipse. Fortunately, Eclipse is designed from the ground up to be extensible using plugins, which latch on to “extension points” within Eclipse to add functionality. However, there was still a huge learning curve involved in determining how to write the ESC/Java plugin.

Eclipse is open source. All of the source code for Eclipse is freely available, making it possible for a developer to see how various parts of Eclipse work. Since Eclipse is such a large project, availability of the source to Eclipse was not in itself enough to learn how to write a new plugin. To find a place to begin, I searched various Eclipse web sites, looking for an already existing plugin with similar functionality to what an ESC/Java plugin would need. I came across the Checkclipse plugin, which runs Checkstyle against Java code. Checkstyle looks for violations of various aesthetic coding issues which can make source code hard for humans to read. Checkclipse automatically runs every time the code is compiled, and upon finding a violation adds a message to Eclipse's "Problems View" alongside the compiler's warnings and errors. I knew that an ESC/Java plugin would need to do the same thing, only instead of running Checkstyle, it would run ESC/Java and interpret ESC/Java's output, placing the output in the "Problems View".

Checkclipse is open source, licensed under the Mozilla Public License (MPL). The license allows me to make and release modifications to Checkclipse. I removed all of the code in Checkclipse referring to Checkstyle. I then added code to make the plugin execute ESC/Java instead and interpret its results. I also changed the Preferences and Properties menus to allow users to specify the location of the ESC/Java installation on their system and to decide on a project-by-project basis whether or not ESC/Java should be used or not. Checkclipse incorporated the code to Checkstyle within the plugin, but because ESC/Java is a large piece of software with a complicated installation required, I chose to make the ESC/Java plugin depend on ESC/Java being installed separately on the computer. The ESC/Java plugin executes ESC/Java as a separate program. Since the software was to be installed on the ITC lab computers and used for other purposes besides just CS 201J software development, the plugin must be explicitly enabled on software projects for which it is to be used.

When ESC/Java is run against Java source code files, it produces detailed warning messages about problems that are found. A message, such as “possible null deference”, is printed. Then, the segment of the code that the message applies to is printed out. Then, other relevant sections of the code are printed out. Eclipse’s Problems View is only designed for simple compiler warning and error messages consisting of the name of the source file where the problem occurred, the line number, and a brief one-line message. However, ESC/Java’s messages consist of multiple lines and refer to multiple areas of source code, potentially spread out among several files. Interpreting these messages in Eclipse proved to be the most complicated task in developing the plugin. We took the approach of representing each line of an ESC/Java warning message as an individual “problem” in the problems view, and put numbers in front of each description so they would sort in order and be seen as the user as representing a single ESC/Java warning message. This approach, depicted in Figure 3.1, was extremely confusing to users, as they expect each problem to be shown only once, and the messages weren’t always sorted properly. This limitation was in place with the version of the ESC/Java plugin used for CS 201J in Fall 2003. The usability issues with this approach forced us to partially abandon the plugin and tell students how to run ESC/Java in a text console within Eclipse, which got rid of advantages like being able to click on warning messages and have the editor jump to the relevant code.



```
    public static void main (/*@non_null@*/ Str
      {
                       FileI
           StringTable names = new StringTabl
           int numnames = names.size ();
           int totallength = 0;

           // Calculate the average length of

           for (int index = 0; index <= numnar
               String name = names.getNthLowe
```

```
Multiple markers at this line
  - 0-2: ^
  - 0-0: AverageLength.java:6: Warning: Array index possibly too
   large (IndexTooBig)
  - 0-1: String filename = args[0];
```

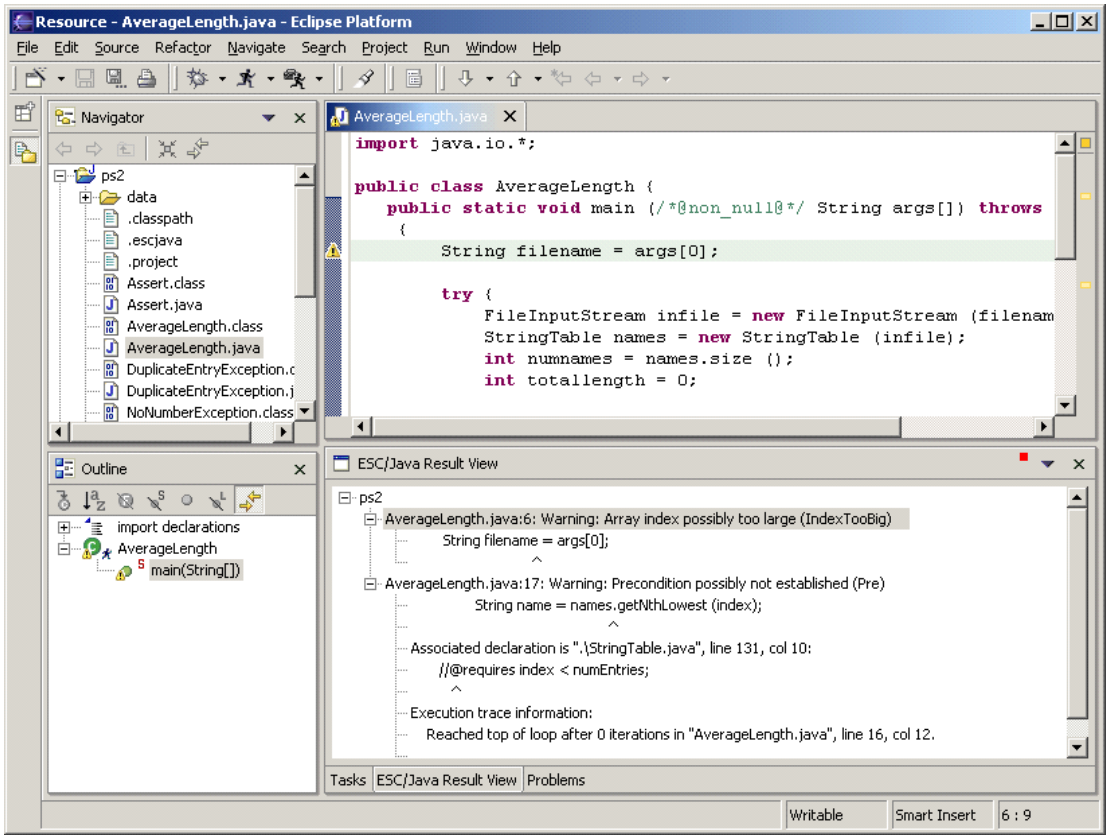
Problems (12 items)

	Description	Resource	In
⚠	0-0: AverageLength.java:6: Warning: Array index possibly too...	AverageLength.java	ps
⚠	0-1: String filename = args[0];	AverageLength.java	ps
⚠	0-2: ^	AverageLength.java	ps
⚠	1-0: AverageLength.java:17: Warning: Precondition possibly n...	AverageLength.java	ps
⚠	1-1: String name = names.getNthLowest (index);	AverageLength.java	ps
⚠	1-2: ^	AverageLength.java	ps
⚠	1-3: Associated declaration is ".\StringTable.java", line 131, col...	AverageLength.java	ps
⚠	1-4: //@requires index < numEntries;	AverageLength.java	ps
⚠	1-5: ^	AverageLength.java	ps

Figure 3.1: Illustration of an older revision of the ESC/Java plugin, as used in CS 201J in Fall 2003.

The final approach to displaying the messages that we took was to create a separate Eclipse view to show the ESC/Java warning messages. We still use the Problems View to show the first line of each message, which indicates the general problem. The "ESC/Java Result View", as we called it, shows the first line of each message and allows users to click on an expansion button to view further lines of any message. With this approach, the Problems View is still somewhat useful, but the ESC/Java Result View must be used to get more details. We did not want to create a separate view, which clutters up the Eclipse workbench, but it proved to be necessary due to a lack of flexibility with extending the Problems View. Double-clicking on a message in the ESC/Java Result View will make the Eclipse editor jump to the relevant section of the source code.

Problem sets and other course materials using ESC/Java were developed for use in CS 201J at the University of Virginia. These problem sets are freely available for download at the CS 201J web site and are being used by software engineering courses at other universities. Problem Set 2: "Using Data Abstractions" introduces students to ESC/Java by showing how it can be used to find programming mistakes in a given piece of software, AverageLength.java. Problem Set 2 is described in further detail in the Results chapter, as it was used to examine the usability of the ESC/Java plugin. Problem Set 3: "Implementing Data Abstractions" introduces students to understanding ESC/Java annotations describing design decisions; specifically, introducing students to how invariants can be used to specify the desired behavior of a data abstraction. Problem Set 4: "Designing with Data Abstractions" asks students to create their own data abstractions to solve a problem, specifically, to find solutions to the Cracker Barrel peg game. Students must provide specifications for the data abstractions they create, and write ESC/Java annotations documenting these specifications.



```
import java.io.*;

public class AverageLength {
  public static void main (/*@non_null@*/ String args[]) throws
    {
      String filename = args[0];

      try {
        FileInputStream infile = new FileInputStream (filenam
        StringTable names = new StringTable (infile);
        int numnames = names.size ();
        int totallength = 0;
```

```
ps2
  AverageLength.java:6: Warning: Array index possibly too large (IndexTooBig)
        String filename = args[0];
                              ^
  AverageLength.java:17: Warning: Precondition possibly not established (Pre)
            String name = names.getNthLowest (index);
                                  ^
    Associated declaration is ".\StringTable.java", line 131, col 10:
        //@requires index < numEntries;
          ^
    Execution trace information:
      Reached top of loop after 0 iterations in "AverageLength.java", line 16, col 12.
```

Figure 3.2: Illustration of the newest revision of the ESC/Java plugin.

Chapter 4: User Experience

We carried out an informal evaluation to gauge the effectiveness of the ESC/Java plugin for Eclipse. A volunteer, a fourth-year computer science student familiar with Java (but who had not previously used ESC/Java) was given a Java source file with several problems. The source file, AverageLength.java, was taken from CS 201J Problem Set 2. The AverageLength program takes a text input file consisting of lines in the format "name: value" where name is a baby's name and value is the percentage of babies born in a certain year with that name. The program contains several problems. Some of these problems are very subtle: the program will execute properly in most cases but will fail under certain conditions. These problems might not be revealed during testing or even during inspection of the source code. However, the problems are revealed by ESC/Java. This problem set demonstrates the usefulness of ESC/Java to students in the course. The first problem detected by ESC/Java is a lack of bounds checking. The AverageLength program takes as an argument the name of a file to be analyzed. However, it never checks to make sure that a filename is actually provided. If the program is run with no arguments, an exception will occur and the program will crash. The second problem is a precondition violation. A loop in the program accesses all of the baby names and percentage values (stored in a table) to average the values. Due to a typo in a loop's termination condition, the loop executes one too many times, and attempts to access the table of baby names and values with an index value one too high. The table has a precondition stating that the index value used cannot exceed the number of items in the table. ESC/Java detects the possible violation of this precondition. The third problem is a potential division by zero. When the AverageLength program finally computes the average length of each name, it divides the total length by the number of names. If no names exist, a division by zero occurs, causing an exception. In the CS 201J problem set, students are asked to explain what problems in the program are causing the ESC/Java warning messages. They are also asked to give example situations that would cause exceptions with the problems in place. Then, students are asked to correct the problems.

To carry out our usability experiment on the ESC/Java plugin, a subject was asked to complete the above portion of the CS 201J problem set. He sat in front of a computer in the ITC Stacks computer lab. It already had Eclipse running with the AverageLength program open. The ESC/Java plugin was enabled and the ESC/Java Result View was open. He was observed while carrying out the tasks, with careful attention paid to any malfunctions that he ran into. He was instructed that the software was the subject of the evaluation, not him. He was asked to speak his thoughts aloud as they went through the tasks, and careful notes were taken about his performance. After the completion of the tasks, the subject was asked to describe his experience with the ESC/Java plugin and suggestions for improvement. The student took 5 to 10 minutes to complete the tasks. He began by carefully reading the instructions given to him. He then examined the ESC/Java Result View and began double-clicking on its messages, which made the Eclipse editor jump to the relevant code. Using ESC/Java's messages, he explained the problems with the code. He then corrected the "Precondition possibly not established" problem. He did not realize that saving a file in Eclipse makes it automatically compile the file and run ESC/Java against it. This fact was explained to him, as well as the fact that he needed to click "Refresh" within the ESC/Java Result View to make the view refresh with the changes. He then explained the problems with the other two messages reported by ESC/Java and how to fix them. He was asked to explain some of the confusing aspects of the plugin:

- He thought the ESC/Java Result View should automatically refresh.
- The ESC/Java Result View showed extra lines within each problem's report that weren't necessarily relevant.
- He disagreed with the way Eclipse automatically rebuilds projects when source files are saved (this feature can be disabled by the user).
- He thought the plugin should jump to the relevant column of the code where a problem exists in addition to the relevant line of code.
- He suggested sorting ESC/Java's warning messages by type, importance, function name or file name.
- He suggested an option to not run ESC/Java automatically and instead make it run whenever the user specifically requests it, since ESC/Java can sometimes be slow to run, especially for large files.

Chapter 5: Conclusions

ESC/Java's use in CS 201J has shown that it is a useful tool for teaching software engineering concepts. However, usability problems have interfered with its use. The ESC/Java plugin for Eclipse should allow students to begin using ESC/Java more easily. It displays the results found by ESC/Java within Eclipse and allows students to quickly jump to the relevant location in the source code where each problem is found. Any tool, like ESC/Java, that can help teach students the importance of creating secure and reliable software should be a strong component of any software engineering course. In the future, the role of ESC/Java in other computer science courses should be examined. For example, ESC/Java could be useful in data structures and algorithms courses.

Much room for future work on the ESC/Java plugin still remains. First of all, the plugin does not extend the text editor within Eclipse. A good future enhancement for the plugin would be to make it extend the editor to recognize ESC/Java annotations, instead of just seeing them as Java comments. The annotations should be colored differently and should be formatted properly. Also, the plugin would then be able to check the syntax of ESC/Java annotations as well as assist the programmer in writing them, by, for example, providing auto-completion capability. Another useful addition would be to make use of Eclipse's Quick Fix capability, which suggests code modifications to the programmer that would fix warning messages reported by the Java compiler. Some ESC/Java warning messages can easily be corrected with a minor code modification or the addition of an annotation. Quick Fix options should be provided by the plugin to programmers to make these modifications for them. Last of all, ESC/Java warning messages often include a backtrace through the program code. For example, if a precondition is violated, ESC/Java prints out the code segment where the precondition is violated, followed by the code location where the precondition is actually defined. Clicking on the line of the warning message showing where the precondition is defined should jump to that location in the source code. The test subject's suggestions, shown in chapter 4, are also valuable ideas that should be incorporated into future versions of the ESC/Java plugin.

Bibliography

[1] Associated Press. "Maryland DMV Set to Reopen." August 23, 2002. `http://www.foxnewschannel.com/story/0,2933,94583,00.html`

[2] Lilian Burdy, Yoonsik Cheon, David Cok, et al. *An overview of JML tools and applications.* Eighth International Workshop on Formal Methods for Industrial Critical Systems. June 2003.

[3] Lilian Burdy. *JACK (Java Applet Correctness Kit).* `http://www.gemplus.com/smart/r_d/trends/jack.html`

[4] CNN. "Computer worm grounds flights, blocks ATMs." January 26, 2003. `http://www.cnn.com/2003/TECH/internet/01/25/internet.attack/index.html`

[5] Compaq Systems Research Center. *Extended Static Checking for Java.* `http://research.compaq.com/SRC/esc/`

[6] David Evans. *Teaching Software Engineering Using Lightweight Analysis.* NSF CCLI Proposal. June 2001.

[7] David Evans and Michael Peck. *Simulating Critical Software Engineering*. University of Virginia Computer Science Technical Report.

[8] Cormac Flanagan, et al. *Extended Static Checking for Java*. PLDI, June 2002.

[9] Aaron Greenhouse, et al. *Using Eclipse to Demonstrate Positive Static Assurance of Java Program Concurrency Design Intent.* http://www.fluid.cs.cmu.edu:8080/Fluid/FluidOOPSLAposter.pdf

[10] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual.* Compaq Systems Research Center Technical Note 2000-002.

[11] Millstein, Todd. *Towards More Informative ESC/Java Warning Messages.* In *Selected 1999 SRC Summer Intern Reports*, compiled by James Mason, Technical Compaq Systems Research Center Technical Note 1999-003, November 1999.

[12] Object Technology International. *Eclipse Platform Technical Overview.* February 2003. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`

[13] Marco van Meegen. *Checkclipse plugin for Eclipse.* http://www.sourceforge.net/projects/checkclipse

[14] Jeanette Wing. *Weaving Formal Methods into the Undergraduate Computer Science Curriculum.* 8th International Conference on Algebraic Methodology and Software Technology, May 2000.