A Case Study Of Defect Detection And Analysis With JWiz

Jennifer Geis
Dept. of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822 USA
+1 808 956-6920
jgeis@hawaii.edu

ABSTRACT

This paper presents a study designed to investigate the occurrence of certain kinds of errors in Java[5] programs using JavaWizard (JWiz), a static analysis mechanism for Java source code. JWiz is a tool that supports detection of certain commonly occurring semantic errors in Java programs.

JWiz was used 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.

We found that all programmers inject a few of the same mistakes into their code, but these are only minor, non-defect causing errors. We also found that the types of defects injected vary drastically with no correlation to program size or developer experience.

Finally, we 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.

Keywords
static analysis, code review, run-time/semantic errors

1 INTRODUCTION

All programmers inject defects into their code. Even experienced developers typically inject a defect about every 10 lines of code[6]. 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 some horrible night spent searching for the cause of a strange behavior in their program. Such a night frequently ends with the programmer groaning in disgust when they finally spot the offending line.

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.

In essence, JWiz serves as a kind of “smart compiler” that can inform the programmer about constructs that will likely cause the program to behave in unexpected ways, as opposed to the syntactic errors that a compiler normally captures. Basically, JWiz is like a Lint[3] for Java. What distinguishes JWiz from Lint, other than the programming language, is the way in which we used it for this research.

JWiz provided us 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?

The answer 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 other 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, 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. We 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."

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:42: addActionListener not called on button ‘stopButton’.

TestFile.java:89: Multiple objects added to same borderlayout area.

TestFile.java:131: Local variable 'varString' overshadows member variable.

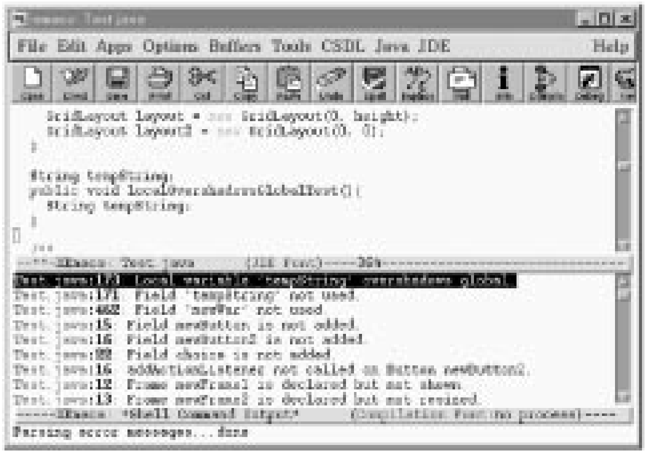
TestFile.java:171: Local variable 'testString' not used.

We have also implemented a JWiz mode for XEmacs which allows the JWiz output to be mouse selectable as shown in Figure 1.

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 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 tendency to forget to add an action listener when she makes her buttons (which causes the buttons not to do anything when clicked). 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.

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.



```
    GridLayout layout = new GridLayout(0, height);
    GridLayout layout2 = new GridLayout(0, 0);
  }

  String tempString;
  public void localOvershadowsGlobalTest(){
    String tempString;
  }

  /**
--**-XEmacs: Test.java      (JDE Font)----36%-------------------
Test.java:171 Local variable 'tempString' overshadows global.
Test.java:171 Field 'tempString' not used.
Test.java:482 Field 'newVar' not used
Test.java:15 Field newButton is not added.
Test.java:16 Field newButton2 is not added.
Test.java:22 Field choice is not added
Test.java:16 addActionListener not called on Button newButton2.
Test.java:12 Frame newFrame1 is declared but not shown.
Test.java:13 Frame newFrame2 is declared but not resized.
-----XEmacs: *Shell Command Output*     (Compilation Font:no process)----
Parsing error messages...done
```

Figure 1: JWiz-mode in Emacs

2 EXPERIMENTAL DESIGN

Hypothesis
The JWiz research project involved the following hypotheses:

1. The use of JWiz in the context of this research would 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 sections discuss how the experiment was designed to address the above hypotheses.

Improvement to the Java Language/Environment/ Programmers
In order to discover problem areas for Java developers and the Java language and/or environment, we needed to find out what defects were being made. Since JWiz provided a mechanism for defect reporting, we 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?

JWiz vs. Manual Debugging
In order to determine if JWiz is any more effective at locating defects than manual debugging, we needed to record data on how long it took people to find and remove the defects which JWiz can detect. We 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.

Most developers would not have this information recorded, but the nature of the class the students were taking required it. This will be elaborated on a little later in the section regarding the Personal Software Process[6].

After the programs were finished, we 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, we could look at the students' responses to the warnings and see if JWiz is any more efficient than their manual debugging efforts.

Experimental Procedures
For this research, we designed a case study. Over a period of four weeks, JWiz was given to two groups of students and one research group of 5 graduate students.

Data
Collecting the defects discovered by JWiz is not enough. That would only indicate whether or not the program works, not if it is really a useful tool. To understand more about the effectiveness of the tool, we 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. We also planned to track "false positives," warnings which the developer decided were not valid.

Size
The size of the program is useful in determining JWiz' effectiveness. By effectiveness, we 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.

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 for the developer to 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.

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.

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.

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

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.

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 evaluate the usefulness of JWiz.

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.

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.

Time

Time required for testing is another indicator of the effectiveness of JWiz. We believe that developers would 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.

The Personal Software Process (PSP)

Some data collection relied heavily on the use of Watts Humphrey's Personal Software Process (PSP)[6]. The PSP requires that the developer follow a strict software development process consisting of the following phases: plan, design, code, compile, test, and postmortem. The developer keeps track of all defects that are found in their programs. For each defect, the data collected includes 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 to find and remove it. Exactly how the PSP is used in this experiment and by whom will be discussed in the following section.

Subjects

For this research, we collected Java code and/or the JWiz results from two classes at the University of Hawii (ICS 311 and 613), and a research group (Collaborative Software Development Laboratory). A variety of approaches for the use and data collection of JWiz were needed because of the different types of developers.

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. We 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. All three were available for them to use at will.

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 members of this group, we had daily, immediate access to all of its members. We were 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. We found a number of false positives which we had not anticipated and we 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. We 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. We 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, we had access to all the code and the authors. In the interest of inconveniencing my fellow group members as little as possible, we ran the JWiz tests myself and then queried the authors as to the validity of JWiz warning messages as needed.

ICS 613

ICS 613, Advanced Software Engineering, was a graduate course conducted by Dr. Philip Johnson at the University of Hawaii. In this course the students used the PSP in the development of all their programming assignments.

As defined by the PSP, the students were required 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, we notifed them of warnings generated by JWiz. We 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. We also asked the students to provide some demographic data regarding their experience level. The students were never given JWiz itself.

We later compared this data with the data obtained from running JWiz on CSDL code. The data differed in the phase of development in which JWiz was run. Since we could control the phase at which JWiz was run, JWiz was always run on post-compile, pre-test code.

Means of Data Collection

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.

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, we 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 would take the file name as input and then run JWiz on it. It would then prompt the user for responses to the survey questions in addition to validating the errors.

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 we would run JWiz. After their assignments were finished, we sent them the results of the run and asked them to verify if the warnings referred to real errors or not.

3 RESULTS

Defect Classification

We 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, we 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.

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.

Out of the 30 tests for defects that can be performed by JWiz, only nine tests generated any warnings. Also, only two of these tests indicated defects which required significant fixes.

Developer Experience

We 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. We had developers with general programming experience ranging from one year to twenty-five. We 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.

Program Size

In comparing the number of new and changed lines of code along with the number of defects made, there didn’t seem to be any relationship.

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. We 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.

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.

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. Some users took to running JWiz after testing 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, we 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. We 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).

JWiz Effectiveness and Accuracy

One of the goals we wanted to accomplish with this research is to determine the accuracy and effectiveness of JWiz.

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.

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.

We 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 during a pilot study. 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 find [an error reported by JWiz]. If I had seen the results of [JWiz] before, I would have gone straight to the point."

Changes to Java

In looking at the warnings which found the most functional errors, 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. We imagine that some developers may not be to 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.

Changes to JWiz

As a result of this experiment, we found that some changes to JWiz should be made. We 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. We 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. We planned to implement a mechanism which allowed users to choose specific warnings. For the purpose of this experiment, we wanted everyone to run all the tests. We 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 we collected prior to the development of JWiz when we found that one student spent almost a half an hour on this particular bug.

We 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 CONCLUSION

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.

Future Directions

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.

We 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 we could evaluate their effectiveness. Also, the small sample size may not be indicative of the tests' general performance.

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. We 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.

Shrinking Test Time

“The longer the defect is in the product the larger the number of elements that will likely be involved."[6]

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.

We 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 we 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).

5 INFORMATION AND QUESTIONS

For more information, contact Jennifer Geis (jgeis@hawaii.edu).

ACKNOWLEDGEMENTS

I offer much thanks to the members of Collaborative Software Development Laboratory: Robert Brewer, Anne Disney, Cam Moore, and especially Philip Johnson for being my advisor and editor.

REFERENCES

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. Darwin, Ian F. Checking C Programs with Lint. O'Reilly, 1988.
4. Geis, Jennifer. JavaWizard: Investigating Defect Detection And Analysis. M.S. Thesis, Technical Report 98-01, Dept. of Information and Computer Sciences, University of Hawaii, 1998.
5. Grand, Mark. Java Language Reference. O'Reilly, 1997.
6. Humphrey, Watts S. A Discipline for Software Engineering. Addison-Wesley, January 1995.