Development of a Processor That Detects Non-Literal Java Errors

Thomas Dy
Department of Information Systems and Computer Science
Ateneo de Manila University
Quezon City, Philippines
Email: dysprosium66@gmail.com

Edward John Robles Jr.
Department of Information Systems and Computer Science
Ateneo de Manila University
Quezon City, Philippines
Email: edyey@yahoo.com

Ma. Mercedes Rodrigo
Department of Information Systems and Computer Science
Ateneo de Manila University
Quezon City, Philippines
Email: mrodrigo@ateneo.edu

***Abstract*—Novice programmers typically have problems diagnosing errors in their programs. Attention should be paid to non-literal errors, or those whose compiler error does not exactly correspond to how to fix the problem. Through the analysis of compilation logs, we determined the errors frequently committed by novice programmers and developed rules to determine the actual error that occurred. Based on those rules, we then developed a processor that checks novice student code for these non-literal errors and produces more informative error reports. Our testing shows that the processor is successful at detecting the correct error at least 80% of the time.**

I. Introduction

Debugging is the process of locating and resolving errors (bugs) within a computer program. It is achieved using a variety of strategies ranging from tracing to commenting out code to systematic testing to an intuitive pattern matching in which programmers simply know that code does not "look right" [1]. Novice programmers apply these strategies with varying levels of skill with most finding debugging difficult to master. Indeed, researchers have found that even novices with a good grasp of programming syntax are sometimes unable to debug effectively [2]. Poor debugging skills lead to poor performance, frustration, and possibly the introduction of new errors [1].

Students' lack of debugging skills are a source of concern for human tutors and teachers. Also, some of the debugging difficulties that novice programmers encounter are related to programming environment and an inability to get help [3] which may make students unwilling or unable to practice programming outside of class time.

Automated tools for helping novices cope with compiler errors have therefore been developed. A project by Coull et al attempts to provide novice programmers with solutions based on their compiler errors [4]. The application parses compiler messages from a Java Integrated Development Environment and then searches for suggestions from a database of errors and their solutions.

A debugging system that tries to understand the programmer's intention has also been developed. The system attempts to compare the programmer's intention with the program's design. Intention-based diagnosis can detect a wide range of errors, including deep faults resulting from design errors [5]. An example of such a system is PROUST [6]. PROUST generates a hypothesis about the user's intention and matches these against the code; it then explains this to the user.

The study we have undertaken is similar to that of Coull et al. We attempt to create a database of common errors and their solutions. In doing so, we hope to provide novice programmers with an automated means of support as they learn to program in Java.

II. Towards More Precise and Understandable Error Messages

There are times that the compiler is not directly helpful at aiding programmers correct an error. The compiler may say "; expected" yet adding a semicolon will cause the same error. When this kind of discrepancy occurs, the compiler is said to have outputted a non-literal error as opposed to literal errors which are those whose true cause is identified by the compiler's error message [7]. Figure 1 shows code that would cause a "; expected" error, however, adding a semicolon would only lead to more errors. The actual problem is there is a missing "+" operator to join the strings "Hello," and "World" which is fixed in figure 2.

```
1 String x = "Hello,"
2            "World";
3 System.out.println(x);
```

Fig. 1. Erroneous code

```
1 String x = "Hello," +
2            "World";
3 System.out.println(x);
```

Fig. 2. Corrected code

The compiler we generally refer to is the javac compiler bundled with Java. There are other Java compilers, however, that report errors with varying degrees of accuracy. The Eclipse IDE, which has it's own compiler for example, provides helpful suggestions to correct errors. These hints can sometimes be confusing, though. For example, Eclipse sometimes reports "@ expected" instead of "'' expected" in the case of having "()"

in the class declaration. The GCJ compiler outputs either the same error as the regular javac compiler, or something closer to the actual error present. Overall however, our evaluation of the two compilers is that they are better, but still not satisfactory in helping the programmer with the error.

Inaccuracy of error reporting has been shown to cause problems for novice programmers. If the compiler's message did not help the student fix the problem, the student may respond by moving on to another error instead of trying to fix the said problem. Also, when the same error is encountered, the student may respond as if the problem was still non-literal [7].

In order to aid students regarding those problems, we aim to devise methods of detecting non-literal errors. By doing this, time can be saved for both the teacher and the student. The teacher will no longer have to manually look through code to look for the problem and figuring out what the problem actually is. More time can then be spent helping the students with their problems. The same can be said for regular programmers, they will not need to spend as much time deciphering and diagnosing their errors.

III. Review of Existing Methods

We found three frameworks that did source code analysis of students' work for assessing quality and aiding the student. One is from the Queensland University of Technology [8] the next one is Expresso [9], developed in Bryn Mawr College and the last one is Check'n'Crash developed by Christoph Csallner and Yannis Smaragdakis [10].

The framework developed in QUT was designed to be a configurable and extensible framework that can automatically assess a student's work through static analysis and software metrics [8]. Static analysis is done by parsing the student's code into an abstract syntax tree and its structure is compared with the model solution for the given problem. Various software metrics can also be applied such as measuring the complexity of the code.

Another framework, Expresso was developed in Bryn Mawr College to overcome the problem of cryptic compiler messages [9]. The approach Expresso takes is to do a better job of generating error messages and suggesting possible solutions to those errors. It is implemented as a multi-pass preprocessor. Comments are first stripped, then the program is stored into memory and finally mistakes are detected.

The last framework, Check'n'Crash is a combination of static checking and concrete test-case generation. It uses the Extended Static Checker for Java (ESC/Java) and JCrasher. ESC/Java is a compile-time program checker that detects precondition violations. On the other hand, JCrasher is an automatic testing tool for Java that attempts to detect bugs in a program by causing the program to crash. JCrasher defines heuristics that determine whether a Java exception should be considered as a bug or that JCrasher violated the code's preconditions. Check'n'crash tries to improve on ESC/Java and JCrasher by reducing the testing time and producing more comprehensible error reports [10].

These frameworks have their limitations, however. The QUT framework only works for fill-in-the-blanks style programming assignments, while Expresso has not yet been fully tested in a real environment. Check'n'Crash and other similar static analysis programs are more geared towards checking semantic and logic errors of working programs as opposed to compiler errors.

IV. Methodology

A. Data Collection

The data we used was collected in 2007-2008 from Ateneo de Manila University freshman and sophomore Computer Science and Management Information Systems majors taking their CS21a course, the Introduction to Computer Science. Over the course of nine weeks, students were asked to complete 5 programming laboratory exercises in which they applied newly-taught Java programming concepts. They performed these exercises in computer laboratories with 1:1 computer-to-student ratios. Each computer was connected to a network and was installed with BlueJ, an integrated development environment (IDE). The version of BlueJ that the students used was specially instrumented to send each logs of each compilation to a central server [11]. These logs contained the source code, error and line number, if any, for every compile each student does. Further details regarding the actual data collection process are detailed in [12].

B. Determining the Top Errors

The next step involves filtering the data to get the most common errors that students make. Errors that are essentially the same will be grouped together. For example, an undeclared variable error will be grouped together with all other undeclared variable errors found. All other error types will then be grouped in the same manner. From that list of grouped errors, we will take the most common errors and look for instances of non-literal errors.

TABLE I
Top Errors

Error	**(Percentage of all 14665 errors)**
unknown variable	2772 (18.90%)
';' expected	1710 (11.66%)
'[', ']', '(', ')', '', '' expected	1403 (9.57%)
unknown method	1382 (9.42%)
incompatible types	1131 (7.71%)
missing return statement	999 (6.81%)
illegal start of expression	762 (5.20%)
unknown class	617 (4.21%)
identifier expected	543 (3.70%)
class or interface expected	393 (2.68%)

C. Analyzing Compilation Logs

For each error in the top errors list, a sample set of codes where the error occurred was read. If the error was correct then the code will be disregarded. On the other hand, all other cases where the error could not be solved by complying with

```
1 public class Test() {
2
3 }
```

Fig. 3. Code that causes a '{' expected error

the compiler will be gathered. In these cases, patterns will be observed to see how the errors can be identified.

To illustrate this method, let us assume that the code in Figure 3 is one chosen from those with '{' expected errors. The compilation log tells us that this code has an error in line number 1. Basing on the error message, we can see that it already has a '{', therefore, it is a non-literal error. We then look for the actual error in the code, which is the extra '()' after Test. We then record the compilation ID and the actual error. This process is repeated for all the sample data to be analyzed.

D. Processor

A processor was written in Java to supplement the Java compiler. From the set of rules gathered from analyzing the errors, the processor would act as a guide for the students towards discovering the real error generated by their program. Every time a student compiles, it would check for errors and gave the student possible ways of fixing their code. For example, if the processor encountered a missing semicolon error it would check if the error fit a rule where the error was not really a missing semicolon; it then outputs the real error.

E. Testing

For testing, we planned to test it in a classroom setting with two groups. The control group would have the regular compiler, while the test group would use our processor. This would be done for a whole semester, and then the grades of both groups will be compared to see if there was any improvement.

V. Results

The top errors that we determined from our data are in Table I. From those, we have analyzed the top 5, namely the "cannot find symbol" errors (unknown variable, unknown method, unknown class), "incompatible types" errors, "';' expected errors" and the "'(', '[' and '{' expected" errors. We then compiled a list of the non-literal errors stemming from these errors [13]. Now, we have generated rules for all of the above errors except the "';' expected" errors since it was too complex. An example of the rules that we made are shown in Figure 4.

If the processor encounters a '{' expected error, it first checks what the second word after "class" is. If the word is a '(', the student mistakenly tried to put a parameter list after a class name. If the word is "throws" the student tried to throw an exception in class declaration. If there was some other word, the student most likely used an invalid class name. If the word is a "=", the class was used as a type. If there was

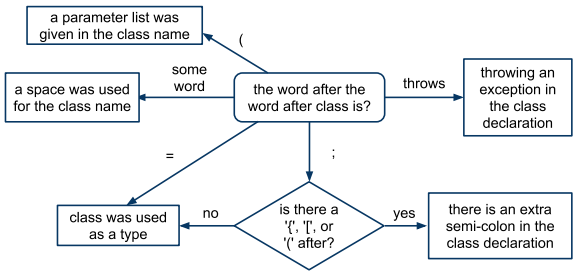

Fig. 4. brace expected diagram

a semi-colon after the classname, followed by a bracket, the class there was an extra semi-colon in the declaration. If not, the student attempted to use the class as a type.

From these rules, we developed a processor that would detect the actual error of a student's source code. For each error, we made a module that specialized in detecting non-literal errors for that specific compiler error. We were not, however, able to make a module for the ';' expected error because it was too complex.

After initial development, we tested the processor against the data we used for analysis. From this, we discovered that we had missed some non-literal errors, which we added into the processor accordingly. This develop-test cycle would be repeated until we observe that it is consistently correct with the test data.

We would have then proceeded to actual testing; however, due to time constraints, the test previously outlined in our methodology was unfeasible for us. Therefore, we decided to just run it against a different set of compile logs from the ones we analyzed and see how accurate the processor is. We chose to test it using the logs from School Year 2009-2010; the results are shown in Table II.

TABLE II
Processor Accuracy

Error	Percent Accuracy
cannot find symbol	86% (out of 100)
';' expected	not implemented
'(' expected	44% (out of 100)
'(' or '[' expected and '[' expected	100% (out of 80)
incompatible types	87% (out of 100)

The results of our testing were generally positive, with most of the modules garnering an accuracy above 80%. The only bad result was with the '(' expected detection, which had a high amount of errors uncaught by the rules we had made. We previously thought that it appears only under one particular situation, but apparently it occurs in at least 3 more.

Another noticeable result is the one for the '(' or '['

expected and '{' expected errors which reached 100%. We believe that this is not a real result as when the processor encounters those errors, it will always respond with only 1 generic message. There is definitely room for improvement in this respect.

In general though, most of the mistakes made by the processor were due to uncaught errors. Most of those could be easily remedied however as most of them are slight variations of previous rules. For example, we successfully catch the mistake of using '=' instead of '==' for comparison in an if statement; we failed to consider doing that in the for and while statements.

The latest list of non-literal errors for each error are shown in tables III to VII.

A. Limitations

Through our analysis of the compile logs and its behavior, we discovered certain limitations with the data we were working with. The actual Java compiler outputs the column number of the error as well as the line number, and this could have been an additional help in determining the actual error committed. We could have compiled the source code ourselves, but sometimes, other supporting source code is not included in our data.

Our data also only contains one error message per compile, whereas the compiler outputs multiple error messages for every compile. The current processor handles only 1 error message per compile, but it can be reworked to handle multiple error messages at a time. However, some of those errors may be caused by other errors, so those cases will need to be handled carefully.

Another limitation we encountered was that there is an extra element of human knowledge to determining the actual error. Different actual errors may have the same structure, but we can differentiate them based on understanding the intention of the programmer. For example, `public int MobilePhone getPhone() {}` would cause a ';' expected error. Basing only on the structure, it can be that the programmer used an invalid name for the method, or there are too many modifiers in the method declaration. For us, it is obvious that there is an extra int in the method declaration since we can infer that the programmer wants to return a MobilePhone because the method name is `getPhone`.

In line with the previous problem, another problem is the subjectivity of determining the error that occurred. Some errors are ambiguous even to us since we don't really know the exact intention of the person who wrote the code. Moreover, there is the problem of the granularity of the error detection, how specific or general should we be? This was quite evident with the problem we had with the 100% result.

VI. Conclusion

Our tests show that detecting non-literal errors is indeed possible and feasible. The processor reached an accuracy of around 80% and we think it is possible to reach an accuracy of above 95% if all possible errors are taken into account. We believe that if fully developed, it will definitely be a useful tool for novice programmers, teachers and maybe even regular programmers.

There is definitely much room for future work. Modules for the other errors could be developed to broaden the capabilities of the processor. The current modules seem crude and have room for improvement as well. Of course, for it to reach a broader audience, it should be able to integrate into IDEs such as BlueJ, which is used in our introductory programming class.

Forgoing the limitations of the BlueJ source data will surely open up more possibilities for the processor. There would no longer be a limit of 1 source file at a time. Additional compiler hints such as the column number would be available, which would make the task of determining the error much easier. Another avenue for improvement would be to parse the code itself so that you would know if a variable is an int, a double, a String, or something else; this may also help in determining non-literal errors.

Acknowledgment

The authors thank Dr. Matthew C. Jadud of Allegheny College, Ms. Emily Tabanao of Mindanao State University-Iligan Institute of Technology, Mr. Jose Alfredo De Vera, Ma. Beatriz Espejo-Lahoz, and the technical and secretarial staff of the Ateneo de Manila's Department of Information Systems and Computer Science for their assistance with this project. We thank the Ateneo de Manila's CS 21 A students, school years 2007-2008 and 2009-2010, for their participation. Finally, we thank the Department of Science and Technology's Philippine Council for Advanced Science and Technology Research and Development for making this study possible by providing the grant entitled "Development and Deployment of an Intelligent Affective Detector for the BlueJ Interactive Development Environment."

References

[1] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander, "Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies," in *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2008, pp. 163–167.

[2] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An analysis of patterns of debugging among novice computer science students," in *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. New York, NY, USA: ACM, 2005, pp. 84–88.

[3] A. Ebrahimi, D. Kopec, and C. Schweikert, "Taxonomy of novice programming errors with plan, web, and object solutions," ACM, Computing Surveys, December 2006.

[4] N. Coull, I. Duncan, J. Archibald, and G. Lund, "Helping novice programmers interpret compiler error messages," in *4th Annual LTSN-ICS Conference. National University of Ireland, Galway.(August, 2003)*, pp. 26–28.

[5] W. Johnson, "Understanding and debugging novice programs," *Artificial Intelligence and Learning Environments, MIT Press, Cambridge, MA*, pp. 51–97, 1990.

[6] W. L. Johnson and E. Soloway, "Proust: Knowledge-based program understanding," in *ICSE '84: Proceedings of the 7th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1984, pp. 369–380.

[7] M. C. Hughes, M. C. Jadud, and M. M. T. Rodrigo, "Novice programmer strategies for string formatting," unpublished.

TABLE III
CANNOT FIND SYMBOL ERRORS

Error	Example

A required package was not imported	`public class Sample {`
	`  Scanner c;`
	`}`

Misspelled class name	`Sacnner c;`
Misspelled method name	`String a = Scanner.nxet();`
Wrong method parameters	`System.out.println("Hello", 5);`
Undeclared variable	`a = 5; // a was not declared`
The fully qualified name was not used	`printf("Hello"); // instead of System.out.printf`
Extra = sign after method call	`System.out.println = ("Hello,");`
Variable declaration and assignment with no name	`CoffeeMachine = new CoffeeMachine();`
Unfinished statement	`liters` `cups = 0 + cups;`

TABLE IV
';' EXPECTED ERRORS

Error	Example
No semicolon after declaration	`public void send() int a`
Invalid variable or method name	`private double Credits Left = 0;`
extra) in method call	
misspelled keywords	`If (pesos >= 25)`

() instead of {} for code blocks	`public int doSomething()`
	`(`
	`  return 5;`
	`)`

unclosed previous block	

No parameters in method declaration	`public Students A {`
	`  A = New Students;`
	`}`

No semicolon on previous line	`int a` `int b; // error line here`
Colon instead of semicolon	`int a:`
Misspelled modifiers in declaration	`Public Static void main( String args[])`
No + in String concatenation	`return "P" credits;`
No ; to separate statements on one line	`return true else return false;`
Used ++ instead of +	`c = b ++ bb;`
Used a comma instead of semicolon in for statement	`for( int i = 1; i <= 50; i++ ) {`
No * for multiplication	`leftPesos = pesos - 6.5x;`
Used x for multiplication	`private double CreditsUsed` `= TotalMinutesCalled x RatePerCall;`
Comma instead of period to call method	`System,out,println();`

TABLE V
'{' EXPECTED ERRORS

Error	Example
Parameter list in class declaration Used (or [instead of { in class declaration	`public class Trial()` `public class Hello (`
Invalid class name	`public class Mobile Phone`
Class used as a type	`class GasStation = new GasStation();`
Placed a throws Exception in class	`public class Driver throws Exception`
Extra semicolon after class declaration	`public class MobilePhone; {`

TABLE VI
'(' EXPECTED ERRORS

Error	Example
Invalid method name	`public void do Something()`
Used void as a type for a variable	`public void value;`
If/For/While statement or method declaration without parameters	`if { doSomething(); }`
If/For/While statement or method declaration without parentheses	`if count == 5 {`
Extra modifier	`public void String toString()`

TABLE VII
INCOMPATIBLE TYPES ERRORS

Error	**Example**

Wrong return type	`public int renew() {`
	`  return "hello";`
	`}`

Did not use boolean in if statement	`if (dd = 5) {`
Assigning null to a primitive type	`int amount = null;`
Did not use int-type in switch statement	`String a = "hello";` `switch(a) {`

Assigning void to a variable	`public void doSomething() {}`
	
	`public void somethingElse() {`
	`  int a = doSomething();`
	`}`

Unfinished statement	`ePass =` `tollFee = 0;`

[8] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' java programs," in *ACE '04: Proceedings of the sixth conference on Australasian computing education*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, pp. 317–325.

[9] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," in *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2003, pp. 153–156.

[10] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 422–431.

[11] M. Jadud, "Methods and tools for exploring novice compilation behaviour," in *Proceedings of the second international workshop on Computing education research*. ACM New York, NY, USA, 2006, pp. 73–84.

[12] M. M. T. Rodrigo, E. Tabanao, M. B. E. Lahoz, and M. C. Jadud, "Analyzing online protocols to characterize novice java programmers," *Philippine Journal of Science*, vol. 138(2), pp. 177–199, 2009.

[13] T. T. Dy, E. J. D. Robles, and M. M. T. Rodrigo, "Detection of non-literal java errors in an introductory programming class," under review.