Backstop: A Tool for Debugging Runtime Errors

Christian Murphy, Eunhee Kim, Gail Kaiser, Adam Cannon
Dept. of Computer Science, Columbia University
New York NY 10027
{cmurphy, ek2044, kaiser, cannon}@cs.columbia.edu

ABSTRACT
The errors that Java programmers are likely to encounter can roughly be categorized into three groups: compile-time (semantic and syntactic), logical, and runtime (exceptions). While much work has focused on the first two, there are very few tools that exist for interpreting the sometimes cryptic messages that result from runtime errors. Novice programmers in particular have difficulty dealing with uncaught exceptions in their code and the resulting stack traces, which are by no means easy to understand. We present Backstop, a tool for debugging runtime errors in Java applications. This tool provides more user-friendly error messages when an uncaught exception occurs, and also provides debugging support by allowing users to watch the execution of the program and the changes to the values of variables. We also present the results of two preliminary studies conducted on introductory-level programmers using the two different features of the tool.

Categories and Subject Descriptors
K.3.2 [**Computers and Education**]: Computer and Information Science Education – *computer science education.*

General Terms: Human Factors, Languages.

Keywords: Exception handling, Debugging.

1. INTRODUCTION
Despite the availability of many high-quality IDEs and frameworks for developing Java applications, many instructors at the introductory level prefer to focus on the Java language itself, so that students are not overwhelmed by needing to learn both a language and a tool (in addition to algorithmic thinking, data structures, the compilation process, etc). Unfortunately, when left only with the raw Java programming language, tasks such as interpreting stack traces become quite daunting when we consider that the language support was not designed with the novice programmer in mind. This leads to frustration for the student, and extra work for the teaching staff, who must spend valuable time explaining the meaning of cryptic Java error messages or teaching more complex debugging techniques and tools.

In this paper we present Backstop, a debugging tool designed for programmers studying Java at the introductory level. Backstop produces a detailed and helpful error message when an uncaught runtime error (exception) occurs; it also provides debugging support by displaying the lines of code that are executed as a program runs, as well as the values of any variables on those lines.

Figures 1 and 2 compare the debugging output produced by Backstop and the Java debugger jdb [19], respectively. Backstop shows the changes to the variables in each line, and also displays the values that were used to compute them. Additionally, it does not require the user to enter any commands in order to see that information. On the other hand, jdb only provides "snapshots" of the variable values on demand, without showing the results of individual computations, and requires more user interaction.

```
Fib.java:27: sum = x + y; (x was 2 , y was 1) (sum is now 3)
Fib.java:29: y = x; (x was 2) (y is now 2)
Fib.java:31: x = sum; (sum was 3) (x is now 3)
```

Figure 1. Debugging output produced by Backstop

```
main[1] step
Step completed: "thread=main", Fib.fibi(), line=26 bci=32
27          sum = x + y;

main[1] step
Step completed: "thread=main", Fib.fibi(), line=29 bci=37
29          y = x;

main[1] step
Step completed: "thread=main", Fib.fibi(), line=30 bci=39
31          x = sum;

main[1] locals
Method arguments:
num = 6
Local variables:
x = 1
y = 1
sum = 2
i = 2
```

Figure 2. Debugging output produced by JDB

More importantly, when a runtime error occurs, jdb only shows the standard Java stack trace; Backstop displays a more friendly error message with information designed to help the beginning programmer fix the problem (see Figure 3). Unlike jdb, which is very powerful but also very difficult to use, Backstop requires very little input from the user but produces a variety of helpful information that will assist the novice programmer in identifying, debugging, and fixing runtime and logical errors, as well as seeing a running program's path of execution.

2. RELATED WORK
There is much work in the area of making compiler errors understandable to the novice programmer, but we are not currently aware of any work in making runtime errors easier to interpret. BlueJ [10], ProfessorJ [6], and DrJava [2] all incorporate more user-friendly compile-time errors (such as syntax and semantic errors) into their programming environments, but none of these tools has support for runtime errors. Similarly, Gauntlet [5, 9], Expresso [8] and HiC [7] seek to identify the most

SIGCSE'08, March 12–15, 2004, Portland, Oregon, USA.

common compiler errors and provide useful messages around those, but those are pre-processors only and do not provide runtime support. Although Backstop has some debugging capabilities, it is not intended to be a replacement for visual tools like DDD [18] or JGrasp [20], which are purely front-ends for command-line debuggers and do not address runtime errors.

Some tools have been designed to quickly identify logical errors but not necessarily to resolve them. InSTEP [14] provides an analysis of a student's running program and can help identify some issues such as an infinite loop or incorrect output, but does not have any tracing facility for seeing intermediate values. DEBUG [11] was written for Pascal and provides analysis of logical errors but does not address runtime errors.

There has also been much investigation of debugging techniques among novice computer science students [1, 12, 16]. Although we are not claiming to provide any new methodology, the tool we present here can be used in conjunction with the approaches in [3], for instance.

The debugging aspect of Backstop is most similar to that of CMeRun [4], which produces a comparable output for C++ programs. However, Backstop includes the handling of runtime error messages, and also provides other advantages such as support for user-defined data types (classes). More importantly, Backstop provides a preprocessor to ensure that the source code is in the appropriate format, whereas CMeRun imposes more restrictions on the formatting of the code.

3. BACKSTOP FEATURES

Backstop consists of two programs that perform the following tasks for helping novice programmers address runtime and logical errors respectively: display error messages concerning uncaught Java exceptions in plain English; and print out the lines of code, including the values of variables, as the lines are executed.

Although Backstop can perform compilation, it is not a compiler: it merely uses the com.sun.tools.javac.Main class. As such, it does not address compile-time errors (semantic and syntactic), and it assumes that the code provided to it will compile. However, given a Java program, Backstop can at once annotate it and encompass it in order to provide debugging support.

Note that in this paper, we use "error" and "exception" interchangeably to refer to any uncaught Java runtime error.

3.1 Interpreting Runtime Errors

The Backstop tool is designed to "catch" any uncaught exceptions. Once the exception (or any other Java runtime error) is caught, Backstop can interpret its meaning and provide a user-friendly message that also seeks to provide some enlightenment as to what caused the error.

A programmer who wants to use Backstop to catch and interpret an uncaught runtime error need only run the Backstop program and provide the name of the (compiled) Java class that contains the "main" method, along with any command-line arguments. Backstop then uses Java reflection to load the class and execute its main method; should any runtime error occur and the ensuing exception is uncaught in the user's program, Backstop will catch it (it catches anything that extends java.lang.Throwable) and use the values in the object and its corresponding StackTraceElements to interpret and display information about the exception.

3.1.1 Features

Some information – like the class, method, and line number in which the error occurred – can easily be gleaned from the stack trace, and this information is displayed first. But rather than stating the name of the exception, Backstop tries to explain it in layman's terms; for instance "The code tried to use an illegal value as an index in an array" instead of "ArrayIndex-OutOfBoundsException", or "The code tried to call a method or refer to a member variable on an object which is null", instead of "NullPointerException". Where possible, additional information from the stack trace (such as the value of an out-of-bounds array index) is given in the message as well.

In the case where the source code is available to Backstop (for instance if the source files are in the same directory), Backstop will show the offending line of code and then try to determine (given the context of the exception) what variables may have caused the error. For instance, if the line "foo[x] = 4;" causes an ArrayIndexOutOfBoundsException and the stack trace has the value -2, then Backstop will point out that "the code was trying to access an element at index -2 of the array called 'foo'" and that "the variable 'x' had a value -2 when the error occurred."

Backstop also provides some assistance to the student as a reminder of how to possibly avoid the same problem in the future. In the above example, it would suggest "Remember, you cannot have a negative index in an array. Be sure the array index is always positive". For a NullPointerException, Backstop would recommend that the user "may need to initialize the object by using the keyword 'new'."

Lastly, so as to relate these exceptions to material that may have been taught in lecture, and so that the student can learn to understand these errors even without Backstop, the type of the exception and the original Java stack trace are presented at the end. Sample output is shown in Figure 3.

```
******************************************************
********************** ERROR **************************
******************************************************
An error occurred at line 124 of Test.java in the method
"main". I'm sorry, the program cannot continue.

The line of code that caused the error is
            int myNum = Integer.parseInt(arg);

It seems that the code wants to convert a String to an
integer.

However, the String "4.5" appears to be a floating point
value, not an integer. You may want to use a different
datatype to store the value, like float.

This is called a NumberFormatException.

The full error is as follows:
java.lang.NumberFormatException: For input string: "4.5"
       at java.lang.NumberFormatException.forInputString(
             NumberFormatException.java:48)
       at java.lang.Integer.parseInt(Integer.java:456)
       at java.lang.Integer.parseInt(Integer.java:497)
       at Test.main(Test.java:124)
```

Figure 3. Runtime error as shown by Backstop

3.1.2 Design Considerations

Special care was taken when writing the error messages that would appear to the user. Because the intended audience of this tool consists of introductory-level programmers, we want to ensure that the appearance of an error does not make the student feel "guilty" or as if he or she had done something "wrong". Therefore, we intentionally avoid wording like "You caused an error" or even "Your code caused an error", but instead use

phrasing like "The code caused an error", so as to deflect blame away from the student, as recommended in [15].

In addition, in the part of the error message that gives hints as to how to fix the error, we stay away from words like "must" or even "should" (such as "the array index must be positive"), which may come across as pushy and pedantic. Instead, the messages strive for a friendly, more helpful tone, with wording like "don't forget" or "keep in mind", which gives the impression that the student already knows these facts.

Backstop is designed to specifically handle 22 different types of Java Exceptions and Errors, in particular those that we feel are most likely to be encountered in a CS0 or CS1 course (the full list of errors and corresponding messages is in [13]). However, when Backstop encounters a Throwable that it does not understand, it will simply print out a message indicating as such, as well as the original stack trace. A future version of Backstop could point the user to the online API documentation for the given error. A production quality implementation could also allow for configurable error messages; currently they are hard-coded.

3.2 Using Backstop to Debug Errors

To aid in the debugging of runtime or even logical errors, Backstop can display the following information for each line of code that is executed: the file name and line number; the code as it is written; and the value of variables in the line (on both sides of an assignment statement, if necessary). This information will help the student see which lines of code are executed, the order of execution, and the values that the different variables are taking on. Most importantly, when a runtime error occurs, the student will be able to see the path of execution that caused the error, as well as the values of all variables at the time of failure. When used simply as a debugger, the student also has the option of inserting breakpoints into the program in order to step through it instead of seeing all output at once.

Before using Backstop, a student can use its preprocessor to ensure that the source code adheres to the following conventions: each line must contain only one statement; each left and right brace must appear on a line by itself; multiple assignments on the same line are expanded to multiple lines; and, a "case" statement must appear on a line by itself. Students' code must also ensure that each statement is contained on only one line, and that blocks of code inside loops are surrounded by curly braces (these are not currently addressed by the preprocessor).

Once the code is in the appropriate format, a student who wants to use Backstop to debug a runtime or logical error (or just to see the execution of the program) starts Backstop and provides the name of the files containing the source code of the Java classes. Backstop then makes backups of the original files and reads the files one line at a time. It copies each line of code and puts a System.out.print command (containing the file name, line number, and code) before the original line of code, and another (containing the values of any variables) after it. These statements are then all written to the new source file. Figure 4 shows an example of code modified by Backstop. This is the code that produced Figure 1, but it is never seen by the student, of course; it is only used internally by Backstop.

The modified code is then compiled by Backstop, and the original files are restored, so that the programmer can modify that code and does not have to see the code modified by Backstop. However, a copy of the Backstop-modified code is maintained. The programmer then runs the program as normal and would see the debugging output. Sample output is shown in Figure 1 above.

```
System.out.print("\nFib.java:27: sum = x + y;");
System.out.print(" (x was " + x);System.out.print(" ,  y was
" + y );System.out.print(")"); sum = x + y;
System.out.print(" (sum is now " + sum + ")");


System.out.print("\nFib.java:29: x = sum;");
System.out.print(" (sum was " + sum);System.out.print(")");
x = sum; System.out.print(" (x is now " + x + ")");


System.out.print("\nFib.java:31: y = x;");System.out.print("
(x was " + x);System.out.print(")");y = x;
System.out.print(" (y is now " + y + ")");
```

Figure 4. Code modified by Backstop (original code in bold).

3.2.1 Design Considerations

A number of considerations were made in the design and implementation of this aspect of Backstop. Perhaps most importantly, it is necessary to insert any System.out.print statements on the same line as the original source code, so as not to affect the line numbers. This was done so that a stack trace would have the correct line numbers with respect to the code in the unmodified program. On a similar note, Backstop displays the line of code *before* it gets executed, rather than *after* (as in CMeRun), so that if the code throws an exception, the last line of code shown will be the one that caused it.

We also considered the manner in which the debugging output was displayed. Whereas CMeRun intersperses the variable values with the C++ code, Backstop leaves the code untouched so that it appears recognizable and readable; the variable values are displayed at the end of the line. Backstop also intentionally uses phrases like "x is now 5" (for variables on the left side of an assignment) and "y was 9" (for those on the right) to show which variables are changing; moreover, the use of "is now" and "was" was chosen for statements like "x = x + 4", so that the student can see the value of x before and after the line is executed. Lastly, we intentionally did not use equals signs to show variable values, since they could be misinterpreted as more Java code.

4. USER STUDIES AND EVALUATION

We conducted two preliminary studies to measure the usefulness of the two main features of the Backstop application: interpreting and fixing a runtime error, and debugging a logical error. In order to test each feature in isolation, the students were given access to only one of the two parts of Backstop in each study. In testing their ability to fix a runtime error, they were unable to see the debugging output that showed the lines of code as they were executed; in testing their ability to debug a logical error, there was no runtime error for them to look at.

Both studies involved the same group of seventeen students (8 male, 9 female) who at the time of the study had recently completed a CS1 course in Java at Columbia University, but had little or no programming experience besides that. The studies were conducted separately on each student, and were all overseen by the first author of this paper (who was also the students' instructor in their CS1 course; although this is somewhat irregular, it was necessary due to time constraints and availability). In each study, the subject was given a task to perform, and we recorded the time to complete the task. Upon

completion of the task, the subject was asked to answer subjective questions about his/her experience.

4.1 Understanding a Runtime Error

In the first task, the students were given a Java program that, for given inputs, would cause a runtime error, resulting in an uncaught exception and termination of the program. The algorithm was first explained to the students, then they were shown the code, and lastly it was demonstrated that an exception would occur for certain inputs. They were then asked to interpret the resulting error message and find and fix the error in the program. In this experiment, we created a control group of four students who did not use Backstop, but only saw the default stack trace produced by Java; the other thirteen students used Backstop.

The program used in this study was designed to read in a sentence from standard input and display the number of occurrences of each letter in the sentence. However, unbeknownst to the students, the inclusion of any non-alphabetic character (such as a blank space or punctuation) would cause a particular calculation to yield a negative number that – when used as an index to an array – causes an ArrayIndexOutOfBoundsException. The Backstop error message is shown in Figure 5. All of the students had had that particular exception explained to them in their CS1 course.

```
**************************************************
********************* ERROR **********************
**************************************************
An error occurred at line 30 of StringCount.java in the
method "main". I'm sorry, the program cannot continue.

The line of code that caused the error is
            occur[(int)(c - 'a')]++;

It seems that the code tried to use an illegal value as
an index in an array.

The code was trying to access an element at index -65
of the array called "occur".

The expression "(int)(c - 'a')" had the value -65 when
the error occurred.

Remember, you cannot have a negative index. Be sure
that the index is always positive.

This error is called an ArrayIndexOutOfBoundsException.

The full error is as follows:
java.lang.ArrayIndexOutOfBoundsException: -65
        at StringCount.main(StringCount.java:30)
```

Figure 5. Backstop output from user study.

Of the thirteen students who used Backstop, nine (76%) were able to identify and fix the cause of the error within eight minutes; the other four could not complete the task within the allotted fifteen minutes. All of the students were able to realize quickly (within two minutes) which calculation was resulting in a negative number, which in turn was causing the exception when used as an array index. The problem some struggled to solve was the issue of "how did this calculation come to produce a negative number?" and they needed to find the answer by debugging the error on their own, since we did not provide them with the debugging functionality of Backstop. Students mainly debugged the problem by inserting System.out.println statements on their own, trying different inputs to see if they could reproduce the error, or thinking about the algorithm. Most likely, of course, the students would have performed better if they had access to the line-by-line debugging output. This particular test is left as future work.

We then asked the thirteen students who had used Backstop questions about the tool. When asked whether they found it "useful", all of the students said "yes". When asked whether the output had misled them, two of the students who did not complete the task said that it had, but admitted that they had not actually read the error message very carefully. We also asked for the students' comments on the tone of the error message produced by Backstop. Six of the thirteen (46%) claimed that it was "friendly" compared to the stack trace produced by Java. However, four said that the error message was too long, and two more pointed out that anyone but a novice programmer may not appreciate the "pedantic" nature of the message (however, Backstop is indeed targeted to novice programmers). Three of the students said that they did not pay any attention to the tone of the message, but they were merely concerned with getting the appropriate information.

For our control group (who did not use Backstop), we selected four of the top students from the CS1 course. We expected that, even without Backstop and just by looking at the Java stack trace, this group would outperform the other group in terms of the time to complete the task. This proved to be the case, as all four were able to fix the error in under five minutes (one student fixed it in less than two). However, upon seeing the error message produced by Backstop, three of the four said that they probably could have solved the problem faster because they were somewhat confused by the stack trace (the array index that caused the exception is displayed but was interpreted by some students as an error code); the only student who said that Backstop may not have helped her claimed that the error message was too long.

Although five of the 17 students said that the error message was too long, we feel that all of the information provided by Backstop is necessary for fixing the runtime error; an alternative suggested by one student is to show only the most crucial information (line number and exception type) and then give the user the option of seeing more (such as the advice for fixing the error). Another possibility is to allow for user-adjusted verbosity levels.

4.2 Finding a Logical Error

In the next task, all 17 students were asked to find a logical error in a program designed to produce the n^{th} Fibonacci number, where n was an argument to the Java program. As in the previous study, the algorithm was explained, the students were shown the code and its output, and lastly it was demonstrated that the output was clearly wrong. They were then asked to use Backstop to debug the code and trace the values of variables in order to see where the logical error had been introduced (we should note that the version of Backstop used in the study did not have the "breakpoint" feature described above; however, the debugging output of the program was small enough to fit into one console window).

The most impressive result of the study was that eight of the 17 students (47%) were able to find the logical error (two lines of code had been juxtaposed) in one minute or less. All of them claimed that the debugging output helped them find the error and that they probably could not have found the error as quickly without the Backstop tool. It bears emphasizing that the students were, in one minute or less, able to find an error in code *that they did not originally write*.

Of the remaining nine students, five completed the task in five minutes or less, two took six to ten minutes, and two did not complete the task within the fifteen minute time limit. The two

students who did not complete the task commented that the debugging output was hard to read/understand, but said that the output did not hinder them in trying to find the error.

All 17 students were asked questions about their experience at the end of the study. When asked whether the output produced by Backstop was "helpful", sixteen said "yes", whereas the only student who did not was one who did not complete the task because he did not understand the output. When asked "did the output mislead you in any way?", all but five of the students said "no", though four of the others said "only at first". The same student who said Backstop was not "helpful" was the same one who claimed to have been misled by the output.

We also asked the students whether there was any other information that could have helped them solve the problem faster. Somewhat surprisingly, five of the students said that it would have been useful for the program to indicate that a logical error had occurred; of course, the program cannot *know* that a logical error has occurred without knowing what the expected output should be, but the students did not realize that. This indicates that perhaps the students did not fully understand the task, or at least overestimated the tool's capabilities.

The students were split on their opinions concerning the appearance of the output. Five students (29%) said that more spacing would have helped make it easier to read; however, another four (23%) claimed that more spacing would have made it harder to read. Three (17%) said that the spacing would not make a difference (the remaining students had no opinion).

5. CONCLUSIONS AND FUTURE WORK

We have presented Backstop, a debugging tool targeted at novice and introductory-level Java programmers. The results of our study suggest that it facilitates the interpretation of runtime errors and aids in the debugging of logical errors. The primary contribution of our tool is its ability to make the error messages produced by uncaught Java exceptions easier to understand, and to provide friendlier and more useful information about how to fix the cause of the error. We have also demonstrated that addressing the debugging of runtime errors is helpful to CS1 students, who clearly could benefit from tools that allow them to understand the cause of such errors and then get assistance in how to fix them.

Backstop could conceivably have been implemented using the Java Platform Debugger Architecture and the JVM Tools Interface, which may provide tighter integration between its two main parts. Additionally, the exception handling feature could be integrated into other debugging and error handling frameworks such as JGrasp [20]. We are currently investigating integration with Eclipse [17].

Further studies are necessary to evaluate the usefulness of Backstop, using a larger group of students and by comparing results against other debugging tools like jdb. Additionally, a study needs to be conducted in which the runtime error handling is combined with the debugging feature to see how the students perform when using the combination of tools.

The authors would like to thank Phil Gross and the students who participated in the user studies. Murphy and Kaiser are members of the Programming Systems Lab, funded in part by NSF CNS-0717544, CNS-0627473, CNS-0426623 and EIA-0202063, NIH 1 U54 CA121852-01A1.

6. REFERENCES

[1] M. Ahmadzadeh, D. Elliman, C. Higgins, "An Analysis of Patterns of Debugging Among Novice Computer Science Students", *Proc. of ITiCSE '05*, Portugal, June 2005, 84-88.

[2] E. Allen, R. Cartwright, B. Stoler, "DrJava: A lightweight pedagogic environment for Java", *Proc. of SIGCSE 2002*, Covington KY, Feb 2002, 137-141.

[3] R. Chmiel, M.C. Loui, "Debugging: from novice to expert", *Proc. of SIGCSE 2004*, Norfolk VA, Mar 2004, 17-21.

[4] J. Etheredge, "CMeRun: Program Logic Debugging Courseware for CS1/CS2 Students", *Proc. of SIGCSE 2004*, Norfolk VA, March 2004, 22-25.

[5] T. Flowers, C. Carver, J. Jackson, "Empowering Students and Building Confidence in Novice Programmers through Gauntlet", *34th ASEE/IEEE Frontiers in Education Conference*, Savannah GA, Oct 2004, T3H-10 – T3H13.

[6] K. Gray and M. Flatt, "ProfessorJ: A Gradual Introduction to Java through Language Levels", *Proc. of OOPSLA '03*, Anaheim CA, Oct 2003, 170-177.

[7] R. Hasker, "HiC: A C++ Compiler for CS1", *Journal of Computing Sciences in Colleges 18:1*, Oct 2002, 56-64.

[8] M. Hristova, A. Misra, M. Rutter, R. Mercuri, "Identifying and Correcting Java Programming Errors for Introductory Computer Science Students", *Proc. of SIGCSE 2003*, Reno NV, Feb 2003, 153-156.

[9] J. Jackson, M. Cobb, C. Carver, "Identifying Top Java Errors for Novice Programmers", *35th ASEE/IEEE Frontiers in Education Conf.*, Indianapolis IN, Oct 2005, T4C-24 – 27.

[10] M. Kolling and J. Rosenborg, *BlueJ*, http://www.bluej.org

[11] T. Lukey, K. Loose, D.R. Hill, "Implementation of a debugging aid for logic errors in Pascal Programs", *Proc. of SIGCSE 1987*, St. Louis MO, 1987, 386-390.

[12] R.F. Mathis, "Teaching Debugging", *Proc. of SIGCSE 1974*, 1974, 59-63.

[13] C. Murphy *et al*., Columbia University Dept. of Computer Science tech report cucs-027-07, Sept. 2007.

[14] E. Odekirk-Hash, J. Zachary, "Automated Feedback on Programs Means Students Need Less Help From Teachers", *Proc. of SIGCSE 2001*, Charlotte NC, 2001, 55-59.

[15] B. Shneiderman, "Designing computer message systems", *Communications of the ACM 25:9*, Sept 1982, 610-611.

[16] J. Wilson, "A Socratic approach to helping novice programmers debug programs", *Proc. of SIGCSE 1987*, St. Louis MO, 1987, 179-182.

[17] http://www.eclipse.org

[18] http://www.gnu.org/software/ddd/

[19] http://java.sun.com/javase/6/docs/technotes/tools/windows/jdb.html

[20] http://www.jgrasp.com