A New Look at Novice Programmer Errors

DAVIN MCCALL, King's College London
MICHAEL KÖLLING, King's College London

The types of programming errors that novice programmers make and struggle to resolve have long been of interest to researchers. Various past studies have analyzed the frequency of compiler diagnostic messages. This information, however, does not have a direct correlation to the types of errors students make, due to the inaccuracy and imprecision of diagnostic messages. Furthermore, few attempts have been made to determine the severity of different kinds of errors in terms other than frequency of occurrence. Previously, we developed a method for meaningful categorization of errors, and produced a frequency distribution of these error categories; in this paper, we extend the previous method to also make a determination of error difficulty, in order to give a better measurement of the overall severity of different kinds of errors.

An error category hierarchy was developed and validated, and errors in snapshots of students source code were categorized accordingly. The result is a frequency table of logical error categories rather than diagnostic messages. Resolution time for each of the analyzed errors was calculated, and the average resolution time for each category of error was determined; this defines an error difficulty score. The combination of frequency and difficulty allow us to identify the types of error that are most problematic for novice programmers.

The results show that ranking errors by severity—a product of frequency and difficulty—yields a significantly different ordering than ranking them by frequency alone, indicating that error frequency by itself may not be a suitable indicator for which errors are actually the most problematic for students.

CCS Concepts: • **Social and professional topics → Computing education**.

Additional Key Words and Phrases: programming errors, novice programmers, Java

ACM Reference Format:
Davin McCall and Michael Kölling. 2010. A New Look at Novice Programmer Errors. *ACM Trans. Comput. Educ.* 9, 4, Article 39 (March 2010), 31 pages. https://doi.org/0000001.0000001

1 INTRODUCTION AND PRIOR WORK

1.1 The Study of Student Errors

The study of errors made by student programmers is of considerable interest to researchers in the field of Computing Education; an understanding of the errors that students tend to encounter, and how they deal with them, is useful in tailoring pedagogy and instructional programming tools.

Recent research [5] has indicated that educators may not have a good grasp on which errors students encounter most frequently. Earlier work [23] examines the belief that misunderstandings or incorrect uses of language constructs cause the majority of student programming errors (in compilable code), and find it unwarranted. It is therefore clear that it is necessary to use hard data for ascertaining frequencies for different types of error.

Some researchers have looked into the most common or serious problems that students have when learning how to program, identifying problematic concepts [21] or common misconceptions

Authors' addresses: Davin McCall, King's College London; Michael Kölling, King's College London.

1946-6226/2010/3-ART39 $15.00
https://doi.org/0000001.0000001

[22] with the aim of improving teaching and learning through a better understanding of the cognitive and conceptual difficulties that students face. Others have looked at student reactions on encountering an error message [16, 19], with the goal of improving our educational programming systems to provide better assistance to students.

Errors may or may not cause compilation failure (if they do not, they manifest as incorrect or unwanted program behaviour at run-time)—and the techniques for examining the two kinds of errors necessarily differ significantly. Ko and Myers [14] proposed a framework for studying the causes of software errors which focused on runtime issues (with studies around Alice [7], an environment that largely precludes syntax errors).

While run-time problems are important, a number of studies focus mainly on compile-time errors. Novice programmers may struggle with both kinds of errors, but compile-time errors are in a sense the real "show stoppers"–a program that does not compile does not even begin to give the student any meaningful result for their efforts. Many studies have been conducted on the nature of compile-time errors made by students in the early phases of learning to write computer programs, that collated statistics on the frequency with which various diagnostic messages were encountered [1, 12, 13, 24]. Note that we have preferred the term "diagnostic message" over "compiler error", commonly used elsewhere, since it is less ambiguous especially in the context of this study.

Watson, Li and Godwin [24] used error frequency together with an estimate on time spent working on errors in an attempt to predict student performance (via the "Watwin" algorithm), with some success. They classified errors via the compiler-produced diagnostic message and estimated time spent by measuring time between related compilation-event pairings or the first compilation event in a pair and an invocation, in cases where an invocation was performed before the subsequent compilation. The frequency and time spent on each type of error was not reported.

Denny, Luxton-Reilly and Tempero [9] looked at the frequency and time of different syntax errors, classified via the diagnostic message, by students in attempts to solve exercises. Attempts were submitted via a web-based drill-and-practice tool and errors were considered fixed when the diagnostic was no longer produced upon a subsequent resubmission.

Becker and Mooney [3] counted frequency of errors categorized by the diagnostic message, and used principal component analysis as a means of grouping related diagnostic messages, effectively forming error categories distinct from (although still dependent on) the messages.

In an earlier part of this work [20] we identified problems with the approach of using diagnostic messages to categorize student errors: they can be inaccurate, they can be imprecise, and they can vary between toolsets (different compilers produce different messages when processing the same erroneous source code). Even using the same compiler, different logical errors frequently result in the same diagnostic message, and the same logical error can—depending on context—produce a different message. Therefore, the results of frequency analysis of diagnostic messages are as much a product of details of the implementation of the compiler as of student behavior.

1.2 Manual Classification of Errors

Our previous study [20] measuring the frequency of logical errors relied on the development of error categories and manual classification of error events from the dataset into these categories, using a methodology based on Thematic Analysis as outlined by Braun and Clarke [4]. The category scheme and the categorizations were validated by checking pairwise agreement between a number of coders with experience in teaching Java programming.

Manual classification overcomes several problems with classification by using the diagnostic message produced by the compiler:

- The conceptual nature of the error can be better assessed; diagnostic messages often tend to focus on technical details such as mismatch between the input source and the expected grammatical structure.
- A greater level of accuracy can be achieved. Compiler diagnostic messages can be misleading, sometimes suggesting or implying a correction that in fact does not resolve the problem satisfactorily; an experienced human instructor will use extended context to determine a more appropriate description of the problem.
- Similarly, a greater level of precision can be achieved. Compiler diagnostics often do not make distinctions between several different kinds of errors. For example, a misspelled variable name and a failure to declare a variable are conceptually different errors, but the standard Java compiler produces the same message in both cases.

A recent study [8] claimed that for more than 20% of errors, the diagnostic message produced by the Java compiler was not sufficient to correctly identify the associated error. For the reasons outlined above, manual classification of errors yields more useful results, even though it is labor intensive—and very slow—when compared to automatic classification using the diagnostic message.

1.3 Error Severity

Meaningful categorization and frequency analysis of logical error types is not enough to identify the errors that cause the most problems for students. Frequency is one part of the equation; the other part is the *difficulty* of resolving the error. The severity of a particular type of error can be considered as a product of the frequency and difficulty of that type of error:

$$severity = frequency \times difficulty$$

Formulated in this manner, the severity of a particular type of error is a measure of how much effort students can be expected to expend on resolving that type of error. This leads to the following research question:

RQ1. Are some categories of errors distinctly more severe than others?

If the answer is *yes*, then we can ask the follow-up question:

RQ2. What errors or error types are the most severe?

While the *frequency* of an error type is easy to determine, determining the *difficulty* that it presents to students is more complicated. Given a data set that contains a series of compilation events, an obvious indicator of the difficulty of a particular error is its *time-to-fix*, that is, the time between the error first being encountered and the first compilation afterwards in which the error is corrected. Other research [24] has shown that time spent fixing errors is a factor related to overall student performance; it is logical that more time spent on solving particular types of error indicates that those errors are more problematic.

A concern with using time-to-fix as part of a metric is that it is difficult to measure. A first approximation is the time between the observation of the error and the next successful compilation. Several sequences of events are, however, possible in which the time between the occurrence of an error and the first subsequent successful compilation does not accurately reflect the time in which an error is resolved. If an error occurs in student code, the student may simply remove the erroneous code (perhaps by reverting the code to the last "known good" state, i.e. the state at which the code was previously compiled without error). This should not be counted as resolving the error; presumably, the change that introduced the error was meant to be functional, and the purpose of the reversion is so the student can try again to make the same functional change, starting from a point at which they know the code is at least syntactically correct.

Furthermore, if an error occurs and a subsequent compilation also fails, it is possible that the error was fixed; the compilation failure may be caused by a different error also present in the code.

In this paper, we present the results of an automated measurement of the time-to-fix of various types of error, where the errors are categorized using a method refined from our previous work [20], and use error frequency and time-to-fix statistics to estimate the severity of different classes of error.

1.4 Improving Compilation Diagnostics for Students

The work presented here required the manual categorization of a large number of programming errors by human experts. Any reasoning by those involved in categorization was recorded along with their choices. A future analysis can potentially investigate the possibility of improving automated error diagnosis by implementing the same reasoning, where possible, in a tool or programming environment for use by students. The error severity statistics presented in this paper should help to guide the focus of such developments, which are planned as future work.

2 METHOD

To perform the analysis of error frequency and severity, data from the Blackbox Project [6] was used. A random sample of coding sessions were extracted and stored in a separate database, and category development and categorization was performed using a web-based tool developed to support this work. Inter-coder reliability of the categorization was assessed, and the categorization and error severity were then analyzed.

2.1 Data Collection

Blackbox [6] 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 worldwide users of *BlueJ* [17], a development environment designed for students and used in a large number of institutions worldwide. Users of BlueJ are prompted to opt in to the data collection, if they so wish, when they start the software for the first time.

The collected data includes information about a variety of user actions including compilation. The events recorded are time-stamped and associated with a session. Source code comments go through an automated redaction process to preserve anonymity of the student (who is identified in the data only by a generated numeric identifier).

Source files and edit actions are recorded in the Blackbox database, and the complete project source code at any recorded event time can be reconstructed. At this time approximately 1.8 million users have participated in the Blackbox data gathering and the dataset contains more than 179 million compilation events.

For the study presented here, 199 user sessions containing a total of (just over) 1000 compilation events with errors, recorded in the period from 2013-06-11 to 2014-05-30, were randomly selected from the total Blackbox data set for analysis. Each of the sessions was associated with a different user (as identified by the unique identifier recorded in the Blackbox data). This number of events was considered as a reasonable target given the time required for the manual categorization.

2.2 Category Development

The first phase of the analysis was the development of the error categories that will then be used to categorize the errors in the dataset. The categories were developed using an iterative process involving continuous refinement during repeated reading of the data set, using a method based on Thematic Analysis [4]. Because our prior work [20] also developed a category scheme, this

was used as the basis for the scheme developed in the work presented here; the category scheme developed was a refinement of the scheme used in the earlier work. The web-based tool developed for the earlier work was used to browse the dataset (and later perform category selection).

2.2.1 Category Hierarchy. The error categories were organized into an explicit hierarchy, where more precise categories constitute the subcategories of a less precise error. For example, the category of "incorrect attempt to use variable" was given the more precise subcategories "variable not declared" and "variable name misspelled" (amongst others). The subcategories are more specific, but the higher-level category is still useful for cases that cannot be classified into any of the more precise categories.

The purpose of the hierarchy was to allow for analysis of the frequency of errors at the broader category level, and to ease the process of category selection for the individual error events.

The hierarchy was developed simultaneously with the development of the categories themselves. In cases where a more specific variant of an existing category was identified, it was created as a subcategory of the original; in cases where two or more categories had some notable similarity, a parent category was created. Once the category development was complete, a minor re-organization of the hierarchy was performed to make it more suitable for use in manual category selection, and the number of top-level categories was reduced to a small set so that a rapid selection of top-level category could be made.

2.2.2 Validation of Category Scheme. Once the category scheme was stable and sufficient to cover the majority of errors observed in the data set, it was validated in order to verify that it could be used to identify errors with a high degree of reliability and objectivity.

223 (22.3%) of the compilation events in the dataset were categorized according to the category scheme by multiple coders from a pool of 5 researchers with experience in Java programming (see section 2.3 for details of the categorization process). The participating researchers were assigned a starting point event and requested to categorize errors from that event on, processing as many events as they could in the period of time they were able to allocate to the task.

Participants were given written instructions on the use of a web-based interface used for category selection. These instructions asked the participants to:

- Favor accuracy and precision over speed;
- Focus on the error (or errors) at the site of compilation failure as identified in the compiler diagnostic;
- Choose a single category for each individual error (if the participant believed that were multiple errors present at the same location, they should make multiple category selections, one for each error); and
- Provide a short explanation of why they made the category selections that they did, in terms of contextual clues within the source.

Once the initial categorizations had been performed, the pairwise agreement between coders was calculated by pairing the categorizations from different coders on the same event and counting the total agreeing categorization pairs as a proportion of the total pairs[1]. The category selection process allows the selection of more than one error for an event; for such cases, the pairwise agreement/disagreement ratio was determined by applying Algorithm 1 to each recorded event, as per our previous work [20], and summing the agreement and disagreement counts from all events.

[1]We found that the use of Krippendorf's Alpha, an established statistical measure of agreement, was not possible given the nature of our data and labelling process. The calculation of a Krippendorf's Alpha value requires that the data be composed of units which are each singly-coded by any number of coders. Instead, our data can be decomposed into units which are *multiply* coded by coders. The issue is that there is no obvious way to suitably pair individual codings from different coders so that pairwise agreement/disagreement counts suitable for use in the standard calculation of the alpha. Simply

ALGORITHM 1: Pairwise agreement determination (original)

Input: The number of researchers who have categorised the event (N); the categories assigned to the event by each researcher (R*n*, for *n* in 1..N).
Output: The number of agreements and disagreements to be counted for the event.
C = the vector $\{R_1 .. R_N\}$ with elements ordered by size, smallest to largest;
for *each element* R_i *in* C **do**
  for *each further element* R_j $(j > i)$ *in* C **do**
    Count one agreement for each categorisation in R_i that occurs also in R_j;
    Count one disagreement for each categorisation in R_i not also occurring in R_j;
  end
end

The initial level of agreement measured in this way was, although reasonable considering the nature of the task, not as high as was expected when compared to our previous work [20]. After examining the category selections and reasoning provided by the participating researchers, it was observed that in many cases one researcher had used reasoning based on context that another researcher had ignored, perhaps because they had not noticed it or missed its significance. Except in one case, the researchers participating in the categorization process had volunteered their time; given the menial nature of the task, it was tempting to neglect to check for details in other parts of the source code that might lead to a different error categorization (rather than, for example, choosing a category which closely matched the diagnostic message from the compiler). It was decided that a second phase of selection, where researchers had access to the reasoning used by others, would be beneficial.

In the second phase of categorization, each of the participating researchers reviewed the selections they had made in the first phase. For each error event where two or more researchers had made selections that did not agree, these researchers were presented with the alternative selection(s) together with the reasoning that had been provided by the other researchers when making the conflicting category selections. Each researcher was then given the option, on an event-by-event basis, of either changing their selection to match one of the other selections, or to keep their original selection. This review process was conducted using a modified version of the web interface used for performing the initial selections. Researchers were not made aware of the identity of the other researchers who had made conflicting selections (they were shown only the category selections made, and the reasoning given).

The algorithm outlined above for counting pairwise agreement for events that are categorized with multiple errors by one or more researchers is not suitable for calculating the pairwise agreement after the second phase of categorization, because it makes an assumption that, if one researcher has selected more errors than another, it is because the latter researcher neglected to categorize the excess errors (and not because they necessarily disagree that the excess errors were present). This assumption is invalid if the latter researcher has seen the selections made by the former together with their reasoning; if they did not at that point decide to alter their own selection to

treating multiply-coded units as separate units is not viable, since it is not clear how to assign each coding within the original unit to the separated units, and the value would be sensitive to any particular selection in that regard. Neither can Cohen's Kappa, another well-established coefficient of agreement, be applied, for the same reason (and also because it requires the same number of coders per unit, which is not the case in our data set). While the ratio of pairwise agreement to disagreement, as calculated by the algorithm we have presented, does not differentiate coincidental agreement, the distribution of label frequencies as reported in section 3.2 suggests that coincidental agreement levels should be low and that the pairwise agreement ratio, at the reported levels, is therefore a meaningful indicator of good agreement.

match, it may be because they disagree about the presence of the excess errors. For this reason the algorithm was modified, by changing the ordering of the vector C to largest-to-smallest (rather than smallest-to-largest). This change causes excess categorisations of one researcher compared to another to be counted as a disagreement (previously, these categorisations did not count as either an agreement or as a disagreement).

The practice of allowing, in a study with multiple coders, a second coding phase where prior results are visible to coders is unusual. In this case, it is justified since we are asking for a professional, objective opinion; we did not expect coders to change the coding except in cases where the reasoning behind alternative categorizations that became available to them was compelling.

As expected, the second phase of categorization yielded an improved agreement level, even when using the modified, more "pessimistic", agreement counting algorithm. The agreement levels are reported in section 3.5.

2.3 Categorization

Once the category hierarchy was finalized, a single researcher proceeded to categorize errors until they had categorized 1000 events with one or more of the error categories. The 1000 events included all those also categorized by participants in the category validation process. The categorizations made by this single researcher form the basis of the analysis on error frequency and severity presented in this paper.

2.3.1 Web-based Interface for Categorization. A web-based interface was developed for categorization of error events (and for editing the category hierarchy). This interface divides the events by programmer (identified only by a unique, randomly-generated number) and lists events according to time of occurrence. Selecting an event shows the source code surrounding the error location and the diagnostic message issued by the compiler. It also offers links to view the full source code for any file in the project at the time of the event.

The categories available for selection are displayed as part of the interface in a tree with collapsible branches (initially collapsed so that only the top-level categories are immediately visible). The tree view control allows for multiple selections to be made simultaneously. Reasoning for the selection(s) made can be entered as free-form text. Navigation buttons allow to save selections and go to the next or previous event.

2.3.2 Recurring Errors. In our previous work [20] we noted a tendency for the same or similar error to occur two or more times in succession (or within a short span of time) within a programming session, with little or no changes made to the source code. One possible reason for this is a student pressing the “compile” button again after encountering an error; another might be due to the student undoing and then re-doing a change that introduced an error (with intervening compilation events). We refer to these and similar instances as “recurring errors”.

Recurring errors should not be counted twice when counting error frequency, since it is not really the case that the same error has been made twice. For the purpose of automatically filtering these errors, we used the formal definition of recurrence from our previous work [20], where it 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 generated for the altered source refers to that same line (though the message may be different).

- 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 filtered automatically from the data set before categorization commenced.

2.3.3 Multiple Errors in Single Events. Multiple errors can be associated with a single error event. Consider the following example:

answer=this.getuserinput

This single line, taken from the dataset, contains no less than four logically distinct errors: the method name was mis-capitalized (it should be "getUserInput" rather than "getuserinput"); parentheses to denote a method call are missing; a required semi-colon is missing at the end of the line; and finally, the variable "answer" to which the statement refers was not defined.

For the purpose of error categorization the researchers were instructed to categorize all errors associated with the source code location identified in the compiler diagnostic, although not necessarily in close proximity to that location; for example, an attempt to call a method which is marked by the compiler as erroneous might instead be judged to be an error in the method's declaration such that the declaration and call do not properly match.

2.4 Data Analysis

Once error categorization was performed, the data was analyzed as follows:

2.4.1 Frequency of Errors. The frequency of errors was calculated by counting the relative occurrence of each category (from the categorizations made by a single researcher). A list of errors ranked by frequency was produced.

2.4.2 Coverage Level of the Most Frequent Error Categories. The coverage (relative number of covered 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.

2.4.3 Error Difficulty. In section 1.3 the concept of error severity as a product of frequency and difficulty was introduced. While counting occurrences to determine frequency is straight forward, a suitable metric for estimating difficulty is required. In the analysis presented in this paper, adjusted average resolution time (explained below) was used as the metric for difficulty—an error that is difficult to resolve should generally take more time to fix than an error that is easy to resolve. In addition, errors remaining unresolved were taken into account. A time-to-fix estimate for different errors was made using an automated analysis run on the categorized errors in the data set. For any given error event, the resolution status was classified by this analysis into one of the following categories:

- *Reverted* — the error had been removed by changing the code back to the state it was in before the error was introduced;
- *Resolved* — changes made in a later event removed the error from the source code (without simply reverting the source code to the state prior to the introduction of the error);
- *Unresolved* — the error remained present in subsequent recorded events, up to the point that the programmer ended their session or began working on a different area in the code;

- *Uncertain* — The resolution status was not established as either *Resolved* or *Reverted*, but it was not possible to ascertain the continued presence of the error in later events.

For each user session in which an error had been categorized, the resolution classification procedure examined recorded compilation events in turn, beginning at the earliest categorized error event. The algorithm used for classification relied on the previous identification of recurring errors (2.3.2), and is detailed in Appendix A.

The time between the origin event of a resolved error and its resolution event, capped to a maximum of 5 minutes, was used as the estimate of time-to-fix for the error. The 5 minute cut-off is somewhat arbitrary, but it is necessary to choose some limit to resolve the problem of outliers, which can be extreme. Consider the hypothetical but realistic example of a student who gets stuck on an error at the end of a lab session, who then moves on to another class, before finally returning to the problem on the next day. In such a case, the resolution time exposed via the data would include many hours not actually spent in attempting to resolve the error, and would artificially inflate the mean resolution time for the relevant error category. Less than 4.7% of individual categorised errors had resolution times affected by clamping with the chosen cut-off of 5 minutes.

Another study [9] also measuring time-to-fix of errors ignored resolution times of longer than 5 minutes, with similar reasoning to ours for limiting them to a maximum. Ignoring longer times rather than limiting them may reduce artificial inflation of average times but also risks losing information about particular errors that were particularly problematic for the novice to resolve.

For each error category, the time-to-fix of each resolved instance was summed and recorded as the total resolution time for the category. Also recorded for each category were the count of reverted instances and instances where resolution was uncertain. An *adjusted average resolution time* was calculated by using the maximum resolution time of 5 minutes for *unresolved* and *reverted* error instances, summed with the time-to-fix for *resolved* instances and divided by the total (excluding instances with *uncertain* resolution status). The difficulty value of an error category was taken as the adjusted average resolution time.

The algorithm to classify resolution status of errors, as outlined above, accounts to an extent for the reversion behavior which is sometimes observed in the data set and which with a more naïve approach would be interpreted as error resolution. Although the results cannot be completely accurate, they give an approximate indication of the relative difficulty of different kinds of errors.

2.4.4 Severity. Finally, after calculating frequency and difficulty, these two measures were combined into a single value, severity. Severity was calculated by taking the product of the error category frequency and the adjusted average resolution time in seconds. The frequency is expressed as a ratio of the number of events in which an error belonging to the category was identified to the total number of events containing errors (1000). Because the adjusted average resolution time is limited to 5 minutes, the theoretical maximum severity of an error category is 300.

Calculating a single severity value is useful to arrive at a linear ranking, approximating an answer to the frequent question "Which parts of a programming language do students struggle with most?". However, collapsing these two measures into a single value also loses information: In some cases considering the relationship between frequency and difficulty can give a deeper insight. For this reason, we created error graphs, depicting the outcome of this analysis on a two-dimensional graph for every error category.

3 RESULTS

3.1 Error Categories

A total of 90 error categories were identified during the category development phase described in section 2.2. In comparison to our previous work [20] the categories were formally organized

into a hierarchy and during category selection were presented in a tree representing this hierarchy. Broadly speaking, errors can be divided into three overall categories—syntactical, semantic and logical; these three overall categories could have been used as the top-level categories in the complete category hierarchy. However, a finer-grained selection is more useful even at the high level of categorization to allow for better analysis of problem areas for the novices, and to simplify the process of category selection by limiting the tree depth (which reduces the amount of navigation required to locate the suitable low-level category). For this reason, the top-level categories were chosen as follows:

- Variable: Incorrect attempt to use variable
- Variable: Incorrect variable declaration
- Method: Incorrect method call
- Method: Incorrect method declaration
- Constructor: Incorrect constructor call
- Constructor: Incorrect constructor declaration
- Incorrect attempted use of class or type
- Semantic error
- Simple syntactical error
- Statement outside method/block
- Uncategorized

Problems involving variables, method calls and constructors were thus distinguished at a high level and were separated at the same level between problems of *use* and *declaration*. Problems involving use of types were assigned another category. Errors not involving any of these constructs could be divided amongst the high-level divisions of *semantic error* or *simple syntactical error*.

Errors of a logical nature were not observed in the data set and so no category was created for such errors. A possible explanation for lack of logical errors is that such errors might not produce a compilation failure (instead causing program malfunction at execution time). Logical errors may also require a more involved analysis to diagnose; manual classification of a large data set does not lend itself to such detailed analysis, since it would require significantly more time and effort.

The category defined as *statement outside method/block* perhaps seems oddly precise when compared to the other high-level categories, but this error was observed in the data set and did not fit into any other high-level category, so remained as a category in its own right.

The remaining top-level category, *Uncategorized*, allows for a selection to be made when an appropriate category cannot otherwise be decided. (A selection of Uncategorized counts as a category selection for purposes of analysis.)

3.2 Frequency of Error Categories

A total of 1105 error categorizations were made for the 1000 source code snapshots making up the data set (the difference in number being due to multiple errors being categorized for a small number of events, as discussed in 2.3.3). The most frequent error was "Variable not declared", accounting for approximately 8.4% of all errors. In our previous work [20] the same category was also the most frequent, in that case accounting for 11.1% of the recorded errors.

3.2.1 Most Frequent Error Categories. Table 1 shows the relative frequency of the top 10 errors occurring in our data set (consisting of 199 user sessions, selected at random from the Blackbox data, as discussed in section 2.1). A complete list is presented in Appendix B.

Other studies have examined the frequency of errors categorized using the diagnostic generated; in prior work [20] we have shown evidence that this method of categorization is not ideal, a finding further substantiated by the results presented here.

Table 1. Frequency of Top 10 Error Categories

Error Category	Frequency
Variable not declared	8.4%
Variable name written incorrectly	7.4%
; missing	7.3%
Simple syntactical error	6.5%
Semantic error	4.8%
Variable: Incorrect variable declaration	4.7%
Variable: Incorrect attempt to use variable	4.6%
Method name written incorrectly	4.6%
Method: Incorrect method declaration	4.5%
Class or type name written incorrectly	4.4%

Denny, Luxton-Reilly and Tempero [9] published such a distribution of errors made by novices, finding the "Cannot resolve identifier" error as most frequent (at 24%). This likely corresponds, to some extent, to a combination of four of our categories—"Variable not declared", "Variable name written incorrectly", "Method name written incorrectly" and some instances of "Method: Incorrect method declaration". These four categories have a frequency total of 24.9%, very close to the 24% reported for "Cannot resolve identifier" in the former work, though their survey counts all errors in each submission whereas our results include only errors focused on a localized area of the source code (around the site identified in the diagnostic issued by the compiler, as described in section 2.3). However, the next most frequent error reported by Denny et al is "Type mismatch", with frequency of 18.4%—making it an interesting deviation from our own results; the error type probably corresponds best to the "Type error" category (not appearing in the top 10), which has a much lower frequency of 3.8%. They find "Missing ;" as the 3rd most frequent error at 13.0%, again notably higher than for the "; missing" category (7.3%, also the 3rd most frequent).

Jadud [13] published another distribution of errors made by novices, finding the "missing semi-colon" 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. By contrast, our own results show that "variable not declared" (corresponding to "unknown symbol – variable") is the most frequent error category, with "; missing" ("missing semicolon") moving down to the third place with a frequency of 7.3% (compared to the 18% reported by Jadud).

Jackson, Cobb, and Carver [12] found a somewhat different distribution in their own study, also based on diagnostic messages: they list "cannot resolve symbol" as the most frequent at 14.6%, with "; expected" next at 8.5%. Other messages reported as having a relatively high frequency were "illegal start of expression", ") expected", "class or interface expected" and "<identifier> expected", all of which also occur in Jadud's [13] top-10 list. If we assume that "cannot resolve symbol" diagnostic message corresponds to the two categories "variable not declared" and "variable name written incorrectly" from our scheme, then the frequency and order of the top two errors reported by Jackson, Cobb and Carver [12] match well with the top three category frequencies from our own study. The remaining categories in our results cannot be matched exactly with diagnostic messages.

Although the frequencies vary, there are several diagnostic messages that appear in the three compared top 10 lists and, allowing for probable correspondences between certain messages and

Table 2. Error Frequency and Difficulty: top-level categories, amalgamated

Category	Occurrences	Difficulty	σ
Simple syntactical error	226 (20%)	53.88	56.79
Variable: Incorrect attempt to use variable	220 (20%)	51.59	43.90
Method: Incorrect method call	199 (18%)	73.88	62.60
Semantic error	140 (13%)	91.57	62.96
Method: Incorrect method declaration	87 (7.9%)	79.29	75.01
Incorrect/attempted use of class or type	66 (6.0%)	69.19	51.69
Variable: Incorrect variable declaration	58 (5.2%)	86.74	69.04
Constructor: Incorrect constructor call	9 (0.8%)	103.89	71.12
Constructor: Incorrect constructor declaration	4 (0.4%)	109.33	95.33
Statement outside method/block	2 (0.2%)	14.00	N/A

Note: Difficulty is the adjusted average resolution time, which combines resolution times and unresolved errors. For unresolved errors, a nominal resolution time of 300 seconds was assigned. *Statement outside method/block* is included for completeness as it was not included under a broader category. Figures for each listed category amalgamate errors that were labelled with the listed category or any of its sub-categories. The standard deviation of difficulty for each category is shown in the column headed "σ".

categories in our scheme, the results from both these works appear to be consistent to some degree to our own (especially with the three most frequently occurring error categories).

The diagnostic message frequencies generated for the error events analyzed in this work 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. However, as we have shown in previous work [20], a suitably designed categorization scheme allows a more precise and accurate classification of errors than does a reliance on compiler-generated diagnostics. Furthermore the frequency of error categories or diagnostics is not necessarily by itself a good indicator of the severity of errors; that a particular type of error occurs frequently does not mean it is difficult to resolve.

The error categories were organized in a tree hierarchy. The full tree of categories is shown in Appendix C. Table 2 shows the first-level categories, and the total number of errors identified in each first-level category or any of its sub-categories.

3.3 Difficulty of Errors

As described in 2.4.3 the resolution status of errors was determined and (for cases where the error was resolved) the average time-to-fix for each error category was calculated. Of the 91 error categories, 63 had fewer than 10 observed occurrences within the data set. 51 error instances were marked as unresolved due to a gap of more than 5 minutes between recorded events.

3.3.1 Resolution Status. Error instances were classified as resolved, unresolved, uncertain or reverted (where reverted is a special case of unresolved). Table 3 shows the 10 error categories with the highest percentage of unresolved error instances (including reverted instances), out of those categories with 10 or more occurrences.

3.3.2 Resolution Time. Table 4 shows the top 10 error categories by average resolution time. Again, error categories with fewer than 10 occurrences were excluded.

3.3.3 Adjusted resolution time. Average time-to-fix gives an indication of the severity of different error categories. The resolution rate is, however, also an important consideration; a particular error

Table 3. Resolution failure: The top 10 precise error categories by frequency of failure to resolve the error

Error	Total	Resolved	Unresolved	Reverted	% not resolved
Missing 'return' statement	23	13	6	0	26%
Uncategorized	94	56	18	1	21%
Type error	38	25	7	0	18%
Method call: parameter number mismatch	27	19	2	2	15%
Variable: Incorrect variable declaration	48	27	6	1	15%
Method not declared	29	20	3	1	14%
Use of non-static method from static context	16	13	2	0	13%
Method call targeting wrong type	25	18	2	1	12%
Semantic error	49	34	5	0	10%
Method call: parameter type mismatch	10	8	1	0	10%

Note: Errors not shown as resolved, unresolved or reverted had uncertain resolution status.

Table 4. Time-to-fix: The top 10 precise error categories by time required until the error was fixed

Error	Total	Resolved	Unresolved	Reverted	Average resolution (seconds)	σ
Method call: parameter type mismatch	10	8	1	0	84	66.64
Method call targeting wrong type	25	18	2	1	80	89.99
Method declaration: missing return type	10	6	0	0	74	116.77
Method: Incorrect method declaration	46	24	3	0	70	79.87
Missing 'return' statement	23	13	6	0	65	78.12
Class from other package used without 'import'	13	5	0	0	61	45.93
Use of non-static method from static context	16	13	2	0	60	74.84
Semantic error	49	34	5	0	52	50.07
Method: Incorrect method call	20	15	0	1	51	44.66
Uncategorized	94	56	19	1	48	64.95

Note: Errors not shown as resolved, unresolved or reverted had uncertain resolution status. The column headed "σ" shows the standard deviation of the resolution times in each category.

category might exhibit a low average time-to-fix but simultaneously a low resolution rate – that is, errors belonging to such a category are often unresolved, arguably a worse outcome than being resolved slowly. Such errors should not necessarily be considered low-severity when compared to another error category with a higher average time-to-fix but significantly better resolution rate.

To produce a single value usable as a metric for estimating overall resolution difficulty of the different error categories, an unresolved error (including a reverted error) was treated as an error with a 5-minute resolution time (5 minutes was the maximum recorded resolution time, as described in section 2.4.3). Thus the *adjusted resolution time* for an error is the observed resolution time capped at 5 minutes, or 5 minutes if no resolution was observed and the error was *unresolved* (errors with resolution status of *uncertain* do not have an adjusted resolution time).

For each error category, the adjusted average resolution time was produced by dividing the total adjusted resolution time by the number of resolved and unresolved errors (including reverted errors, but not including errors with *uncertain* resolution status). The adjusted resolution time of all error categories with 10 or more occurrences is shown in Table 5.

A plot of adjusted average resolution time (*difficulty*) versus occurrence count for all categories (excluding those with only uncertain resolutions) is given in Figure 1, which organizes data points from each categories by their corresponding top-level category. The same data is shown in table form, together with standard deviation of difficulty for each category, in Appendix B. The pink shaded area shows error categories with more than 10 occurrences and more than 10 seconds adjusted average resolution time. The average adjusted resolution time for all errors falling under each top-level category is shown in Figure 2, with data corresponding to that of Table 2.

3.3.4 Error Severity. The average adjusted resolution time of an error category gives an indication of the difficulty of the category, in terms of the time spent dealing with each occurrence of the error. A category with a high level of difficulty may however have a low frequency, and therefore not require as much overall effort on the part of the programmer as another category with lower difficulty but significantly higher frequency.

As discussed in 1.3, we have defined the severity of an error type as a product of frequency and difficulty:

$$severity = frequency \times difficulty$$

Since the difficulty is expressed as a time, the severity is an indicator of the relative amount of time spent on resolving (or attempting to resolve) errors in each error category. Because the adjusted average resolution time is used as the metric for difficulty, it has a theoretical maximum value of 300. Since frequency is a proportion, the severity value also has a maximum possible value of 300. The severity of the 20 most severe error categories is shown (in descending order) in Table 6, and the full table of error category severity is included in Appendix D.

3.4 Error Coverage of Most Frequent Categories

For a given set of categories, the *coverage* is the percentage of all individual error events falling into one of the categories in the set. The coverage level for the N most prevalent error categories, for various values of N, is presented in Table 7.

3.5 Intercoder Reliability

As detailed in section 2.2.2, inter-coder reliability was assessed using a modified pairwise agreement calculation. The initial calculation is optimistic in the sense that a difference in the number of error categories tagged by participants in the categorization process is not considered a disagreement; a second calculation (pessimistic) was performed in which such a discrepancy does count as disagreement.

Table 5. Adjusted average resolution times

Category	Avg. time (seconds)	N	σ
Missing 'return' statement	130	23	134.55
Uncategorized	114	94	125.13
Method call: parameter type mismatch	109	10	95.02
Method call targeting wrong type	104	25	116.79
Method: Incorrect method declaration	96	46	105.08
Variable: Incorrect variable declaration	95	48	121.40
Use of non-static method from static context	93	16	109.00
Type error	93	38	117.16
Method not declared	89	29	105.30
Type mismatch in assignment	88	12	111.51
Semantic error	84	49	96.05
Method call: parameter number mismatch	80	27	109.86
Mismatched parentheses in or around expression	68	10	129.73
Simple syntactical error	67	66	96.63
Method: Incorrect method call	67	20	75.62
Method declaration: missing return type	63	10	116.77
Class from other package used without 'import'	61	13	45.93
Variable not declared	56	85	87.57
Variable: Incorrect attempt to use variable	53	47	85.23
Class or type name written incorrectly	50	44	91.59
Extraneous closing curly brace	49	21	98.35
Method name written incorrectly	47	47	63.62
Missing parentheses for method call	43	21	80.20
Variable name written incorrectly	43	75	77.31
Wrong return type declared	42	15	67.27
Missing operator between expression elements	41	12	66.60
Missing closing curly brace at end of class	33	12	76.44
; missing	30	74	77.80

Note: Errors are ordered by adjusted average resolution time, which combines resolution times and unresolved errors. For unresolved errors, a nominal resolution time of 300 seconds was assigned. Only categories with at least 10 instances are shown. The columns headed "N" and "σ" show the number of occurrences and the standard deviation of the adjusted resolution times in each category, respectively.

There were two phases of categorisation—the original categorisation and the revised category selection—where the second phase was designed to achieve a higher agreement level. However, the second phase renders the optimistic calculation invalid and only the pessimistic value should be considered.

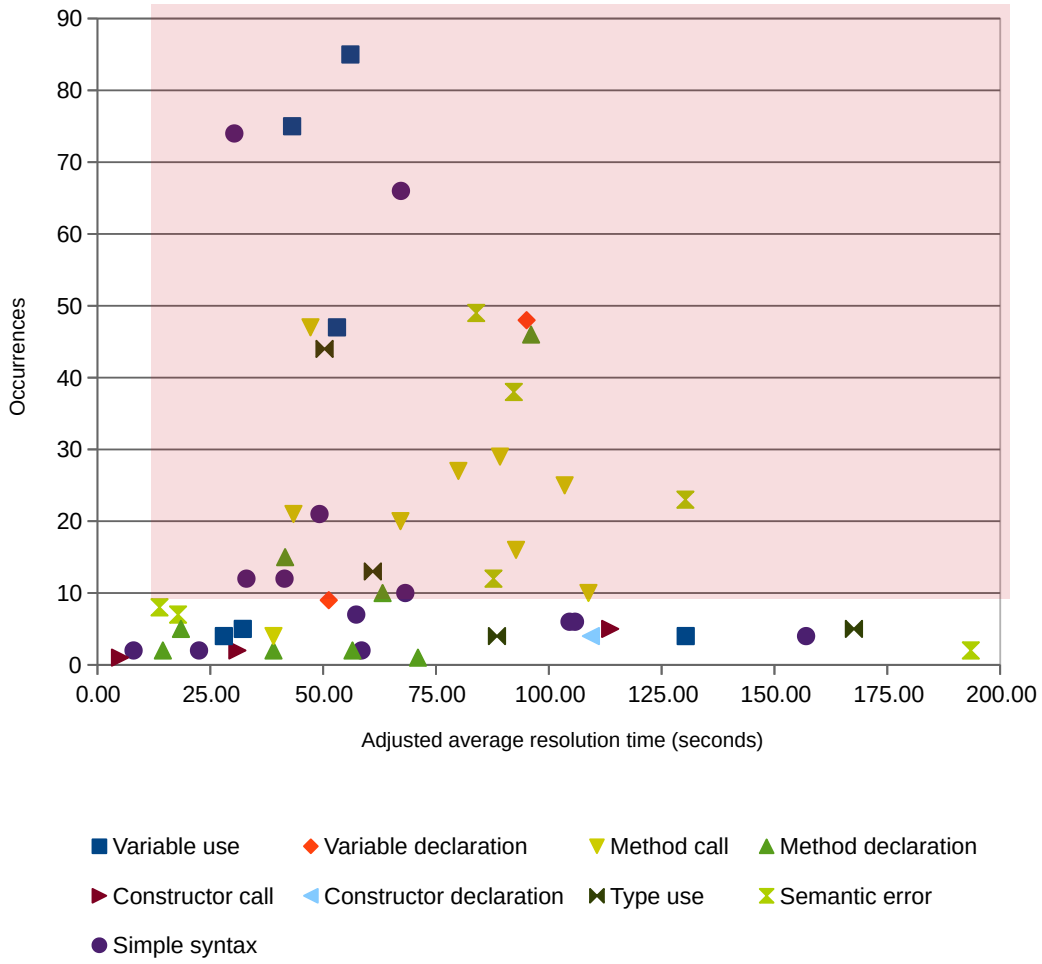

Fig. 1. Adjusted resolution time and frequency of occurrence of error categories (all categories)

The overall agreement levels were:

- 1st phase: 64.85% (optimistic) 59.34% (pessimistic)
- 2nd phase: 78.19% (optimistic) 74.33% (pessimistic)

The agreement levels when only the most frequent categories were considered (and excluding the "uncategorized" categorization) were as follows:

- Top 10 categories: 91.47% (optimistic) 87.73% (pessimistic)
- Top 20 categories: 87.54% (optimistic) 82.40% (pessimistic)
- All categories, excluding "uncategorized": 84.07% (optimistic) 79.60% (pessimistic)

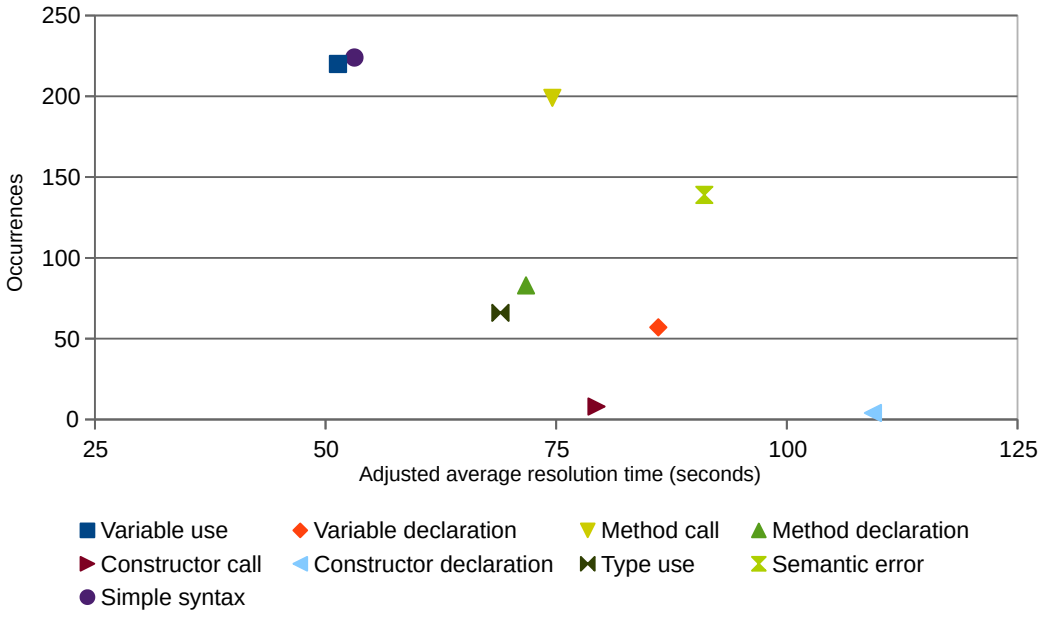

Fig. 2. Adjusted resolution time and frequency of occurrence: mean values of the nine top-level categories

4 DISCUSSION

4.1 Coverage Level of Error Categories

One of the primary sources of motivation of this work was to provide a basis for improving both pedagogy of programming education and tools to support the teaching and learning of programming. For pedagogical interventions it is helpful to know the kinds of errors students most struggle with, so that increased time and effort might be targeted specifically on those areas. For advances in programming tools, such as improved presentation of error messages presented to students, it is helpful to know which problems to concentrate on, and what impact an improvement in messaging for specific kinds of errors may have.

The coverage table (Table 7) shows that the top ten logical errors account for 60% of all error occurrences. Thus, if error diagnostics could be improved for only these ten types of error, students would benefit from improved feedback 60% of the time they see an error message. If better messages could be produced for the top twenty error types, feedback would be improved for 80% of experienced errors. And even if only the top five logical errors were detected and reported accurately, students would benefit in nearly 40% of all error instances.

This data shows that it might be worthwhile to work on dedicated detectors in programming environments that identify specific logical errors and provide more helpful messages for those. Enhanced diagnostic tools have been produced before [3, 8, 10, 11], with varying degrees of effectiveness; guiding the development of such tools with an analysis of logical error severity could improve their effectiveness. Even if the number of types of error detected by a diagnostic tool is fairly limited, targetting the right logical errors could mean that the the impact on programming experience of beginners may be sufficiently significant to make the required development effort worthwhile.

Table 6. Error Category Severity — top 20

Category	Occurrences	Severity
Variable not declared	85	4.763
Variable: Incorrect variable declaration	48	4.564
Simple syntactical error	66	4.437
Method: Incorrect method declaration	46	4.419
Semantic error	49	4.111
Type error	38	3.506
Variable name written incorrectly	75	3.234
Missing 'return' statement	23	2.996
Method call targeting wrong type	25	2.588
Method not declared	29	2.586
Variable: Incorrect attempt to use variable	47	2.494
; missing	74	2.244
Method name written incorrectly	47	2.218
Class or type name written incorrectly	44	2.214
Method call: parameter number mismatch	27	2.159
Use of non-static method from static context	16	1.484
Method: Incorrect method call	20	1.343
Method call: parameter type mismatch	10	1.088
Type mismatch in assignment	12	1.053
Extraneous closing curly brace	21	1.033

Note: The severity is calculated as a product of adjusted average resolution time and occurrence count. The adjusted average resolution time is the average of all resolution times when unresolved errors are considered as resolved in 300 seconds (i.e. the upper limit for resolution); instances with uncertain resolution do not take part in the calculation. Categories with occurrences all having uncertain resolution status are excluded.

Table 7. Error coverage for most frequent categories

Number of categories	Coverage level
5	39%
10	60%
15	74%
20	82%
25	87%
30	91%

Note: The coverage level shows the percentage of error instances covered by the most frequent N categories (e.g. the 5 most frequent error types represent 39.02% of all encountered error events).

4.2 Severity of Error Categories

4.2.1 Severity Rankings. Although frequency of error categories gives an indication of the error types that students encounter most often, it does not give any indication of how much of a problem these errors present for students overall. An error may be encountered frequently but be solved, in general, quite easily; conversely, a less-frequently occurring error may require substantially more effort to resolve each time it occurs.

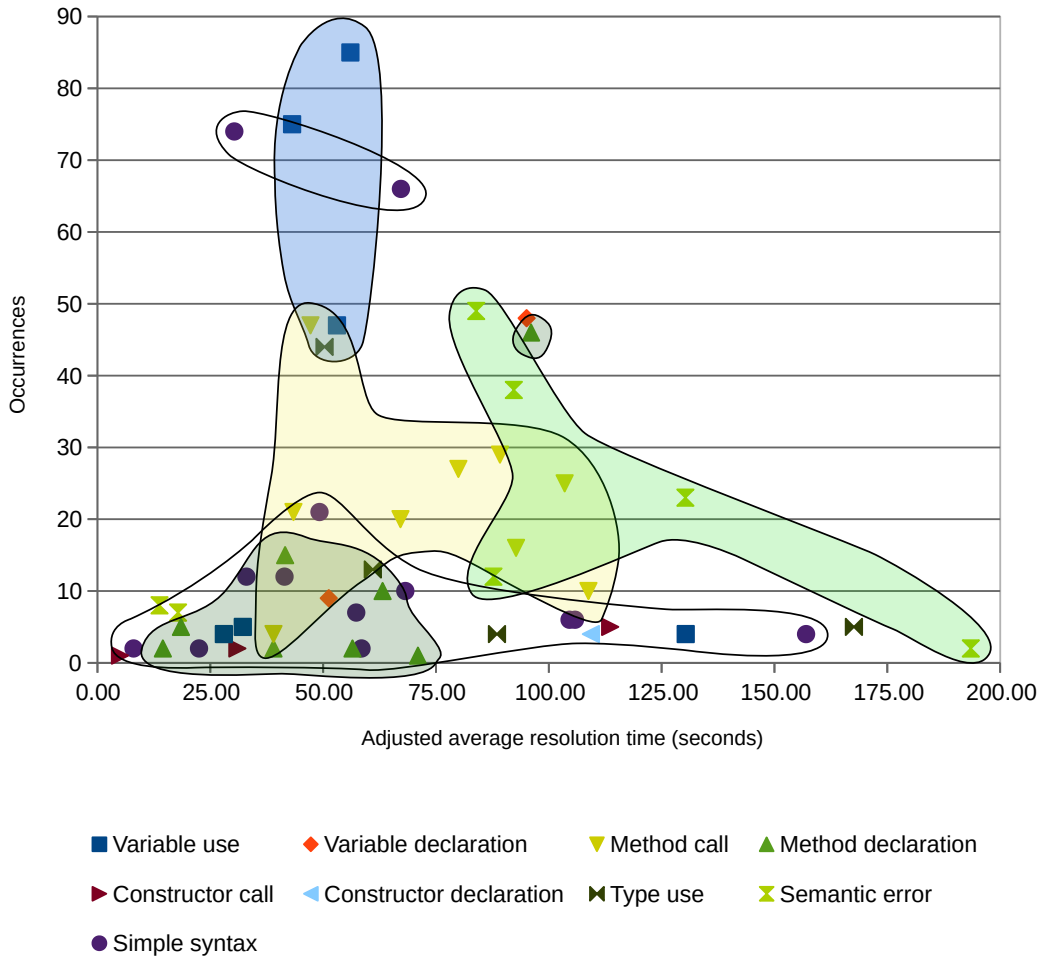

Fig. 3. Adjusted resolution time and frequency of categories (visually grouped by top-level category)

The severity of errors, as depicted in Figure 1 as a combination of frequency and adjusted resolution time, allows some interesting observations not only about the impact of single error types, but also about areas of concepts in programming that cause difficulties for students.

Errors and error categories towards the top right quadrant of the graph are more severe than those at the left and bottom. Errors along the x-axis are less severe because they do not occur very often, while errors near the y-axis are less severe because they are typically resolved quickly.

In Section 1.3, we discussed error severity as a product of the frequency of the error type and the difficulty in resolving errors of that type. We have used the adjusted average resolution time (3.3.3) as the sole metric for deciding error difficulty in this work, and the error categories are ranked (Table 6) by severity calculated on this basis.

The most severe error category is *Variable not declared* (severity score 4.763; visible as the top-most square in Figure 1), which is also the most frequently occurring category with 85 categorized instances. The adjusted average resolution time is just over 56 seconds, which shows that substantial

time is expended on resolving errors in this category. The next four categories—*Incorrect variable declaration* (severity score 4.564), *Simple syntactical error* (4.437), *Incorrect method declaration* (4.419) and *Semantic error* (4.111)—all have a slightly higher adjusted average resolution time, extending up to 96 seconds in the case of *Incorrect method declaration*, though with a corresponding lower frequency of occurrence.

Other than the first (*Variable not declared*), each of the top 5 error categories as ranked by severity are top-level categories which correspond to a broad error description and that have a number of sub-categories, which provide a finer-grained error description. Partly this is due to these top-level categories generally having a noticeably higher occurrence count than the subcategories (with some exceptions—in particular, *Variable name written incorrectly* with 75 occurrences, and *Semicolon missing* with 74 occurrences); this in turn is indicative of difficulty in providing a precise diagnosis for errors from amongst the available (sub-)categories. *Variable not declared* is a clear standout in this regard, since it is both a precise subcategory and the highest occurring categorization overall (as well as having the highest severity score).

One general pattern that we noticed was that semantic-related errors appear to present more difficulty than syntactic errors. In the category of Variable use, for example, *Variable name written incorrectly* (which will include simple typing errors that are easily fixed) and *Variable not declared* (which indicates a slip or omission that is also easily repaired) have low difficulty, though with higher frequency, than *Use of variable from another class*—an error which indicates a lack of understanding of underlying concepts. As another example, in the top-level category of *Method Calls*, the syntactic problems (including writing a parameter type in the method call and leaving out the parenthesis of a call) have lower difficulty than semantic problems, such as trying to call non-static methods from a static context, and using the wrong type of either the receiver or a parameter of a method call.

Many categories have few occurrences, making their indication of severity unreliable. A larger data set may be required to sufficiently assess the severity of these categories, though the manual categorization process does not lend itself to large data sets.

Identifying the cause of the apparent difficulty in precisely categorizing errors would require further investigation to determine. The method used to develop the category hierarchy (2.2) was designed to provide suitable categories for commonly occurring errors, rendering it doubtful that lack of availability of appropriate categorizations was to blame. The possibility that it is innately difficult to precisely categorize an appreciable portion of novice programmer errors needs to be considered (though this does not imply that attempting such categorization is a worthless endeavor, since some types of error can be clearly identified, and since information on error severity even at the broad level can be useful for improving programming pedagogy and tools). If it is difficult to precisely diagnose a novice error—perhaps because the novice's intention, or the nature of the misconception that caused the error, are unclear—then a case can be made that any diagnostic information provided to the student (such as messages from a compiler or other development tool) should be similarly imprecise, while still being broadly accurate.

Figure 3 depicts the same data as Figure 1, but areas representing top-level error categories are highlighted to encompass the data points of the individual errors they contain. In some broad categories, but not all, the individual subcategories show some tendency to cluster, indicating that they have similar frequency and difficulty (and therefore severity). This may indicate that miscomprehending certain concepts described by the top level category may lead to production of a range of errors in related categories, which seems plausible, although we note that there are some clear exceptions, such as *Incorrect method declaration* (the top-level category) having a high severity compared to its various subcategories (including *Missing comma in parameters*, *Missing method body*, *Missing return type*, *No type specified for parameter*, *Return type should be void*, *Wrong number of parameters*, *Method duplicated exactly*, *Missing opening curly brace after method*

header and *Wrong return type declared*). The subcategories of *variable use* do not cluster at all, indicating that this category cannot meaningfully be treated as a single concept for the purposes of pedagogical interventions; it is not uniformly easy or difficult, but rather encompasses some errors at either end of the scale.

Figure 2 depicts top-level categories only, with occurrence and resolution time from subcategories folded in to the top-level category. This identifies the broad categories of *Simple Syntax*, *Variable use*, *Method call* and *Semantic error* as being the most severe. The *Constructor declaration* category also appears severe but has very few occurrences and its severity score may be inaccurate as a result.

4.2.2 Error Severity versus Diagnostic Message Frequency. Various other studies [12, 13] have attempted to characterize novice errors by examining the frequency of different diagnostic messages produced by the compiler. This approach is of limited usefulness due to the potentially weak correlation between the compiler diagnostic and the nature of the particular error being diagnosed. A compiler can produce two different diagnostic messages for what is logically the same error occurring in two subtly different syntactical contexts, for example; similarly, it might produce the same diagnostic message for two quite distinct errors that could be recognized by an experienced programmer as logically distinct error types. Furthermore, different compilers—or different versions of the same compiler—may produce different diagnostics when presented with the exact same source code input.

In the work of Jadud [13], "; missing" was found to be the most frequent diagnostic message encountered by the cohort in the study, followed by "unknown symbol — variable", "bracket expected", "illegal start of expression" and "unknown symbol — class". Another study, by Jackson, Cobb and Carver [12], presented "cannot resolve symbol" as the most frequent diagnostic, followed by "; expected", with "illegal start of expression" and "(expected" also in the top 6; between the two, the precise ordering is slightly different, and the wording of the messages varies somewhat, but there is an apparent level of agreement as to which diagnostics were produced most frequently, which was further substantiated by our previous work [20]. However, our results in this work show (Table 6) that a ranking of error categories by severity provides a markedly different ordering of error types. For example, "; missing" (which, as noted, appears as the most or second-most frequently occurring diagnostic) is found at position 12 in the severity rankings. The highest-ranked error category in terms of severity is *Variable not declared*, which does coincide to some degree with the frequently occurring diagnostics "cannot find symbol – variable" / "cannot resolve symbol", but which is more precise. Other high-severity error categories have no clear diagnostic message counterpart, and might be associated with several different messages.

The use of manually categorized errors, together with a determination of severity by combining both frequency and difficulty, provides a much better basis for understanding which types of error (and to a limited extent, which concepts, or at least syntactical constructs) it is that novices most struggle with than does a simple frequency count of diagnostic messages.

4.3 Risks and Limits to Validity

There are several factors imposing risks and limits to the interpretation of data presented in this work.

The data comprising the student source code within which errors were categorized was sampled from data maintained by the Blackbox Data Collection project [6], which collects anonymized data from BlueJ users. The programming background, age, and other factors that may affect programming ability or behavior of participants in this data collection are not known, though it is expected most are in the first year of university study.

The source code analyzed was written in the Java programming language, and the types of errors encountered by the programmers, as well as their severity, might differ when compared to other programming languages.

While a large amount of data was processed, some error categories manifested only a very small number of times. The calculation of average resolution time for such error categories is prone to high margins of error due to the small sample sizes; however, the low frequency also reduces the severity of such error categories, so that such categories are necessarily also low in the severity rankings, which for purposes of this work makes them less interesting and this uncertainty less important.

The determination of resolution time for an individual error instance uses heuristics to decide when (and if) the error has been resolved, and is subject to the limitation that only compilation triggers an assessment of resolution. The time measurements therefore include any time between the compilation event that was considered to resolve the error and the previous compilation event; in some cases this could include time spent editing other parts of code after solving the error (but before choosing to compile) or while being distracted from the coding task, for example. It is expected that such an effect applies to all error categories similarly, and thus may not strongly affect ranking of errors. Absolute time data, however, may be affected by this effect.

The heuristics for identifying recurring errors (see section 2.3.2) are not perfect and may introduce some inaccuracy in the results due to occasionally classifying two individual errors as a single (recurring) error with a resolution time extended to the total resolution time of the two errors.

5 FUTURE WORK

A ranking of novice programmer errors by severity is a useful result for the future development of programming pedagogy, but could also be used to guide development of programming tools and environments designed for novices.

The goal of the next phases of the authors' work is to improve automatically generated diagnostic messages. Various other efforts [2, 8, 10, 11] have attempted this with varying levels of success, but have used anecodotal evidence to determine high-impact errors, or chosen them based on frequency of encountering diagnostics. The results from this work provide information on which error categories these improvements should be targeted at in order to provide the most benefit to novices; the following phases will utilize this information. The planned phases are:

(1) Examine the records on the contextual information that was noted by researchers categorizing errors to be useful in the categorization (those performing the categorization were asked to provide a brief description of any context used in the categorization of each error).
(2) Investigate and make a determination of whether such contextual information can be used in an automated diagnosis, particularly of high-severity error categories.
(3) Implement a system to automate diagnosis of high-severity error categories, with the aim of producing diagnostic information that makes an improvement over the error diagnostics provided by compilers and environments currently used for teaching programming.

The high prevalence and severity of syntax errors (particularly the top-level *simple syntactical error* category) strongly suggest that syntax remains a significant hurdle for novices. While improved diagnosis of syntax errors will be of benefit, there are other possibilities for alleviating this problem, including syntax-directed or structured editing (such as in the Stride language editor [16]—as implemented in the Greenfoot programming environment [15]—as well as in block-based environments such as Scratch [18] and Alice [7]). A study similar to that detailed in this work but applied to students using such an editing environment and programming language could provide

insight into the actual benefit of such approaches, and reveal the differences in frequency and severity of errors when compared to a traditional language and environment.

6 CONCLUSION

Information on which errors novice programmers encounter most frequently has long been considered useful. This work improves on previous studies in this area by avoiding the use of compiler diagnostic messages as proxies for the type of error encountered.

Combining the measure of difficulty, based on time-to-fix, together with frequency of different error types has yielded a ranking of errors by severity. Our time-to-fix calculation is complex and takes into account several scenarios not covered by previous work. The results confirm that an ordering which takes difficulty of fixing the errors into account is markedly different from a ranking by frequency alone.

The results further indicate that a significant amount of more severe errors are difficult to classify precisely. Semantic errors are among the most severe, but syntactic errors also feature with high severity. The greatest number of errors is syntactical. These findings have implications for the future design of programming pedagogy and tools. In particular, focusing on better diagnosis for—and instruction regarding—a small number of high-severity errors may prove highly beneficial for student programmers.

A ALGORITHM FOR RESOLUTION STATUS CLASSIFICATION

The algorithm presented in this appendix was used for classification of the resolution status of errors, as discussed in 2.4.3.

ALGORITHM 2: Error resolution status classification

Input: The compilation events in a session.
Output: The resolution status for each error.
Let the *active error* be the first error encountered in the session, as decided by the error categorization (note that in a few cases this is actually more than one single error category selection — in this case actions applied to the active error are applied to each selected category);
foreach *event in sequence, after the active error event* **do**
  if *more than 5 minutes elapsed since the previous event* **then**
    Assume that the programmer has taken a break during the session and mark the *active error* as unresolved. Let the *active error* become the next event in the sequence with an error (possibly the currently examined event);
    continue
  else if *the event is in a different project than the previous event* **then**
    Mark the *active error* as unresolved. Let the *active error* become the next event in the sequence with an error (possibly the current examined event);
    continue
  else if *the event does not contain an error* **then**
    Check if the source code has been reverted to the same state it was in immediately prior to the active error;
    if *the source code has been reverted* **then**
      Mark the *active error* as reverted. Let the active error become the error category selection(s) from the next event in the sequence with an error, and continue from that event;
      continue
    else
      Mark the *active error* as resolved, unless the resolution time exceeds 5 minutes, in which case mark it as unresolved. Let the active error become the next event with an error and continue from that event;
      continue
    end
  else if *the event is marked as recurring* **then**
    Check if the recurrence marking is due to the source code being reverted to an earlier state[a];
    if *the source was reverted to an event state preceding the active error* **then**
      Mark the *active error* as *reverted*. Let the *active error* be the error from the reverted-to event;
    else
      Consider the reversion to be part of the process of solving the *active error*, and continue with the next event in sequence;
    end
  else if *the diagnostic issued for the error indicates the same source file and the same or earlier line number than the active error* **then**
    Assume that a new error was introduced by the programmer. Mark the *active error* as *uncertain resolution*, and then let the *active error* be the error in the current event;
  else
    The diagnostic indicates a different source file or a later line in the same source file. Mark the *active error* as *resolved*, and then let the *active error* be the error in the current event;
  end
end

[a]Whether this is the case is recorded during the marking of recurring errors as in 2.3.2

B ERROR CATEGORIES

This appendix contains a list of all error categories, ordered by number of occurrences of errors of this category within the data set (comprising 1000 events). Note that the categories are ranked by calculated severity in Appendix D. A small number of categories were not found within the data set and so have 0 recorded occurrences. Top-level categories in the hierarchy are marked "(*)". Also shown, when sufficient data is available, are the adjusted average resolution time (*difficulty*) and its standard deviation.

Error Category	Occurrences	difficulty	σ
Uncategorized (*)	94	113.68	125.13
Variable not declared	85	56.04	87.57
Variable name written incorrectly	75	43.12	77.31
; missing	74	30.32	77.80
Simple syntactical error (*)	66	67.23	96.63
Semantic error (*)	49	83.90	96.05
Variable: Incorrect variable declaration (*)	48	95.09	121.4
Variable: Incorrect attempt to use variable (*)	47	53.06	85.23
Method name written incorrectly	47	47.20	63.62
Method: Incorrect method declaration (*)	46	96.07	105.08
Class or type name written incorrectly	44	50.32	91.59
Type error	38	92.25	117.16
Method not declared	29	89.17	105.30
Method call: parameter number mismatch	27	79.96	109.86
Method call targeting wrong type	25	103.52	116.79
Missing 'return' statement	23	130.26	134.55
Extraneous closing curly brace	21	49.19	98.35
Missing parentheses for method call	21	43.42	80.20
Method: Incorrect method call (*)	20	67.13	75.62
Use of non-static method from static context	16	92.73	109.00
Wrong return type declared	15	41.58	67.27
Class from other package used without 'import'	13	61.00	45.93
Missing closing curly brace at end of class	12	33.00	76.44
Missing operator between expression elements	12	41.44	66.60
Type mismatch in assignment	12	87.73	111.51
Method call: parameter type mismatch	10	108.78	95.02
Method declaration: missing return type	10	63.17	116.77
Mismatched parentheses in or around expression	10	68.20	129.73
Variable needs a unique name	9	51.25	46.34
Array used in place of array element	8	13.75	9.39
Missing closing curly brace at end of method	7	57.33	118.95
Unhandled exception	7	17.86	12.63
'=' used in place of '=='	6	105.80	129.74
Missing closing curly brace after control stmt body	6	104.60	133.46
Incorrect/attempted use of class or type (*)	5	167.60	136.28
Constructor parameter number mismatch	5	113.60	120.12
Method declaration: Wrong number of parameters	5	18.50	12.02
Use of non-static variable from static context	5	32.20	19.29
Constructor: Incorrect constructor declaration (*)	4	109.33	165.12

Class not defined	4	88.50	141.09
Keyword written incorrectly	4	157.00	202.23
Method call: Parameter types included	4	39.99	27.06
Use of variable from another class	4	130.33	147.55
Wrong variable used	4	28.00	19.31
Constructor: Incorrect constructor call (*)	2	31.00	18.38
Class does not implement required method	2	193.50	150.61
Extra opening curly brace '{'	2	58.50	33.23
Extraneous or misplaced ')'	2	8.00	N/A
Method declaration: missing method body	2	300.00	0.00
Method declaration: no type specified for parameter	2	N/A	N/A
Method declaration: return type should be void	2	14.50	7.78
Method duplicated exactly	2	56.50	2.12
Missing closing curly brace	2	22.50	24.75
Missing opening curly brace after method header	2	39.00	N/A
Statement outside method/block (*)	2	14.00	N/A
: in place of ;	1	300.00	N/A
Comment incorrectly written	1	N/A	N/A
Constructor parameter type mismatch	1	300.00	N/A
Invalid type cast	1	300.00	N/A
Local variable declaration with illegal modifier	1	N/A	N/A
Method declaration: Semicolon at end of method header	1	71.00	N/A
Missing parentheses for constructor call	1	5.00	N/A
Assignment to a collection element	0	N/A	N/A
Call to 'super' in constructor not first statement	0	N/A	N/A
Class should be declared to implement interface	0	N/A	N/A
Collection used in place of element	0	N/A	N/A
Constructor call attempted using variable name	0	N/A	N/ A
Constructor call to non-existent copy constructor	0	N/A	N/A
Constructor call without using 'new'	0	N/A	N/A
Constructor is declared to return 'void'	0	N/A	N/A
Extra closing parenthesis	0	N/A	N/A
Extraneous '(' between 'else' and 'if'	0	N/A	N/A
Method call attempted using variable name	0	N/A	N/A
Method call: missing comma between parameters	0	N/A	N/A
Method call: parameter doubles as declaration	0	N/A	N/A
Method declaration: Missing comma in parameters	0	N/A	N/A
Method declaration: semicolon in place of comma	0	N/A	N/A
Missing '.' between names in qualified name	0	N/A	N/A
Missing closing parenthesis in constructor call	0	N/A	N/A
Missing space after 'new'	0	N/A	N/A
Object instantiation uses variable name	0	N/A	N/A
Object instantiation uses variable name; intends variable access	0	N/A	N/A
Type mismatch: arithmetic performed on String; use length	0	N/A	N/A
Use of variable in context requiring a type	0	N/A	N/A
Value from raw collection must be cast	0	N/A	N/A
Variable access used as statement	0	N/A	N/A
Variable declaration: name and type in wrong order	0	N/A	N/A

Variable declaration: name missing	0	N/A	N/A
Variable declaration: variable name contains whitespace	0	N/A	N/A

C ERROR CATEGORY HIERARCHY

This appendix contains the complete category hierarchy. Each category name is followed by *(x; y)*, where x is the total number of occurrences of the error category and all its subcategories, and y is the number of occurrences within the category (excluding occurrences in subcategories).

- Variable: Incorrect attempt to use variable (220; 47)
  - Object instantiation uses variable name; intends variable access (0; 0)
  - Use of non-static variable from static context (5; 5)
  - Use of variable from another class (4; 4)
  - Variable name written incorrectly (75; 75)
  - Variable not declared (85; 85)
  - Wrong variable used (4; 4)
- Variable: Incorrect variable declaration (58; 48)
  - Local variable declaration with illegal modifier (1; 1)
  - Variable declaration: name and type in wrong order (0; 0)
  - Variable declaration: name missing (0; 0)
  - Variable declaration: variable name contains whitespace (0; 0)
  - Variable needs a unique name (9; 9)
- Method: Incorrect method call (199; 20)
  - Method call attempted using variable name (0; 0)
  - Method call targeting wrong type (25; 25)
  - Method call: missing comma between parameters (0; 0)
  - Method call: parameter doubles as declaration (0; 0)
  - Method call: parameter number mismatch (27; 27)
  - Method call: parameter type mismatch (10; 10)
  - Method call: Parameter types included (4; 4)
  - Method name written incorrectly (47; 47)
  - Method not declared (29; 29)
  - Missing parentheses for method call (21; 21)
  - Use of non-static method from static context (16; 16)
- Method: Incorrect method declaration (87; 46)
  - Method declaration: Missing comma in parameters (0; 0)
  - Method declaration: missing method body (2; 2)
  - Method declaration: missing return type (10; 10)
  - Method declaration: no type specified for parameter (2; 2)
  - Method declaration: return type should be void (2; 2)
  - Method declaration: Semicolon at end of method header (1; 1)
  - Method declaration: semicolon in place of comma (0; 0)
  - Method declaration: Wrong number of parameters (5; 5)
  - Method duplicated exactly (2; 2)
  - Missing opening curly brace after method header (2; 2)
  - Wrong return type declared (15; 15)
- Constructor: Incorrect constructor call (9; 2)
  - Constructor call attempted using variable name (0; 0)

  - Constructor call to non-existent copy constructor (0; 0)
  - Constructor call without using 'new' (0; 0)
  - Constructor parameter number mismatch (5; 5)
  - Constructor parameter type mismatch (1; 1)
  - Missing closing parenthesis in constructor call (0; 0)
  - Missing parentheses for constructor call (1; 1)
  - Missing space after 'new' (0; 0)
  - Object instantiation uses variable name (0; 0)
- Constructor: Incorrect constructor declaration (4; 4)
  - Call to 'super' in constructor not first statement (0; 0)
  - Constructor is declared to return 'void' (0; 0)
- Incorrect/attempted use of class or type (66; 5)
  - Class from other package used without 'import' (13; 13)
  - Class not defined (4; 4)
  - Class or type name written incorrectly (44; 44)
  - Use of variable in context requiring a type (0; 0)
- Semantic error (119; 49)
  - Assignment to a collection element (0; 0)
  - Class does not implement required method (2; 2)
  - Class should be declared to implement interface (0; 0)
  - Missing 'return' statement (23; 23)
  - Type error (59; 38)
    - Array used in place of array element (8; 8)
    - Collection used in place of element (0; 0)
    - Invalid type cast (1; 1)
    - Type mismatch in assignment (12; 12)
    - Type mismatch: arithmetic performed on String; use length (0; 0)
    - Value from raw collection must be cast (0; 0)
  - Unhandled exception (7; 7)
  - Variable access used as statement (0; 0)
- Simple syntactical error (201; 66)
  - '=' used in place of '==' (6; 6)
  - : in place of ; (1; 1)
  - ; missing (74; 74)
  - Comment incorrectly written (1; 1)
  - Extra closing parenthesis (0; 0)
  - Extra opening curly brace '{' (2; 2)
  - Extraneous '(' between 'else' and 'if' (0; 0)
  - Extraneous closing curly brace (21; 21)
  - Extraneous or misplaced ')' (2; 2)
  - Keyword written incorrectly (4; 4)
  - Mismatched parentheses in or around expression (10; 10)
  - Missing '.' between names in qualified name (0; 0)
  - Missing closing curly brace (27; 2)
    - Missing closing curly brace after control stmt body (6; 6)
    - Missing closing curly brace at end of class (12; 12)
    - Missing closing curly brace at end of method (7; 7)
  - Missing operator between expression elements (12; 12)

- Statement outside method/block (2; 2)
- Uncategorised (94; 94)

D ERROR CATEGORY SEVERITY

Table 9. Error Category Severity

Category	Occurrences	Severity
Variable not declared	85	4.763
Variable: Incorrect variable declaration	48	4.564
Simple syntactical error	66	4.437
Method: Incorrect method declaration	46	4.419
Semantic error	49	4.111
Type error	38	3.506
Variable name written incorrectly	75	3.234
Missing 'return' statement	23	2.996
Method call targeting wrong type	25	2.588
Method not declared	29	2.586
Variable: Incorrect attempt to use variable	47	2.494
; missing	74	2.244
Method name written incorrectly	47	2.218
Class or type name written incorrectly	44	2.214
Method call: parameter number mismatch	27	2.159
Use of non-static method from static context	16	1.484
Method: Incorrect method call	20	1.343
Method call: parameter type mismatch	10	1.088
Type mismatch in assignment	12	1.053
Extraneous closing curly brace	21	1.033
Missing parentheses for method call	21	0.912
Incorrect/attempted use of class or type	5	0.838
Class from other package used without 'import'	13	0.793
Mismatched parentheses in or around expression	10	0.682
'=' used in place of '=='	6	0.635
Method declaration: missing return type	10	0.632
Keyword written incorrectly	4	0.628
Missing closing curly brace after control stmt body	6	0.628
Wrong return type declared	15	0.624
Method declaration: missing method body	2	0.600
Constructor parameter number mismatch	5	0.568
Use of variable from another class	4	0.521
Missing operator between expression elements	12	0.497
Variable needs a unique name	9	0.461
Constructor: Incorrect constructor declaration	4	0.437
Missing closing curly brace at end of method	7	0.401
Missing closing curly brace at end of class	12	0.396
Class does not implement required method	2	0.387
Class not defined	4	0.354
Constructor parameter type mismatch	1	0.300

: in place of ;	1	0.300
Invalid type cast	1	0.300
Use of non-static variable from static context	5	0.161
Method call: Parameter types included	4	0.156
Unhandled exception	7	0.125
Extra opening curly brace '{'	2	0.117
Method duplicated exactly	2	0.113
Wrong variable used	4	0.112
Array used in place of array element	8	0.110
Method declaration: Wrong number of parameters	5	0.093
Missing opening curly brace after method header	2	0.078
Method declaration: Semicolon at end of method header	1	0.071
Constructor: Incorrect constructor call	2	0.062
Missing closing curly brace	2	0.045
Method declaration: return type should be void	2	0.029
Statement outside method/block	2	0.028
Extraneous or misplaced ')'	2	0.016
Missing parentheses for constructor call	1	0.005

Note: The severity is calculated as a product of adjusted average resolution time and occurrence count. The adjusted average resolution time is the average of all resolution times when unresolved errors are considered as resolved in 300 seconds (i.e. the upper limit for resolution); instances with uncertain resolution do not take part in the calculation. Categories with occurrences all having uncertain resolution status are excluded.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An Analysis of Patterns of Debugging Among Novice Computer Science Students. *SIGCSE Bull.* 37, 3 (June 2005), 84–88. https://doi.org/10.1145/1151954.1067472

[2] Brett A Becker, Kyle Goslin, and Graham Glanville. 2018. The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, 640–645.

[3] Brett A Becker and Catherine Mooney. 2016. Categorizing compiler error messages with principal component analysis. In *12th China-Europe International Symposium on Software Engineering Education (CEISEE 2016), Shenyang, China*. 28–29.

[4] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[5] Neil CC Brown and Amjad Altadmri. 2014. Investigating novice programming mistakes: educator beliefs vs. student data. In *Proceedings of the tenth annual conference on International computing education research*. ACM, 43–50.

[6] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*. ACM, 223–228.

[7] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. 2000. Alice: lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 486–493.

[8] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 273–278.

[9] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education*. ACM, 75–80.

[10] T Flowers, CA Carver, and J Jackson. 2004. Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*. IEEE, T3H–10.

[11] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, Vol. 35. ACM, 153–156.

[12] James Jackson, Michael Cobb, and Curtis Carver. 2005. Identifying top Java errors for novice programmers. In *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*. IEEE, T4C–T4C.

[13] Matthew C Jadud. 2005. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education* 15, 1 (2005), 25–40. https://doi.org/10.1080/08993400500056530 arXiv:http://dx.doi.org/10.1080/08993400500056530

[14] Andrew J Ko and Brad A Myers. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing* 16, 1-2 (2005), 41–84.

[15] Michael Kölling. 2010. The Greenfoot Programming Environment. *Trans. Comput. Educ.* 10, 4, Article 14 (Nov. 2010), 21 pages. https://doi.org/10.1145/1868358.1868361

[16] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. 2015. Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education (WiPSCE '15)*. ACM, New York, NY, USA, 29–38. https://doi.org/10.1145/2818314.2818331

[17] Michael Kölling, Bruce Quig, Andrew Patterson, and John Rosenberg. 2003. The BlueJ system and its pedagogy. *Computer Science Education* 13, 4 (2003), 249–268.

[18] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Trans. Comput. Educ.* 10, 4, Article 16 (Nov. 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[19] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011)*. ACM, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

[20] D. McCall and M. Kölling. 2014. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*. 1–8. https://doi.org/10.1109/FIE.2014.7044420

[21] Dermot Shinners-Kennedy and Sally A. Fincher. 2013. Identifying Threshold Concepts: From Dead End to a New Direction. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research (ICER '13)*. ACM, New York, NY, USA, 9–18. https://doi.org/10.1145/2493394.2493396

[22] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2, Article 8 (July 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[23] James C Spohrer and Elliot Soloway. 1986. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM* 29, 7 (1986), 624–632.

[24] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2013. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *2013 IEEE 13th International Conference on Advanced Learning Technologies (ICALT)*. IEEE, 319–323.

Received February 2007; revised March 2009; accepted June 2009