

JAVAWIZARD:
INVESTIGATING DEFECT DETECTION AND ANALYSIS

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAII IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

MAY 1998

By

Jennifer M. Geis

Thesis Committee:

Philip M. Johnson, Chairperson
James Corbett
Martha Crosby

We certify that we have read this thesis and that, in our opinion, it is satisfactory in scope and quality as a thesis for the degree of Master of Science in Information and Computer Sciences.

THESIS COMMITTEE

Chairperson

© Copyright 1998

by

Jennifer M. Geis

To
Mom and Dad

Acknowledgments

Some people may think that the writing of a thesis is a solitary journey. It is, in the sense that you are stuck doing the work, but without the support of others, I imagine that few would ever finish the job. I am very fortunate, for I have many people who have helped me along the way.

To Philip Johnson, my advisor. I can't begin to recall all the times you acted as my cheering section, urging me on to accomplish things I never would have dreamed of just a few years ago. I was muddling through the undergraduate program when you held out the prospect of joining CSDL. With some trepidation, I accepted, not having a clue what was in store for me. You have changed the way I view education, software development, and my future. I cannot thank you enough.

To all past and present members of CSDL. You have critiqued my work and suggested many improvements which I would not have thought of myself. This thesis has been greatly enhanced as a result of your willingness to give of your time and experience.

To the person who started me on JWiz, Bruce Foster. Who would of thought that a summer in New Hampshire would lead to this? I thank you for taking the gamble of having me as an intern, and for offering me the opportunity to continue my efforts back in Hawaii.

To Dale Skrien, Jeremy Lueck, and Mitchell Goodman. You provided me with the underlying functionality on which I run JWiz. Thank you for considering my suggestions and requests for your system.

To Bill McKeeman, Will Kling, Steve Rodgers, and all the people at Digital who offered their ideas and support.

I would like to thank my mother and father for attempting to look interested when I talked about JWiz, sympathizing with me when I was worried, and making an effort to understand my topic. I realize much of what I said probably had little meaning to you, but I appreciate your letting me use you as sounding boards. I would also like to thank you both for letting me set my own path through life and supporting the choices I make. For these reasons, this thesis is for you.

Abstract

This thesis presents a study designed to investigate the occurrence of certain kinds of errors in Java[7] programs using JavaWizard (JWiz), a static analysis mechanism for Java source code. JWiz is an extensible tool that supports detection of certain commonly occurring semantic errors in Java programs. For this thesis, I used JWiz within a research framework designed to reveal (1) knowledge about the kinds of errors made by Java programmers, (2) differences among Java programmers in the kinds of errors made, and (3) potential avenues for improvement in the design and/or implementation of the Java language or environment.

I performed a four week case study, collecting data from 14 students over three programming projects which produced approximately 12,800 lines of code. The JWiz results were categorized into three types: functional errors (must be fixed for the program to work properly, maintenance errors (program will work, but considered to be bad style), and false positives (intended by the developer). Out of 235 JWiz warnings, there were 69 functional errors, 100 maintenance errors, and 66 false positives. The fix times for the functional errors added up to five and a half hours, or 7.3 percent of the total amount of time spent debugging in test.

I found that all programmers inject a few of the same mistakes into their code, but these are only minor, non-defect causing errors. I found that the types of defects injected vary drastically with no correlation to program size or developer experience. I also found that for those developers who make some of the mistakes that JWiz is designed for, JWiz can be a great help, saving significant amounts of time ordinarily spent tracking down defects in test.

Table of Contents

Acknowledgments	v
Abstract	vii
List of Tables	xii
List of Figures	xiii
List of Abbreviations	xiv
1 Overview	1
1.1 Debugging...Ugh	1
1.2 The JavaWizard Solution	1
1.3 A Usage Scenario	3
1.4 The Origins of JWiz	6
1.5 Research Thesis	6
1.6 Organization of this Document	7
2 JavaWizard	8
2.1 JWiz Example	8
2.2 Usage Scenario	9
2.3 What JWiz Looks For	11
2.4 Design	11
2.4.1 Class Structure	12

2.5	Development Issues	13
2.5.1	Extensibility	13
2.5.2	Performance	14
2.5.3	What Makes a Defect?	14
2.5.4	Complexity: Number of Tests	15
3	Experimental Design	16
3.1	Overview of Hypotheses	16
3.1.1	Improvement to the Java Language/Environment/ Pro- grammers	16
3.1.2	JWiz vs. Manual Debugging	17
3.2	Experimental Procedures	17
3.3	Pilot Study	18
3.3.1	ICS 414 and CSDL	18
3.3.2	Initial Observations	18
3.3.2.1	Common Mistakes	19
3.4	Data	19
3.4.1	Size	19
3.4.2	Total Number of Errors in Test	20
3.4.3	Development Phase	20
3.4.4	Developer Experience	20
3.4.5	False Positives	21
3.4.6	Time	22

3.5	Subjects	22
3.5.1	ICS 311	22
3.5.2	Collaborative Software Development Laboratory	23
3.5.3	ICS 613	24
3.6	Means of Data Collection	25
3.6.1	ICS 311	25
3.6.2	ICS 613	27
4	Results	28
4.1	Defect Classification	28
4.2	Raw Data	29
4.3	Categorization	30
4.3.1	Developer Experience	30
4.3.2	Program Size	31
4.3.3	Development Phase	32
4.4	JWiz Effectiveness and Accuracy	32
4.5	Changes to Java	34
4.6	Changes to JWiz	34
4.7	Anecdotal Experiences	36
4.7.1	Code Clean-up	36
4.7.2	Defect Denial	36
4.7.3	Post-Test Usage	38
4.7.4	The “String Equals” Bug	39

5	Related Work	41
	5.1 Automated Debugging	41
	5.2 Manual Debugging	44
6	Conclusion	46
	6.1 Contributions of this Research	46
	6.2 Future Directions	46
	6.2.1 Error Correctness Estimation	46
	6.2.2 Publicly Available JWiz	47
	6.2.3 Shrinking Test Time	47
A	JWiz Defect Classes	49
B	ICS311 Handout	51
	Bibliography	57

List of Tables

2.1	JWiz Package Statistics	13
4.1	Generated Warnings	29
4.2	Test Phase Statistics	33
4.3	JWiz Statistics	33
A.1	JWiz Defect Tests	49

List of Figures

1.1	JWiz-mode in Emacs	5
2.1	Selecting the File to Run JWiz on	10
2.2	Selectable Results of Running JWiz	11
3.1	The GUI Application	25
3.2	The Text-only Application	27
4.1	General Experience vs. Java Experience	30
4.2	New and Changed LOC vs. Defects Made	31
B.1	FileDialog	53
B.2	JWiz Results	54

List of Abbreviations

API	Application Programming Interface
BNF	Backus-Naur Form
DRL	Defect Recording Log
CSDL	Collaborative Software Development Laboratory
FTR	Formal Technical Review
GUI	Graphical User Interface
HTML	HyperText Markup Language
ICS	Information and Computer Sciences
KLOC	Thousands of Lines Of Code
LOC	Lines Of Code
PSP	Personal Software Process
URL	Uniform Resource Locator
WWW	World Wide Web

Chapter 1

Overview

1.1 Debugging...Ugh

All programmers inject defects into their code. Even experienced developers typically inject a defect about every 10 lines of code[8]. Half of these defects are normally found by the compiler, while the rest must be found through reviews, testing, or by the users.

Every programmer hates debugging their work. If you were to guarantee a software developer that she would never have to spend another minute tracking down bugs in her code, she would probably worship you for life. In many cases, the most time consuming part of debugging usually isn't removing the defect, but tracking it down in the first place. All programmers can remember that horrible night spent searching for the cause of some strange behavior in their program. The night frequently ends with the programmer groaning in disgust when they finally spot the offending line.

1.2 The JavaWizard Solution

JavaWizard (JWiz) can't prevent those late nights, but it can make them happen a little less often. JWiz is a Java source code analyzer. It scans through code looking for common programming constructs which, though legal in the Java language, are still likely to cause errors. Due to its nature, JWiz is intended for use after the

first clean compile and before testing. JWiz requires the code to be compilable, and thus it does not concern itself with syntactic errors. Instead, JWiz notifies the user of possible run-time problems. For example, the following code prompts a warning from JWiz.

```
public boolean isEqual(String stringIn){
    String myString = "hi";
    if (stringIn == myString){
        return true;
    }
    return false;
}
```

JWiz would give the warning “Comparing strings using ‘==’ instead of the ‘equals’ method.” Usually, programmers want to compare the *contents* of two strings while the above code compares the memory addresses of the strings. Since the two strings do not reference the same object, the method will always return false. This is the case even if the strings’ contents are identical. I refer to this as the “String equals” bug.

Used in this fashion, JWiz serves as a kind of “smart compiler” that can inform the programmer about constructs that will likely cause your program to behave in unexpected ways, as opposed to the syntactic errors that a compiler normally captures. Basically, JWiz is like a Lint[4] for Java. What distinguishes JWiz from Lint, other than the programming language, is the way in which I have used it for this research.

JWiz provided me with the opportunity to study, for the errors it catches, what kinds of programmers make them, what kinds of programs they are made in, and (ultimately) how the programmer and perhaps even the Java language itself could change so that these errors would not occur.

One might ask, given a mechanism to catch these errors, why bother worrying about how to change programmers or the language? Why not just use the tool to catch the errors?

My response is that JWiz can be used that way, but the kinds of semantic errors JWiz catches are only a narrow subset of all the possible errors a programmer could make. Although JWiz may signal that there are a lot of errors present that it knows about, that could mean that there are a lot of errors present that it doesn't know about.

On the other hand, the kinds of improvements a programmer might make in response to JWiz feedback (such as changes to coding style, or the use of reviews), or the kinds of changes that could ultimately be made to the Java language (such as redesigns of the class libraries or interfaces) or environment could not only eliminate the JWiz errors, but also other errors not caught by JWiz. For this reason, I believe JWiz has important potential for software quality improvement beyond its application as a Lint for Java programs.

Another argument for programmer/language improvement is that JWiz is not infallible. JWiz is a new tool. There are some circumstances I did not consider. I could not anticipate all the possible ways a programmer could make a particular error. What happens if the developer gets a false sense of security? They might run JWiz in the process of looking for a bug, and then think "It can't be here since JWiz would have told me about it, so I won't waste my time looking."

1.3 A Usage Scenario

In UNIX or DOS, JWiz can be run as an application invoked via the command line. The user goes to the directory containing the files on which they wish to run JWiz and types the following command.

```
'jwiz *.java'
```

JWiz scans the files and prints out a listing of the warnings generated. The listing might look like this:

```
TestFile.java:23: GUI component stopButton not added to a container.  
TestFile.java:30: addActionListener not called on button 'goButton'.  
TestFile.java:89: Multiple objects added to same BorderLayout area.  
TestFile.java:131: Local variable 'varString' overshadows field.  
TestFile.java:171: Local variable 'testString' not used.  
TestFile.java:228: Comparing strings with == instead of the 'equals'  
method.  
TestFile.java:354: Comparing strings with == instead of the 'equals'  
method.  
TestFile.java:357: Comparing strings with == instead of the 'equals'  
method.
```

I have also implemented a JWiz mode for XEmacs which allows the JWiz output to be mouse selectable as shown in Figure 1.1 on page 5.

Once the JWiz results are displayed, the user then goes through the list and determines which were “real” errors (which must be fixed in order for the program to function properly), which were “maintenance” errors (which do not cause problems in the running of the program, such as unused parameters, but which indicate “bad style”), and which were “false positives” (where JWiz flagged something the programmer really intended to do as an error).

When used with the intent of self improvement, the user may note that she has a number of errors where she compared two strings using two equals signs ‘==’ instead of the ‘equals’ method. One possible step she might take at this time is to start a checklist of her common errors. She might use this checklist for future code reviews.

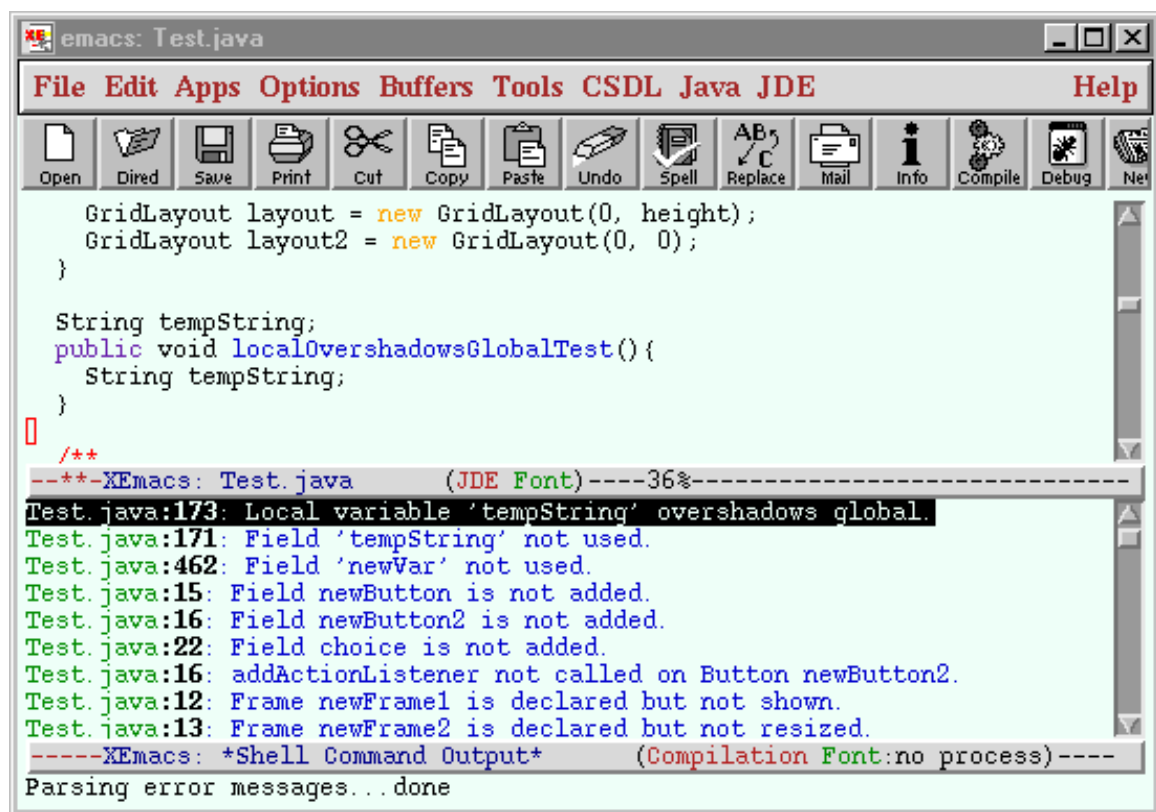


Figure 1.1: JWiz-mode in Emacs

By saving the JWiz output into a log, she would be able to see patterns of errors. For example, she might notice that where one error is found, more generally occur. Or she might notice that she has a habit of making the same errors across multiple projects.

1.4 The Origins of JWiz

In 1995, I participated in a course on Watts Humphrey's Personal Software Process (PSP)[8]. During this course, students followed a strict software development process consisting of the following phases: plan, design, code, compile, test, and postmortem. Students were required to keep track of all defects they found in their programs. For each defect, the data collected included a description of the defect, the phase of development during which the defect was injected into the program, the phase in which it was removed, and the amount of time it took the developer to find and remove it.

In looking over the data, I noticed that the defects that took the most time to find and remove were the ones that made it past the compilation phase and didn't rear their ugly heads until the testing phase. These were the defects that resulted in late nights for me.

In the summer of 1997, I was invited to do an internship at Digital Equipment Corporation in Nashua, New Hampshire. I found the description of JWiz among the list of possible projects that Digital offered me. With my previously collected defect data, I figured this was the perfect project and accepted the offer.

1.5 Research Thesis

The hypotheses of this research are:

1. The use of JWiz in the context of this research will reveal areas of improvement for Java programmers and the Java language and/or environment.
2. JWiz uncovers certain classes of defects more efficiently than manual debugging.

To test these hypotheses I performed a case study on several groups of software developers and obtained the results of running JWiz on their code.

1.6 Organization of this Document

Chapter Two provides an introduction to the JWiz program with a focus on user scenarios. This chapter also covers some of the design decisions made during the course of the program's development and some of the problems I encountered.

Chapter Three focuses on the experiment. There, I discuss the research hypotheses and how they were tested. This chapter contains the results from a pilot study and how it affected the experimental design.

Chapter Four presents the experimental results. Sections include a break-down of data according to developer skill, program size, etc., along with answers to the questions raised during the pilot study. I also relate some unanticipated results and anecdotal experiences.

Chapter Five contains an overview of related work. I describe a number of software engineering tools with capabilities or underlying theories similar to those of JWiz. Each of these tools is compared and contrasted with JWiz.

Chapter Six presents the conclusions of this research. I discuss the contributions of this research and some ideas for future research.

Chapter 2

JavaWizard

2.1 JWiz Example

JWiz does static analysis on Java source code. This is accomplished through the use of a parse tree and symbol table. When JWiz is first invoked, it generates the parse tree and symbol table. Next, JWiz begins a recursive descent through the parse tree. The items in the tree are checked to see if they could be the starting element of any of the errors that JWiz knows about.

The following is an extension of the “String equals” bug example given in the overview: the comparison of two strings using two equals signs instead of the equals method.

```
public boolean isEqual(String string1, String string2){  
    if (string1 == string2){  
        return true;  
    }  
    return false;  
}
```

The above code results in the following parse tree. The parse tree information given is somewhat condensed from what JWiz really uses, but all the necessary information is intact.

```

Method: boolean isEqual(String string1 String string2)
  IfStatement
    EqualityExpression
      Primary prefix: string1
      Primary prefix: string2
    Block
      ReturnStatement: boolean
ReturnStatement: boolean

```

In this case, JWiz scans through the parse tree until it encounters the `IfStatement` node. When this is found, the node's child is retrieved. As this child node is an `EqualityExpression` it knows the user used two equals signs to compare something, however, it does not yet know what the user is comparing. Next it gets the arguments of the `EqualityExpression`. JWiz finds two variables called `string1` and `string2`. It uses the symbol table to obtain the types of these variables. Since both turn out to be `Strings`, JWiz issues the warning “Comparing two strings using ‘==’ instead of the ‘equals’ method.”

2.2 Usage Scenario

Assume that Anne is writing a program for a class project. She uses Emacs enabled with JWiz mode. She has finished coding and has obtained a clean compile. At this point, Anne invokes JWiz. She types the keystrokes “C-c C-j,” and a buffer appears with the mouse-selectable warnings as shown in Figure 1.1 on page 5. She clicks on the warning “Field ‘newButton’ is not added to a container,” and her cursor is moved to the line that generated the warning. She realizes that the button isn't going to show up in her display, and fixes the defect.

Another user, Joe, doesn't use Emacs. When he gets his first clean compile, he types ‘jwiz’ and hits enter. This causes a window to pop up and prompt him for the

file to run JWiz on. He chooses to run JWiz on the file LeapGUI.java as shown in Figure 2.1.

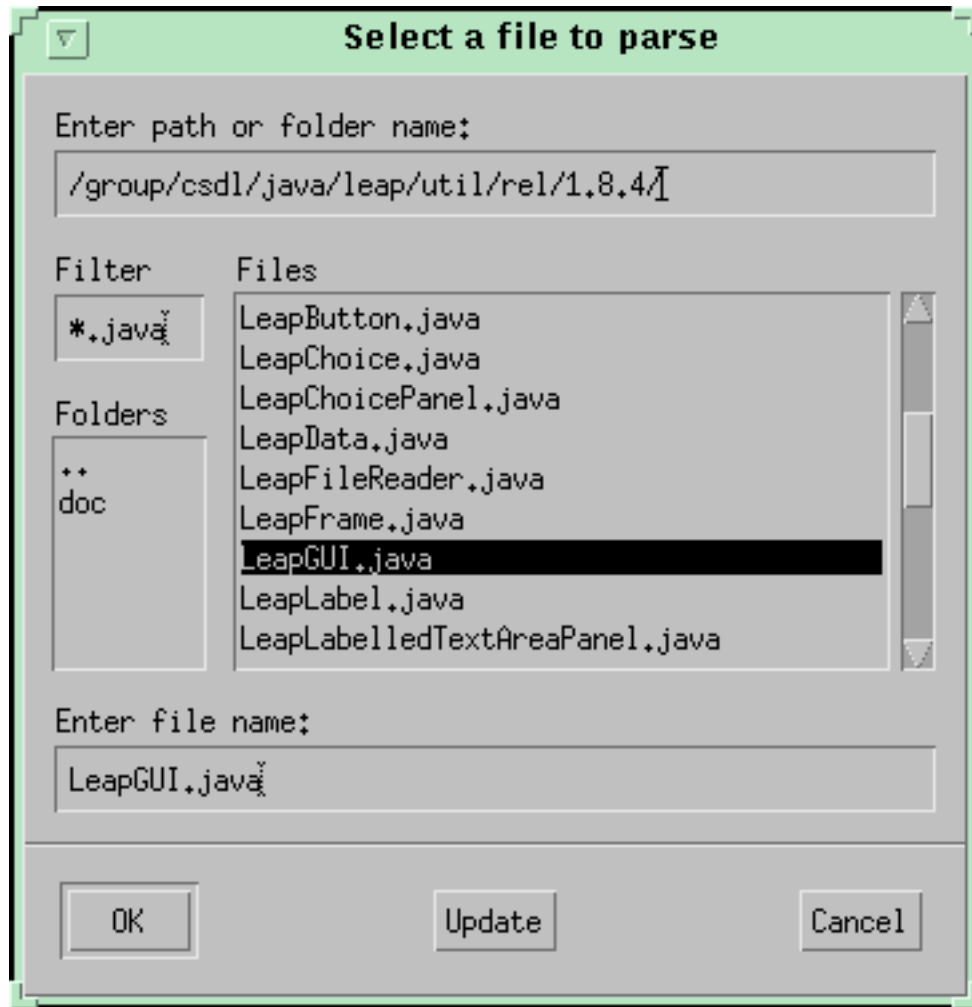


Figure 2.1: Selecting the File to Run JWiz on

He selects LeapGUI.java and hits the OK button. JWiz examines the file and brings up another window listing any defects that it has found (Figure 2.2 on page 11).

If Joe wanted to run JWiz on all his files he could have typed

```
'jwiz *.java'
```

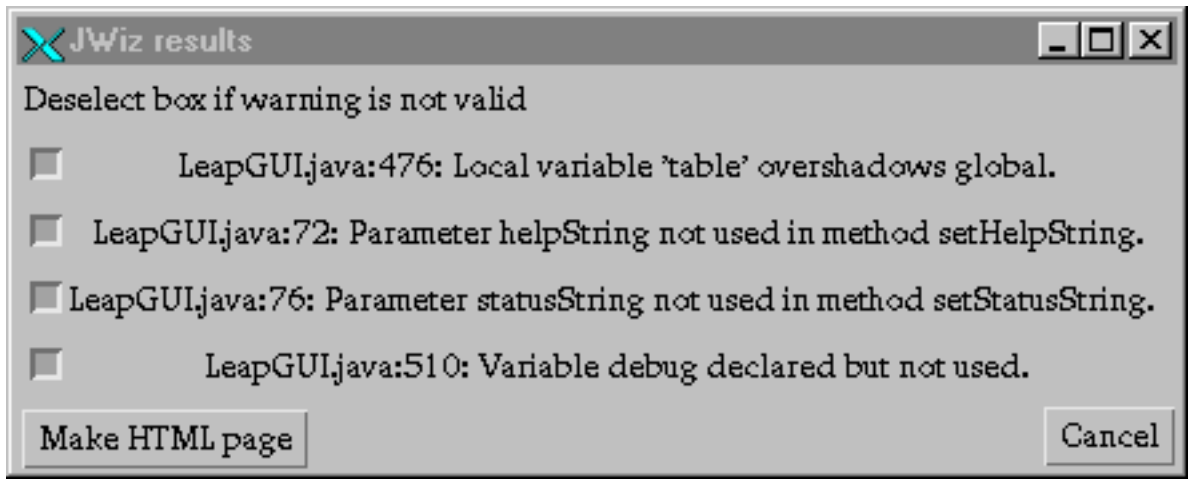



Figure 2.2: Selectable Results of Running JWiz

He could also have bypassed the file selection window by using

```
'jwiz filename.java'
```

2.3 What JWiz Looks For

JWiz focuses on run time errors. JWiz cannot look for syntactic errors by definition, since it uses a syntax tree that can only be built from a syntactically correct program. Many of the JWiz defect checks were derived from the data collected during my previously mentioned Personal Software Process class experience. I also obtained a significant number of warnings from the book “Java AWT Reference” by John Zukowski[12]. Appendix A contains a listing of all the defects JWiz currently tests for.

2.4 Design

JWiz is 100 percent pure Java code, so it should run on any operating system. JWiz utilizes parse tree and symbol table generators that were developed in the summer of

1997 at Digital Equipment Corporation by Jeremy Lueck, Mitchell Goodman, and Dale Skrien. JWiz runs the parser on the Java source code which generates the parse tree and the corresponding symbol table. JWiz then recursively traverses the tree looking for specific nodes. When one of these nodes are found, JWiz runs tests which check to see if this node is followed by others in a pattern which indicates the presence of a possible error. The symbol table is used for variable lookups and is written exclusively in Java as well.

2.4.1 Class Structure

JWiz is composed of three separate packages: JWiz, Parser, and Test.

The user invokes JWiz from the command line. JWiz then creates an instance of the Parser, passing it the name of the file to be analyzed. The Parser scans the file, setting up the parse tree and symbol table for JWiz to use.

JWiz then performs a recursive descent on the parse tree. With each node retrieved from the tree, JWiz passes control to Test.

The Test package is actually a bunch of small programs (tests) that are invoked according to the type of the node received by the JWiz class. There are three types of tests: class level, method level, and node level. The type of the node that is given to Test determines what kinds of tests can be run. For example, if the node is an instance of a class declaration, JWiz invokes the test that checks to see if there are any variables defined in the class' methods that overshadow the class variables. Likewise, if the node is a method declaration, it checks to see that all the parameters are used.

Below is a summary of the statistics of the three packages according to lines of code (LOC), number of classes, and number of methods.

Table 2.1: JWiz Package Statistics

	JWiz	Test	Parser
LOC	235	4155	17355
classes	2	39	93
methods	20	264	1301

Altogether, the three packages that make up JWiz take just over 1.2 MB of space. The application programming interface (API) for all three packages can be found under their respective package names at <http://www.ics.hawaii.edu/~csdl/java/jwiz/>.

2.5 Development Issues

Extensibility, performance, defect identity, and complexity were four of the major issues I confronted while developing JWiz.

2.5.1 Extensibility

Ideally, JWiz will be freely available through the Internet for all who are interested. Should this happen, developers will want to add their own tests to JWiz to accommodate the defects that affect them the most.

For this to happen, an intermediate layer will be required. Currently, in order to add a new JWiz test, the developer must understand the Backus-Naur form (BNF) for the Java language. Without knowing the patterns of the nodes in the parse tree, it is not possible to develop a test.

The Parser package offers another problem for the test developer. The developer would have to reference the API and become familiar with the methods used for the retrieval of variables and the like.

Most developers would prefer less cognitive overhead. This suggests the need for some layer of abstraction. Perhaps a set of classes of methods could be developed as a future project.

2.5.2 Performance

JWiz takes roughly twice as long to run as the Java compiler. For example, on a Sun Microsystems SPARCstation20, a 1,700 line program takes 32 seconds to compile, and 72 seconds to be checked by JWiz. JWiz could be sped up a little by making the Parser class static. The original version of the Parser was static, but in order to facilitate running JWiz through an applet, the Parser had to be made non-static. It would be a fairly simple task to revert the Parser back for anyone who wants an application-only version.

I tend to think that speed of execution is not important. Considering the amount of time a developer spends searching for code defects, it's worth taking the few extra seconds to run JWiz.

2.5.3 What Makes a Defect?

As mentioned previously in the chapter, the defects that JWiz checks for were selected from a variety of sources, including PSP defect logs and books listing Java “gotchas”. I selected defect tests if they matched one of the following criteria.

In the case of defects derived from PSP data, if they:

- Occurred in multiple PSP defect logs.
- Took an individual a non-trivial amount of time to fix.
- Represented a defect experienced by colleagues or myself.

In the case of defects derived from the “Java AWT Reference” book, if they:

- Represented a hard to find defect.
- Are relatively easy for JWiz to detect.

Originally, I intended to include only tests which checked for constructs which “are almost always wrong.” The problem with this plan is that I did not have any data on the likelihood that a construct was incorrect. Essentially, I had to guess the frequency of a construct representing an error. As a consequence, I found that some of the defect checks which I included are frequently false-positives. I have yet to decide on the future of these particular JWiz tests which offer little benefit.

2.5.4 Complexity: Number of Tests

I believe that there is an upper limit to the number of tests that JWiz can run on any given program. Although the user can add as many tests as they desire, the time to run JWiz will grow with each test. I believe JWiz could become too slow if it gets too large. To solve this, I will equip JWiz with the ability to turn off tests. The interface will show all available tests and permit the user to toggle the tests they wish to run. This will allow users to tailor the system to their own personal preference by selecting those defects which they frequently commit and omitting those tests which offer them little or no benefit.

Chapter 3

Experimental Design

3.1 Overview of Hypotheses

My research hypotheses are as follows:

1. The use of JWiz in the context of this research will reveal areas of improvement both for Java programmers and the Java language and/or environment.
2. JWiz uncovers certain classes of defects more efficiently than manual debugging.

The following four sections will discuss how the experiment was designed to address the above hypotheses.

3.1.1 Improvement to the Java Language/Environment/Programmers

In order to discover problem areas for Java developers and the Java language and/or environment, I needed to find out what defects were being made. Since JWiz provided a mechanism for defect reporting, I recorded all the defects (JWiz functional errors) with the intent of answering the following questions:

1. What defects occurred most frequently?
2. What defects could be avoided by changes to the Java language/environment?

3. How can developers change their programming habits to avoid these defects?

3.1.2 JWiz vs. Manual Debugging

In order to determine if JWiz is any more effective at locating defects than manual debugging, I needed to record data on how long it took people to find and remove the defects which JWiz can detect. I accomplished this by having students send me a copy of their code after they got their first clean compile, but before they started doing any testing. The students then went about their normal development. They were required to record all the defects they made, what the defects were, and how long they took to find and fix. After the programs were finished, I sent each student a listing of all the warnings that JWiz generated from their pre-test code. For each warning, they were asked to verify if it was a defect that they found, and if so, how long it took them to locate the source of the problem and fix it.

As JWiz finds errors essentially instantly, I could look at the students' responses to the warnings and see if JWiz is any more efficient than their manual debugging efforts.

3.2 Experimental Procedures

For this research, I designed a case study. A pilot study was followed by the experiment. Over a period of four weeks, JWiz was given to two groups of students and one research group of 5 graduate students.

3.3 Pilot Study

3.3.1 ICS 414 and CSDL

For the pilot study, data was collected from an upper level computer science course at the University of Hawaii. The course, Introduction to Software Engineering (ICS 414), focused on the PSP. As part of the students' activities, they were required to maintain a log of all the defects they detected during the development of nine Java programs. For each defect, the students would record a description of the defect, the phase in which the defect was injected into the program, the phase in which the defect was found and removed, and the amount of time it took to locate and remove it.

The graduate students in the Collaborative Software Development Laboratory (CSDL) research group also adopted the PSP. Since, all members of this software engineering group kept records of defects they made during program development, I utilized these records in my pilot study.

3.3.2 Initial Observations

The Defect Recording Logs (DRLs) from ICS414 and CSDL provided some interesting data. I focused my attention on defects found and fixed during the testing phase. I discovered that certain defects were popping up in multiple DRLs. These defects ranged in severity (in terms of amount of time to find and fix). As any software developer will tell you, the worst part of debugging is usually trying to find the location of the defect, not fixing it. In many cases, I noticed that the time recorded to find and fix a defect was non-trivial, but the defect's description indicated that the fix itself could have taken no more than a few seconds.

3.3.2.1 Common Mistakes

If you were wondering why I use the example of comparing strings using two equals signs instead of the equals method to explain what JWiz does, it is because the pilot study indicated that the “string equals bug” seemed to be the most common error. The severity of the defect varies according to the context in which it was made. The times spent removing this particular defect ranged from a minute or two to almost an hour.

Some other defects were not common, but were time consuming to find so I included them in JWiz. For example, the test “division result assigned to an int instead of a float” arose out of a student spending almost a half an hour on the same problem.

3.4 Data

Collecting the defects discovered by JWiz is not enough. That would only tell me whether or not the program works, not if it is really a useful tool. To understand more about the effectiveness of the tool, I decided to collect some other pieces of information as well, specifically, the size of the program, the phase of development at which JWiz was executed, and the developer’s experience in terms of number of years of programming, experience with Java, and number of languages. I also planned to track “false positives,” warnings which the developer decided were not valid.

3.4.1 Size

The size of the program is useful in determining JWiz’ effectiveness. By effectiveness, I mean the number of defects found per thousand lines of code. If JWiz reports only

one valid error, the effectiveness of JWiz to that program's developer varies depending on whether the program was one thousand lines of code or ten.

3.4.2 Total Number of Errors in Test

The effectiveness of JWiz also depends on the percentage of all errors found in the test phase that were detected by JWiz. If there were 40 errors found during test and JWiz caught only one, there is still much the developer must do.

However, the thing to keep in mind regarding effectiveness is that even if JWiz finds only a small proportion of errors, the errors it does find can still save significant amounts of time in debugging.

3.4.3 Development Phase

The phase of development in which the developer uses JWiz can have a big impact on the number and types of defects JWiz finds. If JWiz is used after compile and before test, it is probable that more warnings will be generated than if JWiz is used after testing is already completed.

Whether a code review is performed before or after using JWiz can have an affect on JWiz' effectiveness as well. If the developer has used JWiz before and noticed that there is a specific error that she makes frequently, than she might be watching out for it during a review, hence eliminating it before JWiz is run.

3.4.4 Developer Experience

There are a variety of things to consider regarding developer experience: amount of time doing programming in general, amount of time programming in Java, and the number of languages the developer has worked with.

I believe that developer experience will be a factor in what kinds of errors JWiz is likely to find. If the developer is a first year introductory student, she will probably not be using inheritance and inner-classes, so advanced defect checks are not likely to be invoked. On the other hand, she will probably make the mistake of not creating a listener for events or adding multiple components to the same area in a BorderLayout.

If the developer is experienced with Java, she could make the same mistakes as a novice, but she is more likely to make mistakes such as calling `Thread.suspend()` (this causes your program to hang). A beginning student would probably not be using threads, so she is unlikely to encounter this problem.

3.4.5 False Positives

The accuracy of JWiz can be determined by comparing the number of functional errors, maintenance errors, and false positives. If for every 10 warnings that JWiz generates, half are functional errors, then JWiz is accurate 50 percent of the time (maintenance errors are not counted towards accuracy). This measure of accuracy must be combined with effectiveness in order to find the usefulness of JWiz. If JWiz generates 20 warnings for a one thousand line program, at 50 percent accuracy the result will be 10 legitimate errors which the developer will not have to track down.

Although false positives are a nuisance, they can not all be avoided. Occasionally, what JWiz identifies as an error is something the programmer meant to do. The canonical error that I use to explain what JWiz does is the usage of `'=='` instead of the `'equals'` method in comparing two strings. In some cases, the programmer really did mean to use the double equals sign.

Other false positives are a result of a limitation in the current design of JWiz. JWiz runs on one file at a time, so if you have a package with multiple classes, JWiz may generate warnings for things which if the class was stand-alone, it would be an error, whereas in a package, it may not be an error. For example, you might declare

a variable in one class that isn't used in that class, but is used by another class in the package. Since JWiz does not check for interdependencies among classes in a package yet, erroneous warning messages will be produced.

3.4.6 Time

Time required for testing is another indicator of the effectiveness of JWiz. I believe that developers will spend less time debugging the defects for which JWiz is designed. If the developer runs JWiz after the first clean compile and before starting testing, JWiz will locate specific defects, saving the developer the time it would have taken to locate them manually.

3.5 Subjects

For this research, I collected Java code and/or the JWiz results from ICS 311, ICS 613, and CSDL. A variety of approaches for the use and data collection of JWiz was needed because of the different types of developers.

3.5.1 ICS 311

Algorithms and Data Structures (ICS 311), a class in which all assignments were done in Java, was taught by Dr. Feng Gao at the University of Hawaii. I made a short presentation to the class regarding what JWiz was and how to use it. JWiz was provided to the students in a GUI application, a text-only application, and an applet for them to use at will.

3.5.2 Collaborative Software Development Laboratory

The Collaborative Software Development Laboratory (CSDL) is a graduate student research group within the Information and Computer Sciences Department at the University of Hawaii.

Being a member of this group, I had daily, immediate access to all of its members. I was already familiar with the demographic data for each member and had the luxury of asking questions or soliciting opinions at any time. One feature of this group is that members have access to all code developed in the laboratory.

The experimental design for this group involved several methods of data collection. The first method was a result of the CSDL system release process. CSDL developed a program called JDS to automate releasing Java software systems. We added a JWiz run to this release process which would send me an email of the results.

This CSDL data was collected over a two month period. I found a number of false positives which I had not anticipated and I received bug reports as well. The release data provided me with a means of refining my system and building experience regarding the use of JWiz on the same system over consecutive releases.

The second method of data collection within CSDL was manual. I ran JWiz over all the Java source code in the CSDL archives. The code was written by many different people with a variety of programming experience. This code was already tested. I ran JWiz to see what sort of errors may be left undetected and how useful it would be to run JWiz on tested code. Due to the nature of the laboratory, I had access to all the code and the authors. In the interest of inconveniencing my fellow group members as little as possible, I ran the JWiz tests myself and then queried the authors as to the validity of JWiz warning messages as needed.

3.5.3 ICS 613

ICS 613, Advanced Software Engineering, was a graduate course conducted by Dr. Philip Johnson at the University of Hawaii. This course required the students to keep track of all defects made during the development cycle. For each defect, the students recorded when the defect was made, when it was found, a description of the defect, and the amount of time it took to find and remove it.

The members of ICS 613 were offered extra credit in return for sending me the source code of their assignments immediately after the first clean compile but prior to testing. Since the students were following the PSP, all coding is completed prior to the first attempt at compiling.

After the students completed their projects, I notified them of warnings generated by JWiz. I asked the students to note which of these warnings were valid. For the warnings which identified real defects, the students indicated whether they had found the defect, and if so, how long it took them to remove it. I also asked the students to provide some demographic data regarding their experience level.

I later compared this data with the data obtained from running JWiz on CSDL code. The ICS 613 data and the CSDL data (as well as the ICS 311 data, I soon found out) differed in the phase of development in which JWiz was run. Since I could control the phase at which JWiz was run for ICS 613, JWiz was always run on post-compile, pre-test code. For CSDL and ICS 311, JWiz was run mostly on post-test code.

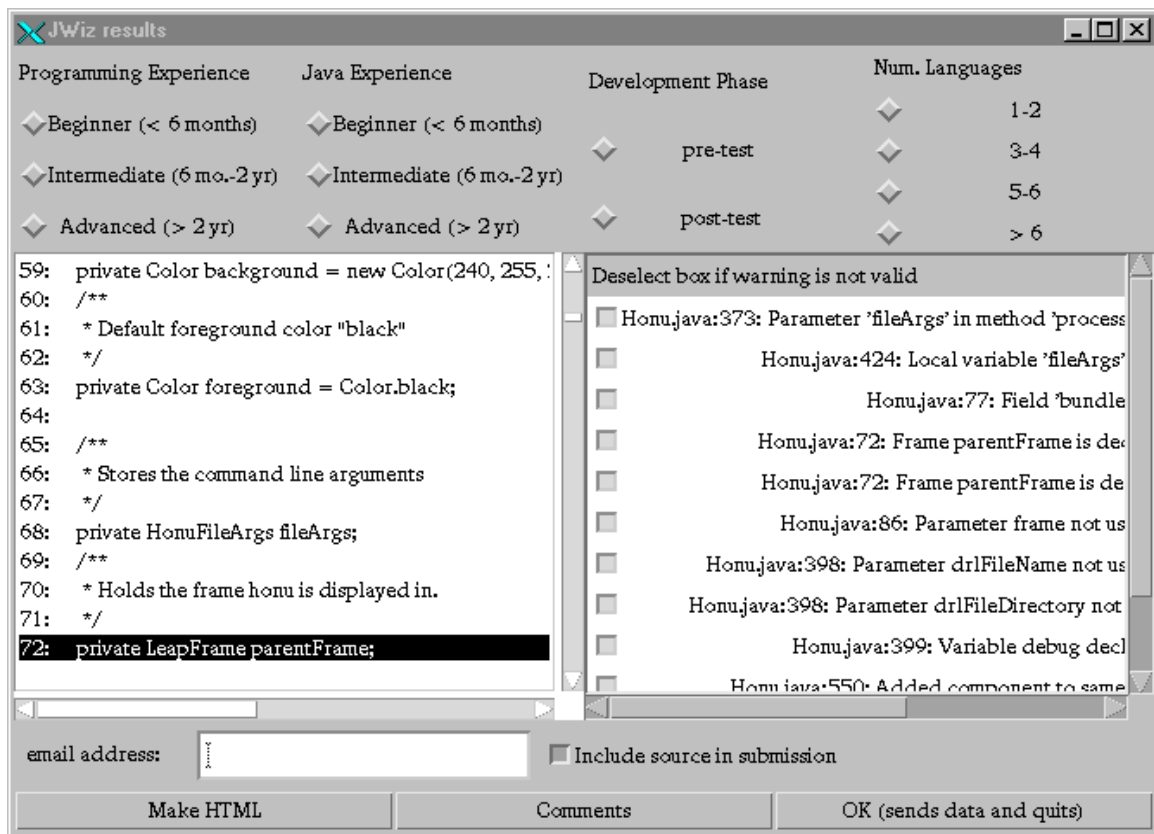


Figure 3.1: The GUI Application

3.6 Means of Data Collection

3.6.1 ICS 311

JWiz was offered to the ICS 311 students in several formats: A GUI application, a text-only application, and an Internet-based applet.

The GUI application displays the user's source code along with any warnings that were generated. It also displays a short survey and provides the user with the option of submitting their code along with the survey and defect data. A screen dump of the GUI application can be seen in Figure 3.1.

In addition to showing the line numbers for the code, the application also provide the ability to go directly to each line which generated a warning by moving the mouse over the description of the problem. Should the warning be invalid, the student would then deselect the corresponding checkbox.

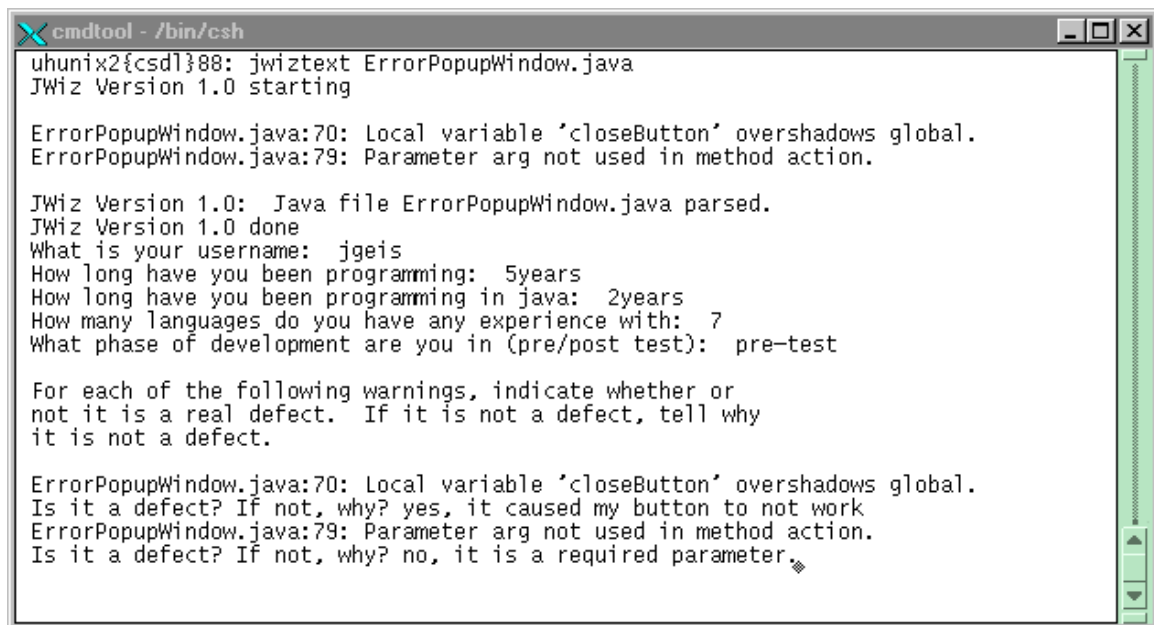
Students in ICS 311 were offered extra credit if they used JWiz. In order to report the use to their professor, I provided a field where students would enter their email address.

For each warning shown, the student was asked to indicate whether it was a real defect or not by toggling the checkbox next to the description of the problem. If the student was using the text only version, she would be queried about the validity of each warning generated.

After indicating the validity of the errors and filling out the survey, the student could quit the program and the data would be sent to me.

The Internet-based applet was similar to the application in all ways except for the manner of entering the file to be parsed. The application took a path and file name whereas the applet required an Internet URL due to applet security restrictions.

The text based application was offered in response to an unexpected problem in data collection. ICS 311 programs were required to run on Unix. I thus assumed that the applet and GUI application would suffice for the ICS 311 student's needs. It turned out that many students were doing their development on PC's, then telnetting over to the Unix environment. Under these circumstances a GUI is not viable. Combined with this was the fact that many of the students did not know how to make a file Internet accessible, so they couldn't use the applet. In response to these events, I developed a text-only application which would take the file and run JWiz on it. It would then prompt the user for responses to the survey questions in addition to validating the errors. Figure 3.2 is a screen shot of the text-only application in use.



```
cmdtool - /bin/csh
uhunix2{csdl}88: jwiztext ErrorPopupWindow.java
JWiz Version 1.0 starting

ErrorPopupWindow.java:70: Local variable 'closeButton' overshadows global.
ErrorPopupWindow.java:79: Parameter arg not used in method action.

JWiz Version 1.0: Java file ErrorPopupWindow.java parsed.
JWiz Version 1.0 done
What is your username: jgeis
How long have you been programming: 5years
How long have you been programming in java: 2years
How many languages do you have any experience with: 7
What phase of development are you in (pre/post test): pre-test

For each of the following warnings, indicate whether or
not it is a real defect. If it is not a defect, tell why
it is not a defect.

ErrorPopupWindow.java:70: Local variable 'closeButton' overshadows global.
Is it a defect? If not, why? yes, it caused my button to not work
ErrorPopupWindow.java:79: Parameter arg not used in method action.
Is it a defect? If not, why? no, it is a required parameter.
```

Figure 3.2: The Text-only Application

3.6.2 ICS 613

In contrast to ICS 311, the students of ICS 613 were never offered JWiz for their use. Instead, as I've already mentioned, the students would send me their post-compile, pre-test code upon which I would run JWiz. After their assignments were finished, I sent them the results of the run and asked them to verify if the warnings referred to real errors or not.

Chapter 4

Results

4.1 Defect Classification

I classified JWiz warnings into the following three categories: functional errors, maintenance errors, and false positives.

A functional error is a defect which will result in the program not doing what the developer intended. These are the real defects that programmers must fix in order for a program to work properly. For each of these defects, I obtained (when possible) data on how long the programmer spent locating and fixing the defect.

A maintenance error is a construct which will not prevent the program from functioning properly but is still not correct. For example, these defects involve situations where variables are declared but never used in a method. These will not cause the program to malfunction, but they are still errors in the sense that they make the program more difficult to understand and modify. We call these maintenance errors because they could cause problems if the program is to be revised in the future.

A false positive is a construct which JWiz flags as an error but is actually what the programmer intended. For example, JWiz flags a warning when it finds a local variable that overshadows a class variable. While this can signal a real problem, sometimes the programmer actually wants to do this. When a programmer says, “I meant to do that,” the warning is classified as a false positive.

Unless explicitly stated otherwise, all of the following results are obtained from ICS 613.

4.2 Raw Data

Out of the 235 warnings generated by JWiz, the warnings were spread fairly evenly across the three categories. Functional errors accounted for 29 percent of the warnings, maintenance errors for 43 percent, and the remaining 28 percent were false positives.

Table 4.1 shows that out of the 30 tests for defects that can be performed by JWiz, only the nine tests shown in the table generated any warnings. Also, only two of these tests indicated defects which required significant fixes.

Table 4.1: Generated Warnings

Test	fe	me	fp	debug time(minutes)
Putting a semicolon immediately after an if, for, or while statement.	0	1	5	0
Using == instead of .equals for string comparisons.	0	0	4	0
Variable/Param overshadows class variable.	55	14	12	163
Declared variables/fields/parameters that are never used.	12	78	27	158
GUI components that are not added to a container.	0	7	4	0
Assigning division result to an int.	0	0	4	0
Frame/DialogBox/FileDialog not sized and/or shown.	2	0	6	4
AddActionListener not called on Button	0	0	3	0
Multiple objects added to same BorderLayout area	0	0	1	0

The Java compiler improved in the time between my pilot study and final experiment. Thus, the “String equals” bug only generated four warnings which were all false positives. I will discuss this further in the section regarding observations.

4.3 Categorization

4.3.1 Developer Experience

I categorized developer experience using three factors.

1. General programming experience (years).
2. Java programming experience (months).
3. Number of languages known.

The amount of experience varied greatly for the developers in this research. I had developers with general programming experience ranging from one year to twenty-five. I didn't see any correlation between the number of years of general experience and the number of years of Java experience. For the majority of the students, ICS 613 was their first exposure to the Java programming language.

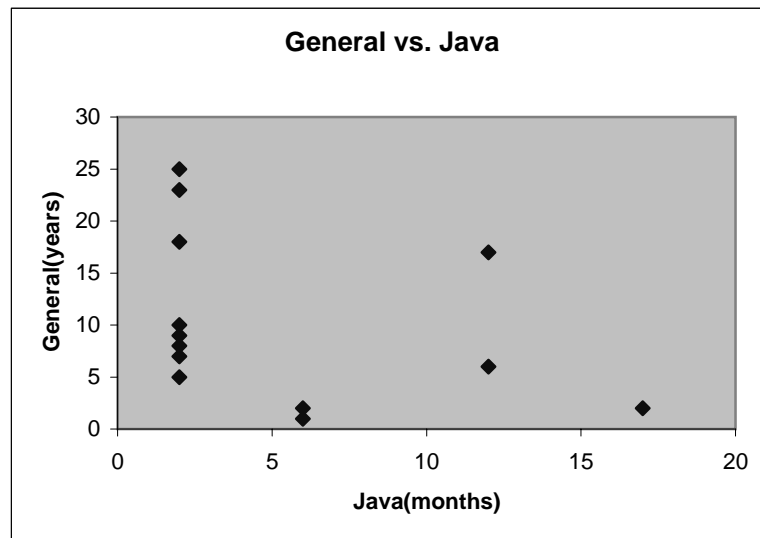


Figure 4.1: General Experience vs. Java Experience

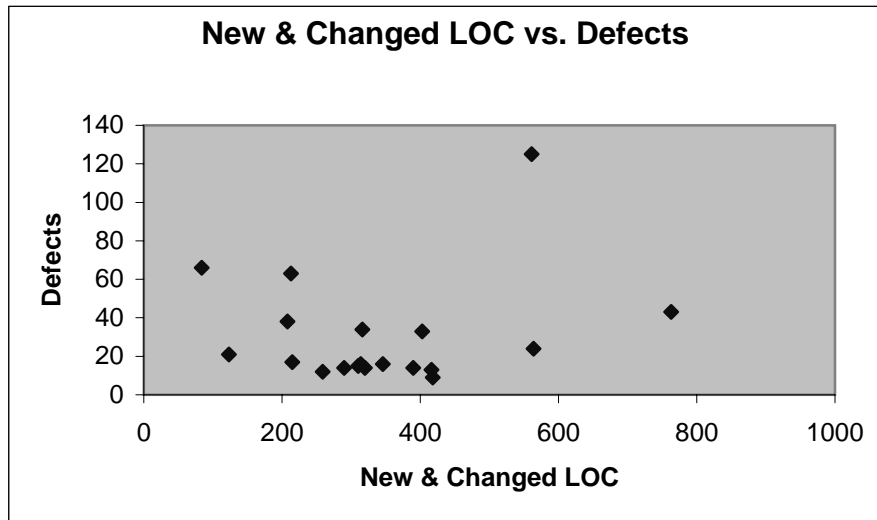


Figure 4.2: New and Changed LOC vs. Defects Made

4.3.2 Program Size

Figure 4.2 is a sample of the number of new and changed lines of code along with the number of defects made. There doesn't seem to be any relationship between program size and the number of errors that people make.

One possibility for this is that the students were not accurate in the recording of the number of new and changed lines of code. For example, one student listed the number of lines of code reused from a previous project as zero, then he listed the new and changed as also zero. Yet, his total lines of code were over 300. I omitted "impossible" numbers such as this.

Another potential source of error is the recording of defects. Perhaps the students were less than exact in noting when an error occurred. If this is the case, no correlation would be shown in this data, while there may be a relation with the number of defects that were really made.

4.3.3 Development Phase

As expected, the development phase during which JWiz was run was a big factor in the kinds of warnings generated. When JWiz was used before testing began, the types of JWiz warnings were split up evenly with roughly the same percentages of functional errors, maintenance errors, and false positives. The number of JWiz warnings generated and their types can be found in Table 4.1 on page 29.

In contrast, when JWiz was run after testing, it was, without exception, a waste of the developer's time. No functional errors were ever found by JWiz at this stage. The only useful outcome of running JWiz after testing was the cleaning up of code. As I discuss later in the section regarding anecdotal experiences, some developers took to running JWiz so they could remove unreferenced variables and parameters.

However, there might be some benefit to running JWiz while still in the test phase. In looking through one student's defect recording log, I noticed that he had created one defect in the process of fixing another. The original defect involved the GUI. While fixing this error, he decided to create another button, and this is where the second defect occurred. He forgot to add the button to the window, a defect which JWiz looks for. He spent close to an hour on this defect, almost forty percent of the total amount of time he spent testing. I noticed similar events on other DRLs as well. It might be worthwhile to run JWiz during test when you encounter a problem that may be found by JWiz (for example, a component not appearing in the display).

4.4 JWiz Effectiveness and Accuracy

One of the goals I wanted to accomplish with this research is to determine the accuracy and effectiveness of JWiz. The following results are obtained from ICS 613.

Effectiveness is a measure of valid defects found per thousand lines of code, accuracy is a comparison of the number of functional errors with maintenance errors and false positives.

JWiz analyzed 12848 lines of code from ICS 613. The students reported spending 125 hours in test. Of this time, 76 hours were spent debugging and 240 defects were removed. On average, the students recorded removing one defect in test every fifty-four lines.

JWiz generated 235 warnings. Of these warnings, 69 were functional errors, 100 were maintenance errors, and 66 were false positives.

Regarding effectiveness, JWiz found one functional error every 186 lines of code. This accounted for 29 percent of all defects found by developers in test during the study.

Table 4.2: Test Phase Statistics

Total Time(hrs)	Debug Time(hrs)	Num.Def.Removed	Test Def/KLOC
125	76	240	1/54

Table 4.3: JWiz Statistics

Warnings	fe	me	fp	fe/KLOC	fe debug time (hrs)
235	69	100	66	1/186	5.5

Although JWiz caught 29 percent of all reported defects found in test, the time spent locating and fixing those defects amounted to only five and a half hours, or 7.3 percent of the total amount of time spent debugging in test.

I believe this apparent discrepancy is a result of the nature of the defects that JWiz found. It turned out that the warnings which most frequently indicated real

defects were the ones which were included in JWiz because they occurred in most developer's programs. It is possible that since these defects are fairly common, people have become somewhat skilled at finding and fixing them. Perhaps the more time consuming defects are the odd, rarely occurring ones. This would account for why JWiz found a large percentage of the reported test defects, yet resulted in a much smaller percentage of the debugging time.

All in all, JWiz appears to be fairly promising. As one student put it "It took me 30 minutes to fix [an error reported by JWiz]. If I had seen the results of [JWiz] before, I would have gone straight to the point."

4.5 Changes to Java

It is not too difficult to identify some potential improvements to the Java programming language/environment. One such improvement would be to disallow unused variables and parameters. Additionally, another possibility would be to eliminate the possibility of using the same parameter name as the class variable. I imagine that some developers may not be too keen on these ideas however. Perhaps a better solution would be to equip the compiler with something similar to the deprecation warnings that arise when the code uses 1.0 event handling.

4.6 Changes to JWiz

As a result of this experiment, I found that some changes to JWiz should be made. I noticed that some of the false positives are avoidable. For example, one false positive occurred whenever an interface class was created. For each of the methods in the class, JWiz generated "Parameter not used in method" warnings. This happened because no code is allowed within the method of an interface, but it is not an error. JWiz can check if the class is an interface, and eliminate this particular false positive.

This same warning was also considered a false positive when the parameter was required as part of an event handling method. For example, consider the action method used for Java 1.0 event handling. The method requires two arguments, an Event and an Object. If the user does not reference the Object argument, JWiz would issue a warning. I had anticipated some of the event methods, but not all (such as the Object argument to the action method), so JWiz generated warnings when the developer did not use one of the parameters required for certain event handling methods. Making these two changes would eliminate over seven percent of false positives that occurred during the course of the experiment.

Another result of the experiment may be to prove the usefulness of a functional addition to JWiz. I planned to implement a mechanism which allowed users to choose specific warnings. For the purpose of this experiment, I wanted everyone to run all the tests. I found that certain tests were useful only to a few people. One test in particular, “Assigning a division result to an int,” was always called a false positive by the developers in the study. However, this test made it into JWiz from data I collected prior to the development of JWiz when I found that one student spent almost a half an hour on this particular bug.

I wanted people to be able to add tests that they would find useful. These tests could be made publicly available for anyone to include in her copy of JWiz. Combined with error toggling, users would then be able to share tests, turning them off if they proved to be more of a nuisance than a help.

Some new tests were implemented based on the results of the experiment. Some of the participants in the study took to writing me about defects which caused them problems. For example, one student spent a half an hour tracking down a NullPointerException. In this case, he was referencing 'this' while initializing a class field. Since the object was not yet set up, there was no 'this' to reference.

4.7 Anecdotal Experiences

Some interesting events occurred during the use of JWiz that provided insight into its potential.

4.7.1 Code Clean-up

One such situation occurred while a student was cleaning up his code at the end of the development process. The student encountered a variable which he believed was not used anywhere in the class. Hesitant to delete the variable lest there was a reference to it, he recalled that JWiz checked for unreferenced variables. This was not the intent of that particular JWiz test, but it still served a useful purpose to the student, assuring him that the variable was indeed never referenced in the class.

4.7.2 Defect Denial

One observation I made during this research was that developers were reluctant to admit they had made a mistake. On multiple occasions, a developer would claim that a JWiz warning was a false positive. Yet, when I looked at the program, I found that it would not function properly as it was. This could be attributed to psychological factors[11], forgetfulness[1], measurement dysfunction[2], or a variety of other reasons.

For the ICS 613 class, each student sent me an email of their post-compile, pre-test code. I ran JWiz on the code, and sent the JWiz results after the assignment was completed. I asked one particular student to verify whether several “GUI component not added to a container” warnings were valid or not. The student replied that the warnings were not valid, that he had really added them to a panel. I then looked at the post-compile, pre-test code and found that this was not the case. It’s possible that there was some miscommunication, perhaps the student wasn’t clear as to what

I was asking, or maybe he didn't understand the error message. Another possibility is maybe the student forgot that he had made the error at all[1].

In another instance, JWiz provided an experienced Java programmer with a warning "Comparing two strings using `==` instead of the equals method." He incorrectly claimed that the code wouldn't compile using the equals method. This may be explained by "The Psychology of Computer Programming"[11], which illustrates how programmers are unlikely to accept blame when an error occurs.

Interestingly, some students report JWiz warnings to be false positives when it indicated real defects in their program. One possibility ties in with another CSDL member's research on the validity of PSP data collection[5]. She found that many students submitted "suspicious" defect recording logs. For example, a student reporting spending over an hour in compile but recording only two compile errors, each taking 2 minutes to fix. It is possible that students are not recording all their defects. This could be attributed to measurement dysfunction[2]. For example, maybe the students thinks it looks bad to report many defects.

A student may not record all their defects as a result of insensitivity to when a defect occurs. Say, for example, a compiler error occurs. It is a rapid fix, so the student thinks, "That only took a few seconds, it's not worth recording." Maybe the student made the same mistake many times in their code. Not wanting to record each occurrence, he records just the first one and omits the rest. Another possibility is the student just did not bother to record the data at the time the error was located and then forgot about it. The idea of the student forgetting to record a defect is very plausible. There is a fair amount of cognitive overhead in performing the PSP and it's possible the student just had too many things to think about[1].

If the student did forget to record one of their defects, it is possible she could forget about making the error entirely. In this case, maybe the student looked at

the JWiz warnings, scanned through her list of defects, and not seeing anything that matched the warning, responded that the warning was a false positive.

Another factor to consider is the effect of telling people that some of the warnings could be false positives. If the student is in a hurry, or thinks she is finished with her program, she may discount the warnings and reply that they are erroneous without really checking.

4.7.3 Post-Test Usage

Another situation which I found interesting occurred with the ICS 311 students. I made a short (10 minute) presentation to the class during which I emphasized the importance of running JWiz after the first compile without errors and before beginning to test. I also provided a handout that described what JWiz was, and how to use it.

Although many students used the program (they received extra credit for doing so), not one student used the program before testing. Every student ran JWiz on already tested code. This indicates that the students used JWiz solely to obtain extra credit, without regard for its' usefulness in debugging. This is consistent with Gould's findings regarding how people debug computer programs[6]. As expected with such usage, JWiz yielded very little in the way of results for these developers. All that was generated were some maintenance errors and false positives; no functional errors. I imagine that the students have a very dim view of the usefulness of JWiz as a result.

I also found that some of the students started using JWiz like a compiler. When multiple maintenance warnings were generated the student would go fix them and then run JWiz again. Occasionally, more warnings were generated and the student would repeat the process.

4.7.4 The “String Equals” Bug

The defect regarding the comparison of two strings using double equals generated four warnings, and they were all false positives. This was really quite a surprise since this was indicated to be a very common time-consuming defect during my pilot study. Why would the results be so different between the two studies? The answer is SunSoft’s javac compiler has gotten a bit smarter in assigning objects and strings.

The following code example shows an example of code that would have caused a defect in the pilot study.

```
public void actionPerformed(ActionEvent evt){
    boolean flag = false;
    String command = evt.getActionCommand();
    if (command == "Enter Program Information"){
        flag = true;
    }
    processFlag(flag);
}
```

The flag would never be set to true regardless of the value of the string ‘command.’ In the actual experiment however, due to changes in the compiler, this code would work as planned.

Using double equals to compare two strings is still a bad idea however as shown by the next code example.

```
public boolean isEqual(String string1, String string2){
    if (string1 == string2){
        return true;
    }
}
```

```
    return false;  
}
```

Assuming the parameters are different objects, the above method will never return true, even if the contents of the two strings are identical.

This is an example of what I hope to accomplish regarding changes in the Java language/environment. The “String equals” bug was a defect which everyone had made at one time or another. The developers of the Java language realized this and modified the compiler to reduce some of these problems. Hopefully, by pointing out some other potentially hazardous elements of the language, the language/compiler developers will account for that in future releases.

Chapter 5

Related Work

This chapter will discuss those systems/methods which are similar in purpose to JWiz. Specifically, I will discuss means of debugging, both automated and manual. The systems/methods that I will discuss are jtest![10], CodeWizard[3], Lint[4], and personal code review[8].

5.1 Automated Debugging

Automated debugging involves the use of a computer program which checks for sources of error in your programs through a variety of methods. Some of these systems perform the task through static analysis, JWiz is an example of this type of system. Other systems operate dynamically, doing their work while the program is actually running.

jtest! is a dynamic debugging tool which operates by generating input for Java applets and applications. “Whenever an input (or sequence of inputs) generated by jtest! causes an uncaught runtime exception, jtest! automatically reports the type of exception thrown and the input which caused it.”[10] The idea behind this system is to make your programs “bomb-proof,” assuring that regardless of the input, the program will not fail ungracefully.

Both jtest! and JWiz aim to improve the quality of Java programs and make the developer’s life easier. They differ, however, in the methods used. The emphasis of jtest! is to “bomb-proof” programs against any and all types of user input. JWiz

does not have the ability to check user-input. In addition to the types of problems that are checked, the difference between the tools is that jtest! is a dynamic analysis tool, while JWiz provides static analysis.

CodeWizard for Java is very similar to JWiz. It is a static analysis tool which looks for constructs which could indicate the presence of a defect in Java source code. It “advises the developer of violation”[3] of a number of rules. Following are the list of rules from the ParaSoft description of CodeWizard for Java[3].

- Avoid unused variables
- Explicitly initialize all variables
- Control flows into case
- Case label inside loop
- Avoid hiding inherited instance variables
- Avoid hiding inherited static member functions
- Constructor should explicitly initialize all variables
- Instantiated classes should be made final
- Avoid assignment in if-condition
- Avoid public instance variables
- First present all public methods/data in a class
- Avoid hiding member variables in member functions
- Implement interfaces nontrivially or abstract
- Use “equals” over “==”

- Use “StringBuffer” instead of “String”
- Provide “default:” label for each switch statement
- Provide incremental in for-statement or use while-statement
- Enforce name format of classes
- Enforce name format of instance variables
- Enforce name format of methods
- Enforce name format of local variables
- Enforce name format of interfaces
- Enforce name format of method parameters

Although JWiz and CodeWizard for Java have a few tests in common, CodeWizard for Java looks for different types of problems than JWiz. CodeWizard, unlike JWiz, seems to concern itself with “style” issues, such as name formats of classes, methods, etc.

Another difference may be usage requirements. JWiz requires the code to compile without errors and, as far as I could tell, CodeWizard does not have this requirement.

Lint’s original purpose was to locate bugs and inefficiencies in C source code. Later, it was expanded to handle portability issues. It also looks for code which could cause run-time errors and notifies the developer. Although the software is for different languages, the two programs are similar in their efforts to locate run-time defects through automated notification.

5.2 Manual Debugging

A personal code review can be performed before or after starting to compile. There is much debate on the topic. Either way, it is clear that personal code reviews can result in the removal of a significant number of defects. Personal code reviews are usually performed by printing out a hardcopy version of the program and stepping through the code line by line.

One of the advantages of doing a code review is that when a problem is discovered, you have the defect's location, and the context in which the defect was made. This typically sets you up for the proper fix. Are code reviews more effective than unit testing? Most developers can find two to four defects for every hour of test. In contrast, reviews typically yield six to ten defects for each hour of code review[9].

An example of the effectiveness of code reviews involves a defect which took three engineers three months to find. When the developers did find this defect, the same code was being inspected by another team consisting of five engineers. This second team was not told about the presence of the defect, yet they found it within two hours. In addition to this defect, the team also found 71 others. As is the case with most defects discovered during testing, finding the defect was the most difficult task, the fix was trivial[8].

JWiz follows the same concept as a code review, find defects before test. Besides the fact that one method is automated while the other is manual, there are distinct differences between the two methods. First, personal code reviews are more effective. This is because JWiz is limited in what it can look for. It only knows what it has been programmed for. As in the example in the previous paragraph, personal code reviews frequently result in the discovery of defects that the reviewer was not specifically looking for.

The two methods can be complementary to each other. First, personal code reviews involve the writing down of the defects that are found. As a result, developers could look through the results and note which defects occur the most often. Then the developer could add tests for those defects into JWiz. In addition to this, JWiz could generate new checklist items to be used during the review. JWiz could be used in conjunction with a code review since even with this process, defects can be missed.

Chapter 6

Conclusion

6.1 Contributions of this Research

The major contribution of this research is JWiz itself. If the data produced by this research is representative, then JWiz can truly support the Java software development community. This is evidenced by JWiz reporting defects which amounted to seven percent of the total amount of time the ICS 613 developers spent debugging in test.

By releasing JWiz to the general development community, it is possible that JWiz will provide even better results as programmers create the tests which are most useful to themselves.

6.2 Future Directions

6.2.1 Error Correctness Estimation

One avenue of research could be an investigation into the likelihood of JWiz warnings being functional errors or false positives. Perhaps by stating that a given error has a high percentage of being a real error, a developer would be more inclined to look into it.

I could use current results to attach a “probability” to each test. However, this would only allow me to rate a few of the tests, as a majority of the tests never

generated any warnings with which I could evaluate their effectiveness. Also, the small sample size may not be indicative of the tests' general performance.

6.2.2 Publicly Available JWiz

A possible topic arising from the idea of making JWiz available to the general public is the potential of collecting data on the kinds of defects made by developers. I believe my sample size was too small and restricted to yield any real data on this topic. Perhaps there could be a JWiz Internet site which would maintain a collection of JWiz tests as well as collect the results from JWiz usage. By allowing people to contribute tests, perhaps the effectiveness of JWiz could be increased, resulting in more expensive defects to be found.

Obviously, one future effort could be the development of a package to assist developers in the creation of new JWiz tests.

6.2.3 Shrinking Test Time

“The longer the defect is in the product the larger the number of elements that will likely be involved.”[8]

Humphrey's quote refers to a fact widely known among developers. It is also known that the test phase can be the most time consuming and frustrating of all the development phases. Anything that eliminates defects prior to test is a good thing.

I found that, on occasion, JWiz could save developers non-trivial amounts of time in test. One developer in my study spent just over six hours debugging. When I sent him the results of the JWiz run, he reported that one of the warnings was indeed an error. The error cost him an hour and a half, 15 percent of the total amount of time he spent in test. It turns out that this one defect accounted for seven percent of his total development cycle for that program.

A future avenue of research could be the investigation of potential time savings as a result of using JWiz. Perhaps the amount of time spent in test will shrink, or maybe people will start to make new errors, resulting in no time savings at all. One could also investigate a reduction of testing as a percentage of the total development cycle or whether JWiz results in an increase in productivity (lines of code per hour).

Appendix A

JWiz Defect Classes

Table A.1: JWiz Defect Tests

Test	fe	me	fp	debug time(minutes)
Omitting the space between the word 'case' and the integral value being tested in a switch statement.	0	0	0	0
Putting a semicolon immed. after an if, for, or while statement.	0	1	5	0
Using == instead of .equals for string comparisons.	0	0	4	0
Controlling counting loops with floating point variables.	0	0	0	0
Declaring variables/fields/parameters that are never used.	12	78	27	158
Assigning a numeric type a value larger than its capacity.	0	0	0	0
GUI component not added to a container	0	7	4	0
Missing "super" before superclass method call.	0	0	0	0
Assigning division result to an int.	0	0	4	0
Frame/DialogBox/FileDialog not sized and/or shown.	2	0	6	4
MenuBar declared but setMenuBar not called.	0	0	0	0
Throwing an exception within a Finally block.	0	0	0	0
Setting Two checkboxes in same checkbox group to have initial values set to true.	0	0	0	0

Table A.1: JWiz Defect Tests (Continued)

Test	fe	me	fp	debug time(minutes)
Using or & instead of or &&.	0	0	0	0
Doing an assignment to a boolean inside of an if statement instead of comparison.	0	0	0	0
'if (null instanceof Object)'.	0	0	0	0
Local variable declaration overshadows global field of same name.	55	14	12	163
Passing two zeroes to GridLayout constructor.	0	0	0	0
Passing non-Rectangular shape to Graphics.setClip(Shape).	0	0	0	0
Calling 'setLayout' on a ScrollPane.	0	0	0	0
AddActionListener not called on Button.	0	0	3	0
Calling 'Thread.suspend()' which can cause program to hang.	0	0	0	0
Directional args for adding to a BorderLayout not "North","South",etc.	0	0	0	0
Multiple components added to same location of BorderLayout.	0	0	1	0
Added same component to container multiple times.	0	0	0	0
Removed an element from a vector within a for loop without decrementing loop counter or returning.	0	0	0	0
Removed an element from a vector within a for loop that counts up to vector's original size.	0	0	0	0
Added a component to itself.	0	0	0	0

Appendix B

ICS311 Handout

JavaWizard
Jennifer Geis
Collaborative Software Development Laboratory
jgeis@uhics.ics.hawaii.edu
Post 307B
956-6920

JavaWizard (jwiz) is an automated code checker for the Java programming language. Basically, jwiz scans through your code and looks for things which will likely cause unintended behavior during run-time. Jwiz assumes your code is syntactically legal. In other words, your code must compile correctly before you can run jwiz on it. When jwiz finds something that might be an error, it writes out a warning message that contains the name of the file, the line number on which the error occurred, and a description of the error.

An example of the kind of error that jwiz can detect is the use of a double-equals sign instead of the 'equals' method to compare two strings. Jwiz will give you the warning message like:

```
'FileName.java:43:Comparing two string using '==' instead of 'equals'.'
```

There are several versions of JWiz: a gui application, a text application, and an applet.

To run the gui application you must do one of the following:

1. Include '/home/35/csdl/bin' in your path.

or,

1. Copy the file `'/home/35/csdl/bin/jwiz'` into your `home/bin` directory.
2. Make sure your path includes your `home/bin` directory.

or,

1. Type `'/home/35/csdl/bin/jwiztext'`

To run the text only application (ideal if you are telnetting into uhunix):

1. Include `'/home/35/csdl/bin'` in your path.

or,

1. Copy the file `'/home/35/csdl/bin/jwiztext'` into your `home/bin` directory.
2. Make sure your path includes your `home/bin` directory.

or,

1. Type `'/home/35/csdl/bin/jwiztext'`

To run the applet:

1. Go to `'http://bianca.ics.hawaii.edu/csdl/applets/JwizHomePage.html'`
2. Enter in a url to your file and hit the `'run jwiz'` button (due to applet security restrictions, you must enter a url, not a path).

Now that you have `jwiz` available in your environment, to run `jwiz`, type `'jwiz'` at the command prompt (if you are using the gui application). At this point, a window will appear showing the files in the current directory.

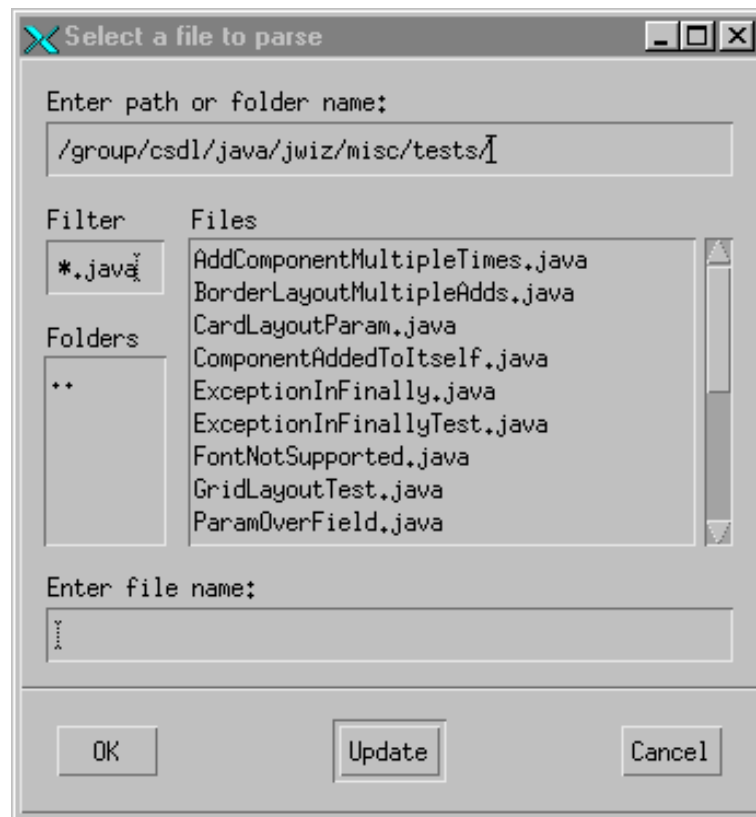


Figure B.1: FileDialog

Select the file you want to run Jwiz on and press the OK button. You can also start jwiz by giving the file name as an argument: For example 'jwiz FileName.java' for a specific file, or 'jwiz *.java' if you want to run jwiz on everything in the current directory.

Jwiz will give you some status messages while it's running, then another window will be displayed. If no warnings were generated, a window will be shown with the message "No warnings were generated." If jwiz did find anything however, the window will contain the code of the file you ran jwiz on, the warnings generated, and a few survey questions.

By moving the cursor over the warning message, the line that generated the warning will automatically be highlighted. Go through the list and verify which are

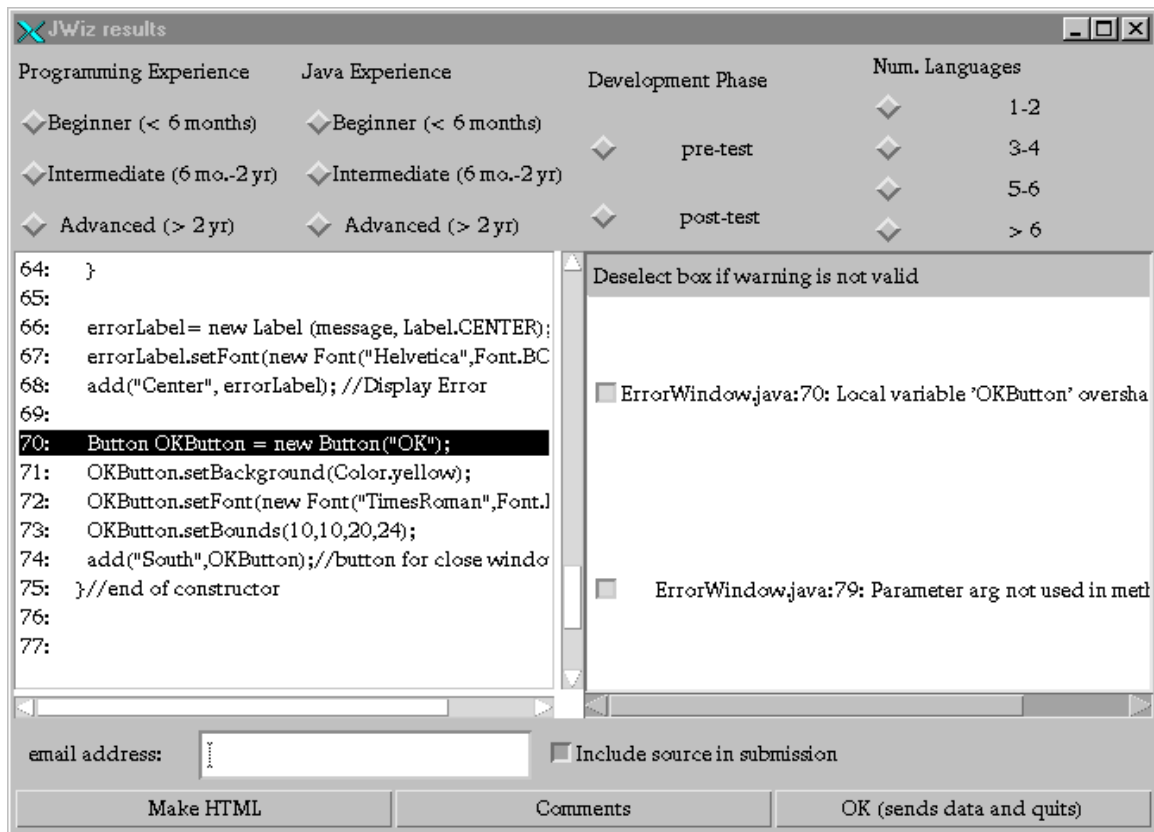


Figure B.2: JWiz Results

valid and which are not. Since jwiz looks for things which are usually errors but not always, sometimes it flags things that you really meant to do.

Each warning is assumed to be valid and so is selected by default, if a warning is not valid, please deselect it before quitting the program.

For the purpose of this experiment, you will be required to fill out the survey data each time you run the program. Since your answers about programming experience do not change, JWiz writes out a file to the current directory with these answers and uses them as defaults the next time you run the program. However, there is one survey question you need to check each time you run JWiz, since it could change between executions of JWiz. This is the "development phase" question. For example, say you run JWiz for the first time on a program right after it compiles correctly and

before you've actually run it on some data to test it. At this point, you'd mark the "Development Phase" as pre-test. Then, if you used JWiz again after running your program a few times to test it on some data, you would mark the Development Phase as "post-test". You would mark the phase as "Post-test" for this and all future runs of JWiz even though you might not yet be done with testing.

The survey questions are:

1. Programming experience: I would like you to indicate how long you have been programming (any languages), less than six months, between six months and two years, or more than two years.
2. Java programming experience: How long you have been programming with Java, less than six months, between six months and two years, or more than two years.
3. Development phase: This is the phase of development you were in when you ran `jwiz`. "Pre-Test" means that you are running JWiz after the code compiles cleanly for the first time, but before you have started testing it by running the program on some data. "Post-Test" means that you are running JWiz after having started to test your program by running it on data. Once you've run your program on some data for the first time, you will always indicate the phase as "Post-Test", even if you need to change and recompile your program again.
4. Number of languages: Please indicate the number of programming languages that you have experience with. One to two, three to four, five to six, or more than six languages.

The buttons at the bottom of the window are as follows:

Make HTML: Writes out whatever warnings you have indicated to be valid in HTML format.

Comments: Pops up a window through which you can send me any questions, comments, or bug reports.

OK: Sends me the list of valid/invalid warnings, a copy of the source (if the "Include source in submission" checkbox was checked), and the survey data. In the interest of privacy, I will not report any details identifying you, your program, or your errors. I request your email address solely in order to contact you if there are questions about your data.

Bibliography

- [1] Anderson, John R. and Jeffries, Robin. Novice lisp errors: Undetected losses of information from working memory. *Human-Computer Interaction*, 1(2):107–131, 1985.
- [2] Austin, Robert D. *Measuring and managing performance in organizations*. Dorset House, 1996.
- [3] Codewizard for Java. WWW page, 1997. <http://www.parasoft.com/wizard/papers/java.htm>.
- [4] Darwin, Ian F. *Checking C Programs with Lint*. O'Reilly, 1988.
- [5] Disney, Anne. Personal software process and data quality problems. Technical Report 96-15, UH Information and Computer Sciences Program, 1998.
- [6] Gould, John D. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7:151–182, 1973.
- [7] Grand, Mark. *Java Language Reference*. O'Reilly, 1997.
- [8] Humphrey, Watts S. *A Discipline for Software Engineering*. Addison-Wesley, January 1995.
- [9] Humphrey, Watts S. *Introduction to the Personal Software Process*. Addison-Wesley, 1997.
- [10] jtest! WWW page, 1997. <http://www.parasoft.com/jtest/home.htm>.

- [11] Weinberg, Gerald M. *The Psychology of Computer Programming*. Van Nostrand Reinhold Company, 1971.
- [12] Zukowski, John. *Java AWT Reference*. O'Reilly, 1997.