Quantifying The Incidence of Novice Programmers' Errors

Dr. Warren Toomey

School of IT, Bond University

Abstract

Existing research shows that students learning to program for the first time often make the same types of errors. Tools have been written to give students useful feedback when they make these errors, but no research has been done to determine the effectiveness of these tools. This paper is the preliminary result of a long-term study to answer the research question: will timely reporting of novice programming errors in an understandable format reduce the incidence of these errors? We report on a tool called Arjen which is designed to quantify the errors made by novice programmers. We give the results of a pilot study where first programming students used Arjen as part of their learning environment.

Keywords: Programming, Common programming errors, Computer Science Education, Java.

1 Introduction

Students learning to write computer programs for the first time face a range of difficulties: the basic concepts of programming (flow of execution, functions and parameters, recursion etc.), the specific syntax and semantics of the language being taught, as well as the process of implementing the solution to a problem as a computer program. This task is made more difficult when the compiler or development environment issues error messages for innocuous transgressions (e.g. a missing semicolon), and when the error messages are so cryptic that the student cannot interpret them and relate the messages to the concepts of programming (e.g. the message "char cannot be dereferenced").

At the same time, novice programmers are apt to make a common set of mistakes as they absorb the conceptual and language-specific material given in their first programming course (Spohrer & Soloway 1986, Hristova, Misra, Rutter & Mercuri 2003, Jadud 2005). Given this fact, it would seem to be intuitively obvious that, if useful and descriptive feedback about these mistakes were given to novice programmers, then they would learn not to make these mistakes. This should reduce the time spent on mundane mistakes and reinforce the learning of the core concepts in programming. However, research in this area tends to assert this hypothesis as self-evident and does not measure the reduction in common errors when feedback is given to novice programmers (Jackson, Cobb & Carver 2005, Nienaltowski, Pedroni & Meyer 2008).

This paper is the preliminary result of a long-term study to answer the research question: will timely reporting of novice programming errors in an understandable format reduce the incidence of these errors? To do this, a tool needs to be developed that identifies common errors in programs submitted by novice programmers, and returns descriptive error information and advice to them. Then, a longitudinal study with a large number of novice programmers can be conducted to determine if the incidence of common errors reduces over time when such a tool is made available to them.

In this paper we describe Arjen: a tool to identify common programming errors. We also outline the quantified results of a pilot study where Arjen was provided to a small number of students learning their first programming language, Java.

2 Previous Work

Hristova *et al.* describe a tool called Expresso which identifies common errors made by novice programming students learning Java (Hristova et al. 2003). The researchers conducted an initial survey of lecturers, teaching assistants and students to determine a suitable set of common errors and created a list of around 20 identified errors. With this list in hand, Hristova *et al.* assessed the possible approaches to implementing a tool to detect these errors, including a simple source code preprocessor and a post-processor of the Java compiler's error output. The researchers chose to write Expresso as a Java parser to be run before the actual compilation stage. Expresso was written as a hand-crafted parser in C++ to target the specific list of 20 errors. Along with the Expresso analyser, Hristova *et al.* created a test suite of 20 example source code files to ensure that Expresso found each of the errors. While Expresso is an excellent idea (and inspired our present research), it has several shortcomings. The hand-written parser is quite fragile and it would be very difficult to modify Expresso to add other identified errors. The error descriptions which Expresso returns to the programmer are single lines almost as cryptic as the error messages returned by the Java compiler. The research does not quantify the benefit that Expresso provides the novice programmer, nor does it provide any anecdotal evidence on the benefit of such a tool.

Flowers *et al.* describe Gauntlet, a tool which also identifies common errors made by novice programming students learning Java (Flowers, Carver & Jackson 2004). Gauntlet is integrated into an existing Interactive Development Environment (IDE) and identifies 9 common programming errors. Gauntlet's error reporting is multi-line, descriptive and provides possible remedies to the error. Flowers *et al.* designed the error reports in Gauntlet to be humorous not dry, and to "incite the student's competitive spirit". Students are encouraged to see Gauntlet as a challenge which only a worthy student can pass. Gauntlet's error reports are periodically altered to keep the system entertaining, and students can also contribute verbiage for error reports. After 18 months of Gauntlet use, Flowers *et al.* posit several benefits. By having simple mistakes identified and explained, students can recognise these mistakes; the incidence of these errors has been reduced, allowing students to focus on problem solving with computers. The

quality of the students' code has increased, allowing the teaching of material at a higher level. Gauntlet has seen a reduction in instructor workload as students require less assistance in solving common errors. However, Gauntlet still suffers from a fragile error detection mechanism, and the benefits to students are qualified, not quantified.

3 Arjen

We have developed a programming tool called Arjen. Like Expresso and Gauntlet, Arjen's purpose is to identify common mistakes made by novice programmers. In doing so, we have addressed the perceived deficiencies of the previous tools and constructed Arjen to support the long-term research objective. Arjen is written to target one specific programming language, Java, which is the first programming language taught at our university.

Arjen's architecture is best described by following the flow of information when a programmer invokes Arjen on their program. On this request, an Arjen client built into the programmer's IDE sends the program's source code, via a web session, to the centralised Arjen server. This client/server design was chosen to allow the collection of research data; the full Arjen functionality could be built into the programmer's IDE.

The Arjen server receives the program's source code and passes it to a number of plugin components. Each component analyses the source code for common errors, and returns back to the Arjen server a list of identifiers for the errors found in the source code. The server collates the errors from the plugin components and, using the error identifiers, looks up a description of the errors and possible solutions from a database. Using these error descriptions, the Arjen server returns its analysis of the programmer's source code to the Arjen client where it is displayed to the programmer.

3.1 Errors Detected by Arjen

Arjen has been primed with a list of 29 common programming errors. These were collected from the list of errors identified by Hristova *et al.* and Flowers *et al.* (Hristova et al. 2003, Flowers et al. 2004). We also ran the Java compiler across 10 years worth of student assignment submissions to identify other programming errors not in the above lists. Below is the list of errors recognised by Arjen; each error's title as shown to the programmer is given, along with an example of the error if possible.

1. Assignment in IF Statement: `if (y = 7)`
2. Use of Comparison After Boolean Operator:
   `if (age >= 13 && <= 17)`
3. Use of Bitwise Operators:
   `if ( (x & 0x1F) == 0x03)`
4. Cannot Find A Certain Identifier
5. Please use Braces not Parentheses
6. Cannot Treat char Like a String:
   `char ch = 'a'; if ( ch.equals('a'))`
7. Else Code Without a Matching If Statement
8. You have an Empty Statement:
   `for (int i=0; i<10; i++) ;`
9. Empty Statement After IF: `if (x == 7) ;`
10. System.exit() needs a Value: `System.exit();`
11. Missing Identifier: `public class { ...`
12. Probable Code in Wrong Place or Missing Braces, Parentheses
13. Probable Imbalance with Braces
14. Incomparable Types: `if ("X" == 7.3)`
15. Incompatible Types: `char ch = 7.3;`
16. Missing Left Brace '{'
17. Possible Loss of Precision: `int x = 7.3;`
18. Malformed FOR Loop: `for (i=0; i<10) ...`
19. Possible Misspelt Word or Command:
    `string str;`
20. Not a Statement: `int i; i == 0;`
21. Package Does Not Exist
22. Not Enough Closing Braces
23. Right Parenthesis Expected:
    `System.out.println("Your age is " age);`
24. Missing Semi-colon on a Line:
    `System.out.println("Hello world")`
25. Checking for String (in)equality:
    `String str="a"; if (str == "b")`
26. Missing Double Quotes on String Literal:
    `String str= "a;`
27. Unrequired Extra Type Keyword Used:
    `System.out.println("age: " + int age);`
28. Duplicate Variable in the Same Scope:
    `int x = 7; char ch; int x;`
29. Cannot Use Something Which Gives 'void' in an Expression:
    `int x = System.out.println(3 + 7);`

3.2 Current Arjen Plugins

Existing tools such as Expresso and Gauntlet provide only one mechanism to analyse source code to find common programming mistakes. During Arjen's development, we realised this limits the type of errors that can be found. For example, if the following line occurs in a Java program:

```
system.out.println("Hello world");
```

then any language parser, including the compiler, would indicate that the "system" class cannot be found. In reality, the programmer has misspelt the identifier "System". To ensure Arjen best reports the errors found, we built the Arjen server with a "plugin" architecture. This allows the input program to be analysed for errors by several mechanisms. At present, there are three available plugins.

The first Arjen plugin is a Yacc-based parser for the Java language which identifies violations of the Java grammar; examples include the use of brace symbols '{' and '}' where parentheses are expected, and malformed statements such as `for (int i=0; i<10)` The plugin also reports errors which are legal Java grammar but which have dubious style. For example, in the first programming course students are not taught bitwise operators such as '&' and '|'; their use often indicates that the programmer wanted to use the logical operators '&&' and '||'.

Originally, we intended this parser plugin to have full knowledge of the Java semantics including scope, variable

types and expression types (to identify errors such as narrowing or type mismatches). We realised this would require us to write a full Java compiler, minus the code generation section. Instead, we chose to write the next Arjen plugin.

The second Arjen plugin passes the program's source code to the existing Java compiler, which is run in a mode that reports as many warnings and errors as possible. The compiler attempts to compile the source code; its error report is parsed using regular expressions to create the list of errors for the Arjen server. This plugin gives the best static analysis of the source code but it also highlights the need for multiple analysis, as the compiler cannot report on stylistic errors.

The third Arjen plugin analyses a program's source code using simple regular expressions. It detects a few errors such as commonly misspelt words and phrases (e.g. `System.out.println`), and the use of comparison operators immediately following a boolean operator (e.g. `if (x > 7 && < 10)`).

Together, the three Arjen plugins detect and report on the 29 errors at present, but the system can be extended either with the detection of new errors in the existing plugins or with new plugins added to Arjen. During the pilot deployment of Arjen we logged any compiler errors and warnings that were unrecognised by Arjen and extended the Arjen compiler plugin to catch and report them.

3.3 An Example Error Report

Arjen's purpose is not simply to translate cryptic compiler error messages into a more human-readable format. Arjen also contextualises an error report with obvious details such as the line number and details such as the offending identifiers, keywords or operators. For most errors Arjen explains the most probable cause of the error and, if that is the cause of the error, a way to rectify the error.

Error descriptions are stored in HTML format in a database on the Arjen server; each description is in a templated format so contextual information can be embedded into the error report sent back to the programmer. Below is an example description of one error, the *bad_comparison_after_andor* error:

> On this line, you have the `%s` operator directly followed by the `%s` comparison operator, which is not legal in Java.
>
> The `&&` (AND) and `||` (OR) operators in Java join together two boolean expressions, i.e. they produce a true or false result. It looks like you don't have a complete boolean expression on this line after the `&&` or `||` operator. A good example of where this occurs is this:
>
> ```
> if (age >= 13 && <= 17)       // Wrong
>   System.out.println("teenager");
> ```
>
> After the `&&` operator there needs to be a proper boolean comparison. In this case, the code needs to compare the `age` variable against the value 17, i.e.
>
> ```
> if (age >= 13 && age <= 17)  // Right
>   System.out.println("teenager");
> ```
>
> Go back and check this line and make sure that the code after the `&&` or `||` operator is a proper boolean expression.

The two `%s` fields are filled in with information supplied by the plugin reporting the error.

As Arjen is to be used in first programming courses, the error descriptions attempt to relate each specific problem back to the essential programming concepts introduced in the course. In the above example the concept of a boolean expression is reiterated to the programmer, both as an aid for understanding the error and also to reinforce the course material.

4 Preliminary Results

The Arjen prototype has been deployed in a small class of students learning the Java programming language for the first time. The class size was small and students could choose to opt-in on the Arjen trial. Over the 13-week semester, 27 of the 35 students chose to use Arjen. The Arjen server logged the programs submitted for analysis along with a hashed version of each student's identifier. It also logged the errors found in each submission. At the end of the semester, students were asked to give feedback on their opinion of Arjen as a tool to help them with their programming tasks. With a small sample size of 27 students, these results are only indicative; a much larger study is required to answer the original research question.

During the semester, students had on- and off-campus access to Arjen for each of the three assignments (due in weeks 4, 9 and 12) as well as for the practical exam in week 13, but not for the mid-term and end-term written theory exams. While 27 students in total used Arjen, the tool was used by no more than 9 students each week. The following table shows Arjen's use over the semester with:

- the number of students who used Arjen each week,
- the number of Arjen invocations each week, and
- the number of errors reported by Arjen each week.

Week	Students	Invocations	Errors
1	6	20	0
2	6	16	0
3	9	26	35
4	9	34	209
5	6	8	11
6	3	3	10
7	2	3	2
8	7	10	19
9	6	23	64
10	2	3	7
11	0	0	0
12	3	10	543
13	4	4	16

Unsurprisingly, Arjen's use mirrors the dates for the practical work. Of the 29 errors Arjen can detect, only 18 error types were found in the programs submitted by students during the semester:

Error	Count over Semester
Empty Statement	679
Unknown Identifier	201
Missing Semicolon	80
String (in)Equality	55
Misspelt Word/Command	32
Missing Braces/Parens	29
Not a Statement	21
Missing Identifier	16
Right Parens Expected	14
Not Enough Closing Braces	11
Braces Imbalance	8
Assignment in IF	8
Incompatible Types	6
Else without Matching IF	6
Missing Double Quotes	3
System.exit() needs Value	3
Possible Loss of Precision	2
Duplicate Variable in Scope	1

Although the sample size is quite small, there are some surprises in these results. From teaching this course twice a year for a decade, we feel the incidence of duplicate variables in scope and loss of precision seems to be too low, and the number of empty statements is extremely high. On the other hand, our observations of other errors such as misspelt words, missing semicolons and the use of the == and != operators to test for String equality are always very high. In general, the ranking of the error counts measured by Arjen correlate well with our own teaching experience.

What we are quite pleased to see in these results is the number of errors reported by Arjen which are not found by the student's existing IDE environment: misspelt words and commands, the use of == and != operators to compare Java Strings, and the use of the assignment operator in an IF statement.

In the end-of-semester student evaluation, the students who used Arjen throughout the semester indicated that they found the tool very useful: Arjen helped them to find their errors, and it also helped the students to understand what caused the errors.

Overall, the results from the pilot study show a majority of students experimented with Arjen, but few students used Arjen regularly throughout the semester. When used, Arjen successfully identified many of the common mistakes made by first programming students with descriptive feedback returned to the students. Arjen also caught errors which had not been identified by the existing IDE.

5 Future Work

The development of Arjen is part of a much larger study into the effectiveness of such a tool to reduce the incidence of common programming mistakes. Before this can be undertaken, some deficiencies in Arjen need to be addressed.

Arjen is, at present, at the prototype/proof of concept stage of development. Due to time constraints, we wrote Arjen to receive and check only a single Java source code file at a time. This prevents Arjen from checking for errors which occur when a Java program is composed of several class files. The Arjen prototype needs to be rewritten so a complete, multi-file Java program can be analysed.

Different Arjen plugins can classify the same error in different ways. For example, a misspelt identifier will be identified by the regular expression plugin as a misspelling, and also by the compiler plugin as an unknown identifier. At present Arjen returns details of both to the programmer. We need to introduce a prioritisation scheme into Arjen so the best classification of an error is chosen, discarding the other reports. In this instance, only the misspelling report should be returned to the programmer.

Arjen has three analysis plugins which identify the majority of common programming mistakes. There is scope for other plugins which can identify other programming errors. For example, a plugin for code coverage analysis would be an ideal addition to Arjen. If Arjen was extended in this way, then it would be a useful tool to more advanced programmers as well as novice programmers. In this situation, Arjen would benefit from a configuration framework so it could be tailored for different programming courses. For example, Arjen could identify different groups of errors and return different types of reports for a second programming course on data structures as compared to a first programming course.

Apart from the plugins, Arjen is quite agnostic to the programming language. Arjen could be modified so that, when a program is received for analysis, it identifies the language (Java, C++, Python etc.), and sends the source code to the plugins relevant to the language. In this way, Arjen could become a framework for identifying common programming mistakes across multiple languages.

At present, Arjen has been implemented using a client/server architecture, collecting the raw data on student submissions and identified errors. In the long-term, Arjen should be built into the programmer's IDE to remove the dependency on network connectivity.

The pilot deployment of Arjen also shows the need to encourage students to use tools which will help them to learn. In the first few weeks of semester we recommended Arjen to the students, but we did not "push" the use of Arjen for fear of skewing the results we were collecting. We had hoped Arjen's obvious benefits would encourage all students to use it, but this was not the case. In the future longitudinal study, we believe that students should be given good documentation and training on all tools which can assist them to stay on top of the learning curve: the IDE, the debugger, the unit testing framework and Arjen.

6 Conclusion

We developed the Arjen tool for the same purpose as tools like Expresso and Gauntlet. Arjen extends this existing work in a number of ways. The descriptions of common programming errors are detailed and descriptive, relate the errors back to the core programming concepts being taught, and also offer useful and understandable advice on how a novice programmer can rectify the errors. Arjen provides a plugin architecture; this allows for multiple mechanisms to identify programming errors, and future work will prioritise the results so programmers receive the best description of each error. Arjen also logs the programs submitted, the errors identified and a hashed version of a student identifier; this allows the lecturer to quantify the errors being made by the novice programmers.

While a long-term longitudinal study is required to evaluate the effectiveness of tools such as Arjen to reduce the incidence of common programming mistakes, the preliminary results show Arjen effectively identifies these errors, and can quantify the incidence of these errors.

References

Flowers, T., Carver, C. & Jackson, J. (2004), Empowering students and building confidence in novice programmers through gauntlet, *in* '34th ASEE/IEEE Frontiers in Education Conference', IEEE, pp. T3H/10–T3H/13.

Hristova, M., Misra, A., Rutter, M. & Mercuri, R. (2003), Identifying and correcting java programming errors for introductory computer science students, *in* 'Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education', SIGCSE '03,

ACM, New York, NY, USA, pp. 153–156.
URL: *http://doi.acm.org/10.1145/611892.611956*

Jackson, J., Cobb, M. & Carver, C. (2005), Identifying Top Java Errors for Novice Programmers, pp. T4C–24–T4C–27.

Jadud, M. C. (2005), 'A first look at novice compilation behaviour using bluej', *Computer Science Education* **15**(1), 25–40.

Nienaltowski, M.-H., Pedroni, M. & Meyer, B. (2008), Compiler error messages: what can help novices?, *in* 'Proceedings of the 39th SIGCSE technical symposium on Computer science education', SIGCSE '08, ACM, New York, NY, USA, pp. 168–172.
URL: *http://doi.acm.org/10.1145/1352135.1352192*

Spohrer, J. C. & Soloway, E. (1986), 'Novice mistakes: are the folk wisdoms correct?', *Commun. ACM* **29**, 624–632.
URL: *http://doi.acm.org/10.1145/6138.6145*