Meaningful Categorisation of Novice Programmer Errors

Davin McCall, Michael Kölling
School of Computing
University of Kent
Canterbury, United Kingdom
{dm391,m.kolling}@kent.ac.uk

***Abstract*—The frequency of different kinds of error made by students learning to write computer programs has long been of interest to researchers and educators. In the past, various studies investigated this topic, usually by recording and analysing compiler error messages, and producing tables of relative frequencies of specific errors diagnostics produced by the compiler. In this paper, we improve on such prior studies by investigating actual logical errors in student code, as opposed to diagnostic messages produced by the compiler. The actual errors reported here are more precise, more detailed and more accurate than the diagnostic produced automatically.**

In order to present frequencies of actual errors, error categories were developed and validated, and student code captured at time of compilation failure was manually analysed by multiple researchers. The results show that error causes can be manually analysed by independent researchers with good reliability. The resulting table of error frequencies shows that prior work using diagnostic messages tended to group some distinct errors together in single categories, which can now be listed more accurately.

Keywords—programming, errors, novices, Java

I. Introduction And Prior Work

A. The Study of Student Errors

The study of errors made by students in their attempts to write programs during the early phases of learning to develop software has a long tradition. Various researchers (for example Jadud [5], Shinners-Kennedy and Fincher [9], and Marceau, Fisler and Krishnamurthi [8]) have investigated varying aspects of student errors, using different programming languages and environments.

The aim of investigating student errors usually falls into one of two categories:

- Some researchers aim at identifying the most common or most serious problems that students have when learning to program, including recognising difficult concepts [9] or misconceptions [10]. The ultimate aim is to improve teaching and learning through an awareness of the cognitive and practical challenges learners face.
- Another group of researchers investigate compiler error messages and students' reaction to encountering them [5,8]. This includes wording and presentation of errors, with the ultimate aim at improving our educational programming systems by producing better messages that provide improved help to students.

Both of these aspects still present interesting research challenges. There still is no real agreement about the most common problems students encounter, and how to address these in our teaching is an open question. Compiler error messages presented to students are still very obviously less helpful than they could be in most development systems used for programming education.

The work presented here aims at addressing both these areas for the Java programming language. Our work includes a method of identifying common student problems that differs from those presented in previous studies, and aims at ultimately improving both the understanding of problems students face in introductory Java programming, and the messages that our development tools produce.

In this paper, we present an analysis of the most common student errors, using a more reliable methodology than previous studies. This forms the basis of future work to improve compiler error messages. The result is also relevant for teachers and researchers wishing to improve programming pedagogy.

B. The Problem with Investigating Error Messages

Various studies [1,4,5] have investigated student errors in the past. Almost all of these studies used the compiler error messages as the data at the heart of their research, usually by recording and analysing error messages produced, and identifying frequent or persistent messages.

This approach presents a significant problem. The relationship of the compiler error message to the logical error is not strong enough.

For the purposes of more precise discussion we define the following terms, which we use in the remainder of this paper:

- The ***Error*** is the nature of the problem in the source code that causes the compilation or execution to fail.

- The ***Diagnostic Message*** is the human-readable message presented to the programmer when execution or compilation fails.
- The ***Programmer Misconception*** is the nature of the misunderstanding that resulted in the erroneous code being written.

For the aims of their work, researchers are usually interested in the error (and then, following this, the programmer misconception), but studies usually evaluate only the diagnostic messages. The reason for this is simple: recording the diagnostic messages can easily be automated, and analyses can be made at a large scale. Identifying the actual error that causes the diagnostic is more difficult, and cannot be automated.

However, this problem deserves to be taken more seriously than it previously has. Imprecision in the relationship between errors and diagnostic messages exists in both ways:

- A single error may, in different context, produce different diagnostic messages.
- The same diagnostic message may be produced by entirely different and distinct errors.

C. Mismatch of errors and diagnostic messages

It is easy to find examples of ambiguity of errors and diagnostic messages. Consider, for example, the following code fragment in Java:

```
for (int i=0, i<10, i++) {
}
```

This code contains an error: The student has typed a comma in the loop header where Java requires a semicolon. When compiled[1], the following diagnostic message is produced:

```
';' expected
```

Consider further the following code segment:

```
int number of points = 0;
```

This code results in exactly the same diagnostic message, even though the error is different (the student included spaces in a variable name, which Java does not allow).

For purposes of analysing student misconceptions or common problems, these are two very different errors which should be distinguished, yet counting of diagnostic messages will record them as the same.

The reverse problem – the same logical error producing different diagnostic messages – is also frequent. Consider:

```
JFileChooser.showOpenDialog;
```

and

```
if(n > myList.size - 1) ...
```

Both these fragments contain the same logical error (the student forgot to type parentheses after a method call), but the diagnostic message is different. The first fragment produces

```
not a statement
```

while the second causes a response of

```
size has private access in
java.util.ArrayList
```

This differences causes obvious problems for analyses of student difficulties. In fact, the situation is even worse than this: The exact same code (in the same context) will produce different error messages when compiled with different compilers (and sometimes even with different versions of the same compiler).

Which diagnostic message is displayed is typically decided by the compiler writer, and usually does not take pedagogical considerations into account. In fact, which message is produced often depends on the internal structure of the compiler, and is to a significant extent an artefact of its implementation details. This introduces a significant random factor into research based on diagnostic messages, and imposes a limited utility on such studies.

D. Identifying Errors

For the study presented here, we identify and analyse errors (as opposed to diagnostic messages). The potential benefits are clear – we can see the actual logical errors made by students, and speculate on programmer misconceptions – but the problems are also apparent: identification of errors cannot be easily automated, and thus involves manual classification. For each error, a researcher has to look at the source code and make a judgement about the actual cause of the error. This classification is more work intensive, so it is difficult to reach the same scale, and necessarily involves a degree of subjectivity.

It is not immediately clear whether actual errors can reliably be identified by researchers with sufficient accuracy and objectivity. This leads to the following research questions:

1. *Can a manageable set of error categories be developed to group student errors?*
2. *Can student errors be categorised objectively and reliably?*

If the answer to these two questions is 'yes' (and the categorisation is produced), then we can ask a third question:

3. *Which are the most common errors that beginning student make?*

In order to be useful, the number of different error categories used must be neither too small nor too large. This is expressed in question 1. If nearly every error produces its own category, the analysis would have no value, as is also true if too many errors fall into the same category. We consider a number of categories in the dozens (rather than single figures or hundreds) to be desirable.

A second question is whether a meaningful classification can be done that is sufficiently independent of individual researchers' judgement. If this is the case, the analysis of the data presents more useful results than analyses of diagnostic messages.

[1] For this paper, all diagnostic messages used as examples were produced with *javac*, version 1.7.0_45

[2] A full list of all categories defined can be found at

In this paper we present work to address all three of the above research questions. We present a set of error categories, a method to validate them for objectivity, and an analysis of a set of errors using these categories.

E. Providing Better Diagnostics

As a side effect of this study, when researchers classify errors they can also record the contextual information used when making a judgement about the actual cause. It can then be investigated whether (or in which cases) use of this contextual information can be automated, and whether this analysis can be used to ultimately produce better automated diagnostic messages.

This paper does not include such an analysis, but the work presented here forms the basis of this work in future.

II. METHOD

To perform the analysis described above, student error data was collected, error categories have been developed and checked for inter-coder reliability, student data was categorised and an analyses of this categorisation has been performed.

A. Data Collection

Two separate collections were performed. Both sets of data collection used the BlueJ IDE [6] to collect data from programmers using that environment. Thus, the data presented here is specific to the use of Java within BlueJ.

The first data collection exercise augmented BlueJ with an extension that captured snapshots of the students' source code at compilation times as they performed their coursework in a university course. Participants were students of two introductory Java programming courses, one of the University of Kent, UK, and one at the University of Puget Sound in Washington, USA, in autumn 2013. Participation in the data collection was voluntary. Overall about 240 students participated in the data collection. (Determining the exact number is difficult since the data was anonymised, and for technical reasons some participants may have received two anonymous ID numbers.)

Included in the collected data were details of each compilation event, consisting of a time stamp, outcome of the compilation (success, or error message) and the source code at the time of the event.

Overall, 37,432 compilation events were recorded, with 21,623 of these associated with a compilation error (the remainder being successful compilations). 197 of these were subsequently analysed.

This first data set was used for the first phase of the analysis, the category development (see section II-B).

A second data collection exercise was then performed to gather a larger data set for more large-scale and more varied (and therefore more reliable) analysis. For this collection, the Blackbox [3] project data was used. Blackbox is a data collection project initiated by the authors and others to support this work and other educational studies, by multiple researchers.

Blackbox provides educational programmer user data at a large scale by collecting the data from BlueJ users worldwide. All users of a recent version of BlueJ are asked to participate in this experiment as a routine part of using the environment, and participation is voluntary (through an opt-in mechanism).

Currently, approximately 250,000 users are participating in the Blackbox data gathering, and the Blackbox dataset contains information about more than 20 million compilation events.

For the study presented here, 23 sessions comprising a total of 136 events recorded between 2013-06-11 and 2014-01-01 were randomly selected from the Blackbox data for analysis. This dataset in conjunction with the first dataset forms the basis for the categorisation, described in section II-C.

B. Category Development

The first phase of the analysis is the development of categories for the categorisation of the error (as distinct from the diagnostic message). The method used for this is based on Thematic Analysis methodology [2]. Categories are developed and refined as familiarisation with the data set is improved by repeated reading of the data set.

A simple web-based application was developed that presents compilation failure events (students errors) to researchers, lets researchers inspect the event details (such as source code) and allows the classification of the error using a category system (including definition of new categories). The method is data driven, deriving the categories from the actual errors that were observed. This necessitated occasional need to revisit and re-categorise prior errors as the category system evolved.

The initial phase of this categorisation was performed by two researchers in collaboration until a suitable abstraction level for categories emerged, and then continued by a single researcher.

1) Category hierarchies

Error categories were allowed to form a hierarchy. Sometimes the cause of the error could be determined fairly precisely, for example "*method call: parameter number mismatch - call incorrect*" in a case where a method call is written with the wrong number of parameters, and it is apparent that the method call expression at the call site (and not the method definition) is incorrect. A more general error is "*method call: parameter number mismatch*", for cases when it cannot be decided whether it is the call site or the definition which is erroneous. These errors form a hierarchy, since one is a more specific variant of the other.

In each case, the most specific error category that could be determined with confidence by the researcher was assigned. For some analyses, sub-categories may be included in their parent categories.

Category hierarchies were developed naturally as the categories themselves were formed, during the familiarisation with the data set. If events were encountered for which an error category existed which closely matched but was either too specific or not specific enough, a new category was created and a hierarchy formed.

2) Validating the categories

Once the category system stabilised, the category system was validated. The premise of manual error categorisation is only useful if the identification of errors achieves a high degree of reliability and objectivity.

The validation of the categories was performed by calculating inter-coder reliability (the degree with which two researchers categorising the same set of error events independently agree on the category assignment).

To check inter-coder reliability, 150 error events were independently coded by three researchers, and a percent agreement calculation [7] was performed.

Since multiple categories could be assigned to a single event (see II-C.1), the calculation of pairwise agreement required handling of cases where multiple categories were assigned by one researcher, and another researcher or researchers assigned fewer categories. Pairwise agreement in these cases was counted according to the following method:

1. Let N be the number of researchers who have categorised an event
2. Let R*n* be the set of categories assigned to a particular event by researcher n (for each n in 1..N)
3. Let C be the vector containing all the elements {R_1, …, R*n*), ordered by size (smallest to largest).
4. For each element of C, as C*i*:
    a. For each further element of C, as C*j* (j > i)
        i. Count one agreement for each element of C*i* that also appears in C*j*, and one disagreement for each element of C*i* that does not appear in C*j*.

Use of this method effectively treats an event that has been multiply categorised by one or more researchers as multiple data points (each representing a logically distinct error) that have been categorised separately, and assumes that equivalent categorisations from multiple researchers are referring to the same data point.

C. Categorisation

Once error categories were defined and validated, 136 additional error events from the second data set were coded by a single researcher using these categories. The categories at this stage were mostly stable, with only a small number of new categories introduced to cover errors not seen in the first data set.

This categorisation, together with the categorisation performed on the first data set during the category development process, forms the basis of the analysis presented below.

1) Multiple errors in single events

In some instances, multiple errors were present in or near the same location in a single error event. An extreme – but real – example from the data illustrates this:

```
answer=this.getuserinput
```

This code includes four separate errors: the method name is mis-spelt, parentheses are missing after the method call, a semicolon is missing at the end of the statement, and the variable used in the assignment is undefined.

Finding four errors at an error event may be rare, but the occurrence of two concurrent errors is reasonably frequent. In these cases, we coded the event by counting an instance of an error for each category that was found (that is: registering all errors present, not just the single one that would have resulted from counting diagnostic messages).

2) Recurrence

Another situation that requires consistent handling during analysis is the recurrence of the same error multiple times, in a very short space of time. Sometimes, students recompile erroneous code without editing the segment that caused the error (either making other changes or, occasionally, not making any changes at all). Just compiling the same error multiple times should not count as multiple errors, as this would invalidate the error frequency count.

Three easily identifiable variations of the recurrence phenomenon were noticed during the initial category development (II-B). First, there were cases where the exact same source had been recompiled (either immediately, or with an intervening change or changes that were then reverted); secondly, there were cases where a minor edit was performed to the single line which was identified as erroneous by the diagnostic message produced by the compiler, which did not resolve the original error and which usually resulted in the same diagnostic message being produced again; finally, edits were sometimes made elsewhere in the source, rather than on the line identified as erroneous by the diagnostic message, which was left unchanged (leaving the error intact and causing the same diagnostic to be produced again).

Therefore, recurrence was defined as any of:

- Compilation of the exact same source file as had previously been compiled (at any stage in the session), resulting in the same diagnostic message being produced by the compiler
- Compilation where the only source line changed from the previously recorded event was the one identified by the diagnostic message as containing an error, and where the diagnostic message was the same as for the previous event.
- Compilation where the source line identified as containing an error for the previous event has not been changed, and for which the diagnostic message is the same as for the previous event.

Recurring error events were detected automatically and omitted from the analysis.

D. Data analysis

Once error categorization was performed, the following analyses were made on the data:

1) Frequency of errors

The frequency of errors was calculated by counting the relative occurrence of each category (from all data points included in the categorisation). A list of errors ranked by frequency was produced.

2) Number and frequency of diagnostic messages

The diagnostic messages associated with the categorised errors were collated and counted. This had to be performed manually, due to diagnostic messages often containing variable elements (such as user identifiers).

3) Coverage level of the most frequent error categories

The coverage (relative number of analysed events) of the top N most frequent error categories was measured. The coverage expresses what proportion of actual error events falls into the top N categories. This analysis is useful when devoting effort on improving pedagogical interventions or error diagnostics: It is useful to judge the actual impact of an intervention or a potential improvement of an error message.

This calculation was performed for N=5 through N=30 in increments of 5.

III. Results

A. Error Categories

A total of 80 error categories were identified during the category development phase of the research (described in section II-B). Broadly, errors can be divided into *syntactic*, *semantic*, and *logic* errors.

1) Syntax error categories

A variety of common syntax errors were identified. One of these, "semicolon missing", corresponds closely to the *javac* diagnostic message "';' expected". In general, however, the identified syntax error categories do not correspond well with any particular diagnostic messages that can be produced by javac. Examples include "keyword written incorrectly", "method call: missing comma between parameters", "method call: parameter types included", "method call: semicolon in place of comma", and "mismatched parentheses in or around expression". A general category, "invalid syntax", was created as a super-category for all syntax errors (and was used to categorize errors that did not fit into any of the identified subcategories)[2].

2) Semantic error categories

As well as syntax error categories, a broad range of semantic error categories were identified.

Semantic error categories showed a greater tendency to correspond, to some degree, with compiler diagnostic messages than did the syntax error categories. Examples include "type mismatch in assignment" (*javac*: "Incompatible types") and "variable not declared" (*javac*: "cannot find symbol – variable"). More precise sub-categories were identified in some cases – for instance, "variable name written incorrectly" is a sub-category of "variable not declared", applicable when it is apparent the programmer intended to refer to a particular variable but wrote the name incorrectly, due to misspelling or wrong capitalization (for example). The compiler's diagnostic message in these cases does not change, showing that the categorization scheme can achieve a higher precision in describing these errors.

Another example of the category scheme allowing for higher precision than the compiler diagnostic message presently does is for method calls with the wrong number of parameters. While javac does identify this error ("method xyz in class Abc cannot be applied to given types; … reason: actual and formal argument lists differ in length"), the compiler makes no distinction between incorrect method calls and declarations; a researcher performing categorization, on the other hand, may be able to use certain clues to allow such a distinction; for instance, with a call to library method that is part of the standard Java API, it can generally be assumed that the call must be at fault since the declaration is known to be correct.

B. Frequency of error categories

A total of 368 categorizations were applied to 333 events. The frequency of the top 10 error categories is shown in Table I[3].

The most frequently occurring error is "Variable not declared". This error alone accounts for 11.1% of all error instances.

C. Diagnostic message frequency

Each of the 333 compilation events analysed had an associated compiler diagnostic message. To judge whether the manual classification provides an improvement over classification of diagnostic messages, it is useful to compare Table I to the equivalent table if diagnostic messages are used for classification. The messages often incorporate variable, contextual text such as user identifiers (variable, method and class names); the general form of such messages was manually extrapolated. The frequency of the top 10 messages is shown in Table II.

TABLE I. Frequency of Error Categories

Category	Frequency
Variable not declared	11.1%
; missing	10.3%
Variable name written incorrectly	8.4%
Invalid Syntax	7.9%
Method name written incorrectly	4.9%
Missing parentheses for constructor call	4.1%
Unhandled exception	3.0%
Class name written incorrectly	2.7%
Method call: parameter type mismatch	2.4%
Type mismatch in assignment	2.4%

[2] A full list of all categories defined can be found at http://bluej.org/davmac/2014_novice_errors/categories.html

[3] A full table of error frequencies found is online at http://bluej.org/davmac/2014_novice_errors/frequencies.html

TABLE II. CATEGORY COVERAGE FOR TOP N CATEGORIES

N (number of categories)	Coverage level
5	44.1%
10	58.9%
15	66.4%
20	73.0%
25	78.7%
30	82.3%

D. Category coverage

Coverage levels for the top N categories were calculated for N=5, 10, 15, 20, 25 and 30. The coverage level represents the relative number of events that are categorised by at least one of top N categories. The results are shown in Table III.

The results show that, for instance, the top 10 error categories encompass nearly 59% of all error events that students encounter. Thus, a significant impact could be achieved if reporting or understanding of just these errors could be improved.

E. Intercoder reliability

After three separate researchers performed categorisation of the same error set, the average pairwise agreement (II-B.2) was calculated. The agreement across all events and all categories was 70.06%. This figure could be considered an indicator of a modest level of agreement. However, agreement is not uniformly distributed over all error categories. Agreement tended to be higher in more common categories, and lower in less frequent, more obscure cases. Since it is likely that future work (both pedagogical or in tool improvements) will concentrate on the most frequent errors, it is interesting to investigate agreement levels separately for this group.

For the top 10 error categories, pairwise agreement was 92.5%. For the top 20 categories, agreement is 87.4%. This indicated a good level of researcher agreement in classification of the errors.

The method used to calculate agreement is conservative, and higher agreement could be claimed using different methodology. To allow simplified calculation, pairwise agreement did not take the category hierarchy (see II-B.1) into account. A selection by one researcher of a category which was a subcategory of that selected by another researcher was counted as disagreement, whereas in fact this would be indicative of partial agreement. The impact of this is potentially quite significant. For the "Class name written incorrectly" category for example, all three researchers agreed on the category for exactly 50% of the events for which this category was selected. For the remaining 50%, two researchers chose this category and the third chose the "Class not defined" super-category. Thus, at the higher category level for this particular category, there is 100% agreement.

As a result, the agreement level reported above is a conservative lower bound.

TABLE III. DIAGNOSTIC MESSAGE FREQUENCIES

Diagnostic message form	Frequency
cannot find symbol - variable {variablename}	24.0%
';' expected	14.7%
cannot find symbol - method {method (…)}	8.7%
')' expected	5.4%
cannot find symbol - class {classname}	4.8%
illegal start of expression	3.6%
'(' or '[' expected	3.3%
{method(…)} in {class} cannot be applied to ({parameter types})	3.3%
unreported exception {classname}; must be caught or declared to be thrown	3.3%
reached end of file while parsing	2.4%

IV. DISCUSSION

A. Diagnostic Message Frequencies in Comparison to Other Studies

Jadud [5] published a distribution of errors made by novices, which was determined solely by examining diagnostic messages produced by javac. He finds the "missing semicolon" message to be the most frequent at 18%, with "unknown symbol – variable" next at 12%. "bracket expected", "illegal start of expression" and "unknown symbol: class" follow at 12%, 9% and 7% respectively.

Jackson, Cobb, and Carver [4] found a somewhat different distribution in their own study, also based on diagnostic messages: they list "cannot resolve symbol" (which possibly includes both "unknown variable" and "unknown class" errors) as the most frequent at 14.6%, with "; expected" next at 8.5%. "illegal start of expression" and ") expected" also feature in the top 6 most frequent diagnostic messages, with "class or interface expected" and "<identifier> expected" both appearing in the top 5 (they appear in Jadud's distribution at position 8 and 9 respectively).

Although the frequencies vary, there are several diagnostics that appear in both the Jadud and Jackson-Cobb-Carver top 10 lists. The diagnostic message frequencies in this paper show a relatively high incidence of "cannot find symbol", "';' expected", "')' expected" and "illegal start of expression" diagnostics, similar to the other two studies, showing at least a moderate level of consistency between the three studies.

There are several factors that could explain some of the differences in the observed distributions across the three studies: different compiler versions may have been used, and different cohorts participated. Neither Jadud nor Jackson et al describe any attempt to eliminate recurrent errors from their results as this paper does (II-C.2).

B. Error categories compared to diagnostic messages

The top 8 categories (see Table I) cover just over 50% of events; in contrast, the top 4 diagnostic messages cover roughly the same amount (52.8%). This fact alone demonstrates that the categories described here are in general more precise than the compiler's diagnostics. While each category describes an error that will cause a diagnostic message to be generated by javac, the diagnostic message does not in general reflect the same precision as the category (see examples in III-A.2).

There are also categories for which the associated diagnostic messages are not just imprecise, but are inaccurate. The "keyword written incorrectly" category, for instance,

corresponds to the following diagnostic messages in the data set used for analysis:

- cannot find symbol - variable flase
- cannot find symbol - variable True

These occurred due to misspelling and incorrect capitalization of the literals ‘false’ and ‘true’, respectively. The compiler message, however, refers to the lack of definition of a variable. While it is true that no variable defined as ‘flase’ or ‘True’ existed, it is not the case that a variable reference was being attempted here.

C. *Coverage level of error categories*

The category coverage analysis shows that a small proportion of the categories cover a significant portion of errors (Table III). The top 10 categories cover well over 50% of errors; as more categories are added, the coverage increases, but the increase diminishes as N increases (where N is the number of categories). This implies that pedagogical interventions (or error diagnostic tools) would be effective by focusing on a select number of the most frequent types of error.

D. *Risks and limits to validity*

Several factors impose risks and limits to the interpretation of this data.

First, all data was collected using the BlueJ IDE, consisting of programs in the Java programming language. Thus, all results are restricted to this programming language, and may be influenced by the choice of programming environment.

Data from the first collection exercise was obtained from a relatively narrow set of students performing a relatively small set of programming exercises (two university courses). This data could be biased by student background and by the type of exercises undertaken.

An attempt was made to use data more widely spread in student demographics for the second collection, by using subjects from the general BlueJ user population, but no information about student background or characteristics in known for this data set.

The number of error events categorised is still relatively low. While they appear to be large enough for results to have stabilised, and thus seem suitable for this type of analysis, categorising a larger number of errors to increase the confidence level might be desirable.

V. Future work

While the manual categorisation and frequency analysis of errors is in itself a useful result with the potential to guide future pedagogical interventions, it can also serve as the basis for improving educational programming environments.

The goal of the next phases of the authors' work is to improve automatically generated diagnostic messages. This work involves four distinct phases:

- The severity measure of an error will be defined, and the severity of every error category will be calculated. In this paper, only the frequency of the error has been presented. The severity of the error is a combination of frequency and difficulty, measured in time to solve the error. If an error is frequent, but easily fixed by beginners, it is less severe than errors that are frequent but present more persistent hurdles.
- When researchers categorised the errors, they made use of a variety of contextual information, usually found elsewhere in the source code. For example, when an "Unknown identifier" diagnostic was encountered, the cause of the error could often be determined more specifically, by examining whether a similarly spelt identifier exists. We will record and analyse the contextual information used by researchers for this purpose.
- We will investigate whether the use of contextual information as done by human researchers can be automated in some cases to improve the presentation of error diagnostics.
- We hope to identify some errors where this improvement can be automated, and where the error ranking is relatively severe. For those errors, we hope to implement a system that produces clearly improved error diagnostics.

Both the availability of error frequencies and the process of manual evaluation, as presented in this paper, are necessary for this further analysis.

VI. Conclusion

Determining the relative frequency of errors students make when programming in a particular programming system has long known to be useful. We improved on previous methodologies by analysing logical errors instead of diagnostic messages. This leads to a more reliable ranking of student error frequencies.

The results show that human analysis can quite reliably identify the source of a student error more precisely than typical compilers currently do. This identification is reliable across individual researchers for the most frequent errors.

There is some variation of our results compared to previously published studies. The data indicates that the most frequent errors are syntactical and their cause can often be determined reasonably precisely, which gives rise to some hope that the production of automated diagnostic messages could be improved.

Acknowledgements

The authors would like to thank Neil Brown and Michael Berry from the University of Kent for their time spent in categorising errors, for the purpose of category validation.

References

[1] Ahmadzadeh, M., Elliman, D. and Higgins, C., 2005. An Analysis of Patterns of Debugging Among Novice Computer Science Students. *ACM SIGCSE Bulletin* 37, 3 (2005) 84-88.

[2] Braun, V., and Clarke, V., 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (2006) 77-101.

[3] Brown, N. C. C., Kölling, M., McCall, D and Utting, I., 2014. Blackbox: A Large Scale Repository of Novice Programmers' Activity. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)* 223-228.

[4] Jackson, J., Cobb, M., and Carver, C., 2005. Identifying Top Java Errors for Novice Programmers. *Proceedings of the 35th Annual Conference Frontiers in Education (FIE '05)*. (Oct. 2005) T4C-24 – T4C-27.

[5] Jadud, M. C, 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (Mar. 2005), 25-40.

[6] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003) The BlueJ system and its pedagogy. Computer Science Education, 13 (4). pp. 249-268. ISSN 0899-3408.

[7] Lombard, M., Snyder-Duch, J., and Bracken, C. C., 2006. Content Analysis in Mass Communication: Assessment and Reporting of Intercoder Reliability. *Human Communication Research* 28, 4 (Jan. 2006), 587-604.

[8] Marceau, G., Fisler, and K., Krishnamurthi, S., 2011. Mind Your Language: On Novices' Interactions with Error Messages. *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (ONWARD '11)* 3-18.

[9] Shinners-Kennedy, D. and Fincher, S., 2013. Identifying Threshold Concepts: From Dead End to a New Direction. *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)* 9-18.

[10] Sorva, J., 2013. Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education (TOCE)* 13, 2 (Jun. 2013) 8:1-8:3.