Novice Programmer Errors – Analysis and Diagnostics

A thesis submitted to
The University of Kent
in the subject of Computer Science
for the degree of PhD.

By
Davin McCall

December 2016

ABSTRACT

All programmers make errors when writing program code, and for novices the difficulty of repairing errors can be frustrating and demoralising. It is widely recognised that compiler error diagnostics can be inaccurate, imprecise, or otherwise difficult for novices to comprehend, and many approaches to mitigating the difficulty of dealing with errors are centered around the production of diagnostic messages with improved accuracy and precision, and revised wording considered more suitable for novices. These efforts have shown limited success, partially due to uncertainty surrounding the types of error that students actually have the most difficulty with – which has most commonly been assessed by categorising them according to the diagnostic message already produced – and a traditional approach to the error diagnosis process which has known limitations.

In this thesis we detail a systematic and thorough approach both to analysing which errors that are most problematic for students, and to automated diagnosis of errors. We detail a methodology for developing a category schema for errors and for classifying individual errors in student programs according to such a schema. We show that this classification results in a different picture of the distribution of error types when compared to a classification according to diagnostic messages. We formally define the severity of an error type as a product of its frequency and difficulty, and by using repair time as an indicator of difficulty we show that error types rank differently via severity than they do by frequency alone.

Having developed a ranking of errors according to severity, we then investigate the contextual information within source code that experienced programmers can use to more accurately and precisely classify errors than compiler tools typically do. We show that, for a number of more severe errors, these techniques can be applied in an automated tool to provide better diagnostics than are provided by traditional compilers.

Table of Contents

1 Introduction

Programming is a fundamental part of computer science, and learning to effectively write and debug computer programs forms a significant part of any comprehensive computer science course. However, programming language compilers have historically tended to produce diagnostic messages – commonly referred to as *error messages* – that are not suitable for novice programmers. The nature of this unsuitability is well understood; the messages are sometimes poorly worded; they make reference to technical terms and they use language related to the task of translating the source code into a more machine-level form (be it bytecode or processor instructions) – tasks which are well outside the scope of introductory programming studies; they often diagnose errors based primarily on the point in that process at which the error is detected, leading to inaccurate descriptions of the error; they are often terse and imprecise. In brief, they do not succinctly provide accurate information to the novice programmer regarding what they did wrong and how they can fix it in a manner that is comprehensible to them.

As well as being unhelpful to novices, the imprecision and inaccuracy of compiler diagnostics are problematic for educators who wish to determine the types of error that their students encounter. Automated logging of diagnostic messages is often a relatively straightforward matter, but the diagnostics are often too limited to be useful in helping teachers to adjust their teaching by, for instance, focussing on problem areas; many logically different errors – leading from different misconceptions on the part of the novice – can potentially produce the same diagnostic message. Furthermore, frequency counts of diagnostic messages do not reveal which of the errors related to those messages presented the most difficulty to the novices tasked with resolving them.

Certainly many of the shortcomings of a compiler's diagnostic capabilities can cease to be a significant impediment as the user gains experience, but in dealing with novices, it would be beneficial to improve the situation. It can be argued even experienced programmers can benefit from improved diagnostic messages when, for instance, they are learning a new programming language.

Over a span of many years, various efforts have been made to produce automated diagnostics that are more suitable for novices, for a variety of programming languages (Schorsch 1995, Hristova et al. 2003, Flowers et al. 2004, Dy and Rodrigo 2010). While these past efforts have, in general, claimed at least limited success, they do not appear to have had a significant ongoing impact on the teaching and learning of computer programming. After all these years, error messages are still problematic: students are still getting stuck, error messages are still known to be imperfect. Complaints today of teachers are the same as they were “then”. While it’s impossible to have a message always be precise and correct, there is a general feeling among educators that improvement can be made.

The question arises, then, why previous work to improve diagnostics has failed to have a lasting effect on the quality of diagnostics produced by software tools such as compilers. We posit two reasons as to why this is the case:

1. The production of a compiler is a significant endeavour. It is often undertaken by commercial organisations, though there are exceptions to this, and normally the product is targeted primarily at experienced programmers whose need for precise diagnostics is somewhat less than that of a novice; furthermore, other aspects of the compiler – such as performance of compilation and of the produced object code – are normally prioritised, since these are generally more important to most users.

2. The approach to improving diagnostics often appears to have been somewhat ad hoc. Rather than modify or produce a compiler, it has often revolved around a separate analysis/pre-processing tool which identifies and reports a limited number of errors. It is often not clear from the relevant literature how the decision of which particular errors should be diagnosed was reached, nor of how the diagnosis procedure itself operated (and whether it was indeed sufficiently able to diagnose errors with greater accuracy and precision that methods traditionally used in compilers).

The first point remains pertinent to some degree, but as languages evolve and become more complex, the need for better diagnostics also increases, even for experienced programmers. Also, increases in computing power mean that the cost in terms of compiler performance of providing more accurate and precise diagnostics is less of a concern than it formerly was.

It is this second point which we shall address in this thesis. The first key issue is the development of a method for identifying high-impact errors for which good diagnosis is particularly important. A number of prior studies (Ahmadzadeh et al. 2005, Jadud 2005, Tabanao et al. 2011) have categorised errors by the diagnostic message generated by a compiler, and the frequency of the presentation of different diagnostics has been used as an indicator of the impact of the error category. This approach is potentially flawed in two ways: firstly, use of diagnostic message as a classifier for errors limits the accuracy and precision of error categorisation due to the limited diagnostic capabilities of the compiler, as already discussed; secondly, the frequency of an error category may not directly correlate to its real *severity*, which should be considered as a combination of the frequency with which a type of error is encountered as well as the actual difficulty that it presents to novices in their attempts to resolve these errors (the notion of error severity will be discussed in more detail in the following chapter).

This work provides two key contributions. First, the development of a categorisation scheme for errors which provides greater accuracy and precision of error classification than that achieved using diagnostic messages, together with a study measuring the severity (including the frequency) of errors in the different categories as made by novices. The study applies manual categorisation to avoid the pitfalls presently associated with automated diagnosis. The category scheme is applicable to the Java programming language, and the methodology used to develop it could equally be applied to other programming languages. This study and others based on similar methodology could be useful in identifying areas of programming that students most struggle with, guiding development of instructional material to better prepare students for their encounters with particularly high-severity errors, or to help them avoid making such errors in the first place by emphasising instruction in problematic topics.

In addition to its benefits as a guide for pedagogy, a severity ranking for error types should be considered essential for the design of tools designed to diagnose errors in novice code. The second key contribution of this work is a prototype implementation of a tool designed to diagnose high-severity errors, designed to illustrate the feasibility of developing tools with more accurate and precise diagnostics of high-severity errors than is normally performed by a compiler. The tool performs more sophisticated analysis than has been seen in many previous efforts, and is designed to diagnose errors using heuristics based on reasoning and experience of code structure and patterns.

1.1 Hypotheses

We have discussed the use of logical error categories which provide a distinct classification compared to the compiler-generated diagnostic message. This leads to our first hypothesis:

> **Hypothesis 1** : *Measuring the frequency of logical error categories will yield different results than diagnostic message frequency*.

By "different results" we mean that there will be no marked one-to-one correlation between categories in a categorisation relying on diagnostic message when compared to a categorisation relying applying a category scheme developed to describe errors by their logical nature as ascertained by a suitably experienced programmer.

In Chapter 4.1, we describe the methodology for a study to determine the feasibility of applying manual categorisation of errors using a category scheme developed with a methodology based on the established principles of Thematic Analysis (Braun and Clarke 2006). The results (Chapter 5) show a markedly different distribution of the frequency of manual error categorisations and diagnostic messages.

Since a different distribution between two categorisation schemes does not automatically imply a greater level of accuracy of either, we also wish to demonstrate that the logical categorisation scheme can be considered accurate. This leads to a sub-hypothesis:

> **Sub-hypothesis 1.1** : *A suitably experienced human can reliably recognise the cause of an error in many cases, even when the compiler gives a misleading error message*.

By "suitably experienced human" we mean a person with several years of frequent programming experience and who would self-describe as an experienced programmer. By "reliably recognise the cause of an error" we mean that several such experienced programmers should show a good level of agreement (80% or more) when categorising the cause of errors.

We have also already posited that frequency of an error category does not necessarily indicate its severity, which also takes the difficulty of error resolution into account. This leads to a second hypothesis:

> **Hypothesis 2** : *Error frequency tables are misleading in the sense that they do not accurately represent the most difficult to handle errors; with a different measure, where a severity calculation is made, a different result will ensue*.

In Chapter 2.7 we discuss error severity and formally define severity as a product of frequency and difficulty. In Chapter 4.2.3 we discuss the calculation of difficulty as an approximate average error resolution time, making severity of an error category a relative indicator of the proportion of time spent resolving errors belonging to the category. The results of analysis, including both error category frequency and severity scores, are presented and discussed in Chapter 6.

Our final hypothesis is centred around the idea of generalising human reasoning about error cause so that it can be applied in an automated tool:

> **Hypothesis 3** : *in some cases reasoning that humans make to decide error cause can be implemented as a heuristic in an automated tool to produce more useful diagnostic messages.*

In Chapter 2.6 we discuss human reasoning in error diagnosis and provide examples of error diagnosis using expert reasoning. In Chapter 7 we present the design, development and evaluation of a proof-of-concept tool prototype which uses comprehensive analysis and heuristics derived from reasoning and understanding of program structure to diagnose certain categories of high-severity error.

1.2 Summary of Contributions

The two primary contributions of the thesis were mentioned previously:

- The development of a categorisation scheme for errors which provides greater accuracy and precision of error classification than that achieved using diagnostic messages, and a related study which identifies and ranks the most severe errors encountered by novice Java programmers. This study and related methodology may lead to a better understanding of programming errors by novice programmers.
- A proof-of-concept implementation of a tool for diagnosing a range errors in categories which were found to be of high severity, using heuristics derived from reasoning about the natural diagnosis of these error categories.

Other contributions include:

- The formulation of error category severity as a product of its frequency and difficulty;
- The representation of error category severity as a point on a 2-dimensional plane with axes of frequency and difficulty, allowing the factors determining severity to be visualised;
- A methodology for development of an error category scheme, suitable for adaptation to any programming language;
- A method for approximating the average resolution time of error categories, based on the classification of individual errors to various resolution statuses and estimating individual error resolution time based on the effect of source code changes to the generated compiler diagnostic;

- A number of heuristic techniques that can be used for the differentiation, in an automated diagnosis, between several categories of error that can occur in the Java programming language; and
- The proposal of a *declaration-use match score*, for assessing the appropriateness of matching a declaration with a definition (where the spelling of the identifier does not match exactly) as part of the heuristics.

1.3 Thesis Outline

In this chapter we have briefly discussed the context as well as the motivation behind and contributions of the work presented in this thesis.

In the next chapter, we will discuss in more detail some background to the issues which this work addresses, serving as a more thorough exploration of both context and motivation, and define some terminology that will be used throughout the remainder of the thesis.

In Chapter 3, we present a review of literature relevant to the work presented in this thesis.

In Chapter 4, we present the methodology used for the development of a category scheme for programming errors (including a validation of this scheme), the categorisation of a large number of errors from sources known or highly likely to be novice programmers, and the analysis data after categorisation. This was performed in two distinct phases, the first both as a viability study for the method and to compare the frequency results of manual error categorisation with automated categorisation using standard compiler diagnostics, the second as a larger-scale study which also measured error severity.

In Chapter 5, we present and discuss results from the first phase of the study, showing that manual error categorisation could be performed with high level of agreement between separate suitably-experienced individuals, and that manual categorisation of errors can be more precise and accurate than classification via compiler diagnostic.

Chapter 6 details the results of the error severity study. We show that an error severity ranking of error categories produces a markedly different ordering than a ranking by frequency alone.

In Chapter 7, we discuss the design and evaluation of a prototype tool for automatic error diagnosis of several high-severity errors categories, intended to serve as a study of feasibility for diagnosing such errors using more advanced techniques than what are typically employed in such tools and using heuristics developed by reasoning about diagnosis of the relevant errors.

Finally, in Chapter 8, we discuss future work and offer concluding remarks regarding the evidence gathered in support of the hypotheses presented earlier in this chapter.

2 Background

2.1 Introduction

The work presented in this thesis is centred around two distinct but related themes: identification and categorisation of errors made by novice programmers, and use of expert-identified aspects of error context to enable accurate and precise error diagnosis. In this chapter, an overview of important related concepts will be presented; terminology will be defined, and the notion of an *error* as distinct from a *diagnostic message* and other key concepts will be discussed. Extant problems with diagnostic messages issued by current compilers will be highlighted together with novice responses to these messages that demonstrate a need for improvement of diagnostic capability and reporting in these tools.

We also discuss some aspects of typical compiler design that result in poor diagnostic capability, and briefly examine how an experienced programmer is often able to make a more informed error diagnosis than that performed by a compiler. Finally, we discuss categorisation of errors, and why the diagnostic message is an unsuitable proxy for the underlying error, as well why the frequency of an error category alone is not always a good indicator of how much of a problem errors in a particular category are for novices.

2.2 Novice Programmers and Errors

It has been widely recognised in literature that novice programmers in the early stages of programming courses often struggle with language syntax rules when they write code; similarly, certain misunderstandings of semantics – whether of particular constructs, or of a more general nature – are known to occur amongst novices, and these misunderstandings lead to code which fails to work correctly or which fails to compile. For example, Shackelford and Badre (1993) observed a high rate of failure for students asked to produce code implementing simple calculation tasks, asking *why can't smart students solve simple programming problems*? Similarly, Flowers et al. (2004) noted that in a Java course conducted at a military academy:

> *Many cadets are challenged by the complexity of the assignments and the difficulties of understanding the syntax of the Java programming language. Instructors noted that students made the same mistakes and became frustrated trying to understand the error messages and correct their code, often wasting hours of time on a simple error.*

The identification of diagnostic (“error”) messages as being problematic for students is a recurring theme, as is the difficulty in understanding language syntax. These issues are not unrelated, since diagnostic messages have the potential to assist a programmer in correcting the syntax in their programs – a task that otherwise consumes the time of the novice programmer and, indirectly, their instructors, and which potentially causes much frustration to both.

The potential benefits of improved error diagnostics are broadly recognised; a significant number of prior work exists that attempts to improve the diagnostic messages given to the novice programmer when an error is detected within their code (Flowers et al. 2004, Dy and Rodrigo 2010, Hristova et al. 2003). Often the approach has involved an ad-hoc (or at least undescribed) design attempting to address a somewhat arbitrarily selected set of errors that are perceived to be problematic, and while many have shown some positive results, it is feasible that a more rigorous approach to the selection of an error set and to the method of diagnoses of those errors would yield significant further improvement.

2.3 Terminology

It is helpful at this point to discuss terminology and to distinguish *diagnostic messages* from actual *errors* as well as *programmer misconceptions*. The term *error* has taken on multiple common meanings, including both “the diagnostic message produced by the system when it encounters an error” (also often referred to as an *error message*) and “the nature of the problem with the code” (often characterised by the nature of the mistake made by the programmer); to avoid ambiguity, the term “error message” shall be avoided in this work, and the following precise terms shall be used as now defined:

Error – the nature of the problem in source code which causes compilation or execution failure

Diagnostic Message – the human-readable message produced by a compiler when it encounters an error (syntactic, semantic or logical) in source code which prevents compilation.

Programmer Misconception – the nature of the misunderstanding or lack of comprehension (of the programmer) which resulted in the erroneous code being written.

Using these terms according to their definitions above greatly simplifies the discussion that follows. In this thesis the term “error” should always be considered to refer to the nature of the problem rather than to a message produced by a compiler or interpreter.

2.4 Relationship of concepts

As defined in the previous section, the *programmer misconception* is the nature of the misunderstanding of the programmer which caused the erroneous code to be written. The latter can be divided into two categories:

1. Syntactical, in which the programmer does not understand the syntax required to express the desired construct; an example would be where the programmer omits the parentheses around the condition in a Java 'if' statement.
2. Semantic, being a misunderstanding of the effect of some syntactical construct. This includes cases where the programmer has a flawed or incomplete mental model of the notional machine on which their program will execute. An example is failing to understand that a method-local variable does not retain its value between successive calls to the method.

Note that the occurrence of an error may not necessarily be due to a misconception on the part of the programmer; it may instead be caused by a simple typographical error or misspelling, and there also *logical* errors which may result from an incorrectly implemented algorithm or other mistake not directly caused by a fundamental misconception.

An error always has some manifestation in the source code of the program. Often only a small part of the source code may be truly relevant to the error, but strictly speaking, the *error manifestation* consists of the entire source code (which of course may contain multiple errors).

An individual error is most easily distinguishable by its *fix*, that is, the change or changes to the source code that must be made to remove the error; note that several possible changes may allow code to compile, but not all correspond to a correct fix of the original error.

Any diagnostic messages are produced by the compiler based purely on the error manifestation (the source code); it is possible that a single error may cause one compiler to produce multiple diagnostic messages, and it is also possible that different compilers will produce different diagnostic messages from the same source code. The relationship between these concepts is illustrated in

Figure 2.1.

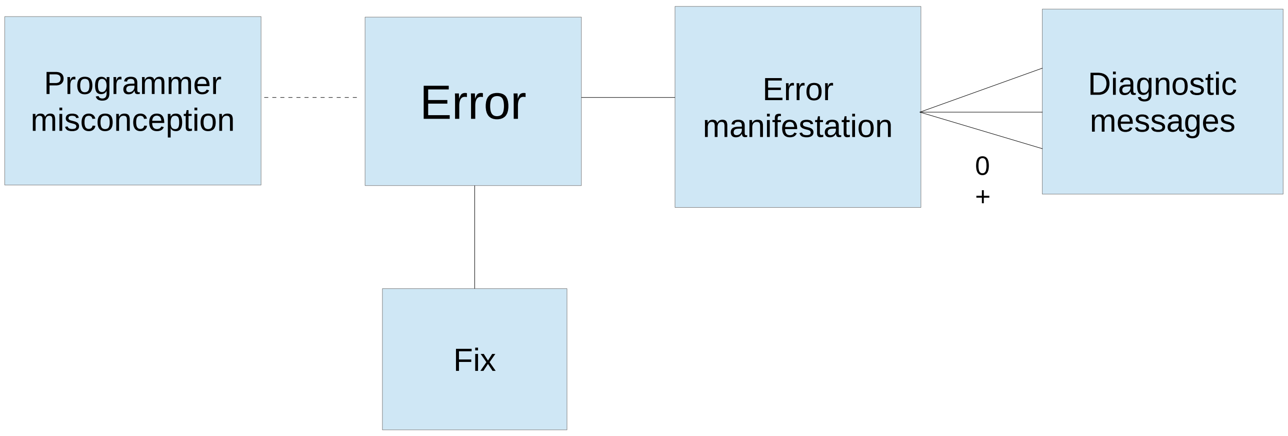


Figure 2.1: Relationship between an error and other concepts

For example, consider the snippet of Java code presented in Figure 2.2, which is intended to define a method. Assume the code was written by a novice programmer.

```
int getValue();
{
     return value;
}
```

Figure 2.2: Example of erroneous code. There is a superfluous semicolon between the method header and method body.

For the code in Figure 2.2, the *javac* compiler produces the following diagnostic message:

missing method body, or declare abstract

The message, which is poorly worded, offers one possible error cause – that the method body is missing – and one unrelated possible solution (declare the method as abstract); the actual *error* is that there is an extraneous semi-colon between the method header and the opening curly brace which marks the beginning of the method body. The *fix* for the error is to remove the extraneous semi-colon. The *programmer misconception* is indeterminable; possibly this was just a typographical error, or possibly the programmer does not have a correct grasp on the syntax and in particular does not understand the significance of the semi-colon.

Note that not all errors need be caused by misconceptions on the part of the programmer. In general, the *programmer misconception* is indeterminable; the error in Figure 2.2 might have been caused by a misunderstanding of the grammar rules regarding method declaration and in particular the use of semi-colons, but might also have been a simple typographical error that was overlooked by the user before they attempted compilation. Similarly, misspelling a keyword or an identifier might cause a compilation error without being caused by a misunderstanding of any language rules or programming concepts.

Also, not all programmer errors result in compile-time errors – a semantic misconception may result in code which compiles, but does not perform as intended when executed. Errors may also be *logical* in nature – they may not be due to any conceptual misunderstanding regarding syntax or semantics, and may not cause the compiler to produce an error message, but will have the effect of causing the program to behave in a way that was not intended. All errors have a *manifestation* – they exist in some form in the source code – but two different errors might have the same manifestation. The most obvious occurrence of this is when a complete program source code (manifestation) contains multiple distinct errors; an example is given in Chapter 4.1.4.1. It is also logically possible that two subtly distinct but correct programs could both have an error introduced which resulted in identical erroneous source code, requiring a different fix in each case, though we have not observed this occurring in practice and cannot conceive of a likely example.

Ideally, a diagnostic message makes a reasonable determination of the likely error, and offers an appropriate fix where possible. In practice, as has been shown, this is not always the case.

2.5 Diagnostic Messages

In an ideal world, after a compilation failure, the diagnostic message given to the programmer would neatly explain the conceptual misunderstanding or mistake that they had made, and offer straightforward advice on how to fix the resulting problem. In practice, though, this seems to be the exception rather than the rule; far from being generally helpful, diagnostic messages produced by compilers have a tendency to be confusing or misleading, especially for novices (Marceau et al. 2011, Pears et al. 2007).

Typical diagnostic messages may:

- contain technical terms with which the novice is unfamiliar;
- be inaccurate (state an incorrect error cause);
- be imprecise (fail to give sufficient detail about the error); or
- suggest an incorrect solution (potentially resulting in further errors).

An experienced programmer may learn to distinguish inaccuracies in diagnostics (and indeed may reach a point where the diagnostic message itself is hardly needed – it is enough to be told “an error was detected here”, which is all that an expert programmer may need to quickly fix an error in their code). However, it has been observed (Marceau et al. 2011) that novices sometimes assume a compiler diagnostic message is correct, and follow its suggestions, even when it it appears patently not so to a more experienced programmer; the same study also noted that other students seem to blithely ignore the message provided by the compiler, a behaviour also observed by Kummerfeld and Kay (2003), who suggest that it may be due to messages being incomprehensible to the students. This shows that when presented with an error message from a compiler, a novice may:

- consider the message to be true and correct, even though it may in fact be incorrect or misleading; or
- disregard the message entirely, perhaps not reading it at all, believing either that it is incorrect or otherwise unhelpful.

For novices in the first category at least, it is clear that improving accuracy of diagnostic messages could be helpful. The precision of the diagnostics is also a factor, however; a vague description of an error will not suggest a useful solution. Schneiderman (1982) reported that, in one experiment, increasing the specificity of messages in a Cobol compiler led to an improvement of 28 percent in the repair scores of novice undergraduate users.

2.5.1 Accuracy and Precision

There is in general a trade-off between accuracy and precision of diagnostic messages. The more precise a given diagnostic, the less likely it is to be an accurate and correct description of the error. Ideally, of course, a diagnostic message has both a high level of precision and accuracy. The most general form of a diagnostic - “there is an error” - can never be inaccurate (barring compiler bugs), but is of no help to a novice programmer due to its lack of precision. On the other hand a more precise diagnostic, such as the “missing method body, or declare abstract” response for the code in Figure 2.2, runs of the risk of being inaccurate (as in that example) and thereby also unhelpful; an inaccurate message may lead to a novice making an erroneous change in their source code, leading to further compilation errors or unwanted behaviour at runtime.

It is not possible to give an accurate and precise diagnosis of an error with total certainty without knowing the intended purpose of a program – or at least without knowing the possible correct program structures, as in the work of (Spohrer et al. 1985) – but an experienced programmer can often do so with a high degree of confidence even when presented with code “from the wild”, using clues in the structure and other context provided in the code itself, together with a knowledge of common errors. Referring again to Figure 2.2, it can be seen that there is an extraneous semicolon after the method header by observing that a valid method body does occur after the semicolon; thus the message that there is a “missing method body” is incorrect, and the suggested fix - “declare abstract” - would in fact lead to another error.

As stated, there can be a trade-off between accuracy and precision of diagnostic messages. Novice behaviour (Marceau et al. 2011) suggests that accuracy and precision are both important; novices may benefit from messages that correctly identify the problem and suggest how to resolve it. While it has been shown that increasing precision of error messages is beneficial (Schneiderman 1982), it is logical that more specific messages would be helpful only if they are accurate.

2.5.2 Generation of Diagnostic messages

Many of the problems with inaccuracy or imprecision of diagnostic messages may be attributed to the manner in which errors are detected by the compiler (particularly in the case of syntactical errors). A compiler is tasked with converting a program written in a human-readable programming language into a form more directly understood by the machine. The programming language is defined by a formal grammar, and syntax errors are detected during the process of parsing the input source code to form an Abstract Syntax Tree (AST) representing the structure of the program, as a failure of the input to match any production of the language grammar.

Parsing is a complicated though well-studied problem, however, the primary focus of parsers is usually the translation of correctly structured input. Tools for generating a parse from a grammar description (a set of production rules), such as *yacc* (Johnson n.d.) and *ANTLR* (Parr n.d.), do exist, but do not automate the generation of diagnostics beyond the identification of simple grammatical failures. Usually, the production rules for the grammar need to be manually augmented with at least one additional rule for each point-of-failure for which a unique diagnostic is desired, or the diagnostic must be generated from a combination of parse state and lexical token lookahead, which in either case requires significant additional development effort; for example, Jeffery (2003) noted that in the Java grammar used by the GCC compiler ('GCC, the GNU Compiler Collection' n.d.), “150 error productions were added to approximately 350 nonerror grammar rules”.

Development of useful error diagnostics tends to be relegated to a lower priority than other concerns such as compilation speed and optimisation of the output (Traver 2010). Typical compiler design theory (Grune et al. 2012) focuses on compilation as purely a process of converting a textual source program into a series of lexical tokens, an abstract syntax tree and finally into target code; there is scant attention paid to the mechanism for reporting of errors – Grune et al. (2012, pp.121) comment that “Some advanced error handling methods consider the entire program when producing error messages, but after 30 years these are still experimental, and are hardly ever found in compilers”. Accordingly, most compilers report errors at the point of detection, and naturally refer to the detected deviation from the language grammar or semantic rules. It is common to see diagnostics of the form “*token-list* expected”, where a set of lexical tokens that would have been legal at a certain point in the source program are listed when a token not in that list was present; however, the real error may manifest as an incorrect token prior to the point of detection.

Similar to the way in which syntax diagnostics tend to assume that previous syntax was correctly specified, semantic errors detected by compilers may be diagnosed based on the assumption that the syntax was used correctly. That is, if the source code appears valid grammatically, then it is assumed that the intended semantics match those implied by the grammar. In Java, for example, a method call is differentiated from a variable use by the presence of a parenthesised argument list; if the argument list is missing, the code is still grammatically correct, and can be parsed as a variable reference, potentially resulting in misdiagnosis as a reference to an undeclared variable by a compiler with typical design.

2.6 Expert Diagnosis of Errors

In Figure 2.2 we showed a small code fragment which resulted in an unsatisfactory diagnostic message from the compiler. The notion that the diagnostic is unsatisfactory stems from our own ability, as experienced programmers who are familiar with the programming language, to make an educated assessment of the likely nature of the error. In some cases this nature may not be clear; the error may be undiagnosable, or may be diagnosable as a set of possible error types (an imprecise diagnosis). In other cases, such as in the code in the example, the error seems unambiguous to an experienced programmer, as various contextual clues in source code to assist with diagnosis of possible errors.

In the example, the method declaration in the first line would be valid in a Java interface, but is missing the “abstract” keyword that is required if it were a declaration of an abstract method in a Java class. There is a complete method body below the method header, however; we can reason that it is a method body, and not a static initialiser block, because it contains a “return” statement which is valid only in the former. The presence of a method body almost immediately trailing the method header suggests that the two are associated, and that the semicolon after the method header (which acts as a separator) should not be present. This reasoning relies on slightly broader context than the parse state / lookahead combination from which a compiler may decide a diagnosis.

Another example of using broader context to provide better diagnostics is identification of misspelled variable and method names. If a reference is made to an undefined variable or method with a name that almost matches that of an existing variable or method, then misspelling should be identified as a potential cause of the error. Consider the Java class shown in Figure 2.3.

```
class A {
     public int value = 5;
     public int getValue()
     {
          return vallue;
     }
}
```

Figure 2.3: Example of mis-spelt variable name. The variable is declared as ‘value’ and then referred to as ‘vallue’.

The source code shown in Figure 1.1 does not compile, due to the variable 'vallue' not being defined in the scope within which it is referenced. However, there is a variable with a very similar name – 'value' – and it is reasonable to think that the programmer intended to refer to this variable instead. A compiler could suggest this misspelling as a possible error cause. The Gauntlet system (Flowers et al. 2004) does this in at least some cases, as do the BlueJ (Kölling et al. 2003) and Greenfoot (Kölling 2010) environments.

The diagnostic reasoning outlined in the two cases above are not definitive, but serves as examples of potential reasoning that might be used by an expert to diagnose an error; in chapter 4 we discuss the collection of data pertaining to the expert reasoning for diagnosis of a large number of errors.

2.7 Categorisation of Errors and Determining Severity

For pedagogical guidance and for the development of tools to be used in a pedagogical environment, it is useful to have an understanding of the types of error that students make most often and have the most difficulty with. This requires the categorisation of errors according to some category schema. Several prior studies (Ahmadzadeh et al. 2005, Jadud 2005, Tabanao et al. 2011) have relied on diagnostic messages to categorise errors; however, relying on diagnostic messages as a categorisation of errors is prone to problems due to their potential inaccuracy and imprecision, as well as the variation of diagnostic messages between compilers and compiler versions. To summarise the potential issues:

- Messages may be inaccurate – they may suggest an incorrect error cause and/or corrective action;
- Different diagnostic messages might be generated for conceptually identical errors occurring in different contexts within the program source;
- Diagnostic messages are compiler dependent; different compilers (including different versions of the same compiler) may produce different messages for the same erroneous source code, as demonstrated in the discussion of the studies above; and
- Even by the a single compiler, The same diagnostic message might be generated for conceptually different errors.

Inaccuracy of diagnostic messages is a significant issue when collating information on novice errors; messages often give an incorrect error cause. In fact, a compiler might not even correctly determine whether an error is syntactic, semantic or logical in nature, since a misunderstanding of syntax rules on the part of the programmer could lead to an error which the compiler perceives as a semantic error, for instance; as an example of this, using the wrong delimiter between multiple arguments in a method call might lead the compiler to interpret them as a single argument, generating an error due to the number of supplied arguments being incorrect.

Furthermore, different compilers produce different diagnostic messages for the same source code. The wording of what is essentially the same diagnostic may differ, but more importantly, there can be errors which one compiler will diagnose as one particular type of error while another compiler would diagnose the same errors as another type of error.

In general, there is a many-to-many mapping between conceptual error and diagnostic message; the frequency of different messages cannot be used to reliably determine the frequency or range of conceptual errors. For this reason, statistics on diagnostics do not necessarily give a good indication of the types of error that novices actually encounter.

It is also important to note that the frequency of an error type does not by itself necessarily indicate the severity of that error type, in terms of the overall impact on student programmer workload and throughput. Certain types of error might be encountered frequently by novices, but be repaired by them easily in most cases; other error types might be less common, but be far more difficult for the novice to resolve. The severity of an error category can be considered as a product of both frequency and difficulty:

$$\textit{severity} = \textit{frequency} \times \textit{difficulty}$$

Defining severity this way is useful because this formula gives desirable output for particular ranges of input values: an error type which does not occur has zero frequency and so zero severity; an error type that occurs frequently and has a high *difficulty* will have a higher severity than another error type with the same difficulty that occurs less frequently, or with the same frequency but lower difficulty, Importantly, if difficulty is an approximation of the average time spent resolving each occurrence of an error type (as we do in our analysis, described in Chapter 4.2.3), then severity of different error types can indicate the relative time spent resolving errors of each type. A novice is more likely to become frustrated when dealing with errors of a type that require more total time to resolve, whether because these errors are encountered very frequently or because they require time to resolve.

To avoid the problems of error categorisation using diagnostic messages as outlined above, in this thesis we will present a methodology (chapter 4) for development of error categories which requires manual classification of errors, allowing for more accurate, though more time-consuming, categorisation, and a method for measuring error difficulty to allow the severity calculation described above.

2.8 Background Summary

Previously in this chapter we discussed the relationship between errors and diagnostic messages, and made a clear distinction between the two; we showed an example where an existing diagnostic was unsatisfactory, and unrelated to the likely error as determined by human reasoning, possibly leading to the confusion of a novice programmer. We showed some examples of human reasoning that lead to better error diagnosis than that given by a compiler. We also discussed the trade-off between accuracy and precision of diagnostic messages and why typical compiler design often leads to inaccurate or imprecise diagnostics. Finally, we discussed why categorisation of errors using diagnostic messages, and why measuring error category frequency without also measuring severity, can paint an imperfect impression of the types of error that most impact novice programmers.

In the next chapter, we will investigate more thoroughly the literature related to these issues, and attempts to address them.

3 Literature Review

3.1 Introduction

In this chapter we will review the literature and previous studies related to the topic of this thesis. This includes material related to the difficulty novice programmers have with understanding programming language syntax and semantics, their interaction with diagnostics, studies on the frequency and kinds of different errors as encountered by novices, the effectiveness of different forms of diagnostic message, and various efforts to provide improved diagnostic messages. We will also discuss various efforts aimed at alleviating the problems of syntax

3.2 The Difficulty of Syntax

One aspect of programming that causes a lot of trouble for beginners is dealing with language syntax. Denny et al. (2011) note that “writing syntactically correct code is a challenge that regularly confronts the novice programmer”, also observing that students who have difficulty understanding syntax may also struggle to understand algorithms and examples that are presented in the pertinent programming language. They conducted a study on students in an introductory Java programming course at University level, which showed that the students who were most able to correctly complete a series of programming exercises were likely to encounter less syntax errors during their attempts to do so than less able students. Also observed was that students who struggled to write compilable code tended to perform poorly in the course, which presents the possibility that making it easier for students to produce compilable code may allow them to perform better.

Kummerfeld and Kay (2003) make a specific claim which lends itself to arguments for both simpler language syntax and better diagnostic messages: that students using a programming language which is unfamiliar to them “waste considerable time correcting syntax errors”. They had a small group of participants with a range of programming experience and familiarity with the language (C) complete a series of exercises. In each of the exercises, participants had to correct one or more syntax errors in a program which was provided to them; in each case the error message provided by the compiler was considered to be unintuitive or likely to cause problems. They found that programmers with more experience in the language showed familiarity with error messages and had strategies for dealing with them; that both experienced and inexperienced programmers sometimes randomly tinkered with the source code when a solution was not obvious to them, that code examples in a reference guide for various error messages were generally helpful to the participants, and that correcting the errors was a time consuming process, sometimes even for experienced programmers.

Regarding specific features of language grammar and meaning, du Boulay (1986) notes some of the syntax (and semantics) in BASIC and Pascal which causes issues for beginners. Some of these can be generalised; for instance, semicolons are used as separators in some Pascal constructs, sometimes in places where the program structure seems to obviate the need for them – the same is true in other programming languages, for instance the semicolons that terminate statements in Java most typically occur at the end of a line of code.

All in all, existing literature makes it clear that language syntax presents a significant problem for novices, and that accurate and comprehensible diagnostic messages are important.

3.3 Programming Difficulties that are not Syntax-Related

Mastering syntax is not the only challenge for the novice programmer. Not only must the correct syntax for various constructs be understood, but the semantics of those constructs must be comprehended as well. Furthermore the programmer must have a good understanding of the underlying *notional machine* (du Boulay 1986); in an imperative programming language for instance, they must understand various concepts such as variables (and how they change or retain value), method calls and control flow; misunderstanding these concepts can cause students to write erroneous code.

Sorva (2013) delivers a review of relevant literature to the concept of the notional machine, and serves as a good introduction to the topic. Amongst other things, a notional machine is an "idealised abstraction of computer hardware and other aspects of the runtime environment of programs", which is particular to a programming paradigm or even language. Misconceptions about the notional machine can occur as students construct their own mental model during the learning process; these misconceptions carry through to the production of errors in code that a student writes. Identifying student misconceptions, then, potentially plays a part in developing appropriate automated diagnosis.

du Boulay (1986) offered a range of common misunderstandings that novice programmers may stumble over. They outlined several kinds of mistake – misapplication of analogies, over-generalisations and misunderstanding of interactions between program parts – and then gave examples of specific mistakes, largely focused on the BASIC and Pascal programming languages. Some of the specific misconceptions identified (for Pascal and BASIC) include:

- Conceiving variables as “boxes” which are initially empty. Additionally the novice may believe that the variable-box can hold multiple values, that the initially empty state is equivalent to a value of 0 (or an empty string), and/or that assignment transfers a value from one variable to another (leaving the first empty);
- Belief that assignment creates a link between the left-hand-side and right-hand-side expressions, as in mathematics; the right-hand-side is not evaluated when the assignment is executed; that changing a value of a variable mentioned on one side of the assignment operator will simultaneously affect the value of the variable(s) on the other side;
- Not understanding flow of control. Du Boulay notes cases where students do not understand that one statement affects the environment in which the next executes, expect that the system will jump around blocks of code that are reachable via other means (particularly in BASIC with its GOTO instruction), or fail to comprehend that a statement reading input from a terminal causes execution of the rest of the program to be suspended until such input is provided;
- Mixing up array indices and cell values, that is, treating a variable that holds an index as if it holds the cell value indicated by the index; and
- Belief that a 'while' loop will be terminated immediately once the control condition becomes false (i.e. a failure to understand that the condition is tested only when the loop body begins or repeats).

Many of the misconceptions listed above could also apply to programming languages other than Pascal and BASIC.

Fleury (2000) conducted interviews with students, asking them to compare two Java programs, determine whether they would produce the same output, and explain their reasoning; a constructivist theoretical framework, as described by Ben-Ari (1998), was assumed for the study. It was found that different students had constructed several incorrect rules regarding Java semantics:

- Identically named methods in different classes can only be distinguished if they have different parameter lists;

- Constructors are invoked only to initialise object variables (not to create the object);
- Only numeric constants or literals can be used as arguments when the formal parameter is of integral type;
- The dot operator can only be used to access methods of an object or class.

Madison (1995) interviewed college students enrolled in an introductory programming course, and reported that several students had developed incorrect models of the mechanism of parameter passing in Pascal; in particular, several believed that common variable parameter (i.e. pass-by-reference parameter) names in different procedures established some fundamental connection between those parameters; they also incorrectly believed that formal parameter names and actual parameter names must not match, perhaps due to overzealous cautions from their instructor regarding style.

Ragonis and Ben-Ari (2005) investigated the understanding of object-oriented programming concepts by high-school students learning to program in Java. They identified a large range of misconceptions in different categories, including confusion between the concepts of object and class, the mechanics of object instantiation and its relationship to constructors, various aspects of class composition, program control flow and data flow, and more. Specific examples of misconceptions include:

- That methods are executed in the order in which they are declared within a class;
- That the attributes (fields) of a composed class contain the attributes of a component class, as well as the contained object itself;
- That methods declared in a component class need to redeclared in the containing composed class.

Holland et al. (1997) also discuss several misconceptions observed in students learning about object concepts in object-oriented programming. These include:

- Conflation of the concepts of object and variable;
- Conflation of the concepts of object and class;
- That work in methods is performed exclusively via assignment to variables;
- Identity/attribute confusion (where an instance variable called “name” is confused with the identity of the object of which it is part).

The number of potential misconceptions is clearly great, and the occurrence of these misconceptions in the mental models of novices is a well-studied problem. A number of visualisation tools have been produced to aid novices in their understanding of particular notional machines for various programming languages; a survey of such tools is produced by Sorva et al. (2013). They note in particular the trouble caused to learners by *hidden behaviours* in a notional machine which students may have difficulty in inferring and incorporating into their mental models without assistance; tools which visual behaviour can make these hidden behaviours apparent, enhancing learning and in some cases aiding in the debugging or understanding of erroneous code.

While visualisation tools are undoubtedly of significant benefit to novice programmers, they can not completely prevent errors from being made. It can be seen that a significant number of misconceptions underlying novice programmer errors have been identified; diagnostic messages which sufficiently explain an error could also help to correct these misconceptions.

3.4 Issues with and Considerations for Diagnostic Messages

In the previous two sections, we discussed the difficulty that understanding syntax presents to novice programmers, and listed several misconceptions that are not strictly syntactical in nature; we suggested that improving compiler diagnostics for these errors would help to overcome these issues. In this section we will discuss literature which discusses problems observed with diagnostic messages in compilers, and provides insights into what makes a “good” diagnostic message for novices – that is, what constitutes a message that is effective in helping a novice to understand and resolve the problem.

Brown (1983) conducted an analysis of the error messages produced by a range of Pascal compilers, and notes a dearth of compilers producing a helpful diagnostic for an error in a simple test program. Brown mentions the use of syntax-directed editing and enhanced visual display of diagnostic information (the latter of which has since become common in IDEs) to improve the effectiveness of error message or prevent the need for them, by preventing the possibility of the error occurring in the first place. Although dated, Brown's observations on the quality of error messages still hold true, and his work serves as a good introduction to the relevant issues.

A key observation made by Brown is that error diagnostic production is often an after-thought of compiler design:

“... errors are artificially classified according to the time at which the compiler detects

them. This is a wrong approach, since the internal workings of the compiler should be of no concern to the user."

Brown also remarks on the problem of inaccurate messages – error messages which state a cause of the error which is different from the true nature of the error – and the compromise that compiler developers tend to make between specificity and accuracy of errors. In other words, making a diagnostic message more general may prevent it from being incorrect in certain cases. While this is superficially true, reducing the precision of messages also reduces their utility; the possibility of developing techniques to improve correctness without sacrificing specificity is left open.

Flowers et al. (2004) also look at the weaknesses of diagnostic message production in an existing compiler. In particular they discuss the "cannot resolve symbol" error produced by the Javac compiler, and showed that it could be caused by several different logical errors: the variable being declared but not initialised before being used, the variable name being misspelled when used, or the variable being declared and initialised in a conditional block but referred to from outside that block. The situation seems to have improved since; the current Javac compiler produces a more specific message in the case of using a declared but not initialised variable, at least, but fails to distinguish the other highlighted cases.

Traver (2010) looks at the issue of compiler diagnostic messages, noting that

"Error messages shown by compilers are, more often than not, difficult to interpret, resolve, and prevent in the future."

Traver identifies five different factors affecting the ability of a novice programmer to effectively deal with diagnostic messages received from a compiler: limitations in training, experience and habits, environment, and in the human cognitive system, and in compilers (including both technical limitations due to the typical implementation of the compilation process, and in a general lack of awareness of the designers of compilers of what constitutes a good diagnostic message). Some principles for diagnostic message development are suggested, based on usability heuristics developed by Molich and Nielsen (1990), including:

- clarity and brevity
- specificity
- context insensitivity

- locality
- proper phrasing
- consistency
- suitable visual design
- extensible help.

A range of diagnostic messages produced by several different compilers are evaluated according to these principles.

The most interesting of Traver's principles (Traver 2010) in regards to the work presented in this thesis are *specificity* (providing specific rather than generic information about the problem – because too-general messages make it difficult to know exactly what has gone wrong), *context insensitivity* (ideally, the same logical error should produce the same or similar error message even when it occurs in a different context), and *locality* (diagnostics should correctly identify the location of the error, thus guiding the user on where to apply the fix). These principles relate directly to the process of generating diagnostics, rather than their presentation.

Marceau et al. (2011) engaged in dialogue with novices as they encountered diagnostic messages during a formal study involving hour-long talk-aloud sessions. They noticed that the subjects tended to believe that the location of an error highlight accurately indicated the part of the code that needed to be changed in order to fix the associated error, and that they tended to be weak with technical vocabulary (and thus unable to understand what messages using such vocabulary were telling them). To counter these problems, they suggest highlighting multiple parts of the code which correspond to constructs mentioned in the error messages, and using simplified vocabulary in the message text.

Tackling another aspect of diagnostic messages, Nienaltowski et al. (2008) performed a study attempting to evaluate how different forms of message presentation affect their efficacy when presented to novices. They define a "short form" (displayed separately from the source and including a source file name and line number, with a brief error message and code snippet), "visual form" (error indicated "in-line", i.e. by highlighting part of the source code and displaying a short message separately) and "long form" (essentially, the same as the short form, but supplemented with additional information and suggestions on how to fix the error); the three forms were evaluated by presenting code samples with different types of errors to students and asking them to choose (from several options) a correct description of the error. Participants were presented with errors of different types together with a diagnostic message in one of the three forms; each error type was represented once in each of the forms during the questionnaire. It was found that the visual form led to students diagnosing the error with greater speed but less accuracy; the results were weak however, and the limitations of having only a single error of each type in each form, and of having a limited number of error types represented, was noted.

Lee and Ko (2011) experimented with changing the tone and some aspects of the visual presentation of diagnostic messages, declaring that:

> *"... [beginners] may view the cold, terse, and often judgemental errors from compilers as a sign of personal failure."*

They implemented a system where the user programs the behaviour of a simulated robotic character using a special-purpose programming language, where diagnostic messages were presented either using an impersonal style, or as speech bubbles from a friendly robot face which generally phrased the problem as a failure of the robot rather than the programmer – in contrast to the suggestions of Traver (2010), who warns mildly against anthropomorphic messages. Drawing on theory in educational psychology, they argue that messages which could be taken as a critique of the programmer may be de-motivational. They found a significant increase in the number of tasks completed by novices when the messages were presented in the unconventional "friendly robot" style, despite no significant difference in time spent on the activity. The game-like nature of the programming tasks may be relevant to the results, however; they report that subjects with the friendly error messages were significantly more likely to want to "help Gidget succeed." It is not certain that re-phrasing diagnostic messages shown during more conventional programming tasks in the same way would deliver any benefit.

It is clearly widely recognised that diagnostic messages are an important aspect of a novice programmer's interaction with a programming system. The works and studies presented in this section should guide the development of systems designed to provide program diagnostics to novice programmers.

3.5 Identifying Errors Made by Novice Programmers

Knowledge of what kinds of error students make would be beneficial as a guide for pedagogy and for development of systems producing diagnostic messages. Errors have been categorised according to various schemes and using several different methods, which will be discussed in the following sections.

3.5.1 Diagnostic Messages Encountered by Novice Programmers

One approach to categorisation of errors has been to collect data on the compiler-generated diagnostic messages which are produced for each error; the category schema for errors then corresponds to the set of diagnostic messages that the compiler can produce. Methods using this approach benefit from being highly automatable, since compilers or development environments can be instrumented to collect data during novice programming activity.

Ahmadzadeh et al. (2005) instrumented the Jikes compiler ('Jikes' n.d.) to record source code and compiler diagnostic message for student errors made during the completion of exercises set across a term in an introductory programming module; they tabulated the frequency of different diagnostic messages in exercises related to certain core Java programming concepts. Messages were divided into three categories – lexical, syntactic and semantic – according to the type of error they suggested. Diagnostics suggesting a semantic error were found to be most common, amongst which "Field Not Found" was the most prevalent message. However, the study focussed largely on novice debugging behaviour when presented with code containing *logical* errors (which do not necessarily cause a diagnostic to be generated).

Jadud (2005) also counted frequency of diagnostic messages during student programming exercises, using BlueJ (Kölling et al. 2003). The most common diagnostic message was "semicolon missing", followed by "unknown variable"; the latter likely equates roughly to the "field not found" message from the Jikes compiler. The primary purpose of the study was to examine novice behaviour on encountering an error. Tabanao et al. (2011) later performed a similar study, in the hope of being able to identify at-risk students; they reported the highest occurring diagnostics as "cannot find symbol – variable" and "; expected", which seem to describe the same two most prevalent errors (though they have swapped position in the frequency ranking).

Jackson et al. (2005) performed a count of diagnostic message frequencies (with a different programming environment) at the United States Miltary Academy at West Point, finding that "cannot resolve symbol" and "';' expected" were the two most frequently generated diagnostics; though the text is slightly different, these do seem to match the messages from the aforementioned studies.

While there are several studies of the relative occurrence of different diagnostic messages and they show some agreement in regards to the most commonly encountered diagnostics – once differences in wording are taken into account – using frequency of diagnostics as an indicator of which types of error students most frequently encounter is problematic, for several reasons (as discussed in chapter 2.7); indeed, the studies cited above – with the exception of that of Jackson et al. (2005) – are not primarily concerned with identifying the most frequent errors.

3.5.2 Categorisation of Logic Errors using Goal-and-Plan Deviation

The work presented in this thesis is concerned primarily with the categorisation and diagnosis of syntactical and semantic errors in novice programs. Programs may also contain logical errors, however, which may result in the program compiling successfully but not behaving as intended. The work of Spohrer et al. (1985) examines the use of Goal-and-Plan (GAP) trees to represent potential problem solutions. A GAP Tree identifies the goals that must be fulfilled to solve a problem, and the various plans that might be used to achieve each goal; plans may have sub-goals, and so on. Various plan subcomponents (input, output, initialisation, update, guard, syntax) can be matched directly to lines of code in an attempted solution, and a GAP tree inferred from a solution attempt (with one plan chosen for each goal) can be matched with and compared to a solution sub-tree.

Spohrer et al. (1985) show that bugs can be categorised by the nature of a difference between between an attempted and correct solution tree: plan components, or whole plans, can be *missing, malformed*, *spurious* or *misplaced*. Each condition can occur with each type of plan subcomponent (including entire plans), giving a total 28 bug categories in a problem-independent categorisation scheme. A problem-dependent scheme can further categorise according to which top-level goal the mismatch is associated with. The authors note that the production of GAP trees has not been automated and that "developing GAP trees is still an art"; regardless, the technique can give insight into the misconceptions underlying bugs in novice programs.

3.6 Improving Programming Diagnostics

3.6.1 General Automatic Diagnostic Generation

A number of approaches have been taken in trying to improve the diagnostic messages – by increasing precision and accuracy, and altering wording – that are presented to novice programmers when they produce erroneous code. Several past efforts, which we will discuss shortly, developed tools which use traditional techniques such as error detection during parsing (as discussed in chapter 2.5.2) and limited semantic analysis to detect and identify errors, or rely on the diagnostic generated by the compiler to identify an error, before providing a crafted diagnostic which is generally less terse and more informative than of the compiler. These tools are general in the sense that they work with any body of code in the supported programming language, which means that they do not require additional preparation to be useful when used with new exercise material, for example.

Schorsch (1995) produced *CAP*, a tool to check for errors and style problems in Pascal programs. The target errors were based on an informal study of frequent errors encountered during laboratory exercises and in assessments. The tool is limited to a subset of the language, as used in the programming class at the United States Air Force Academy. It identifies errors via a limited static analysis, with syntax error recovery during parsing, making it usable even when the code is not compilable. The diagnostic messages are parameterised (with relevant identifiers, operators or other details).

On a technical level, CAP appears unremarkable by modern standards, however, it provides much more detailed messages than typical compilers, attempts to “[describe] the problem in a user-friendly way”, and offers advice on potential solutions. It also detects various logical errors by scanning for certain problematic code patterns. The tool received generally positive ratings in a survey given to students. The technical limitations (support for a limited language subset and limited recognition of built-in Pascal modules or “units”) are considerable, but it serves as an early example of a tool specifically created to provide better diagnostics for students.

A tool set to pre-scan and identify potential problems in Java code is described by Lang (2002). Examples of problems detected by the tool set include mixing of upper/lower case in keywords and having numeric literals begin with '0' (which in Java denotes a literal in octal rather than decimal, though a novice may not realise this). The tools also assist with tasks such as correctly indenting code. Unfortunately there is no data to show whether novices found the tool set useful, nor any description of the internal design; one of the errors it detects is instance shadowing (where a local variable is defined with the same name as a field), which would require parsing to identify variable or field scope.

Hristova et al. (2003) developed the *Expresso* system as a multiple-pass pre-processor to provide error diagnosis for Java code. The source is reduced to a vector of lexical tokens (with whitespace and comments removed) before various error diagnostic passes are run. Further details of the design are not reported, but from analysing the tool's own source code[1], it can be seen that the diagnostic passes rely on detecting lexical patterns to detect grammatical constructs such as variable and method declarations, rather than parsing according to the formal language grammar; this has the benefit of being able to identify constructs in the presence of grammatical errors, though does not applying scoping rules which could lead to misdiagnosis (in the worst case, identifying an error where there is one), and limits the reliability of semantic analysis which is used to detect mismatched parameter types for example. Despite these limitations, the tool performed well when run against a test suite of erroneous programs assembled by the developers.

Flowers et al. (2004) developed the *Gauntlet* system to diagnose Java errors; a series of errors was identified by the faculty as common or problematic and "checkers" for various of these errors were implemented as part of the system. The diagnostic messages produced by Gauntlet checkers are more detailed than the compiler messages, identifying multiple possible errors where appropriate, and occasionally introduce elements of humour. The details of the design of the system are not reported. The system reportedly improved the quality of work produced by students, though the authors later acknowledge that "the Gauntlet system was not deciphering the most common errors that the students were encountering" (Jackson et al. 2005), perhaps due to a discrepancy between the errors identified as most common by the faculty – which guided the development of the system – and the errors students actually produced.

Another tool, developed by Dy and Rodrigo (2010), was developed to diagnose errors identified as causing the compiler to produce particularly unhelpful diagnostic messages ("non-literal errors"). It works as a post-compiler, using the diagnostic provided by the compiler in junction with the source code and line number of the error to produce a "more descriptive or targeted error message". It was tested by assessing the output compared to other compilers when run on a corpus of source files containing non-literal errors which had been assembled for this purpose, and found to correctly identify the error in most cases. The use of the diagnostic originally produced by the compiler for purposes of producing an enhanced diagnostic has some drawbacks, however; it is not certain what diagnostic will produced for a particular error (as discussed in chapter 2.5) and the diagnostic may change with newer versions of the compiler.

1 The source code for the Expresso tool is archived at (seen 21/9/2016): https://web.archive.org/web/20150623044422/http://www.cs.princeton.edu/~amisra/expresso/expresso.cpp

In the work of Denny et al. (2014), development of enhanced diagnostic messages for the *CodeWrite* tool (Denny et al. 2011) is discussed and their effectiveness evaluated. The approach, similar to that of Dy and Rodrigo (2010), was to use the compiler diagnostic as well as the corresponding erroneous source code as input to generate an enhanced diagnostic message. Evaluation in an empirical study showed that the use of CodeWrite with enhanced diagnostics did not reduce the number of consecutive non-compiling submissions, nor the total number of non-compiling submissions, made by students during completion of laboratory exercises when compared to the use of the same tool when it displayed the original compiler messages; neither did it reduce the number of attempts to resolve errors corresponding to the most frequently generated compiler diagnostic messages.

The results of Denny et al. (2014) appear to conflict with the notion that improving diagnostics may benefit novices, but the limitations of their approach (as already discussed) should be noted. Furthermore, another study, by Becker (2016), uses a similar technique to provide enhanced diagnostics in a tool called *Decaf*, but reports a contrasting result. Though the author acknowledges that "in some cases, matching of the source code and compiler error message does not result in information which can be used to create a usable enhanced error message", the tool was reported to reduce both the number of overall errors and the number of repeated errors per compiler message in a controlled empirical study.

The number of attempts to improve compiler diagnostics indicate a widespread belief that doing so would be of benefit – and the evaluations, bar that of Denny et al. (2014), support this position. However, none of these efforts have provided a solution that has gained widespread use. There may be various reasons for this outside of efficacy, though it is notable that these efforts either rely on existing compiler diagnostic capabilities, or have technical limitations. They also target errors which were identified via surveys of teaching staff and/or students, or by collating the frequency of diagnostic messages from an extant compiler. While the tools discussed here are certainly of interest, there is room for improvement by better guiding the types of error which are the most severe – that is, which require the most effort for novices to overcome – and by formalising an approach to diagnosis which does not rely purely on traditional techniques.

3.6.2 Semi-automated Generation of Diagnostic Messages

In the previous section, we discussed various systems developed to provide improved diagnostic messages to novice programmers. In general, in cases where the implementation details were reported, these systems have been implemented either by using a compiler-generated diagnostic as input, or by lexical pattern matching techniques. Since compiler-generated diagnostics regarding syntax errors are usually generated during the parsing process, and since lexical pattern matching is limited in the errors it can diagnose due to lack of real semantic analysis, it is worth examining the possibility of enhancing diagnostics produced during parsing.

Jeffery (2003) describes a software tool, *Merr*, to generate accurate diagnostics based on language grammar, for LR-parseable grammars. A set of example errors and suitable error messages are provided to the tool, along with the language grammar; the tool then generates code which maps parse-state together with lookahead token to a suitable error message. This eliminates or reduces the need to manually derive the parser logic necessary to produce particular diagnostic messages, and provides a separation – to some degree – of the concerns of parsing and syntax error diagnosis.

The use of tools such as Merr may lead to improvements in diagnostic messages, due to the reduced effort required to develop and maintain parsers which is achieved by automating the translation of parse state to message. However, the essential mechanism of producing the diagnostics remains the same as with a manually coded error-diagnosing parser, and has the same limitations; it will not take broader context (i.e. outside the immediate parser state) into account when producing an error diagnosis.

3.6.3 Crowd-sourcing Suggestions for Dealing with Errors

An alternative to offering an alternative diagnostic message text to that produced by the compiler is to annotate the compiler's own diagnostic with solutions that have worked for other users. This is the approach employed by Hartmann et al. (2010), who developed a system called *HelpMeOut* to collect data on code changes that resolved errors corresponding to particular diagnostic messages, and to present matching solutions to users who later encounter the same diagnostic message. Solutions are matched based on both the compiler diagnostic and a lexical token-pattern match between the source codes, which also enables for automatic application of a selected solution. Explanations for solutions in the database can be added manually, and will then be presented to the user alongside the solution when they encounter a similar error. The system can be used for errors detected both compile-time and run-time.

An evaluation of HelpMeOut showed that it was able to produce useful suggestions (leading to a direct solution or providing a clarification leading to a solution) to 47 percent of the errors encountered by student participants. Hartmann et al. (2010) identify the lexical nature and lack of semantic analysis (for instance, all user-defined types are considered equivalent) in the matching process as being limitations which may contribute to this statistic.

3.6.4 Diagnosing Errors by Goal and Plan Difference

Johnson (1990) implemented a system, *PROUST*, which enumerates predicted implementations for a set of goals, specified in a formal language, in terms of the plan pattern that may be used; essentially this is an automated generated of the GAP trees discussed by Spohrer et al. (1985) based on a root-level set of goals and a database of goal-plan matches. Predicted implementations include implementations generated by common (but incorrect) goal reformulations that have been discovered via extensive empirical analysis of novice code. These predicted implementations are matched against student code (in the Pascal programming language) to find the best match, and differences which match a database of plan-difference rules representing common logical errors are reported, together with any goal reformulations necessitated by the match.

In their motivation for the development of PROUST, Johnson (1990) make an argument for thorough error diagnosis:

> *"Our empirical studies of how students debug programs indicate that novice programmers tend to correct the surface manifestations of bugs rather than the bugs themselves; thus proper descriptions of bugs are crucial."*

The approach taken in PROUST allows for precise and problem-specific automated diagnosis of logical errors, the availability of which is undoubtedly beneficial for novice programmers. The main drawback of this approach is the requirement for a goal set to be specified for each problem, and for a database of candidate goal reformulations and plan-difference rules. The system identifies logic errors, but not syntactic or semantic errors; it operates on source code that is already in a compilable state.

3.6.5 Visualisation of Grammatical Structure and Rules

SyntaxTrain (Moth et al. 2011) takes a novel approach to diagnosing syntax errors in Java code by displaying a syntax diagram, generated by parsing code up to the point of a syntax error and representing syntax rules for the offending construct as well as containing constructs in an easy-to-understand graphical diagram, allowing the novice to determine exactly where and how their own code does not conform to the language grammar. A benefit of this approach may be the repeated exposure to formal grammar rules in a digestible form. The technique is applicable only to syntax errors.

3.6.6 Static Analysis

The *FindBugs* tool (Hovemeyer and Pugh 2004) uses static analysis techniques, including control flow and data flow analysis, to identify a range of common bug patterns. Bug patterns detected include null check after use, use of null value, and unconditional self recursion, as well as a range of style problems. This is useful for detecting logical errors, but cannot detect diagnose syntactic or semantic issues. FindBugs makes trade-offs in that it can produce false positives and false negatives: warnings for pieces of code that function correctly, or no warning in a case where a problem does exist.

Static analysis is certainly a useful technique for identifying problems in real source code, and the FindBugs tool was used to identify many real bugs in various pieces of software which see widespread use. It could certainly be used with novice code, however, it is restricted to operating on code that is already compilable.

3.7 Other Approaches to Mitigating the Syntax Problem

We have discussed the difficulty that syntax causes to novice programmers (chapter 3.2) and approaches to mitigating this and other conceptual difficulties by providing better diagnostics (chapter 3.6). There are, however, other approaches to helping novices overcome or avoid problems with syntax.

One technique to reduce the burden of syntax is to delay its introduction. Gaspar et al. (2008) remark on the syntactical difficulties that novices may encounter when using the Java programming language; they used a system called *Raptor*, which allows creating programs by expressing them in the form of flowcharts rather than via a purely textual representation, before moving on to Java. The authors claim that measurements of syntax error occurrence and of the number of different types of question asked in a programming forum support the notion that “syntax doesn't really matter” in the education of novice programmers (though their methodology fails to take into account that syntax familiarity increases over time and their conclusions are weak). Avoiding syntax allows focusing on other elements of programming such as program design, but the student must battle with syntax at some point.

Dillon et al. (2012) look at whether what they term *moderately assistive environments* – environments supporting features such as syntax highlighting, error highlighting, integrated compilation/execution, and auto-completion – are helpful compared to low assistive environments (those without such features). They show that students are able to use a moderately assistive environment more effectively. Code auto-completion, in particular, can be considered a tool which specifically helps students with syntax by supplying well-formed code snippets (such as a method call with parentheses and place-holders for argument values) although Dillon et al. (2012) do not attempt to determine the benefit of this or similar features on their own.

Several pedagogical programming environments have taken on board the idea of allowing the programmer to construct their programs visually, via drag-and-drop interfaces. *Scratch* (Maloney et al. 2010) does so; the designers strove to avoid the need for error messages by making it impossible to construct a syntactically invalid program. Language constructs (including control flow constructs, commands and functions) have a visual representation and will “snap together” as appropriate when dragged near each other. The designers note that “a program that runs, even if it is not correct, feels closer to working than a program that does not run (or compile) at all.” The indivisible control blocks prevent the problem of missing or misplaced closing delimiters which can occur in many text-based languages.

The *Alice* environment (Cooper et al. 2003), incorporates a similar program-editing technique to that of Scratch. Dann et al. (2012) report on their experiences using Alice as part of a mediated transfer designed to guide students in the process of transferring knowledge between Alice 3 and a regular Java programming environment. They developed a plugin allowing students to open an Alice 3 project directly from within NetBeans, a feature-rich open-source development environment ('NetBeans IDE' n.d.). They found that their approach of using a mediated transfer led to consistently higher exam scores when compared to the previous approach in the course, which incorporated exposure to Alice 2 and a Java development environment but which did not incorporate mediated transfer and did not support the automatic import of code from Alice.

An advantage of having a graphical representation, together with a suitably-developed “structured editor” – as in both Scratch and Alice – is that it removes the possibility of a range of syntactical errors that could otherwise occur. Structured editing does not however specifically require a graphical program representation; a program might be displayed using a textual representation inside an editor that is not a general-purpose text editor, but rather is crafted to limit the grammatical errors that can be produced during editing, or at least to make it difficult to produce them (which serves to guide the user in the editing process). Such an arrangement is often referred to in literature as “syntax-guided editing”; the user is guided by the tool towards the correct syntax, but the visual interface may closely resemble a text editor.

Syntax-guided editing is not a new idea; Lang (1986) describes a system called *Mentor*, a syntax-guided editor for Pascal (and other languages) on which development began in 1974. Although the system was used to teach Pascal in several universities with some anecdotal success, they claim that:

> *“Even with a good user interface, the syntax directed paradigm is too complex and bothersome for inputting programs or for performing simple editing tasks.”*

A well-designed syntax-guided editor need not be so “complex and bothersome” as Mentor, however. Kölling et al. (2015) introduces Frame-Based Editing, which aims to combine the beneficial aspects of block-based editing systems (such as Scratch and Alice) with those of more conventional text-based program editing. The system allows insertion of various program constructs using single keystrokes, and presents the user program as a series of nested *frames*, which essentially correspond to block scopes (selection/iteration blocks, methods, classes, etcetera).

Of course, a text editor that doesn't use the syntax-directed approach can still provide some functionality beyond that of a generic editor, in order to increase productivity when editing a program in the supported programming language; for instance, many professional Integrated Development Environments, such as Eclipse ('Eclipse' n.d.) and Netbeans ('NetBeans IDE' n.d.), provide a “code completion” feature which suggests method and/or variable names that might be accessible from a particular point in the program code; an editor may also provide syntax-related hints during editing, such as highlighting errors or indicating the range of particular scopes within the program. Such features may also be useful to novices.

Other “syntax light” approaches use a text-based programming language with (arguably) lower syntactical complexity than rival languages. Grandell et al. (2006) argue that Python, with its “simple and flexible” syntax, is a good choice for use in teaching novices, particularly in combination with its dynamic typing and interpreted nature (which allows for immediate feedback). They found that grades were generally higher in a high school programming course using Python compared to a very similar course using the Java language. It is not clear what, if any, supporting tools (such as development environments) were used nor whether this may have had an impact; however, it is certainly true that Python has a simple syntax for certain tasks (such as list and dictionary creation and manipulation) that are somewhat more complex in Java.

Warren (2001) makes a similar argument to Grandell et al: dynamic typing removes some of cognitive overhead of understanding the typing system of C++/Java; those two languages also have gratuitous complexity (too many primitive types, and high syntactical overhead for a small program), inconsistency (reals and integers as separate types, pass-by-reference versus pass-by-value semantics), and expose irrelevant implementation details (such as arrays being dense and of fixed size). Warren suggests Javascript as a viable alternative to C++/Java as a language to introduce programming to novices with.

The notion that it is better to start teaching novices with simpler languages seems quite common. Bloch (2000) suggests choosing a language which has as few language constructs as possible, and which allows (where practical) introducing them one by one; using this principle he chooses Scheme as a language to teach initial programming concepts (switching later to Java). He perceived that students had less trouble with the syntax of Scheme than previous students had with the syntax of Java at the same stage. Many concepts were introduced earlier than in the previous year, but effects on grades were inconclusive.

Rather than choose an existing language for teaching novices, some instructors have chosen to implement a new one. McIver and Conway (1996) provide a list of language design principles for introductory programming languages. Among these, they note several syntactical/semantic issues with existing languages; they suggest avoiding syntactic synonyms (different ways of expression the same construct), syntactic homonyms (similar syntax yielding different semantics depending on context), elision (allowing the omission of redundant syntactical elements), and violation of expectation. For an example of the latter in C, an assignment to an “int” variable yields a value which can be converted to a boolean as context demands, so that a statement such as:

```
    if (x = 0) { /* conditional code block */ }
```

… will compile, but fail to produce the expected behaviour (of comparing the current value of the variable to zero without modifying it) at runtime.

As an alternative to changing to or creating a new programming language altogether, one approach to alleviate students' struggles with syntax is to remove some constructs from an existing language. This helps to eliminate accidental usage of more advanced language features, and simplifies syntax error diagnosis, since it reduces the number of constructs that might validly appear in certain contexts. The DrScheme system (Felleisen et al. 1998) uses the notion of “language levels” to help catch common beginner mistakes such as attempted use of infix notation for expressions (the system specifically recognises certain such errors). The idea is to have the compiler or interpreter understand a configurable subset of the full programming language, and be able to produce better diagnostic messages as a result. ProfessorJ (Gray and Flatt 2003), built on DrScheme, applies the same theory to the Java language. DePasquale et al. (2004) introduced language levels in a C++ environment in order to assess their impact on student performance, and claim that it has no adverse impact, though their data shows no positive impact either. Hasker (2002) found that students using an environment that compiled a subset of C++ and was designed to produce “more specific” error messages (than commercially available compilers) were able to complete lab tasks slightly faster than those using regular C++.

Moving away from programming languages altogether, Fidge and Teague (2009) attempt to remove the syntax problem by asking participants in their study to construct solutions to a problem in structured English; they show that common logic errors were made with a frequency similar to when programming using a formal programming language, implying that understanding syntax is certainly not the only issue facing novices. However, they found only a small difference between performance of participants in a task using structured English and performance in a similar task when using Python.

Constructing or choosing the programming language to simplify syntax, using graphical program representation, or limiting syntactical problems by constricting the editor are all attempts to overcome or mitigate the difficulty that novice programmers have in dealing with programming syntax, and have met with varying degrees of success. However, novices will undoubtedly move on to more regular program editing environments and languages at some stage; at this point, such techniques are unavailable, and some of the problems that they have helped to avoid may surface.

4 Methodology

In this chapter we will present and discuss the methodology and describe the methods used in an attempt to answer the research questions outlined in the first chapter. The research presented in this thesis is divided into three parts. In the first part, a category scheme for categorising programmer errors was developed, and novice errors were categorised accordingly, to ascertain the viability of manual error categorisation on a large number of errors.

In the second part of the research, a larger data set was similarly categorised using a slightly modified method, and an analysis of the severity of different errors was undertaken.

In the third part of the research, a prototype tool for the accurate and precise automated diagnosis of high-severity errors was developed and evaluated by comparing its output with that from the *javac* compiler.

4.1 Methodology – Part I

4.1.1 Initial Data Collection

Two separate data collections were performed by implementing an extension for the BlueJ environment (Kölling et al. 2003) to collect data from programmers using it for coursework. The extension captured snapshots of the students' source code at compilation times as they performed their coursework in a university course. Participants were students of one of two introductory Java programming modules – CO320 at University of Kent, UK, and CSCU161 at the University of Puget Sound in Washington, USA – during their respective spring terms of the 2012 academic year.

The CO320 module ("Introduction to Object-Oriented Programming") at the University of Kent provides an introduction to software development using the Java programming language and the BlueJ development environment. It is a first year module and is designed to be suitable for students who have no prior experience with programming.

At the University of Puget Sound, the CSCI161 module ("Introduction to Computer Science") is similar to CO320 at the University of Kent in terms of entry requirements and goals. It also teaches programming using Java and BlueJ.

Participation in the data collection was voluntary. Overall approximately 240 students participated

in the data collection (determining the exact number is not possible since the data was anonymised, and some participants may have received two or more anonymous identification numbers if they installed the extension on more than one system).
Students of both CO320 and CSCI161 during their respective Spring Term, 2012, worked on several exercises over the course of the respective modules, each of which involved modifying a project provided to them as a starting point. Students who chose to participate in this study did so by using an alternative starting-point project, identical to the original except for the addition of a BlueJ extension which was loaded automatically upon the project being opened.

The BlueJ extension provided to participants transmitted data to a central server, where it was then stored in a database. Data was transmitted whenever a code compilation was triggered in BlueJ (either by using the “compile” button in the main interface or the similar button in an editor window, or the “rebuild package” menu item in the “tools” menu). The following items are transmitted upon each such event:

- A unique numeric identifier to distinguish the participant, for the purpose of associating events and source code belonging to a single user. The identifier is randomly generated and then stored locally, on the participant's machine, so that multiple sessions (where BlueJ is terminated and restarted) remain associated via the identifier.
- A project identifier string, being the folder name within which the project files are stored (i.e. the project name). The full path to the project was not transmitted as this would potentially compromise anonymity; projects often reside within a home folder which is named for the user's login identifier.
- The names (including the project-relative paths) of any source files within the project that were modified since the previous event data was transmitted, together with the contents of those files at the time of the event. In the case of the first transmission for a particular participant, the contents of all files within the project are transmitted. Contents were anonymised, by replacing the contents of comments within the source code, before transmission.
- The diagnostic message produced by the compiler during the compilation, if any. This is the same message as was presented to the user via the BlueJ interface. If multiple diagnostics are produced by the compiler, only the first is recorded (in-line with BlueJ's behaviour of presenting only this single diagnostic to the user).

- The line number associated with the diagnostic message (if any)
- The project-relative path to the file associated with the diagnostic message (if any)

Because the contents of changed files are transmitted on any compilation event, it is possible to reconstruct the entire project state as it was when the event occurred.

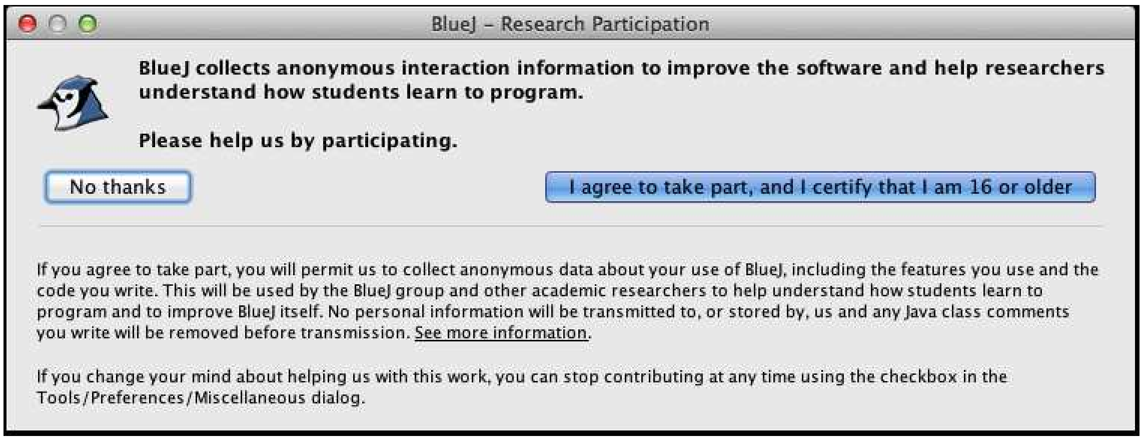
Although it is not transmitted, the time at which the data for each event is received is also recorded in the database by the data collection server, giving a good approximation of the time at which the event was generated. Overall, 37,432 compilation events were recorded, with 21,623 of these associated with a compilation error (the remainder being successful compilations). 197 of these were subsequently analysed as part of the category development phase, which will be described in Chapter 4.1.3.

4.1.2 Large-Scale Data Collection

During the period of the data collection just described and the subsequent development of a category scheme, we also contributed (as part of the BlueJ group at the University of Kent) to the development of the *Blackbox Data Collection Project* (Brown et al. 2014), which effectively extended the ability to collect similar data from all BlueJ users on an opt-in basis. Blackbox conducts on a world-wide scale a data collection similar in nature to that performed at the Universities of Kent and Puget Sound, as discussed in the previous section, but also collects information on a broader range of events occurring within the BlueJ system, including use of certain parts of the environment such as the unit test tool and the debugger. The data collected is made available to researchers from academic institutions for purposes of research.

Whereas the initial data collection required a BlueJ extension, Blackbox directly incorporates the collection process into the BlueJ environment. This has been implemented (by the author of this thesis and other members of the BlueJ team) as of BlueJ version 3.1.0, which was officially released on the 10th June 2013, and the data collection has been running since that time.

Participants in Blackbox give their consent by choosing to do so when they start BlueJ for the first time. They are presented with a dialog (Figure 4.1) which gives information regarding the Blackbox project and the nature of the data it collects, and explains that, if should they choose to take part, their source code will be anonymised before transmission. It also gives a link to a web page providing further information regarding how the data is collected, how data collection can be disabled, and who the data will be made available to.


BlueJ – Research Participation

BlueJ collects anonymous interaction information to improve the software and help researchers understand how students learn to program.

Please help us by participating.

If you agree to take part, you will permit us to collect anonymous data about your use of BlueJ, including the features you use and the code you write. This will be used by the BlueJ group and other academic researchers to help understand how students learn to program and to improve BlueJ itself. No personal information will be transmitted to, or stored by, us and any Java class comments you write will be removed before transmission. See more information.

If you change your mind about helping us with this work, you can stop contributing at any time using the checkbox in the Tools/Preferences/Miscellaneous dialog.

Figure 4.1: Participation consent dialog for the Blackbox project. This dialog is presented to users when they launch BlueJ for the first time.

Since the amount of data and the cohort it related to were significantly larger than in the prior collection, data from Blackbox project was incorporated into the next phase of analysis, involving the categorisation of a large number of errors. 23 sessions, comprising a total of 136 events recorded between 2013-06-11 and 2014-01-01, were selected and retrieved from the Blackbox database for analysis. The selection was made randomly in order to avoid any bias in the nature of programming errors within the data due to particulars of course structure or stage, as well as other factors such as gender, age, and socio-economic background, which would have been difficult to control for given the limited information recorded in the database. This dataset in conjunction with the first dataset forms the basis for the categorisation phase, described in Chapter 4.1.4.

4.1.3 Development of a Category Hierarchy

The data collected during the initial data collections (from participants who opted in to the study, from two programming modules at different universities, as described in Chapter 4.1.1) was analysed for the purpose of developing a category scheme for categorising programming errors. Specifically, the aim was to develop a category hierarchy where top-level categories divide errors broadly and sub-categories allow more specific error categorisation.

The analysis method used for analysing the collected data (consisting of source code snapshots produced whenever a subject compiles their code) for purposes of error category hierarchy development is grounded in the Thematic Analysis methodology described by Braun and Clarke (2006), with variations due to the nature of the data in and purpose of this analysis. The subsections that follow will give an overview of Thematic Analysis, and how the analysis method used in this research relates to the Thematic Analysis methodology as well as details of ways in which this analysis necessarily differs from that methodology.

4.1.3.1 Thematic Analysis

Thematic Analysis is a methodology for analysis of qualitative data; it is formally described by Braun and Clarke (2006), using concepts from earlier work such as Attride-Stirling (2001). Braun & Clarke focus on its use in the field of Psychology, however the methodology is generally applicable to analyses of qualitative data in other fields, and has been adapted for use in this work. A brief introduction to and outline of Thematic Analysis methodology will now be given. Following this, its relation to the analysis method presented in this thesis to form a category scheme for programming errors described.

Thematic Analysis:

- Identifies themes and patterns in the data via an iterative process
- Places emphasis on making explicit any assumptions that the researcher makes; and
- Describes the qualitative data set in rich detail by developing codes for themes and patterns and annotating data items with the codes.

Thematic Analysis is designed for quantitative analyses which seek to identify *themes*, that are broad categorisations of terms and concepts found during the early phases of analysis and are related to the research questions being considered, occurring in a data set usually comprised of material regarding or related to a particular subject of study (such as published media, or transcriptions of interviews conducted by the researcher). By contrast, in this work the data items are complete programs written in a computer programming language, and the intention is not to identify and codify *themes* arising in the subject matter, but rather to identify the (likely) nature of *errors* found within the programs (by applying expert knowledge and experience to draw on contextual information within the source code). The techniques used in Thematic Analysis to identify themes are instead applied for this purpose; in applying its methodology to a data set consisting of student code containing errors, we treat the occurrence of a type of error as analogous to the presence of some particular theme.

Braun and Clarke (2006) identify a number of decisions to be made when undertaking Thematic Analysis, before the analysis phase begins:

1. What counts as a theme?
2. Is the desired outcome a rich description of the data set, or a detailed account of one particular aspect?
3. Is this a 'theoretical' instance of thematic analysis, or an 'inductive' instance?
4. Are the themes semantic or latent?
5. What is the epistemology: essentialist/realist versus constructionist thematic analysis?

Once the questions above have been answered, the analysis itself is performed in a series of phases:

Phase 1: Familiarisation with data. This requires repeated reading of the data set, and taking notes when finding anything of interest that will help in generation of the initial codes.

Phase 2: Generation of initial codes. “Codes identify a feature of the data … that appears interesting to the analyst” Braun and Clarke (2006, pp.88). Collate all extracts together within each code; thus each code is associated with a set of extracts.

Phase 3: Search for themes. Sort the codes into potential themes; consider how different codes might be combined into a theme. Themes may have sub-themes. Some codes may be discarded if they bear no relevance to the themes of interest. The phase ends with a collection of candidate themes and sub-themes.

Phase 4: Review themes. Read the collated extracts for each theme, and determine whether they form a coherent pattern; if not, re-work the theme or re-locate problematic extracts to another theme, or discard them from consideration. Then, re-read the entire data set, to ascertain whether the chosen themes adequately cover and suitably categorise the data set (if not, the coding must be refined, and new themes may be added as necessary). The output of this phase is a *thematic map* or *thematic network* (Attride-Stirling 2001) showing the relationship between the main themes and any sub-themes.

Phase 5: Define and name themes. For the individual themes, conduct a detailed analysis, essentially with a view to explaining how the theme relates to the research questions. Ensure that the themes are distinct and distinguishable in this regard, and that there is “not too much overlap between themes” (Braun and Clarke 2006, pp.92).

Phase 6: Production of report. Make arguments related to the research questions using examples and extracts from the data which are clearly associated with the relevant theme(s).

The five pre-analysis decisions and the six analysis phases form the essential basis of Thematic Analysis.

4.1.3.2 Application of Thematic Analysis to Error Category Development

As discussed in the previous section, Braun and Clarke (2006) identify a number of decisions to be made when undertaking Thematic Analysis. The determination for each question in regards to the category development process .

1. What counts as a theme?

The goal of the relevant analysis in this thesis is to categorise novice programmer errors. Thus, themes are considered analogous to the kinds of error that are present in the data set. These error categories should be specific enough to describe the error usefully, so that it could be understood by an experienced programmer, though the categories may form a hierarchy with less specific error categories potentially having more specific subcategories.

2. Is the desired outcome a rich description of the data set, or a detailed account of one particular aspect?

The desired outcome is a rich description of the data set, in terms of the types of error seen throughout the set, rather than a detailed account of particular types of error.

3. Is this a 'theoretical' instance of thematic analysis, or an 'inductive' instance?

The analysis of novice errors presented in this thesis is an inductive instance, since the error categories (themes) are induced from the data set.

4. Are the themes semantic or latent?

While Thematic Analysis typically looks at data items representing some form of communication (media samples, interviews, etc.), in the work presented in this thesis the data items are computer programs which are not directly written for the purpose of expression of information or ideas. The code itself is not intended as a communication, or at least, it is not useful for our purposes to analyse it as such. If we consider the communication expressed by the data to be that *a programmer made a particular error, at this time, in this source code*, then the themes could be considered semantic; however, we do not believe that classifying the themes as semantic or latent is useful in this instance.

5. What is the epistemology: essentialist/realist versus constructionist thematic analysis?
If, as for the previous question, we take the communication in the data items to be regarding the presence of an error in code, the analysis is conducted within an essentialist/realist paradigm: there are no underlying contexts being considered and we are seeking to show, empirically, the types of error being encountered.

A determination for each of the five pre-analysis decisions identified by Braun and Clarke (2006) has been made and described above, although in doing so it is evident that the analysis presented here differs in some ways from the types of analyses that they have considered, due to the specific nature of the data and goals of the analysis.

For development of a category hierarchy, the data to be analysed was stored as records relating to compilation events, when a participant had compiled the code in their project, successfully or not; each record included a time stamp and a set of source code files (names and contents) belonging to the relevant project, as well as the text and associated line number of any diagnostic message produced by the compiler when attempting to the compilation. A web-based tool was developed (in Ruby, using the Ruby-on-Rails web application framework) to allow browsing the stored data. The tool organises the records according to the anonymous participant identifier they were associated with; for each participant, it is possible to browse all collected events, including the state of the source code in each file in the project at the time of the event, and, in cases where an error is present, the associated compiler diagnostic; in these cases, the lines of code surround the line identified in the diagnostic are presented immediately (the content of all other files can be viewed by following a hyperlink). The tool also allowed for creation, deletion and editing of error categories, and the assignment or re-assignment of one or more error categories to each compilation event corresponding to a compilation failure.

We used the web-based tool to browse and achieve familiarisation with the data, for phase one of Thematic Analysis (Braun and Clarke 2006). We did not browse the entire data set, but selected events at random to observe the manifestation of a variety of errors and to discover the approximate distribution of different kinds of error. We proceeded to create initial categories describing errors that were found in the data set (phase two); categories were created in this way until the set of categories appeared to have stabilised, with new categories no longer being regularly created, and it was deemed that the categories covered a significant majority of errors recorded within the data set. The categories were reviewed and organised in a hierarchy, with several categories being placed together in a newly-created high-level category when considered appropriate; some categories were discarded if they were considered too specific to be useful (phase three). The previously assigned categories were then reviewed and revised (phase four).

The fifth phase of Thematic Analysis, definition and naming of themes, was not necessary for this analysis, since the themes corresponded exactly to error categories which were already suitably defined. The sixth phase of Thematic Analysis is the production of a report; for this work, the category hierarchy developed is used for ongoing analysis. In particular, the hierarchy is validated (as discussed shortly) and then used for categorisation of a large number of data records.

4.1.3.3 Validation of Error Categories

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

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

To check inter-coder reliability, three researchers who were experienced Java programmers were recruited from the School of Computing at the University of Kent to independently code a combined total (accounting for overlap) of 150 error events, and a percent agreement calculation (Lombard et al. 2002) was performed. The researchers were instructed briefly in the use of the web interface for categorisation, and were instructed specifically on the following points:

- They should favour accuracy and precision of categorisation over speed; quality of results was considered more important than quantity.
- They should focus on the error or errors at the site of the compilation as identified by the compilation diagnostic.
- They should choose a single category for each individual error; if they believed that there were multiple errors present at the same location, they should make multiple category selections – one for each error.

- They should provide a brief explanation of why they made the category selections that they did, in terms of contextual clues within the source code, when applicable.

Since multiple categories could be assigned to a single event (as will be discussed in Chapter 4.1.4.1), the calculation of pairwise agreement required handling of cases where multiple categories were assigned by one researcher, and another researcher or researchers assigned fewer categories. Pairwise agreement in these cases was counted according to the following method:

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

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

Note that the analysis (which will be described in 4.1.5) was based on our own categorisations, not those performed by other researchers for purpose of category scheme validation.

4.1.3.4 Deviations from standard Thematic Analysis

Thematic Analysis is designed as a formal set of techniques for the analysis of qualitative data, but the nature of the data obtained for this work, and the desired outcomes of the analysis, require some minor deviation from the standard methodology outlined by Braun and Clarke (2006). The fundamental difference between Thematic Analysis as described and what is being undertaken for this research is that the data being used is not typical qualitative data – that is, it is not media extracts on a particular topic, or answers to survey questions, but rather source code produced in trying to solve some problem. The patterns or *themes* which are identified in this study are not in what is directly expressed by the data records, but in the errors in those expressions.

A notable deviation between the analysis for this work and Thematic Analysis as outlined is that the distinction between codes and themes is less pronounced. In Thematic Analysis, codes are combined to form themes (phase 3), and the themes becomes the focus of further analysis phases; in this work, where the codes may correspond to categories concerning the conceptual nature of errors; these code types are important enough to this analysis that over-generalising them would be detrimental to its value (for instance, it is desirable to know specifically which errors are problematic for the participants; a “missing comma” error and a “missing semi-colon” are both cases of a “missing token” but may occur in very different contexts, so it would be unsuitable to combine them into this broader category). Therefore, the themes should be narrow rather than broad. This close alignment between codes and themes makes analysis easier and removes some of the subjectivity from the analysis, as determination of how to combine codes to themes is simplified.

To remove subjectivity from the coding phase of the analysis, multiple researchers performed the coding, and a test for a reasonable level of agreement between the analysts was conducted (as discussed in Chapter 4.1.3.3). This is not a part of Thematic Analysis as described by Braun and Clarke (2006), although in principle it could be applied to to other analyses based on the Thematic Analysis methodology. Similar strategies have been employed previously in other qualitative studies; for example, Amabile and Kramer (2011, pp.205).

Despite the difference in subject material and purpose of the analysis, the phases of Thematic Analysis map well to the method for category hierarchy development that we have described, as shown in the discussion in Chapter 4.1.3.2.

4.1.4 Error Categorisation

We have discussed the development of a category scheme for the categorisation of programmer errors (Chapter 4.1.3). Once error categories were defined and validated, 136 additional error events from the second data set, selected from the Blackbox Project database (Chapter 4.1.2), were coded using these categories, bringing the total number of categorised events to 333. The categories at this stage were mostly stable, with only a small number of new categories introduced to cover errors not seen in the initial data set.

This categorisation, together with the categorisation performed on the first data set during the category development process, forms the basis of the analysis described in Chapter 4.1.5.

During the process of categorisation, we also recorded short notes (entered via the web tool used for categorisation) detailing any specific considerations that had led to a particular categorisation.

4.1.4.1 Multiple Errors in a Single Event

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

```
answer=this.getuserinput
```

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

Finding four errors at an error event may be rare, but the occurrence of two concurrent errors is reasonably frequent. In these cases, we multiply-categorised the event to reflect each of the errors observed.

4.1.4.2 Error Recurrence

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

Three easily identifiable variations of the recurrence phenomenon were noticed during the initial category development (Chapter 4.1.3). First, there were cases where the exact same source had been recompiled (either immediately, or with an intervening change or changes that were then reverted); secondly, there were cases where a minor edit was performed to the single line which was identified as erroneous by the diagnostic message produced by the compiler, which did not resolve the original error and which usually resulted in the same diagnostic message being produced again; finally, edits were sometimes made elsewhere in the source, rather than on the line identified as erroneous by the diagnostic message, which was left unchanged (leaving the error intact and causing the same diagnostic to be produced again).

Therefore, recurrence was defined as any of:

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

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

4.1.5 Categorisation Analysis

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

1. Frequency of errors

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

2. Number and frequency of diagnostic messages

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

3. Coverage level of the most frequent error categories

The coverage (relative number of analysed events) of the top N most frequent error categories was measured, for N=5 through 30 in increments of 5. 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 targeting particular categories of error, or to identify potential improvements to diagnosis of particular categories of error.

4.2 Methodology – Part II

The second part of the work presented in this thesis extends the methodology of the first part (Chapter 4.1). We have shown that it is possible to produce a category hierarchy enabling a reliable categorisation of programmer errors, and performed an analysis of the relative frequency of errors in the different categories; in the second part, we extend this analysis to make a determination of the severity of different error categories, rather than just their frequency, and we apply this to a much larger data set, comprising 1000 different error manifestations.

4.2.1 Data Collection

For this second study, data was taken from the Blackbox Data Collection Project Brown et al. (2014), as in the secondary data collection in the first study (Chapter 4.1.2). A total of 1000 compilation events corresponding to a compilation error, corresponding to 199 user sessions and recorded in the period from 2013-06-11 to 2015-05-30, were randomly selected and extracted from the Blackbox database and made accessible via the same web interface that was developed for the previous part of the study (Chapter 4.1.3.2).

To recap, the Blackbox data is from users of BlueJ (Vee et al. 2006) globally who choose to allow data on their use of the environment to be recorded. Data pertinent to this study are the source code for each source file in the user's project at the time of a compilation, and any compiler diagnostic generated (which is not used directly for analysis, but acts as an aid to categorisation). All compilation events are associated with a session belonging to a particular user (distinguished by a numeric identifier for purpose of anonymity).

4.2.2 Refinement and Validation of Category Hierarchy

In Chapter 4.1.3 we discussed the development and validation of a category hierarchy for categorising programming errors. For this second study, we used the category hierarchy developed at that stage as the basis for a refined category hierarchy; refinement of this hierarchy was performed during the categorisation of 1000 error events. Categories were added when, during categorisation, it was noticed that the addition of a new category would allow more precise categorisation of several error events. Once categorisation was complete the revised category hierarchy was reviewed to improve consistency of phrasing in category names and descriptions, and eliminate redundant categories.

The validation of error categories was again performed by testing inter-coder reliability. In this instance, five researchers with experience in Java programming were recruited from the School of Computing at the University of Kent; each was assigned starting point within the data set of 1000 compilation events and asked to categorised as many errors as they could within the time available to them.

During investigation of disagreements in categorisation between researchers in the first study, we noticed that some categorisations may have been made without regard to certain factors within the source code. For example, in some cases, an error was classified as a misspelling of a variable name by one researcher, and as use of an undeclared variable by another researcher. We considered it likely that in this instance, one of the researchers had overlooked the existing declaration of a similarly-named variable, leading to categorisation as an undeclared variable use, whereas the other researcher had noticed this declaration and considered it likely that the two similar variable names were intended to refer to the same variable. To overcome problems of this nature, we ran a second phase of categorisation where each participating researcher was shown each event they had categorised, together with the categorisations made by other researchers and the explanation that they had provided for their choices; researchers were then permitted to keep their original categorisation choice, or switch their choice to match that of another researcher (the categorisations and reasoning were presented without revealing the identity of the researcher who had made them). This gave the researchers access to the reasoning of the other researchers, with the intention that they could make use of information that they had overlooked during their initial category selection.

Once category selections had been revised, pairwise agreement level was calculated. Allowing revision of selections produces a more accurate indication of the viability of the category scheme than the single selection as used in the first study (Chapter 4.1.3), since it alleviates the issue of overlooked details resulting in errors being miscategorised as described above. In Chapter 4.1.3.3 we describe an algorithm for calculating pairwise agreement in the presence of multiple error categorisations, which does not count absence of any categorisation by one member of any pair combination as disagreement, based on the perception that some researchers had focussed on a single error without regarding or detecting a second; with the ability to revise selections based on insight provided by others performing the categorisation, this assumption no longer holds, and so the algorithm for calculating pairwise agreement was adjusted accordingly:

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

The difference from the algorithm used previously is underlined. This alteration causes an excess of categorisations made by one researcher when compared to another to be counted as disagreement, rather than to be counted as neither disagreement nor agreement.

4.2.3 Analysis

The following analyses were performed over the categorisations we made over a total of 1000 compilation events from 199 user sessions, after the category revision process already described (categorisations made by other researchers for purpose of category validation were not considered during these analyses):

1. Frequency of error categories

As in the prior study, the frequency of errors was calculated by counting the relative occurrence of each category (from all data points included in the categorisation), and a list of errors ranked by frequency was produced.

2. Coverage level of the most frequent error categories

Again as in the prior study, the coverage (relative number of analysed events) of the top N most frequent error categories was measured, for N=5 through 30 in increments of 5.

3. Error Difficulty

In Chapter 2.7 the concept of error severity as a product of *frequency* and *difficulty* was introduced. While counting occurrences to determine the frequency of an error category is straightforward, a suitable metric for estimating difficulty is required. In the analysis presented of this second study, *time-to-fix* 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 categorised errors in the in the data set. For any given error event, the resolution status was also 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 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.

The resolution time of a resolved error was approximated by the time between the recorded event where the error is first observed and categorised, and the recorded event where the error is classified as *resolved*. The error may have been introduced by editing the source code before the recorded event where it was observed, since events are recorded only upon compilation and not on edit; likewise an error may have been fixed earlier than the event in which it can be seen as resolved. However, the size of the measurement error is bounded in the presence of preceding and succeeding events, and furthermore, evidence exists that novices tend to compile immediately after making edits (Jadud 2005).

An error is considered unresolved if code is reverted to a prior, error-free state, or if a period of 300 seconds (5 minutes) elapses with no compilation events recorded while the error is present. Given novice programmer tendency to make rapid changes while attempting to resolve an error and to compile after a small number of changes (Jadud 2005), a period without compilation of this length of time indicates that the novice may not be able to resolve the error themselves; it is also possible that they are waiting for assistance from a tutor, have moved on to another task, or have gone for a break. Because the cut-off is 5 minutes, we also clamped resolution time (in cases where there is continue compilation activity and the error is apparently resolved) to 5 minutes. While this clamping certainly has an effect on our calculated severity value, it is likely that an error taking longer than this to resolve has been problematic, and the majority of errors should take significantly less time to resolve if the student understands the nature of the error (or is able to solve it with assistance from the compiler-generated diagnostic); clamping should reduce the possibility for outliers to skew the results.

For each user session in which an error had been categorised, the resolution classification procedure examined recorded compilation events in turn, ordered by their timestamps. The algorithm used for classification relied on the previous identification of recurring errors, as described in Chapter 4.1.4.2. It operated as follows:

1. Let the *active error* be the first error encountered in the session, as decided by the error categorisation (note that in a few cases this is actually more than a single error category selection – in this case actions applied to the *active error* are applied to each selected category).
2. Examine the next event in sequence:
    1. If more than 5 minutes elapse since the previous event, assume that the programmer has taken a break during the session and mark the active error as *unresolved*. Let the *active error* become the error category selected in next event containing an error (which is possibly the event being examined currently).
    2. Otherwise, if the event is in a different project than the previous event, mark the *active error* as *unresolved*. Let the *active error* be taken from the next event with an error (possibly the even being examined currently) and continue from that event.
    3. Otherwise, if the event does not contain an error, check if the source code has been reverted to the state it was in immediately prior to the active error.
        1. If the source code has been reverted, mark the *active error* as *reverted*. Let the *active error* be taken from the next event with an error and continue from that event.
        2. Otherwise, mark the *active error* as *resolved*, and record the resolution time as that between the active error and current even, clamped to a maximum of 5 minutes. Let the *active error* become the next event with an error and continue from that event.
    4. Otherwise, if the event is marked as *recurring*, check whether this marking is due to the source being reverted to an earlier state. If so:
        1. If the active earlier state to which the source was reverted precedes the *active error*, mark the *active error* as reverted. Let the *active error* be the error from the reverted-

        to-event.
    2. Otherwise, consider the reversion to be part of the process of attempting to solve the error. Repeat step 2 with the next even in the sequence.
  5. Otherwise, if the event is marked as *recurring* (but does not represent a source code reversion), repeat step 2 with the next event in the sequence.
  6. Otherwise, if the diagnostic issued for the error indicates the same source file and the same or earlier line number than the active error, assume that a new error was introduced by the programmer. Mark the *active error* as *uncertain* resolution, and then let the *active error* be taken from the event being currently examined.
  7. Otherwise, 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.
3. Repeat step 2 until all events in the session have been processed.

The time (clamped to 5 minutes) between the origin even of a resolved error and its resolution event was used as the estimate of time-to-fix for the error. 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 unresolved and reverted instances and of instances where resolution was uncertain.

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

4. Error Severity

The total resolution time of errors in each error category is calculated during the determination of error difficulty, as described above. An adjusted total resolution time, intended to factor unresolved errors into the difficulty metric, is then produced by adding five minutes to the total resolution time for each *unresolved* or *reverted* instance of each error category (five minutes is the maximum allowed for resolution of error according to the resolution status process outlined previously). This calculation is intended to produce a value which reflects the time spent attempting to resolve an error, whether it was resolved successfully or not; it is based on the assumption that an error classified as unresolved occupied at least 5 minutes of the subject's time as they attempted to solve it – which is likely if the error was marked unresolved due to remaining present in the source code for the threshold period, though is difficult to ascertain one way or the other for other cases where errors may be marked unresolved, such as when an error is removed by means of reverting the source code to a prior state.

The adjusted average resolution time for a category is calculated by dividing the adjusted total resolution time by the total number of *resolved*, *unresolved*, and *reverted* instances. The resulting value is used as an estimation of average time-to-fix, and therefore *difficulty*, of the error categories; when expressed as a value in seconds, this allows a calculation of severity, using the simple formula proposed in Chapter 2.7:

$$\textit{severity} = \textit{frequency} \times \textit{difficulty}$$

Using this calculation produces a *severity* value which can be used to assess the relative severity of different error categories. With frequency expressed as a proportion of the events in which an error occurs, the severity is in the range of 0 to 300 due to the 5 minute resolution limit (a difficulty metric of 300 seconds). A list of error categories was produced, ranking them by severity.

4.3 Diagnosis Tool Prototype

The identification of the most severe error categories may be useful for development of pedagogical practice and material related to programming, and also to the development of tools (such as compilers and development environments) designed for use by novice programmers. Having produced a ranking of error category severity, and gathered examples of reasoning used by experienced programmers to diagnose novice errors, we then designed and implemented a prototype tool to assess the feasibility of producing accurate and precise diagnosis of several of the more severe errors, and proceeded with an evaluation of this tool.

Errors to be diagnosed were chosen based on their severity, and their ability to be diagnosed using contextual clues present in the source code that would not typically be used by a compiler in making an error diagnosis, but which could be used by an experienced programmer or an automated system to perform an accurate and precise diagnosis.

The tool was developed by making additions and modifications to the BlueJ open-source development environment (Kölling et al. 2003).

Evaluation of the tool was performed by comparing its diagnosis with the standard compiler-generated diagnosis on a number of source code snapshots taken from the data set of novice code gathered for the process of error category severity determination (Chapter 4.2.1). The source code snapshots used for this purpose had been categorised with high-severity errors that had been selected during the tool design as targets for automated diagnosis. For each snapshot, the compiler diagnostic was compared to that from the tool; for each diagnostic a judgement regarding the accuracy was made (as a binary result of either *accurate* or *not accurate*), and in cases where both the compiler and the tool had produced an accurate diagnostic, the precision of the diagnostic generated by the tool was judged in comparison to that produced by the compiler, as either more precise, equivalent in precision, or less precise.

Details of the design of the tool and results of its evaluation are presented in Chapter 7, and a record of the code snapshots used for judgement together with the judgements regarding accuracy and precision of the diagnostics is included in Appendix D.

5 Results and Discussion - I

In this chapter the results of the first part of the research introduced in this thesis (Chapter 4.1) is presented and discussed. The results include a set of programming error categories and a frequency distribution of the occurrence of errors in those categories across the data set. We also show the coverage level of the most frequent *N* categories for various values of *N*, the frequency of diagnostic messages, and the level of intercoder reliability calculated during validation of the category scheme.

5.1 Results of Analysis

5.1.1 Error Categories

A total of 80 error categories were identified during the category development phase of the research (described in Chapter 4.1.3). Broadly, errors can be divided into *syntactic*, *semantic*, and *logic* errors; logic errors do not usually cause compilation failure, and since the analysis focussed on errors detected during compilation, logic errors are not represented in the resulting error categories. We will now discuss some of the errors that were identified; a full list of the error categories is provided in Appendix A.

5.1.1.1 Syntax error categories

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

5.1.1.2 Semantic error categories

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

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

5.1.2 Intercoder reliability

After three separate researchers performed categorisation of the same error set, the average pairwise agreement was calculated (as discussed in Chapter 4.1.3.3). The agreement across all events and all categories was 70.06%. This figure could be considered an indicator of a modest level of agreement. However, agreement is not uniformly distributed over all error categories. Agreement tended to be higher in more common categories, and lower in less frequent, more obscure cases. Since it is likely that future work (both pedagogical or in tool improvements) will concentrate on the most frequent errors, it is interesting to investigate agreement levels separately for this group.

Where all categorisations outside the top 10 most frequent categories were excluded from the agreement calculation, pairwise agreement was 92.5%. For the top 20 categories, agreement is 87.4%. This indicates a good level of researcher agreement in classification of the errors.

To allow simplified calculation, pairwise agreement did not take the category hierarchy into account. A selection by one researcher of a category which was a subcategory of that selected by another researcher was counted as disagreement, whereas in fact this would be indicative of partial agreement. The impact of this is potentially quite significant. For the "Class name written incorrectly" category for example, all three researchers agreed on the category for exactly 50% of the events for which this category was selected. For the remaining 50%, two researchers chose this category and the third chose the “Class not defined” super-category. Thus, at the higher category level for this particular category, there is 100% agreement. As a result, the agreement levels reported above represent a conservative lower bound.

5.1.3 Frequency of Error Categories

A total of 368 categorisations were applied to 333 events. The frequency of the top 10 error categories is shown in Table 5.1.

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

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

Table 5.1: The top 10 most frequent error categorisations

5.1.4 Frequency of Diagnostic Messages

Each of the 333 compilation events analysed had an associated compiler diagnostic message. To judge whether the manual classification provides an improvement over classification of diagnostic messages, it is useful to compare Table 5.1 to the equivalent table if diagnostic messages are used for classification. The messages often incorporate variable, contextual text such as user identifiers (variable, method and class names); the general form of such messages was manually extrapolated. The frequency of the top 10 messages is shown in Table 5.2.

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

Table 5.2: Diagnostic message frequency. Variable parts of the messages are shown as text in curly braces.

5.1.5 Category Coverage

Coverage levels for the top N categories were calculated for N=5, 10, 15, 20, 25 and 30. The coverage level represents the relative number of events that are categorised by at least one of top N categories. The results are presented in Table 5.3.

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

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

Table 5.3: Coverage levels of top N categories for N=5,10,...,30

5.2 Discussion

5.2.1 Diagnostic Message Frequencies

Although we have argued that diagnostic messages do not provide a satisfactory means of identifying errors, they are not completely disconnected, and comparing the frequency of diagnostic messages in our study with those reported elsewhere shows similarities which suggest that the error frequencies (as determined by categorisation) may be representative of that encountered in other cohorts.

Jadud (2005) published a distribution of diagnostic messages encountered by novices. He finds the “missing semicolon” message to be the most frequent at 18%, with “unknown symbol – variable” next at 12%. “bracket expected”, “illegal start of expression” and “unknown symbol: class” follow at 12%, 9% and 7% respectively.

Jackson et al. (2005) found a somewhat different distribution in their own study, also based on diagnostic messages: they list “cannot resolve symbol” (which possibly includes both “unknown variable” and “unknown class” errors) as the most frequent at 14.6%, with “; expected” next at 8.5%. “illegal start of expression” and “) expected” also feature in the top 6 most frequent diagnostic messages, with “class or interface expected” and “<identifier> expected” both appearing in the top 5 (they appear in Jadud’s distribution at position 8 and 9 respectively).

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

There are several factors that could explain some of the differences in the observed distributions across the three studies: different compiler versions may have been used, and different cohorts participated. Neither Jadud (2005) nor Jackson et al. (2005) describe any attempt to eliminate recurrent errors from their results as was performed in the work presented in this thesis (Chapter 4.1.4.2).

5.2.2 Error Category Frequency Compared to Diagnostic Message Frequency

The top 8 categories (Table 5.1) cover just over 50% of events; in contrast, the top 4 diagnostic messages cover roughly the same amount (52.8%). This fact alone demonstrates that the categories described here are in general more precise than the compiler’s diagnostics. While each category describes an error that will cause a diagnostic message to be generated by *javac*, the diagnostic message does not in general reflect the same precision as the category (see examples in Section 5.1.1).

There are also categories for which the associated diagnostic messages are not just imprecise, but are inaccurate. The “keyword written incorrectly” category, for instance, corresponds to the following diagnostic messages in the data set used for analysis:

> *cannot find symbol - variable flase*
> *cannot find symbol - variable True*

These occurred due to misspelling and incorrect capitalisation of the literals ‘false’ and ‘true’, respectively. The compiler message, however, refers to the lack of definition of a variable. While it is true that no variable defined as ‘flase’ or ‘True’ existed, it is unlikely that a variable reference was being attempted in either case.

5.2.3 Coverage Levels of Error Categories

The category coverage analysis shows that a small proportion of the categories cover a significant portion of errors (Table 5.3). The top 10 categories cover well over 50% of errors; as more categories are added, the coverage increases, but the increase diminishes as N increases (where N is the number of categories). This implies that pedagogical interventions (or error diagnostic tools) would be effective by focusing on a select number of the most frequent types of error.

5.3 Summary

The method used for production and application of a category hierarchy for programming errors has been shown to be viable. When considering the 10 most common categorisations, covering approximately 60 percent of the error events, a pairwise agreement level of over 90 percent was achieved. We have also demonstrated that this categorisation often provides a more specific, and in some cases more accurate, means of classifying novice errors than does performing such classification using the diagnostic message from a compiler.

6 Results and Discussion – II

In this chapter the results of the second study conducted as part of the work introduced in this thesis (4.2) is presented and discussed. The results include a set of programming error categories, revised from those presented in the previous chapter, and a frequency distribution of the occurrence of errors in those categories across the data set. We show the coverage level of the most frequent *N* categories for various values of *N*, and the level of intercoder reliability calculated during validation of the category scheme. Finally, we will show the average resolution time, and resolution rate, and how they are combined to form an adjusted average resolution time; this is used as a metric for the *difficulty* component in a severity calculation for each of the categories (where severity is a product of difficulty and frequency, as discussed in Chapter 2.7). The resulting severity of the error categories, grouped by top-level category, is presented.

6.1 Results of Analysis

6.1.1 Error Categories

A total of 90 error categories were identified during the category development phase described in Chapter 4.2.2. In comparison to the first study, the categories were formally organised into a hierarchy and during category selection were presented in a tree representing this hierarchy rather than a flat list. 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 categorisation 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
- Uncategorised

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 neatly fit into any other high-level category, so remained as a category in its own right – although it is arguably always a syntactic error, since it runs against grammatical rules, it also appears to represent a misconception of the semantics of relevant language constructs.

The remaining top-level category, *uncategorised*, allows for a selection to be made when an appropriate category cannot otherwise be decided. A selection of *uncategorised* counts as a category selection for purposes of analysis; for the pair-wise agreement calculated for purposes of category scheme validation, disregarding *uncategorised* selections would not properly count a disagreement where one researcher thought the error belonged to a particular category but another felt that no category was applicable, for example.

A complete list of the error categories (ordered by frequency of occurrence) is given in Appendix B, and the category hierarchy is shown in Appendix C.

6.1.2 Frequency of Error Categories

A total of 1011 error categorisations were made for the 1000 source code snapshots making up the data set (the difference in number being due to multiple errors being categorised for a small number of events, as discussed in 4.1.4.1) The most frequent error was “Variable not declared”, accounting for approximately 8.4% of all errors. In the first study, the same category was also the most frequent, in that case accounting for 11.1% of the recorded errors.

Table 6.1 shows the relative frequency of the top 10 errors in this study.

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%

Table 6.1: Frequency of error categories. The frequency is expressed in terms of the number of categorisations per error.

There is a high level of concordance between the most frequent error categories in this study and the first study (Chapter 5.1.3). “Variable not declared”, “Variable name written incorrectly” and “; missing” are again the three most frequent error categories (though in the first study the latter two are in reversed order); The “Invalid syntax” category which places fourth in the first study matches neatly with the “Simply syntactical error” from this study; “Class or type name written incorrectly” and “Method name written incorrectly” also feature in the top 10 in both studies. The differences that do exist can be attributed partially to the refined category hierarchy, though there is likely also a level of variability in the data.

The categories of “Simple syntactical error”, “Semantic error”, “Variable: Incorrect variable declaration”, “Variable: Incorrect attempt to use variable” and “Method: Incorrect method declaration”, which all appear in the top 10 categories by frequency (Table 6.1), are top-level categories in the category hierarchy. Their presence in the top 10 is not surprising, since a top-level category offers a broad categorisation which satisfies errors not belonging to a more specific subcategory, though not all of the top 10 categories were top-level. In Table 6.2, the frequency counts for subcategories were folded into the top-level category counts for a better comparison between all the top-level categories.

Top-level category	Total occurrences
Variable: Incorrect attempt to use variable	220 (21%)
Simple syntactical error	201 (19%)
Method: Incorrect method call	199 (19%)
Semantic error	119 (11%)
Uncategorized	94 (9%)
Method: Incorrect method declaration	87 (8%)
Incorrect/attempted use of class or type	66 (6%)
Variable: Incorrect variable declaration	58 (5%)
Constructor: Incorrect constructor call	9 (1%)
Constructor: Incorrect constructor declaration	4 (< 1%)
Statement outside method/block	2 (< 1%)

Table 6.2: Error frequency by top-level category. Subcategory frequencies have been folded into the top-level to provide a direct comparison between the top-level categories.

Table 6.2 shows that errors involving use of variables or method calls, as well as simple syntactical errors, have a similar frequency that is relatively high compared to other top-level error categories. The next most frequent top-level category, "Semantic error", occurs at a significantly lower rate.

6.1.3 Error Difficulty

As described in Chapter 4.2.3 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. 6 instances of resolved errors had a resolution time greater than 5 minutes (the value was clamped for calculating the time-to-fix in severity calculation).

6.1.3.1 Resolution Status

Error instances were classified as resolved, unresolved, uncertain or reverted (where reverted is a special case of unresolved). Table 6.3 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.

Error	Occurrences	Resolved	Unresolved	Reverted	Average Time-to-fix (seconds)	% unresolved
Missing ‘return’ statement	23	13	6	0	65	26%
Uncategorised	94	56	18	1	48	21%
Type error	38	25	7	0	34	18%
Method call: parameter number mismatch	27	19	2	2	33	15%
Variable: Incorrect variable declaration	48	27	6	1	41	15%
Method not declared	29	20	3	1	47	14%
Use of non-static method from static context	16	13	2	0	60	13%
Method call targeting wrong type	25	18	2	1	80	12%
Semantic error	49	34	5	0	52	10%
Method call: parameter type mismatch	10	8	1	0	84	10%

Table 6.3: Resolution failure: The top 10 precise error categories by frequency of failure to resolve the error.

6.1.3.2 Resolution Time

Table 6.4 shows the top 10 error categories by average resolution time. As before, error categories with fewer than 10 occurrences were excluded.

Error	Occurences	Resolved	Unresolved	Reverted	Average Time-to-fix (seconds)	% unresolved
Method call: parameter type mismatch	10	8	1	0	84	10%
Method call targeting wrong type	25	18	2	1	80	12%
Method declaration: missing return type	10	6	0	0	74	0%
(d) Method: Incorrect method declaration	46	24	3	0	70	7%
Missing 'return' statement	23	13	6	0	65	26%
Class from other package used without 'import'	13	5	0	0	61	0%
Use of non-static method from static context	16	13	2	0	60	13%
Semantic error	49	34	5	0	52	10%
(c) Method: Incorrect method call	20	15	0	1	51	5%
Uncategorised	94	56	19	1	48	21%

Table 6.4: Time-to-fix: top 10 error categories when ordered by average resolution time.

A scatter-plot of all error categories showing number of occurrences against average resolution time is shown in Figure 6.1. Different symbols are used to represent the different top-level categories.

The pink highlight area corresponds to more than 10 occurrences and an average resolution time of more than 10 seconds. Thus, the highlight area identifies more severe errors: those errors that either occur relatively rarely, or that are usually quickly fixed, are in the graph area with white background along the axes, while the more severe errors are towards the top right quadrant of the graph. The plot excludes error categories with no recorded resolutions.

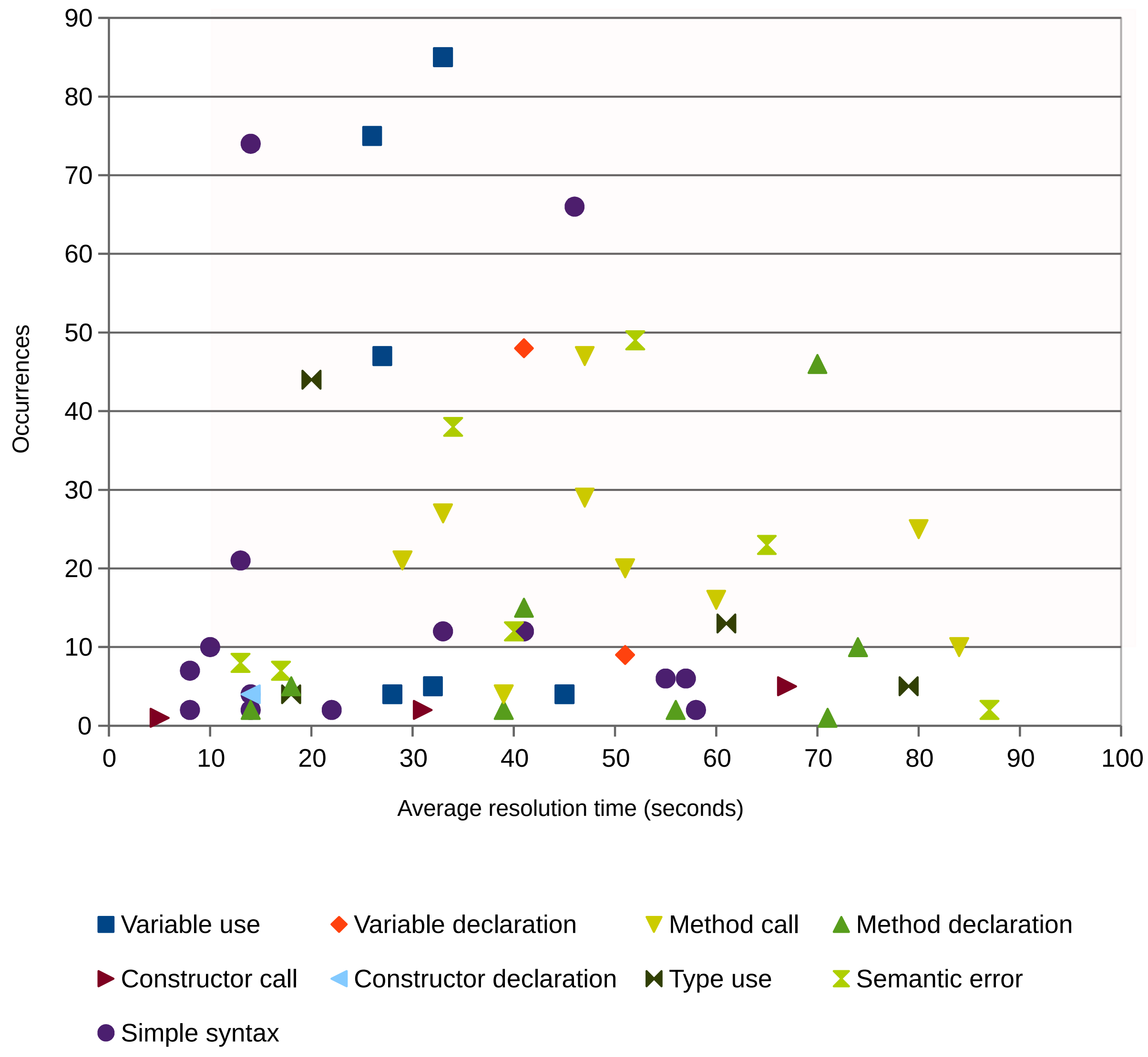


Figure 6.1: Resolution time and frequency of occurrence of error categories

6.1.3.3 Adjusted Resolution Time

Average time-to-fix gives an indication of the difficulty of different error categories. The time-to-fix value for an error category is the average resolution time in seconds, after clamping all resolution times to a maximum of 300 seconds (as discussed in Chapter 4.2.3). The resolution rate is, however, also an important consideration; a particular error 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 beyond which an error would automatically be determined as unresolved, as described in Chapter 4.2.3. 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 (errors whose resolution status is 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 6.5.

Category	**Occurred**	**Resolved (total)**	**Uncertain resolutions**	**Unresolved**	**Reverted**	**Average resolution (seconds)**	**Total resolution time**	**Penalty**	**Adjusted total**	**Adjusted average**
Missing 'return' statement	23	13	4	6	0	65	1495	1800	3295	173
Class from other package used without 'import'	13	5	8	0	0	61	793	0	793	159
Method: Incorrect method declaration	46	24	19	3	0	70	3220	900	4120	153
Uncategorised	94	56	18	19	1	48	4512	6000	10512	138
Method call targeting wrong type	25	18	4	2	1	80	2000	900	2900	138

Category	Occurred	Resolved (total)	Uncertain resolutions	Unresolved	Reverted	Average resolution (seconds)	Total resolution time	Penalty	Adjusted total	Adjusted average
Method call: parameter type mismatch	10	8	1	1	0	84	840	300	1140	127
Method declaration: missing return type	10	6	4	0	0	74	740	0	740	123
Variable: Incorrect variable declaration	48	27	14	6	1	41	1968	2100	4068	120
Method not declared	29	20	5	3	1	47	1363	1200	2563	107
Type error	38	25	6	7	0	34	1292	2100	3392	106
Use of non-static method from static context	16	13	1	2	0	60	960	600	1560	104
Semantic error	49	34	10	5	0	52	2548	1500	4048	104
Type mismatch in assignment	12	9	1	2	0	40	480	600	1080	98
Method call: parameter number mismatch	27	19	4	2	2	33	891	1200	2091	91
Simple syntactical error	66	48	13	3	2	46	3036	1500	4536	86
Method: Incorrect method call	20	15	4	0	1	51	1020	300	1320	83
Mismatched parentheses in or around expression	10	4	5	1	0	10	100	300	400	80
Variable: Incorrect attempt to use variable	47	29	15	1	2	27	1269	900	2169	68
Variable not declared	85	63	16	4	2	33	2805	1800	4605	67
Class or type name written incorrectly	44	25	16	2	1	20	880	900	1780	64
Method name written incorrectly	47	35	12	0	0	47	2209	0	2209	63
Variable name written incorrectly	75	47	25	2	1	26	1950	900	2850	57
Missing operator between expression elements	12	9	3	0	0	41	492	0	492	55
Extraneous closing curly brace	21	14	5	2	0	13	273	600	873	55
Wrong return type declared	15	12	3	0	0	41	615	0	615	51
Missing parentheses for method call	21	18	2	1	0	29	609	300	909	48
; missing	74	50	21	3	0	14	1036	900	1936	37
Missing closing curly brace at end of class	12	11	1	0	0	33	396	0	396	36

Table 6.5: Adjusted average resolution times. Errors are ordered by adjusted resolution time, which combines resolution times and unresolved errors. For unresolved errors, a nominal resolution time of 300 seconds was assigned.

A plot of adjusted average resolution time versus occurrence count for all categories (excluding those with only uncertain resolutions) is given in Figure 6.2. 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 6.3.

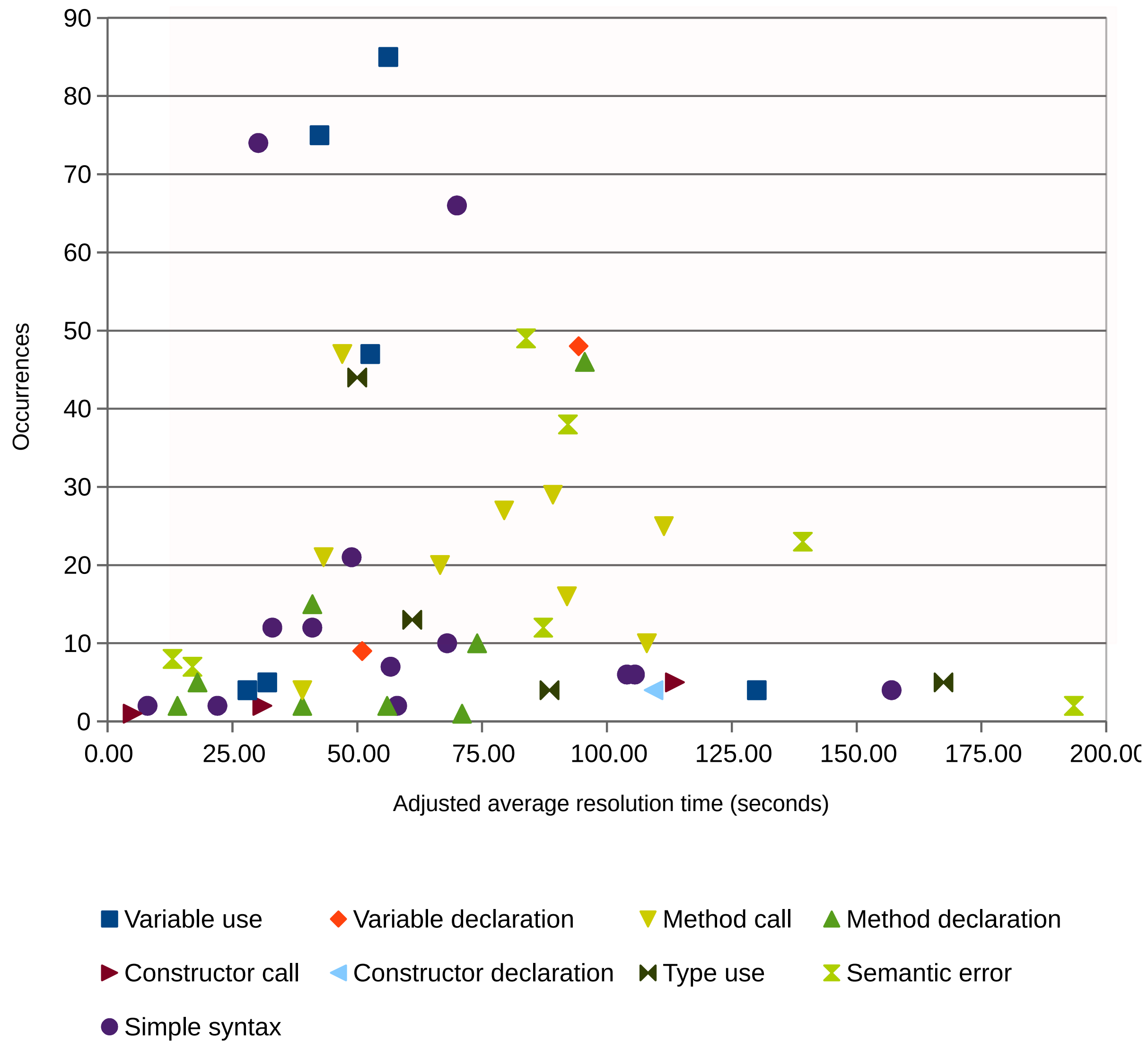


Figure 6.2: Adjusted resolution time and frequency of occurrence of error categories (all categories)

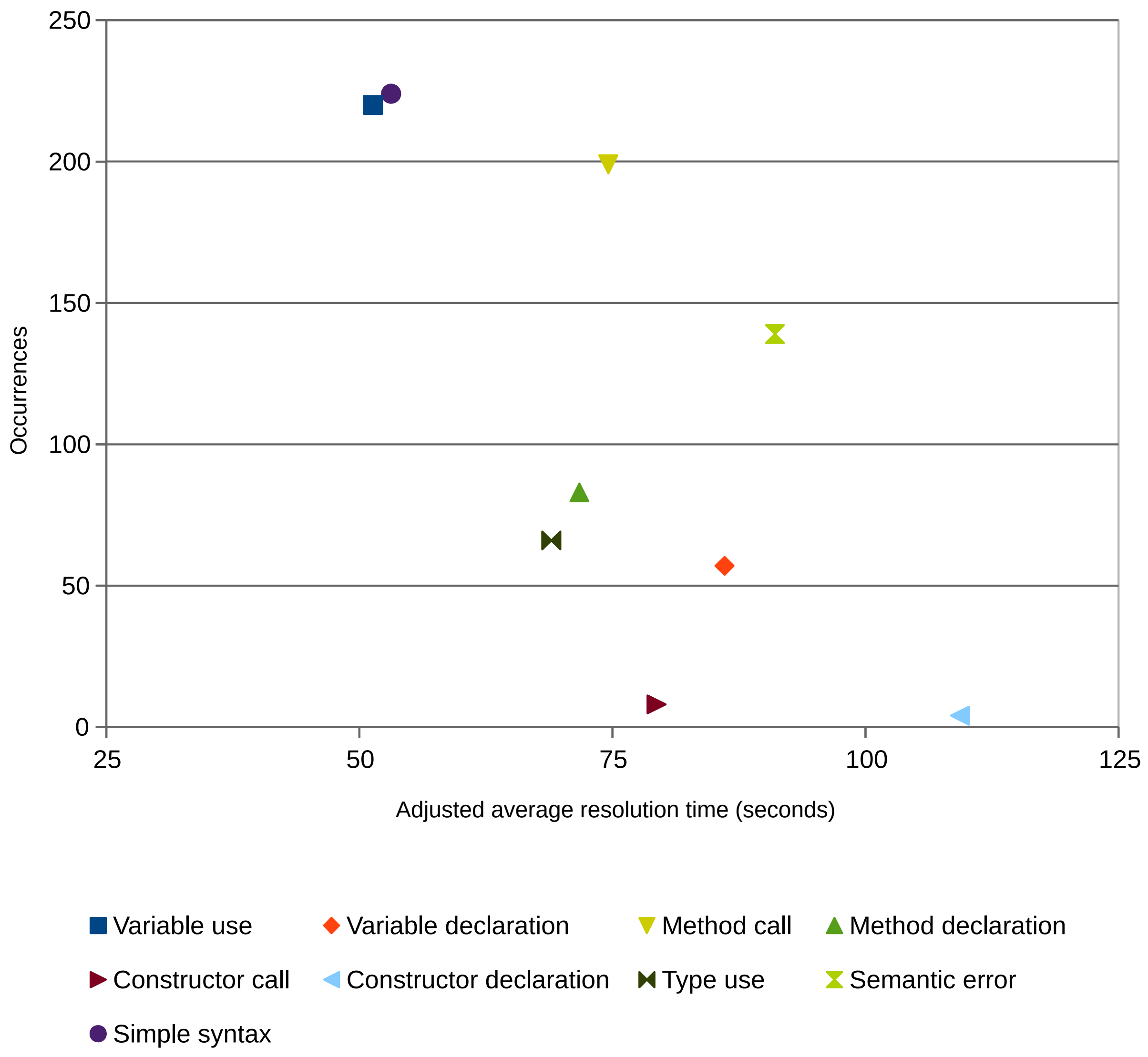


Figure 6.3: Adjusted resolution time and frequency of occurrence: mean values of the nine top-level categories

6.1.3.4 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 Chapter 2.7, 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. The severity of each error category is shown (in descending order) in Table 6.6.

Category	**Occurred**	**Resolved (total)**	**Unresolved + Reverted**	**Uncertain resolutions**	**Average resolution (sec.)**	**Adjustment**	**Adjusted Total resolution time**	**Adjusted avg. resolution time**	**Severity**
Variable not declared	85	63	6	16	33	1800	3879	56	4.8
Simple syntactical error	66	48	5	13	46	1500	3708	70	4.6
Variable: Incorrect variable declaration	48	27	7	14	41	2100	3207	94	4.5
Method: Incorrect method declaration	46	24	3	19	70	900	2580	96	4.4
Semantic error	49	34	5	10	52	1500	3268	84	4.1
Type error	38	25	7	6	34	2100	2950	92	3.5
Missing 'return' statement	23	13	6	4	65	1800	2645	139	3.2
Variable name written incorrectly	75	47	3	25	26	900	2122	42	3.2
Method call targeting wrong type	25	18	3	4	80	900	2340	111	2.8
Method not declared	29	20	4	5	47	1200	2140	89	2.6
Variable: Incorrect attempt to use variable	47	29	3	15	27	900	1683	53	2.5
; missing	74	50	3	21	14	900	1600	30	2.2
Method name written incorrectly	47	35	0	12	47	0	1645	47	2.2
Class or type name written incorrectly	44	25	3	16	20	900	1400	50	2.2
Method call: parameter number mismatch	27	19	4	4	33	1200	1827	79	2.1
Use of non-static method from static context	16	13	2	1	60	600	1380	92	1.5
Method: Incorrect method call	20	15	1	4	51	300	1065	67	1.3
Method call: parameter type mismatch	10	8	1	1	84	300	972	108	1.1

Category	Occurred	Resolved (total)	Unresolved + Reverted	Uncertain resolutions	Average resolution (sec.)	Adjustment	Adjusted Total resolution time	Adjusted avg. resolution time	Severity
Type mismatch in assignment	12	9	2	1	40	600	960	87	1.0
Extraneous closing curly brace	21	14	2	5	13	600	782	49	1.0
Missing parentheses for method call	21	18	1	2	29	300	822	43	0.9
Incorrect/attempted use of class or type	5	3	2	0	79	600	837	167	0.8
Class from other package used without 'import'	13	5	0	8	61	0	305	61	0.8
Method declaration: missing return type	10	6	0	4	74	0	444	74	0.7
Mismatched parentheses in or around expression	10	4	1	5	10	300	340	68	0.7
'=' used in place of '=='	6	4	1	1	57	300	528	106	0.6
Keyword written incorrectly	4	1	1	2	14	300	314	157	0.6
Missing closing curly brace after control stmt body	6	4	1	1	55	300	520	104	0.6
Wrong return type declared	15	12	0	3	41	0	492	41	0.6
Method declaration: missing method body	2	0	2	0	N/A	600	600	300	0.6
Constructor parameter number mismatch	5	4	1	0	67	300	568	114	0.6
Use of variable from another class	4	2	1	1	45	300	390	130	0.5
Missing operator between expression elements	12	9	0	3	41	0	369	41	0.5
Variable needs a unique name	9	8	0	1	51	0	408	51	0.5
Constructor: Incorrect constructor declaration	4	2	1	1	14	300	328	110	0.4
Missing closing curly brace at end of method	7	5	1	1	8	300	340	57	0.4
Missing closing curly brace at end of class	12	11	0	1	33	0	363	33	0.4
Class does not implement required method	2	1	1	0	87	300	387	194	0.4
Class not defined	4	3	1	0	18	300	354	89	0.4
Constructor parameter type mismatch	1	0	1	0	N/A	300	300	300	0.3
: in place of ;	1	0	1	0	N/A	300	300	300	0.3
Invalid type cast	1	0	1	0	N/A	300	300	300	0.3

Category	Occurred	Resolved (total)	Unresolved + Reverted	Uncertain resolutions	Average resolution (sec.)	Adjustment	Adjusted Total resolution time	Adjusted avg. resolution time	Severity
Use of non-static variable from static context	5	5	0	0	32	0	160	32	0.2
Method call: Parameter types included	4	3	0	1	39	0	117	39	0.2
Unhandled exception	7	7	0	0	17	0	119	17	0.1
Extra opening curly brace '{'	2	2	0	0	58	0	116	58	0.1
Method duplicated exactly	2	2	0	0	56	0	112	56	0.1
Wrong variable used	4	3	0	1	28	0	84	28	0.1
Array used in place of array element	8	4	0	4	13	0	52	13	0.1
Method declaration: Wrong number of parameters	5	2	0	3	18	0	36	18	0.1
Missing opening curly brace after method header	2	1	0	1	39	0	39	39	0.1
Method declaration: Semicolon at end of method header	1	1	0	0	71	0	71	71	0.1
Constructor: Incorrect constructor call	2	2	0	0	31	0	62	31	0.1
Missing closing curly brace	2	2	0	0	22	0	44	22	<0.1
Statement outside method/block	2	1	0	1	14	0	14	14	<0.1
Method declaration: return type should be void	2	2	0	0	14	0	28	14	<0.1
Extraneous or misplaced ')'	2	1	0	1	8	0	8	8	<0.1
Missing parentheses for constructor call	1	1	0	0	5	0	5	5	<0.1

Table 6.6: Error Category Severity. 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.

6.1.3.5 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 6.7.

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

Table 6.7: Error coverage for most frequent categories: 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% of all encountered error events)

6.1.3.6 Intercoder Reliability

As detailed in Chapter 4.2.2, inter-coder reliability was assessed using a modified pairwise agreement calculation. The original calculation as used in the first study, shown only for reference, 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; The revised calculation (pessimistic) was performed in which such a discrepancy does count as disagreement.
The agreement levels were as follows:

All categories:	84% (optimistic)	80% (pessimistic)
Top 10 categories:	83% (optimistic)	80% (pessimistic)
Top 20 categories:	88% (optimistic)	82% (pessimistic)

6.2 Sensitivity to Design Choices

The severity values presented in Chapter 6.1.3 are calculated as a simple product of frequency and difficulty, where difficulty is expressed as an adjusted average resolution time, which clamps resolution times to a threshold value of 5 minutes and counts unresolved errors as having a resolution time of that same threshold value. Some reasoning for this choice of formula and threshold value have been discussed in Chapter 2.7 and Chapter 4.2.3. We believe that the proposed formulation of severity is useful and that the 5 minute threshold for error resolution is suitable; however, other formulations for severity are possible; similarly, a differing threshold value might well be used in an analysis of error severity. In this chapter we present a lightweight sensitivity analysis of these experimental design choices.

6.2.1 Alternative Error Resolution Threshold Values

Tables 6.8 and 6.9 show severity rankings of the top 20 error categories when the threshold value for error resolution is changed from 5 minutes to 2 minutes and 10 minutes, respectively, and can be compared with the ranking shown for the original threshold value in Table 6.6.

Note that severity values scale with the threshold; a 5-minute threshold implies a theoretical maximum possible severity score of 300, whereas the maximum severity is 120 for a 2-minute threshold and 6000 for a 10-minute threshold. However, it is not useful to normalise the severity since it depends on measured error resolution time which is the same (bar the effect of clamping) across different threshold values.

The resolution threshold value affects the classification of error resolution status, as explained in Chapter 4.2.3. Tables 6.8 and 6.9 include the count of resolved, unresolved/reverted and uncertain-resolution instances in each error category when the corresponding resolution threshold (2 minutes or 10 minutes) is applied..

Variable not declared remains the highest severity error category in all three cases. Table 6.10 shows the ten most severe error categories with a threshold value of 5 minutes, and their corresponding severity rank when the threshold is selected instead as 2 or 10 minutes. Although there are some minor differences, there is a good level of similarity between the rankings; 8 of the top 10 are common between the 2-minute threshold ranking and 5-minute threshold ranking, and 9 of the top 10 are common between the 5-minute threshold ranking and the 10-minute threshold ranking. 7 error categories are common in the top 10 of all three rankings.

The similarity in severity rankings across the three threshold values indicates that results are largely insensitive to threshold value selection. A larger threshold may give slightly more accuracy in the determination of time-to-fix by avoiding clamping, but may also increases the noise due to large outliers (which could be caused by factors other than difficulty, such as the subject becoming distracted while in the process of solving an error). We believe that the chosen threshold of 5 minutes has been shown to be useful for the purpose of ranking error categories by severity.

	Occurred	Resolved (total)	Unresolved + Reverted	Uncertain	Average resolution (secs)	Adj. Avg. resolution time (secs)	Severity
Variable not declared	85	61	12	12	26	41	3.5
Method: Incorrect method declaration	46	20	13	13	37	70	3.2
Semantic error	49	32	9	8	42	59	2.9
Simple syntactical error	66	43	11	12	24	44	2.9
Variable: Incorrect variable declaration	48	23	14	11	16	55	2.7
Variable name written incorrectly	75	44	7	24	16	30	2.3
Type error	38	24	8	6	27	50	1.9
Method name written incorrectly	47	32	4	11	29	39	1.8
Variable: Incorrect attempt to use variable	47	29	4	14	27	38	1.8
Method not declared	29	19	5	5	39	56	1.6
Missing 'return' statement	23	12	9	2	31	69	1.6
Method call targeting wrong type	25	15	7	3	33	61	1.5
; missing	74	49	6	19	8	20	1.5
Class or type name written incorrectly	44	24	5	15	15	33	1.5
Method call: parameter number mismatch	27	17	7	3	20	49	1.3
Method: Incorrect method call	20	14	3	3	44	57	1.1
Class from other package used without 'import'	13	4	2	7	43	69	0.9
Use of non-static method from static context	16	11	4	1	31	55	0.9
Method call: parameter type mismatch	10	6	3	1	53	75	0.8
Type mismatch in assignment	12	9	2	1	40	55	0.7

Table 6.8: Severity ranking for error resolution threshold of 2 minutes (120 seconds)

	Occurred	Resolved (total)	Unresolved + Reverted	Uncertain	Average resolution (secs)	Adj. Avg. resolution time (secs)	Severity
Variable not declared	85	63	6	16	33	82	7.0
Variable: Incorrect variable declaration	48	27	6	15	41	143	6.8
Simple syntactical error	66	48	5	13	46	98	6.5
Method: Incorrect method declaration	46	24	3	19	70	129	5.9
Type error	38	27	5	6	60	144	5.5
Semantic error	49	34	4	11	52	110	5.4
Missing 'return' statement	23	15	3	5	123	203	4.7
Variable name written incorrectly	75	48	2	25	38	60	4.5
Method call targeting wrong type	25	18	3	4	80	154	3.9
Variable: Incorrect attempt to use variable	47	29	3	15	27	81	3.8
Class or type name written incorrectly	44	25	3	16	20	82	3.6
Method call: parameter number mismatch	27	20	3	4	58	129	3.5
Method not declared	29	21	2	6	68	114	3.3
; missing	74	51	2	21	23	45	3.3
Method name written incorrectly	47	35	0	12	47	47	2.2
Extraneous closing curly brace	21	14	2	5	13	86	1.8
Use of non-static method from static context	16	15	0	1	109	109	1.7
Method: Incorrect method call	20	15	1	4	51	85	1.7
Type mismatch in assignment	12	9	2	1	40	142	1.7
Incorrect/attempted used of class or type	5	3	2	0	79	287	1.4

Table 6.9: Severity ranking for error resolution threshold of 10 minutes (600 seconds)

	Rank: 2 minutes	**Rank: 5 minutes**	**Rank: 10 minutes**
Variable not declared	1	1	1
Simple syntactical error	4	2	3
Variable: Incorrect variable declaration	5	3	2
Method: Incorrect method declaration	2	4	4
Semantic error	3	5	6
Type error	7	6	5
Missing ‘return’ statement	11	7	7
Variable name written incorrectly	6	8	8
Method call targeting wrong type	12	9	9
Method not declared	10	10	13

Table 6.10: Severity rank of selected error categories with an error resolution threshold of 2, 5, and 10 minutes. Categories shown are the ten most severe with a 5 minute resolution threshold.

6.2.2 Alternative Formulations for Severity

Severity has been expressed as the product of *frequency* and *difficulty*, where difficulty of an error category has been expressed as the *adjusted average resolution time* (Chapter 4.2.3) for the category. Tables 6.11, 6.12 and 6.13 show the 20 most severe error rankings when the difficulty is instead expressed as the natural logarithm, square root, or square of the adjusted average resolution time, respectively.

Variable not declared remains the highest severity error category across the three alternative formulations of severity, and *Simple syntactical error* remains in the 2nd position except in one case (difficulty as logarithm of the adjusted average resolution time) where it moves to the 3rd position. In Table 6.14 we present the ranking of the 10 most severe errors (with the original severity formulation) after the severity formulation is changed to each of the three alternatives. Note that for each alternative formulation, the 5 most severe categories in the original ranking remain within the top 7 in the revised ranking, and the 6 most severe categories in the original ranking remain within the top 11 in the revised ranking. In general, the alternative severity formulations have limited impact in assessing the most severe error categories. Because of this, and by reasoning discussed in Chapter 4.2.3, we maintain that the severity formulation used in our analysis is suitable for the purpose of this thesis.

Category	Severity (log t • f)
Variable not declared	342
Variable name written incorrectly	281
Simple syntactical error	280
; missing	252
Variable: Incorrect variable declaration	218
Semantic error	217
Method: Incorrect method declaration	210
Variable: Incorrect attempt to use variable	186
Method name written incorrectly	181
Class or type name written incorrectly	172
Type error	172
Method not declared	130
Method call: parameter number mismatch	118
Method call targeting wrong type	118
Missing 'return' statement	114
Method: Incorrect method call	84
Extraneous closing curly brace	82
Missing parentheses for method call	79
Use of non-static method from static context	72
Wrong return type declared	56

Table 6.11: Top 20 errors by severity when difficulty is calculated as the natural logarithm of the adjusted average resolution time.

Category	**Severity ($\sqrt{t} \bullet f$)**
Variable not declared	637
Simple syntactical error	552
Variable name written incorrectly	489
Variable: Incorrect variable declaration	466
Method: Incorrect method declaration	450
Semantic error	449
; missing	407
Type error	365
Variable: Incorrect attempt to use variable	341
Method name written incorrectly	322
Class or type name written incorrectly	311
Method not declared	274
Missing 'return' statement	271
Method call targeting wrong type	264
Method call: parameter number mismatch	241
Method: Incorrect method call	163
Use of non-static method from static context	153
Extraneous closing curly brace	147
Missing parentheses for method call	138
Type mismatch in assignment	112

Table 6.12: Top 20 errors by severity when difficulty is calculated as the square root of the adjusted average resolution time.

Category	Severity ($t^2 \bullet f$)
Variable not declared	1940877632
Simple syntactical error	1407211963
Variable: Incorrect variable declaration	983929084
Method: Incorrect method declaration	888761798
Semantic error	826081929
Variable name written incorrectly	759861675
Type error	466331621
; missing	369303467
Variable: Incorrect attempt to use variable	287185064
Missing 'return' statement	235791236
Method name written incorrectly	229345007
Class or type name written incorrectly	212960000
Method call targeting wrong type	194005102
Method not declared	193909487
Method call: parameter number mismatch	124197460
Method: Incorrect method call	35444531
Use of non-static method from static context	34668544
Extraneous closing curly brace	22122358
Missing parentheses for method call	17333821
Type mismatch in assignment	13161362

Table 6.13: Top 20 errors by severity when difficulty is calculated as the square of the adjusted average resolution time,

	Rank: $s = t \bullet f$	Rank: $s = \log t \bullet f$	Rank: $s = \sqrt{t} \bullet f$	Rank: $s = t^2 \bullet f$
Variable not declared	1	1	1	1
Simple syntactical error	2	3	2	2
Variable: Incorrect variable declaration	3	5	4	3
Method: Incorrect method declaration	4	7	5	4
Semantic error	5	6	6	5
Type error	6	11	8	7
Missing ‘return’ statement	7	15	13	10
Variable name written incorrectly	8	2	3	6
Method call targeting wrong type	9	14	14	13
Method not declared	10	12	12	14

Table 6.14: Severity rank of selected error categories with alternative formulations of difficulty (and thereby

severity). . Categories shown are the ten most severe with the original formulation (t • f).

6.3 Discussion

6.3.1 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.

As with the first study (see Chapter 5.2.3), a small number of error categories accounted for the majority of errors seen in the data set. The coverage table (Table 6.7) shows that the top ten logical errors account for almost 60% of all error occurrences. This indicates that, 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 a diagnostic message. If better messages could be produced for the top twenty error types, feedback would be improved for 80% of all errors experienced. 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 suggests that efforts to design an automated diagnosis of novice errors need focus on a small number of carefully selected errors in order to be effective at improving the diagnosis given for the majority of errors actually encountered.

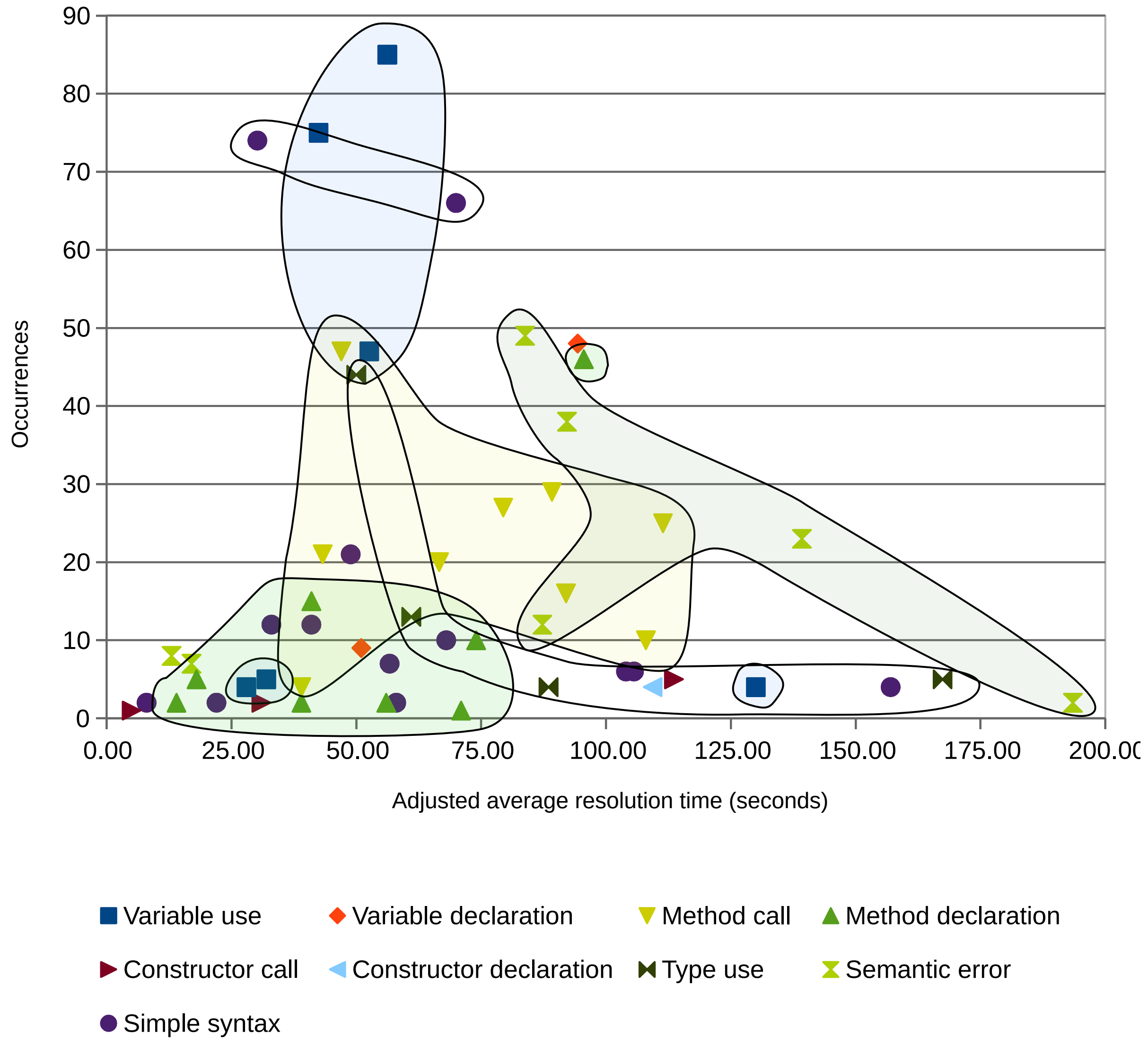


Figure 6.4: Adjusted resolution time and frequency of occurrence by top-level category. The shaded/outlined areas identify apparent spatial grouping of error categories underneath various of the top-level categories as named in the legend of the graph (choice of shading or outline is purely for legibility). Frequency is shown as occurrences per 1000 error-containing compilations events.

6.3.2 Severity of Error Categories

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.

The severity of errors, as depicted in Figure 6.2 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. Figure 6.4 depicts the same data, but areas representing top-level error categories are highlighted to encompass the data points of the individual errors they contain (either by shading or in some cases, to reduce clutter where many regions overlap, a coloured outline). Figure 6.3 depicts top-level categories only, with occurrence and resolution time from subcategories folded in to the top-level category. Errors and error categories towards the top right quadrant of each 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 Chapter 2.7 we discussed error severity as a product of the frequency of the error type and the difficulty in resolving the error. We have used the adjusted average resolution time (Chapter 4.2.3) as the sole metric for deciding error difficulty in this work, and the error categories are ranked (Table 6.6) by severity calculated on this basis.

The most severe error category is *Variable not declared* (severity score 4.8), which is also the most frequently occurring category with 85 categorised 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 – *Simple syntactical error* (severity score 4.6), *Incorrect variable declaration* (4.5), *Incorrect method declaration* (4.4) and *Semantic error* (4.1) – all have a slightly higher adjusted average resolution time, extending up to 95 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 significantly 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 stand-out in this regard, since it is both a precise subcategory and the highest occurring categorisation overall (as well as having the highest severity score).

The cause of the apparent difficulty in precisely categorising errors is difficult to determine without further investigation. The method used to develop the category hierarchy (Chapter 4.2.2) was designed to provide suitable categories for commonly occurring errors, rendering it doubtful that lack of availability of appropriate categorisations was to blame. The possibility that it is innately difficult to precisely categorise a significant portion of novice programmer errors needs to be considered (though this does not imply that attempting such categorisation is a worthless endeavour, 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.

In the following subsections we will look at each of the top-level categories in turn and discuss the relevant severity results for both the top-level category and its subcategories.

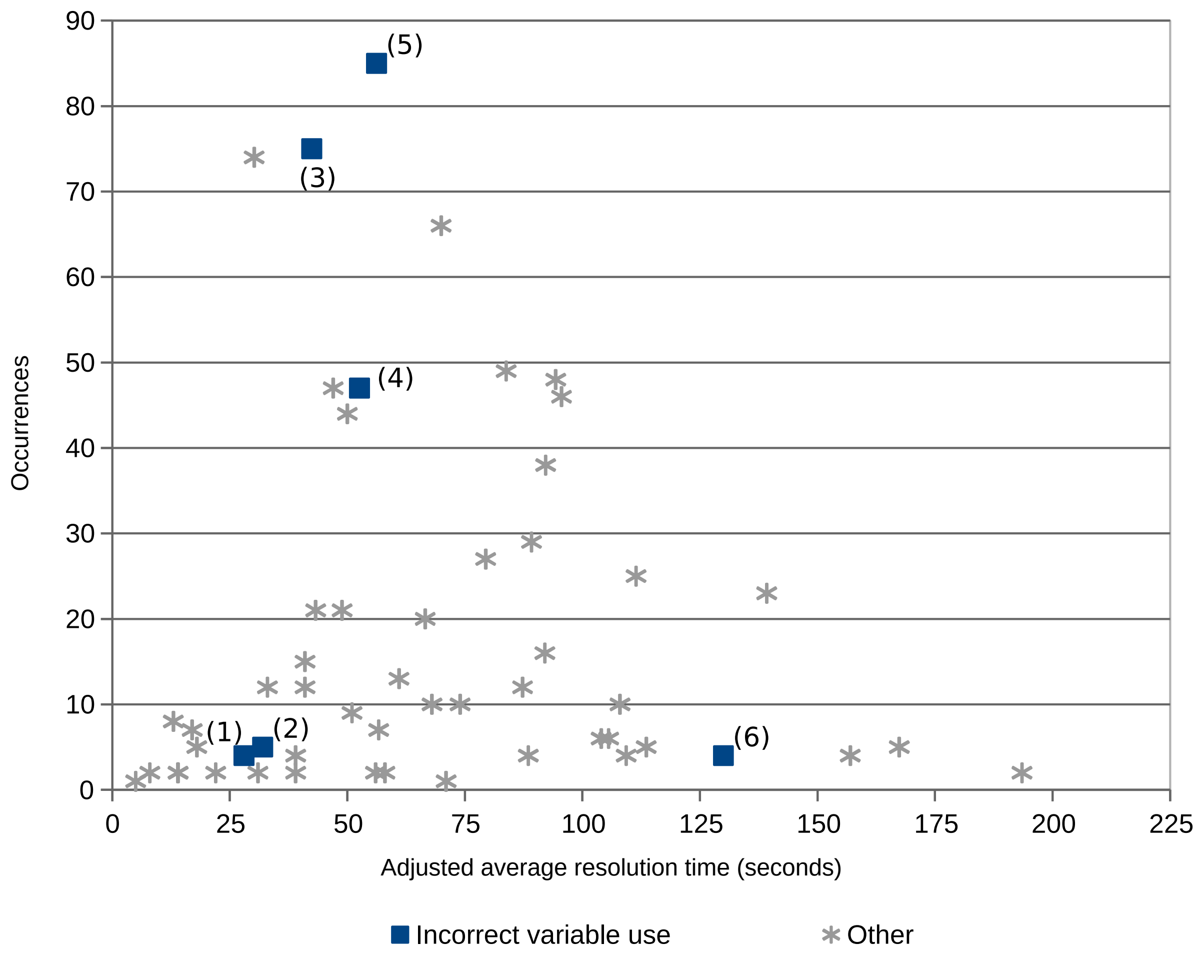


Figure 6.5: Severity of errors in the 'Variable Use' top-level category. The individual errors are: (1) Wrong variable used (2) Use of non-static variable from static context (3) Variable name written incorrectly (4) Variable use (top-level) (5) Variable not declared (6) Use of variable from another class

6.3.2.1 Variable Use

The top-level category 'Variable Use' consists of six data points with a wide spread over the severity landscape (Figure 6.5). The fact that the individual errors do not cluster in a single area shows that 'variable use' cannot meaningfully 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. Some errors (such as (1) and (2)) are not problematic: they are infrequent and—when encountered—fixed relatively quickly. Other errors, however, are of more concern: Errors (3) and (5) occur very frequently and error (6) seems to cause students problems in fixing it. Errors (3) and (5) can be interpreted as more shallow: (3), *Variable name written incorrectly*, will include simple typing errors that are easily fixed and (5), *Variable not declared*, will also be a combination of slips (students merely forgetting to declare) and actual lack of understanding. Even though these errors are fixed more quickly than some others (they are just under the mean resolution time) the time to fix is not trivial. Students spend on average almost a minute fixing these types of error, indicating that some more serious issues exist in this area than mere typing mistakes.

Note that error (3) *Variable name written incorrectly* does not necessarily imply a common spelling mistake, but rather that the variable name in a particular *use* did not match the name of the intended variable in its declaration; an identifier mis-spelling that is applied consistently will not result in a compilation error. However, nearly identical spelling in variable use and declaration can be an indicator of this error; this fact is used in mechanisms employed by our prototype diagnosis tool to diagnose this error (Chapter 7.3.1).

Error (6), *Use of variable from another class*, does not occur very frequently, but when it does, it causes trouble. This points to a lack of understanding of underlying concepts and may identify an area of instruction that would benefit from attention.

Looking at the mean values of all errors in this top-level category (Figure 6.3) we can see that the errors in this category are among the most frequent errors, but are also the most quickly fixed.

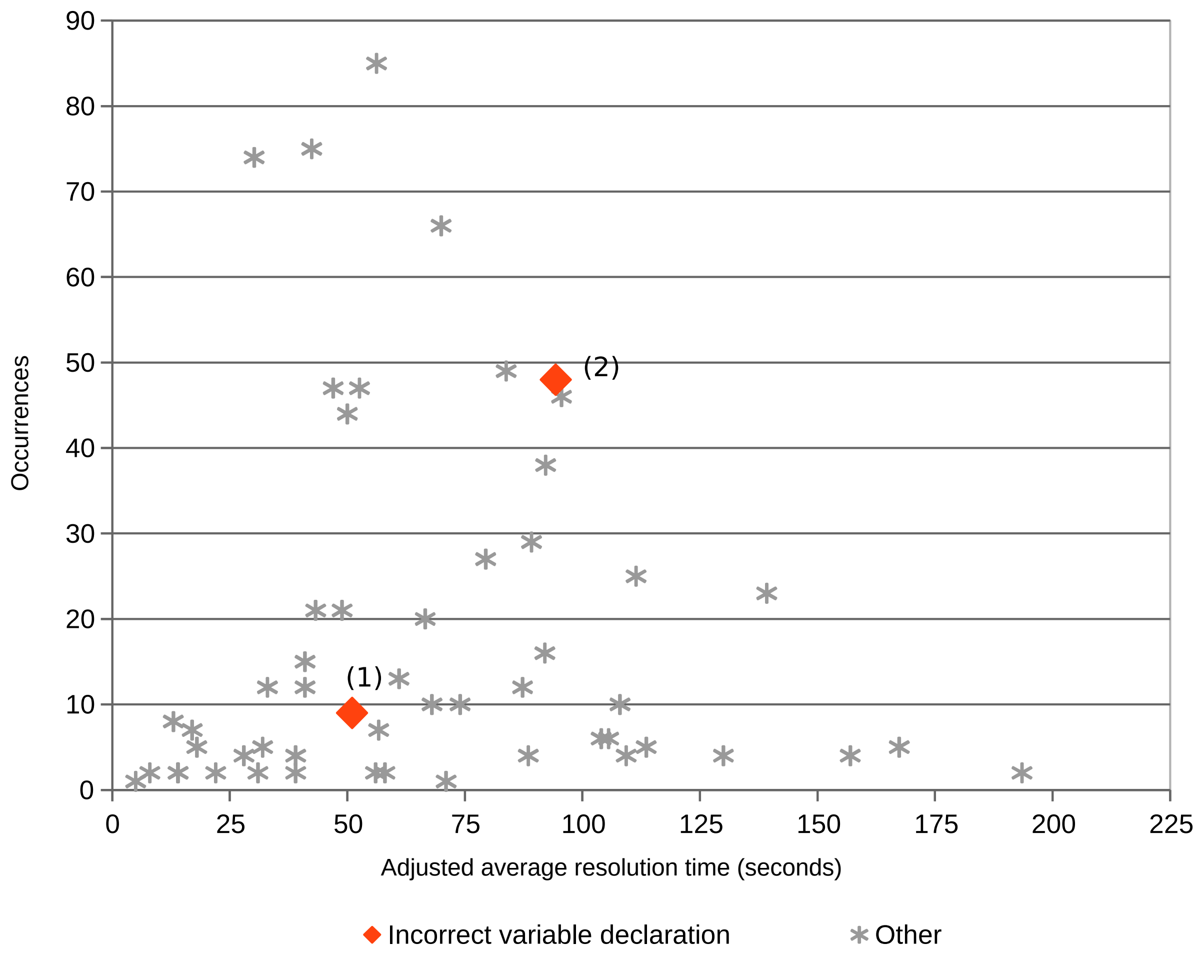


Figure 6.6: Severity of errors in the 'Variable Declaration' top-level category. The individual errors are: (1) Variable needs a unique name (2) Variable declaration (top level)

6.3.2.2 Variable Declaration

The 'Variable Declaration' category, shown in Figure 6.6, includes only one sub-category ((1) *Variable needs a unique name*); all other related errors were categorised under the top-level category. While this makes it impossible to identify more specific problems students might have with variable declarations, we can see that this is an area of significant concern. Variable declaration problems are frequent and take a long time to be resolved, pointing to significant lack of mastery in an important area. Both pedagogical interventions and better support from software tools may be ways to address this problem.

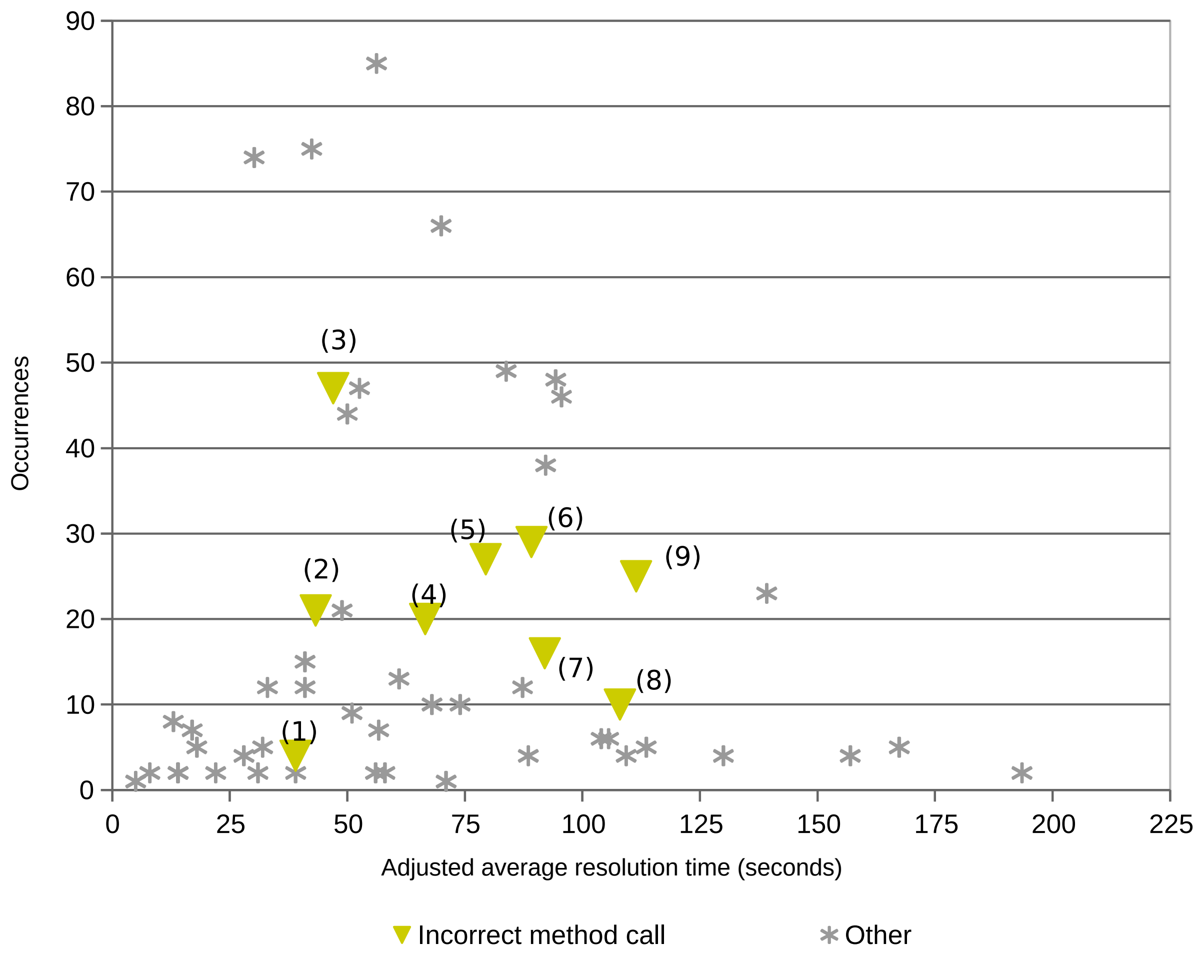


Figure 6.7: Severity of errors in the 'Method Call' top-level category. The individual errors are: (1) Parameter types included (2) Missing parentheses (3) Method name written incorrectly (4) Method call (top level) (5) Parameter number mismatch (6) Method not declared (7) Use of non-static method from static context (8) Parameter type mismatch (9) Method call targeting wrong type.

6.3.2.3 Method Call

The 'Method Call' category (Figure 6.7) forms a cluster of a number of related problems in the area of the graph representing significant severity. Again, this indicates that this concept is one that poses significant challenges to students.

In this case we have a number of more precise sub-categories that give us some clues about specific aspects that are particularly troublesome. While none of the errors is typically resolved in trivial time (all take more than 30 seconds on average), we again find the more syntactic problems at the easier-to-fix end, while errors concerning semantic (often type-related) problems take longer to resolve. The syntactic problems include writing a parameter type in the method call and leaving out the parenthesis of a call. Semantic problems, on the other hand, are represented by 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.

Error (3) *Method name written incorrectly* – which specifies that a method name in an expression was written differently than in its declaration – occurs with a higher frequency than other errors in this category, although with a relatively low average resolution time which reduces its severity. It is comparable in this regard to the *Variable name written incorrectly* error in the *Variable use* category (Chapter 6.3.2.1). This error category represents the only type of error that the BlueJ environment attempts to diagnose itself, using a string-edit-distance metric to find candidate matches between declarations and calls, similar to the matching of variable declaration and use employed by our diagnostic tool and described in Chapter 7.3.1. Since BlueJ already performs this diagnosis and because the technique is so similar to that used for variable names, we opted not to implement a similar check for mis-spelled method names, in favour of focusing effort on other high-severity errors.

In this case, even the syntactic problems (such as including a type in a method call parameter list) are likely to not be mere typing slips, but indicative of a lack of understanding of a fundamental concept. Overall, the area of method calls seems to be a deeply problematic area causing problems for many students. This is confirmed by the mean severity across this top-level category, as presented in Figure 6.3.

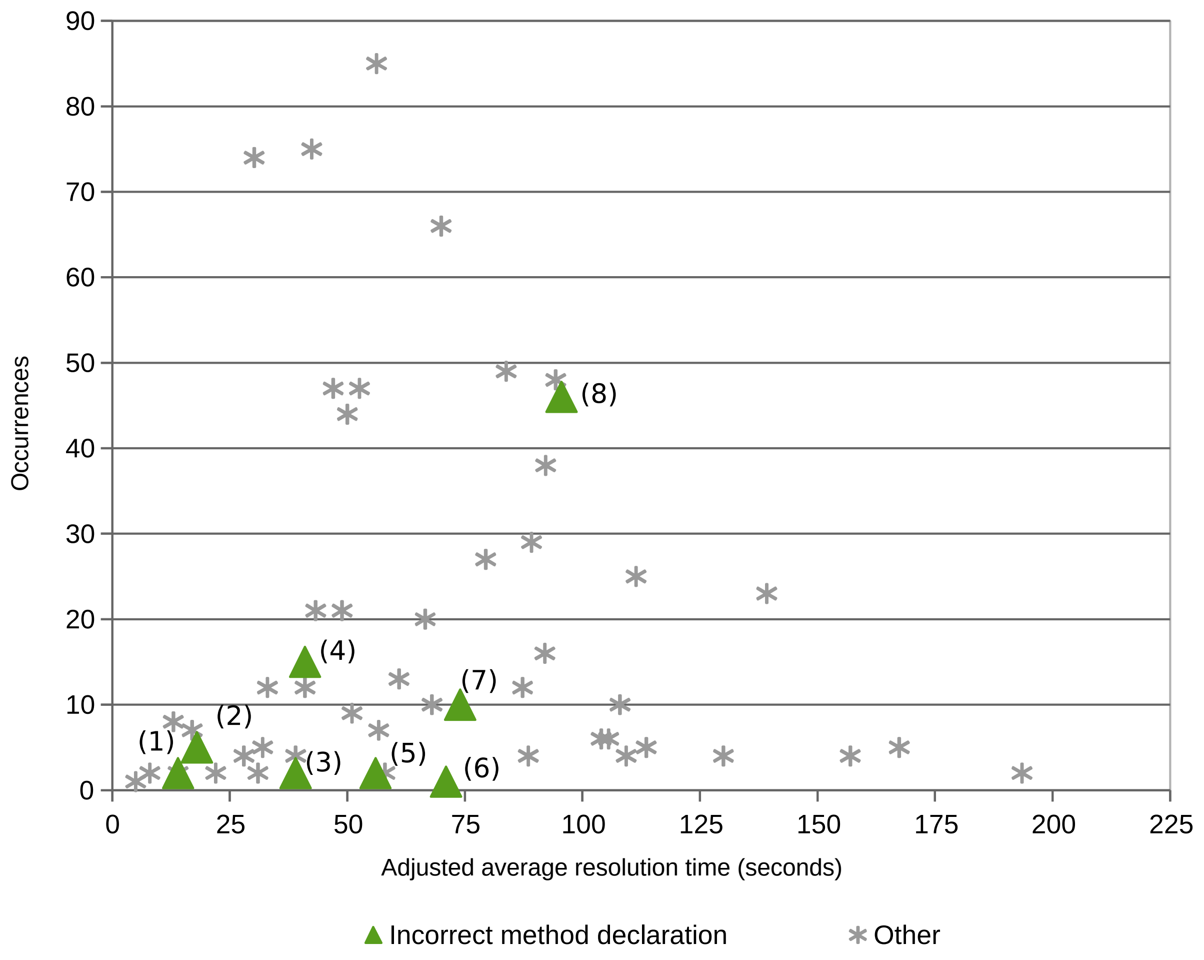


Figure 6.8: Severity of errors in the 'Method declaration' top-level category. The individual errors are: (1) Return type should be void (2) Wrong number of parameters (3) Missing opening curly brace (4) Wrong return type declared (5) Method duplicated exactly (6) Semicolon at end of header (7) Missing return type (8) Method declaration (top level)

6.3.2.4 Method Declaration

The 'Method Declaration' data points (Figure 6.8) look less severe at first glance, since many of them are positioned along the x-axis, making them too infrequent to be considered severe. This first impression is somewhat misleading, however, since the one clearly severe error type, number (8), is the top category.

This simply tells us that only a relatively small number of the Method Declaration errors could be categorised more precisely, while a large number of problems in this area were so varied or defied any attempt at conclusively interpreting the author's intent that they could not be further sub-categorised. They do, however, exist, and in significant numbers.

The only area within this category that can be identified as a distinct sub-problem is the handling of return types. Both errors (4) and (7) show problems with return types, and these are the only sub-categories with more significant frequency.

In the actual code analysed, method declarations were often so fundamentally wrong that the error could not be assigned to a precise sub-category, resulting in the large number of top-level errors. This is indicative of more fundamental problems than a larger number of sub-category errors, where the code is often nearer to a correct solution.

It is also interesting to compare Figure 6.3, which shows the mean of the top-level categories, including all sub-categories. It is evident that method declarations, while not the worst category in either frequency or difficulty, are still an area of significant concern.

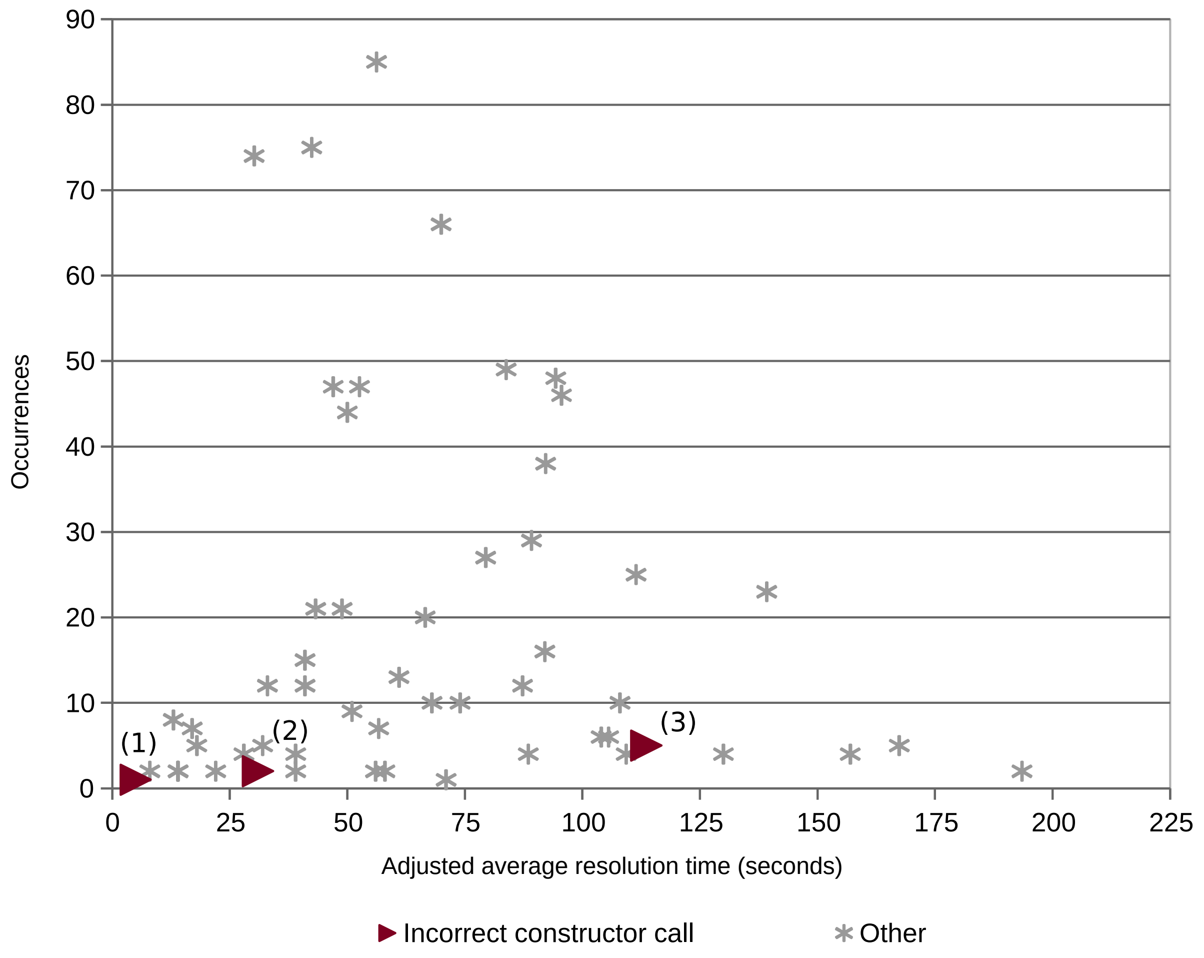


Figure 6.9: Severity of errors in the 'Constructor Call' top-level category. The individual errors are: (1) Missing parentheses (2) Constructor call (top-level) (3) Parameter number mismatch

6.3.2.5 Constructor Call

The ‘Constructor Call’ category (Figure 6.9) has two sub-categories, one of which ((3) *Parameter number mismatch*) shows significantly longer resolution times than the others. However, the frequency of all categories is low, so the severity is limited.

This is also visible in the mean values of the top-level category, compared to the other categories (Figure 6.3). While constructor parameters occasionally cause problems for some students, the overall severity of this category is low.

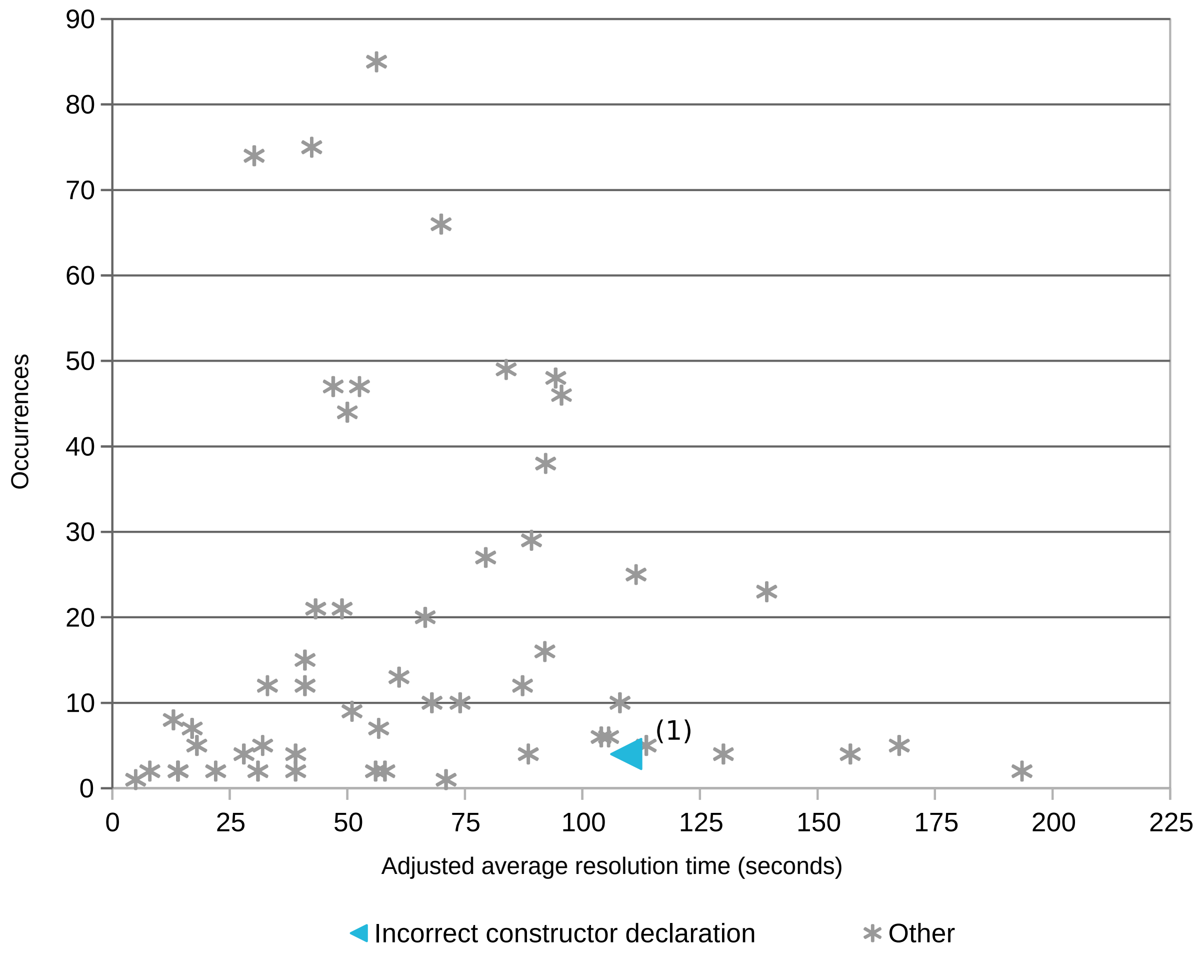


Figure 6.10: Severity of errors in the ‘Constructor Declaration’ top-level category. The category shown (1) is the top-level category.

6.3.2.6 Constructor Declaration

Constructor declaration errors (Figure 6.10) occurred infrequently, and only the top-level category was selected. The resolution time is relatively high, but the low frequency of this error category gives it a low overall severity. In the comparison of mean adjusted resolution time (Figure 6.3) the category appears to have a distinctly high resolution time when compared to the other categories, but due to the low number of occurrences the statistical accuracy of this data point is diminished; in any case, the severity (calculated as a product of frequency and resolution time) is low.

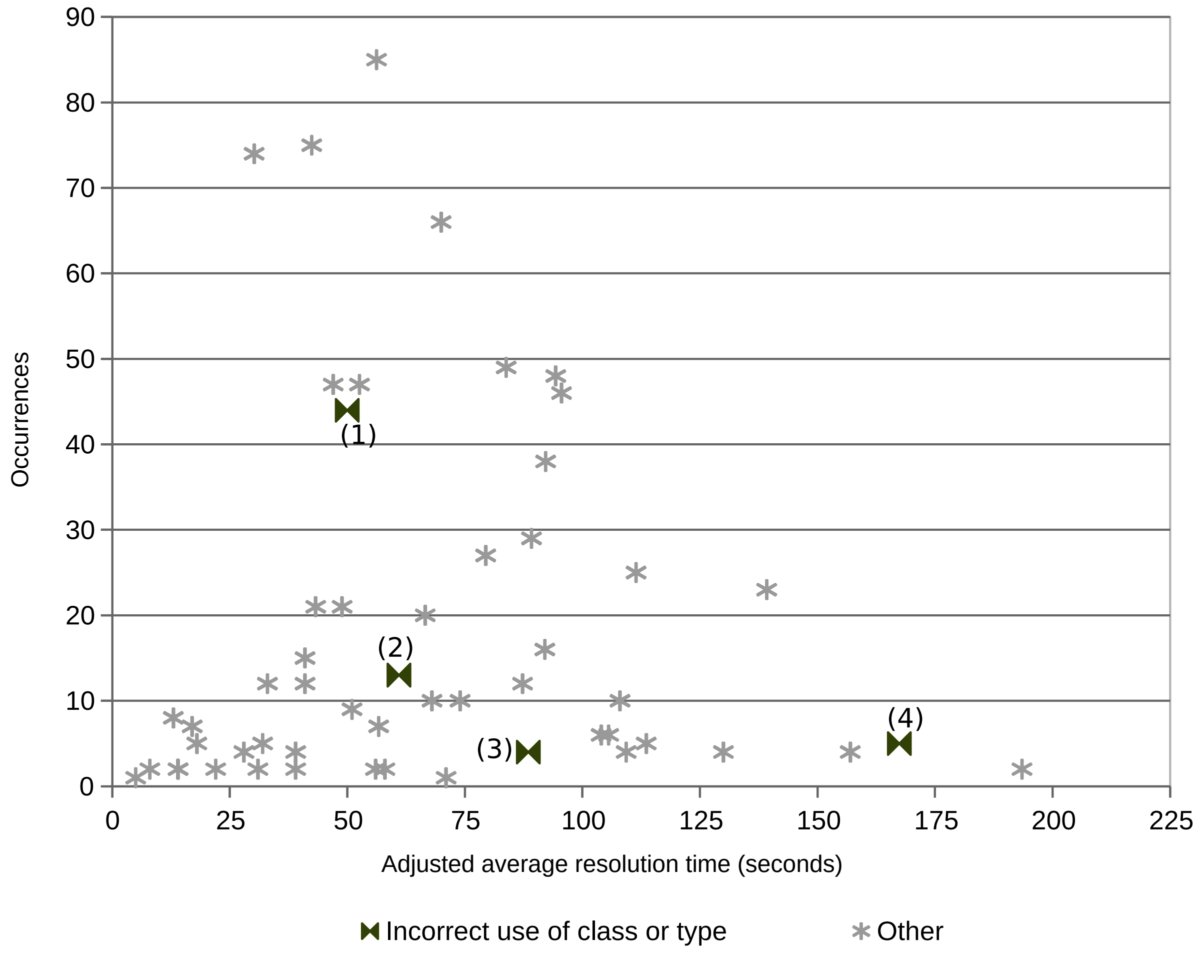


Figure 6.11: Severity of errors in the 'Use of class or type' top-level category. The individual errors are: (1) Name written incorrectly (2) Used without being imported (3) Class not defined (4) Use of class or type (top level)

6.3.2.7 Use of Class or Type

The 'Use of class or type' category (Figure 6.11) has error instances in three subcategories. The top-level category (4) has a noticeably higher resolution time, but a lower number of occurrences than the subcategories (indicating that in most cases, a more precise categorisation was able to be made), and a low severity.

The (3) *Class not defined* subcategory also maintains a low frequency of occurrence and a midrange resolution time. The (2) *Used without being imported* subcategory occurs slightly more frequently, but is still dwarfed in terms of severity by the (1) *Name written incorrectly* subcategory. In the comparison of mean adjusted resolution time (Figure 6.3) the category is clearly among the less severe of the top-level categories, with a moderate frequency and resolution time.

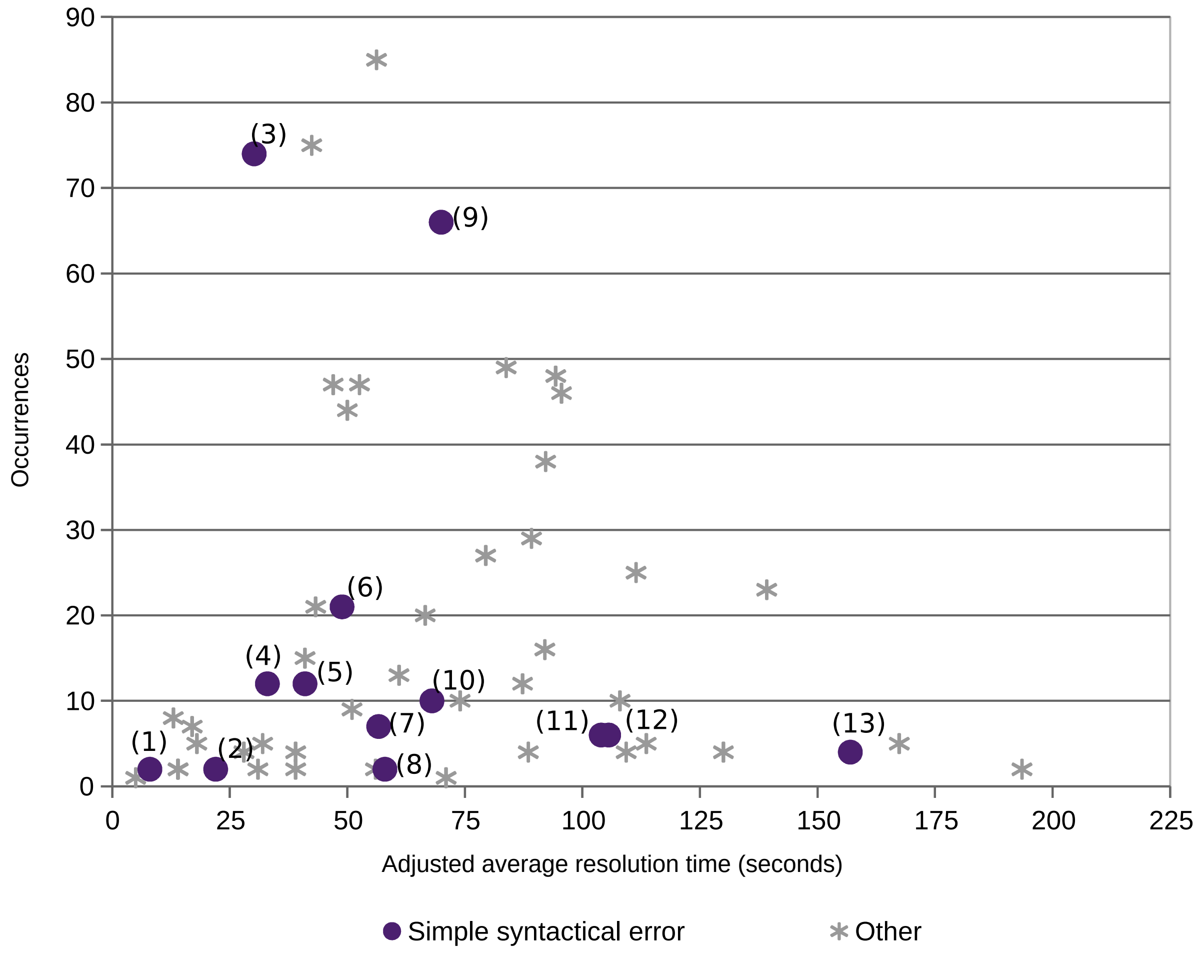


Figure 6.12: Severity of errors in the 'Simple syntactical' top level category. The individual errors are: (1) Extra/misplaced '(' (2) Missing closing curly (3) ';' missing (4) Missing closing curly (class) (5) Missing operator (6) Extra closing curly (7) Missing closing curly (method) (8) Extra opening curly (9) Simple syntactical error (top level) (10) Mismatched parentheses (11) Missing closing curly (control statement) (12) '=' used in place of '==' (13) Keyword written incorrectly

6.3.2.8 Simple Syntactical

The 'Simple syntactical' category is the most frequently occurring top-level category (Figure 6.3) and has numerous subcategories (Figure 6.12). Unsurprisingly, missing semicolons represent the most frequently occurring sub-category.

Overall, the severity of this top-level category is moderated by its low resolution time: errors in this category are, on average, among the most quickly fixed. The typical syntactical errors that we see frequently in student code – unbalanced curly brackets or parentheses, missing semicolons – all have lower-than-average resolution times, while misspelled keywords take longest to fix.

While individual frequency is not high for many subcategories, the large number of categories results in the combined count for the top level category being high. The fact that the top-level category accounts for over 30% of the occurrences within the category as a whole shows that many errors seen in this category were difficult to categorise more precisely

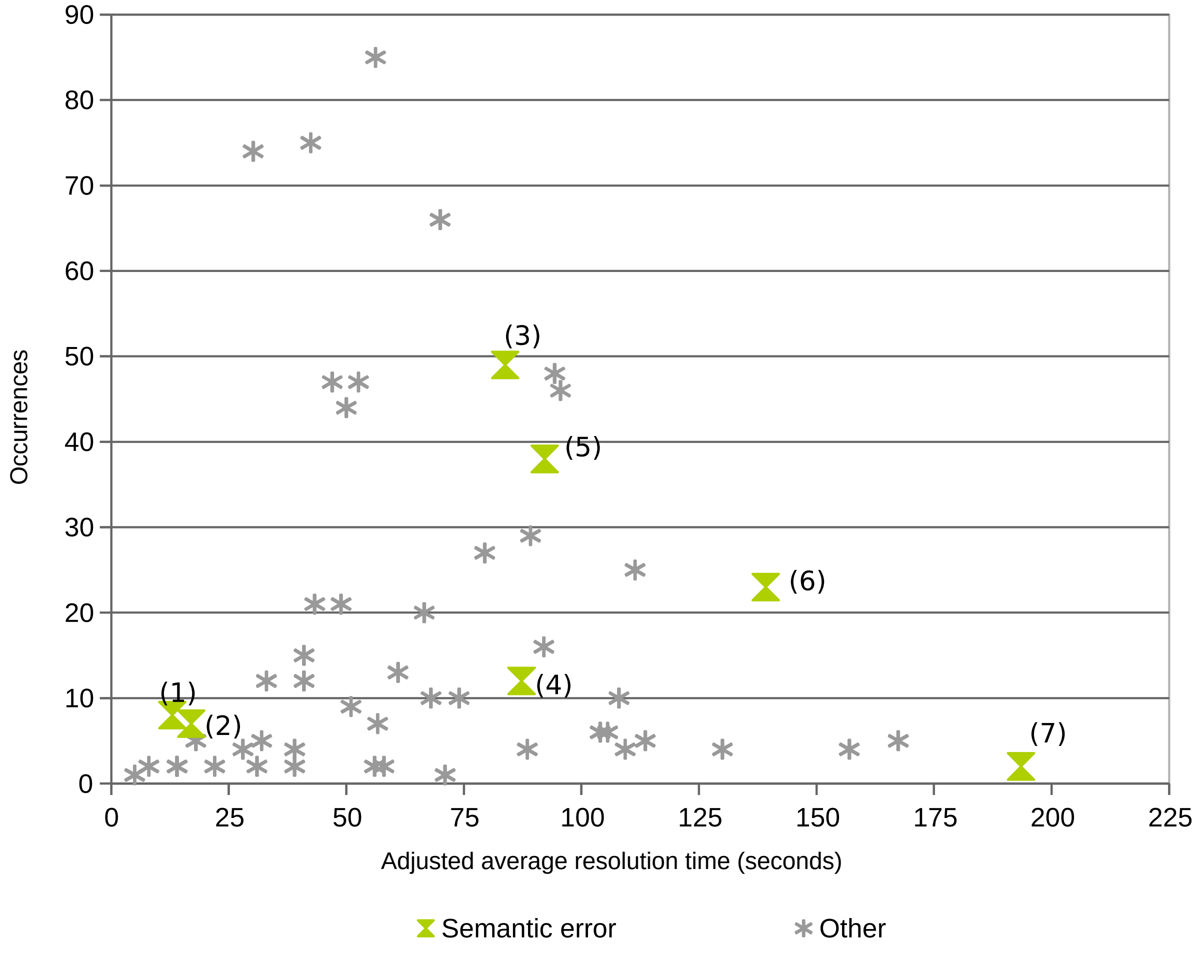


Figure 6.13: Severity of errors in the 'Semantic Error' top-level category. The individual errors are: (1) Array used in place of element (2) Unhandled exception (3) Semantic error (top level) (4) Type mismatch in assignment (5) Type error (6) Missing 'return' statement (7) Class doesn't implement required method

6.3.2.9 Semantic Error

The 'Semantic error' category includes, in its subcategories, some of the most severe errors found in the data set (Figure 6.13). The aggregate data (Figure 6.3) also places this top level category as the most severe.

Error (7), *Class doesn't implement required method*, is the most difficult for students to fix of all errors observed. Its frequency, however, is low. Three other errors in this group, (3), (5) and (6), are at the edge of the most severe errors in our graph, being both relatively frequent and difficult to resolve. Overall, it is clear that semantic errors pose a significantly greater problem to students than other error types.

6.3.3 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 categorised was sampled from data maintained by the Blackbox Data Collection project (Brown et al. 2014), which collects anonymised data from BlueJ users. The programming background, age, and other factors that may affect programming ability or behaviour of participants in this data collection are not known, though it is expected most are in the first year of university study.

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 significant statistical error; 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 effect ranking of errors. Absolute time data, however, may be affected by this effect.

7 Error Diagnosis Tool

In the previous chapter we discussed results which show that certain categories of error have considerably higher severity than others. These high-severity errors have a high combination of frequency and difficulty, and consume a large total amount of time to resolve when compared to less severe error categories; accordingly, it is prudent to focus assistive measures on these types of error.

The categorisation of errors used for the determination of error severity was performed manually (Chapter 4.2) and the human reasoning used to diagnose the errors was recorded. This reasoning includes examples of diagnosis logic that could also be implemented in an automatic system.

In this chapter we present details of the design, development and evaluation of a prototype tool to demonstrate the feasibility of using a more thorough automated analysis to diagnose high-severity errors than what is typically performed by compilers or by other tools developed to provide enhanced diagnostics, to enable a more accurate and precise diagnosis. We first discuss an overall approach to the diagnosis of errors as a logical sequence of steps which can be performed by an experienced human programmer and which can be potentially be automated. We then recap the most severe errors, and discuss which of these are suitable for diagnosis in such a prototype tool, along with an overview of the technique chosen for doing so in each case; we then provide a more detailed discussion of the design of the tool, and show examples of its output. Finally, we show results of an evaluation of the diagnostics generated by the tool compared to those generated by the standard *javac* compiler for a number of errors from our dataset.

7.1 Expert Diagnosis of Errors

During categorisation of errors for the research presented in Chapter 4, we recorded a short statement of our reasoning for choosing a particular categorisation. Other researchers who performed categorisation for the category validation phase (Chapter 4.1.3.3) were also asked to submit such statements of reasoning.

We have not used the reasoning statements directly for the development of the diagnostic tool prototype, but examining them has suggested a general approach for making error diagnoses which can be broken down into a short sequence. We do not claim that all experienced human programmers necessarily evaluate all errors in this precise sequence, but that it is a logical sequence which may be used to produce a diagnosis for a range of errors that is more accurate and precise than that which would be produced by a typical compiler in at least some cases. The sequence is as follows:

1. Look where the compiler diagnostic positions the error, and determine a *fundamental reason* for the error case, given existing knowledge of how the compiler identifies errors, and experience of why it generates certain diagnostics for various types of error. As an example of a fundamental reason behind an error, consider the case of misspelled variable name used in an expression; the *javac* compiler will typically report this kind of error as an *unresolved symbol - variable*. The fundamental reason for the error is that an identifier is used in an expression context and there is no visible declaration for a variable with the exact same identifier. Note that the fundamental reason is related to the manner in which the error was detected by the compiler.

2. Based on the fundamental reason for the error, determine a set of candidate error types that may stem from that fundamental reason. For the example in the previous step, such a candidate set might include:

   - Misspelling of the identifier in the expression;
   - Misspelling of the identifier in the declaration;
   - Attempt to use a variable from a scope in which it is not visible;
   - Missing variable declaration; and perhaps others.

3. Look for evidence in the source code surrounding the error, and in some cases more broadly (the program as a whole, and the standard library classes), which supports the diagnosis of a particular error type. To continue the present example, we might look for a declaration with an identifier with spelling similar but not identical to the identifier used in the expression in which the error was detected; the existence of such a declaration supports the diagnosis of the error as either a misspelling in either the declaration of the expression. Further evidence might then be sought in order to differentiate which of these two cases should be considered more likely.

This sequence of steps can potentially be automated, and doing so forms the basis of the approach for the design of our prototype tool. Detection of the error and determination of the fundamental reason for the error is performed using an analysis step which functions similarly to certain compilation phases (including parsing and symbol resolution); a number of further analyses are then used to decide a diagnosis from one of several possible options.

7.2 Selection of Error Types for Diagnosis

To show the feasibility of improved automated error diagnosis for high-severity errors, we selected a small number of high-severity error categories for which a suitable diagnosis strategy could easily be devised.

Table 6.6 shows the error categories ranked according to severity. The ten most severe errors are shown again in Table 7.1. The most severe error category is “Variable not declared”; this is related to another high-severity error category, “Variable name written incorrectly”. Both of these types of error typically result in a symbol resolution failure in the compiler, leading to a common diagnosis for each; however, the high severity of these error types suggest that it is worth the effort to distinguishing them if possible. A straight-forward strategy to differentiate these error types is, upon the detection of an unresolvable symbol used in a context requiring a value, to look for other variables which are declared with a similar but not identical name in the assumption that the presence of such a declaration indicates a likely misspelling.

In a similar vein, the high-severity error categories “Method call targeting wrong type” and “Method not declared” could lead to a common error diagnosis in a typical compiler, since both manifest in symbol resolution failure. The existence of an identically named method defined in a different receiver type can be used as a strong indicator however that the error in question does result from an attempt to call such a method and that for instance the receiver expression is incorrect or has the wrong type declaration.

Category	Occurred	Resolved (total)	Unresolved + Reverted	Uncertain resolutions	Average resolution (sec.)	Adjustment	Adjusted Total resolution time	Adjusted avg. resolution time	Severity
Variable not declared	85	63	6	16	33	1800	3879	56.22	4.778
Simple syntactical error	66	48	5	13	46	1500	3708	69.96	4.618
Variable: Incorrect variable declaration	48	27	7	14	41	2100	3207	94.32	4.528
Method: Incorrect method declaration	46	24	3	19	70	900	2580	95.56	4.396
Semantic error	49	34	5	10	52	1500	3268	83.79	4.106
Type error	38	25	7	6	34	2100	2950	92.19	3.503
Missing 'return' statement	23	13	6	4	65	1800	2645	139.21	3.202
Variable name written incorrectly	75	47	3	25	26	900	2122	42.44	3.183
Method call targeting wrong type	25	18	3	4	80	900	2340	111.43	2.786
Method not declared	29	20	4	5	47	1200	2140	89.17	2.586

Table 7.1: The ten most-severe error categories. The highlight indicates the error categories selected for automated diagnosis in the diagnosis tool prototype.

Other high-severity errors in the top-ten list are either too broad to be useful (“Simple syntactical error”, “Semantic error”, “Type error”), or were considered too difficult to diagnose automatically to justify the effort in a prototype tool (“Incorrect variable declaration”, “Incorrect method declaration”). One high-severity category in this list, “Missing ‘return’ statement”, is unique in that it appears in general to be correctly diagnosed by the standard compiler; for this reason no attempt was made to diagnose this particular error in the prototype tool.

7.2.1 Misspelled and undeclared variables

Upon detection of an unresolved identifier used in a context for which a value is valid or required, we can examine variable declarations visible within the scope containing the unresolved symbol and check for declared names which have a low edit distance to the unresolved identifier. If there are no such names, we may assume the variable is simply lacking a declaration (though there are also some other possibilities, such as that the programmer was attempting to refer to a variable that is declared elsewhere but is not visible in the current scope).

If a declaration is found of a variable name with a sufficiently low edit distance to the unresolved identifier, there are at least two distinct possibilities: firstly, the variable name may have been misspelled on use, and secondly, the declaration itself may have been misspelled. To attempt to disambiguate these cases, we look for an additional clue: whether the declared variable is accessed elsewhere using its declared name. If that is the case, it is less likely that the declaration is misspelled, since it would mean that the variable had been identically misspelled in two different locations.

If no declaration is found of a variable name with sufficiently low edit distance, we instead assume that the declaration is missing. Since we aim to give a *precise* as well as *accurate* diagnosis, we designed the tool to give an indication of where the declaration should be. A decision is made that either the declaration should be at the method level (that is, within and local to the method that the identifier was used in) or should be as a field at the class level. This determination is made by checking the location of any other unresolved uses of the same identifier; if they all occur within the same method, then the diagnosis suggests declaration within that method; otherwise, diagnosis suggests declaration at the class level.

7.2.2 Unresolved method calls

The errors "Method call targeting wrong type" and "Method not declared" both result in failure to resolve a method name to a suitable method ("Method call targeting wrong type" denotes an attempt to call a method on a receiver, where the type of the receiver does not match the type of which the method is a member). To distinguish between the two, the tool looks for declarations of methods with a name identical to the unresolved identifier in a number of other types (as discussed below); if it finds any such methods, it diagnoses the method call as targeting the wrong receiver.

In looking for candidate methods to match the name used in a method call expression, the tool searches for methods defined in other classes and interfaces within the same package. Because apparent confusion was noticed in the use of String methods on other classes within some of the captured code snapshots in our dataset, the tool was also designed to match methods in the String class ("java.lang.String"). Other commonly-used Java API classes could be considered for similar treatment in a further development of the tool, but we did not consider this necessary for development of the prototype.

7.2.3 Final selection of errors for diagnosis

Taking into account the considerations outlined previously, the following four errors from the list in

Table 7.1 were chosen for automated diagnosis in the prototype:

- Variable not declared;
- Variable name written incorrectly;
- Method call targeting wrong type; and
- Method not declared.

The prototype also diagnoses some limited cases of the “Incorrect variable declaration” – specifically, it can make a diagnosis that a variable name in a declaration is misspelled.

7.3 Details of Prototype Design

The prototype diagnostic tool was developed as a modification of the BlueJ environment (Kölling et al. 2003). BlueJ is an integrated development environment which allows development and execution of Java projects; it features a code editor with syntax and scope highlighting and code auto-completion, a “codepad” (REPL) facility in which statements can be executed and expressions evaluated, and a graphical debugger; for compilation it relies on the standard *javac* compiler, which also generates any diagnostics that BlueJ displays.

Internally, for its scope highlighting and code completion functionality, BlueJ implements a full Java parser which generates a reduced Abstract Syntax Tree (AST) from Java source code. The parser is a hand-written recursive descent parser which has been designed to have good error recovery in the case of grammatical errors in the source. With some modification it was possible to use this parser and the resultant AST to perform symbol resolution for methods and variables, which is the basis of the method for detecting the errors that were targeted for diagnosis (7.2.3). An *analyse* function was implemented in BlueJ to perform an automated diagnosis of these errors, completely without the use of the *javac* compiler, for a single class within the current project.

The first stage of the analysis function is to parse the source code of the class using the modified parser. This produces a reduced AST representing the logical structure of the parsed source code, and a list of symbols which are not resolved during the parsing process. Symbols are separated according to their nature; Java’s grammar rules dictate certain contexts in which a symbol must represent a type, a value, or a method – for instance, in the expression “xyz.class”, the identifier “xyz” must resolve to a type. Once the initial parse phase is complete, resolution of all value and method symbols is attempted; resolution failure signals an error which can then be diagnosed. Successful resolution of a variable marks the variable declaration as *referenced*.

7.3.1 Diagnosis of unresolved variables

Unresolved “value context” symbols can be diagnosed into one of three cases:

- Variable not declared;
- Variable name written incorrectly in use; or
- Variable name written incorrectly in declaration.

In some cases, where there is an apparent spelling mismatch between the variable name in use and in declaration but there are no other uses with either spelling, the tool is not able to distinguish the 2nd and 3rd case above and gives a diagnosis which accordingly states that either the use or declaration may be incorrect.

The tool attempts only to diagnose local variable references, and not qualified field accesses using the “dot” member-access operator, in order to simplify design.

For diagnosis, a multi-map is constructed mapping all unresolved identifiers to their location in the source code (this groups identically spelt unresolved symbols). For each such identifier, the diagnosis procedure proceeds as follows:

1. Each instance of the unresolved symbol is matched to one or more declarations with a similar (low edit-distance) name. Each such match is given a score based on the edit distance and the number of scopes “outwards” that the declaration lies from the unresolved identifier (this favours matching to the innermost declarations, following the logic that a declaration with a narrower scope is more likely to be accessed from within that scope. For instance, a local variable declared within a method should certainly be used within that method; a class instance variable might only be used within some methods of that class).
2. For each unresolved symbol the highest scoring declaration match is retained and the others discarded. Thus each symbol is matched to a single declaration.
3. A diagnosis given for each symbol. If the declaration is marked as correctly referenced elsewhere, the unresolved symbol is diagnosed as a misspelling of the declared variable. If the declaration is matched by multiple identically-spelled symbols, the declaration is diagnosed as a misspelling. Otherwise, a diagnosis is offered that gives both alternatives; either the declaration, or the usage, is likely misspelled.

The maximum string edit distance for identifier matches was selected as one-quarter the length of the unresolved identifier. The score for each match, which we term the *declaration-use match score* *(d)*, was calculated as:

$$d=10-\frac{4e^2}{l}+s$$

Where *e* is the edit distance, *l* is the length of the unresolved symbol, and *s* is the scope score (calculated as five minus the number of scopes "outwards" that the declaration lies from the symbol, to a minimum of zero).

7.3.2 Diagnosis of unresolved methods

Unresolved methods are diagnosed by the prototype into one of two categories:

- Method call targeting wrong type; or
- Method not declared.

Upon detection of an unresolved method call, the diagnosis tool looks for a declaration of a method with the same name in other types: all the classes and interfaces declared in the same package as the analysed source, and the "java.lang.String" class. If there is such a method declared in one or more other classes, the tool suggests that the method call is targeting the wrong type; otherwise, it diagnoses as an undeclared method.

7.4 Example output

We will now present short examples of erroneous code together a diagnosis for each of the possible diagnoses produced by the prototype.

7.4.1 Variable not declared

The "Variable not declared" error category is diagnosed with an appropriate message which also suggests a location for a declaration to be added in order to resolve the error. In Figure 7.1 we show some example code where an undeclared variable is diagnosed, and the suggested location for the declaration is the method.

```
public class A1
{
    int foo(int b)
    {
        return undec + b;
    }
}
```

```
Diagnosis:
Reference to undeclared variable:
There seems to be use of a variable called
"undec", but the variable is not declared.

The name is used on line 5 (in the "foo" method).
I suggest that a declaration should be inserted
in the "foo" method.

Standard message:
cannot find symbol – variable undec
```

Figure 7.1: Example of erroneous code for "variable not declared" diagnosis. The "foo" method makes reference to an undeclared variable, "undec".

In Figure 7.2 an example of an undeclared identifier being used in two different methods is shown; in this case, the declaration is suggested to be added at the class level.

```
public class A2
{
    int foo1(int b)
    {
        return undec + b;
    }

    void foo2()
    {
        undec = 6;
    }
}
```

Diagnosis:

```
Reference to undeclared variable:
There seems to be uses of a variable called
"undec", but the variable is not declared.

The name is used:
- on line 10 (in the "foo2" method)
- on line 5 (in the "foo" method)

I suggest that a declaration should be inserted
at the class level.
```

Standard message:

```
cannot find symbol – variable undec
```

Figure 7.2: Second example of erroneous code for "variable note declared" diagnosis. The "foo1" and "foo2" methods both make reference to an undeclared identifier, "undec".

In both cases, the message produced by the standard compiler is "cannot find symbol – variable undec". The prototype tool generates a diagnostic that is more accurate and informative.

7.4.2 Variable name written incorrectly

The "Variable name written incorrectly" error category is diagnosed by the tool in the case of an unrecognised variable name closely matching a declared variable name. The tool assumes the use and not the declaration is incorrect if the declaration also has an existing correct reference.

```
public class A3
{
    int value;

    int foo1(int b)
    {
        return volue + b;
    }

    void foo2()
    {
        value = 6;
    }
}
```

```
Diagnosis:
line 7: Possible misspelling of value as volue

Standard message:
cannot find symbol – variable volue
```

Figure 7.3: Example of erroneous code for "variable name written incorrectly" diagnosis. The "foo2" method refers to the "value" variable; the "foo1" method misspells the variable name as "volue".

In Figure 7.3 an example is presented where a variable ("value") is referenced both correctly and incorrectly (as "volue") within the same source. The tool makes a diagnosis that "volue" is a misspelling. The standard compiler on the other hand issues the same "cannot find symbol" diagnostic that it gives for undeclared variables.

7.4.3 Incorrect variable declaration

When there appear to be multiple identical spellings in the usage of a variable, but only a non-identically spelled declaration, the tool diagnoses the error as a case of an incorrect variable declaration. An example is shown in Figure 7.4.

```
public class A4
{
    int volue;

    int foo1(int b)
    {
        return value + b;
    }

    void foo2()
    {
        value = 6;
    }
}
```

Diagnosis:

```
line 3: Likely mis-spelling of declaration of
volue; should be value?
```

Standard message:

```
cannot find symbol – variable value
```

Figure 7.4: Example of erroneous code for "incorrect variable declaration" diagnosis. The "foo1" and "foo2" methods refer to the "value" variable; the declaration misspells the variable name as "volue".

Once again, the standard compiler issues a "cannot find symbol" diagnostic for the example. The prototype tool suggests that the declaration is misspelled.

```
public class A5
{
    int volue;

    int foo1(int b)
    {
        return value + b;
    }
}
```

Diagnosis:

```
line 7: Possible misspelling of volue as value
(or vice-versa)
```

Standard message:

```
cannot find symbol – variable value
```

Figure 7.5: Example of erroneous code for ambiguous "incorrect variable declaration" / "variable name written incorrectly" diagnosis. The "foo1" method refers to the "value" variable; the declaration spells the variable name instead as "volue".

If a declaration can be matched with a single use that has non-identical spelling, and the declared identifier is not used with the same (declared) spelling elsewhere, the prototype tool does not make a judgement regarding whether the declaration or usage is incorrect, and offers an alternative diagnosis, as shown in the example in Figure 7.5.

7.4.4 Method call targeting wrong type

When a method call uses the name of a method name not declared in the receiver – which may be the implicit “this” object reference – the tool searches for declarations of the method name in other classes within the package, as well as Java’s standard “java.lang.String” class. If a declaration exists, the prototype tool diagnoses the error as a method call targeting the wrong receiver type. An example is shown in Figure 7.6.

```
public class B1
{
    Object o = "hello";

    public int getLength()
    {
        return o.length();
    }
}
```

Diagnosis:

```
Line 7: Call to method 'length' on a receiver of
type  java.lang.Object,  which  does  not  have  a
method with that name:
A method with that name is declared in the
following types: java.lang.String
Perhaps the method should be called on a
different receiver, or the receiver's type
should be declared to be one of the above types.
```

Standard message:

```
cannot find symbol – method length()
```

Figure 7.6: Example of erroneous code for “method call targeting wrong type” diagnosis. The expression “o.length()” has a receiver type of Object, which declares no “length” method.

For the example in Figure 7.6, the prototype tool’s diagnostic suggests that the “length” method be called on a different receiver or that the type of the receiver be changed (in this case, the latter is arguably the correct solution, since the receiver will hold an object of “String” type at run-time). The standard compiler instead issues a message indicating that the “length” method symbol cannot be found.

7.4.5 Method not declared

In cases where an unresolved method name cannot be matched to a declaration in another type, the prototype tool issues a diagnostic for the "method not declared" error category. An example is presented in Figure 7.7.

```
public class B2
{
    public int foo()
    {
        return bar();
    }
}
```

```
Diagnosis:
Line 5: Call to method 'bar' on a receiver of
type B2, which does not have a method with that
name:
That method does not seem to be declared.
```

```
Standard message:
cannot find symbol – method bar()
```

Figure 7.7: Example of erroneous code for "method not declared" diagnosis. The method "bar" is called with the implicit "this" receiver, but is not declared in B2 or any other class whose declarations are searched.

In this example, the diagnostic message generated by the tool is more informative, but semantically similar to the message generated by the compiler, which gives the same diagnostic as for the previous example ("method call targeting wrong type" category).

7.5 Evaluation of Prototype

We will now discuss the method and results of evaluation of the prototype. The aim of the evaluation is to determine how the accuracy and precision of the diagnostics generated by the prototype tool compared to the diagnostics generated by the standard compiler.

Evaluation was performed by running the diagnostic tool on code snapshots corresponding to compilation events that had been captured for purposes of error categorisation and severity calculation (Chapter 4.2). For each of the five error categories that the tool was designed to diagnose, we selected up to ten recorded events that had been categorised accordingly, and ran the tool on the corresponding source code snapshot. The events selected were the first appropriately categorised event in each recorded user session, up to a limit of 10 sessions, ordered by time. A small number of sessions were skipped over as they appeared to be using external code libraries which were not available to us; when this was done, the next suitable session (if any) was used instead. For two of the selected categories, “method targeting wrong type” and “method not declared”, the number of recorded sessions containing events categorised accordingly was less than ten (nine and six respectively).

Snapshots used for evaluation were taken from different sessions as we felt that using multiple snapshots from a single session could skew the results; a novice programmer might produce the same type of error repeatedly in a similar form which might lend itself to a particular diagnosis using the compiler or the prototype tool.

For each code snapshot used for evaluation, the tool was run to produce diagnostics and the diagnostic (if any) corresponding to the source line identified by the recorded compiler diagnostic as containing an error. In some cases the recorded compiler diagnostic was in a language other than English and in these cases the English-language diagnostic was produced by also compiling the code in the snapshot with the standard compiler (from within the modified BlueJ environment).

The diagnostic message produced by the tool, and the recorded (or regenerated) diagnostic from the compiler, were both assessed for accuracy as a binary choice of either accurate or inaccurate. A message was considered accurate if it was factually correct in the context of the categorisation recorded and without being dependent on internal compiler/tool state (an assessment of accuracy that required the diagnostic to match the specific categorisation closely would have lead to the standard compiler’s diagnostic being judged inaccurate in the majority of cases; the requirement for not being dependent on internal state allows diagnostics that pertain to parse state, for example, to be classified as inaccurate if they do not otherwise appropriately describe the error as categorised).

If both compiler and prototype tool were assessed to have produced accurate diagnostic messages, a further judgement assessing the precision of the diagnostic produced by the tool as compared to that of the compiler was made. The precision of the tool’s diagnostic could be designated as higher, equivalent, or lower than that of the compiler.

In the following sections, we will discuss the results of this evaluation for each of the five error categories that the tool was designed to diagnose in turn, in each case ending with a summary of the accuracy and precision judgements.

The recorded compilation event snapshots used for evaluation, and the judgement regarding accuracy and relative precision of the compiler diagnosis and the prototype diagnosis, are included in Appendix D.

7.5.1 Variable not declared

For the ten code snapshots used for evaluation of the diagnosis of the “Variable not declared” error category, the compiler diagnostic was assessed as accurate in all ten cases. The tool was accurate in nine cases; in the remaining case, it did not produce a diagnostic, because the undeclared variable was used in a method-call receiver context and might therefore have referred to a value or type (the latter in the case of a *static* call, i.e. class-member method call) and the tool only attempts to resolve value references occurring in a strict value context.

The compiler diagnostic, for each compilation event snapshot in this category, is of the form:

cannot find symbol – variable x

In comparison, an example of a diagnostic generated by the tool (and showing the usual form of diagnostic messages generated by the tool for snapshots in this category) is:

Reference to undeclared variable:
There seems to be use of a variable called "ergebnis", but the variable is not declared.

The name is used on line 6 (in the "main" method).

I suggest that a declaration should be inserted in the "main" method.

Note that the compiler diagnostic includes a line number not included as part of the message show above, however, the diagnostic from the tool (which appeared in the above form in eight cases) is more informative, and was judged more precise, particularly as it suggests a suitable location for correction. In the one remaining case, the tool and compiler diagnostic diagnosed two logically distinct errors appearing on the same line of source code, and so comparison of the precision of the diagnostics was not appropriate.

Summary:
Compiler accuracy: 10/10
Tool accuracy: 9/9, with no diagnosis given in one case
Precision: The tool gave higher precision diagnostics in 8 cases; one case not assessed.

7.5.2 Variable name written incorrectly

Ten distinct code snapshots with errors categorised as “Variable name written incorrectly” were used to evaluate the prototype’s diagnostics of errors in this category. The standard compiler diagnostic message was considered accurate in nine out of the ten cases; the tool produced accurate diagnostics in six cases, and did not diagnose the remaining four cases.

The error causing the compiler to produce an inaccurate diagnostic was found in the following line of source code:

```
Word Count ++;
```

Here, the “wordCount” variable name had been written incorrectly as “Word Count”, which contains whitespace that confuses the compiler’s parser. The compiler diagnostic for this error indicates “; expected”, a message that is clearly a result of the parser state and which is inaccurate in this context. The tool did not produce a diagnosis in this case; the parse failure prevented any relevant symbols being added to the unresolved symbol set.

The three remaining cases where the tool did not produce a diagnosis were those in which the incorrect variable name was qualified (referencing a variable defined in another type). The tool design was limited to resolve only unqualified symbols, and so it did not detect these errors or provide diagnoses. That this occurred in three of ten cases suggests that proper handling of qualified variables is essential for a fully-developed diagnosis tool; we believe that this would be a straight-forward extension to the existing design.

In all cases the compiler diagnostic was in the form of “cannot find symbol – variable x”, except the case discussed above, and a single other case in which the diagnostic was in the form of “cannot find symbol – method x()”. In contrast, the prototype diagnostic tool produced three distinct forms of diagnostic: in three cases, the “variable not declared” form shown in Chapter 7.4.1; in two cases, the “variable xxx possibly misspelled as yyy” form shown in Chapter 7.4.2; and, one single case of the “method not declared” form which is shown in Chapter 7.4.5.

The “cannot find symbol – method” and “method not declared” diagnostic messages generated by the compiler and prototype tool respectively correspond to a single event snapshot which appears to have been miscategorised and do not reflect a failure of either to produce an accurate diagnostic.

The tool's diagnostic was judged equivalently precise in this case, as both diagnostics convey the same essential information.

When the prototype tool's diagnosis was of either the "variable not declared" or "variable xxx possibly misspelled as yyy" form, the corresponding compiler diagnostic was of the "cannot find symbol – variable x" form. The first of the tool's diagnostic forms was judged equally precise as the compiler's, and the second (issues in two cases) was judged more precise.

Summary:
Compiler accuracy: 9/10
Tool accuracy: 6/6, with no diagnosis given in 4 cases
Precision: Tool higher in 2 cases, equivalent in 4 cases, and lower in no case.

7.5.3 Incorrect variable declaration

Ten code snapshots with errors categorised as "Incorrect variable declaration" were used to compare the prototype's diagnostics of errors in this category with those of the standard compiler. The standard compiler diagnostic message was considered accurate all cases; the tool produced accurate diagnostics in five cases, and did not produce a diagnostic for the remaining five cases. Note that the error category is broad and the tool was designed to diagnose a special case of the category (Chapter 7.2.3).

The tool did not produce "perfect" diagnostic form (Figure 7.4), which assesses the declaration of a variable to have used incorrect spelling, in any case. It did however produce the secondary form (Figure 7.5), which states that either the declaration or use is incorrect, in one instance. In all other cases, it produced the "variable not declared" form of diagnostic (Figure 7.1), which may still be accurate in the presence of an incorrect variable declaration but is less precise.

The compiler diagnosis was judged accurate in all cases. The tool diagnostic was accurate in all cases where it gave a diagnosis (although borderline in one case). In all but one of those cases, the tool's diagnostic was assessed as more precise than that produced by the compiler.

In the one case where the compiler diagnostic was assessed to be of higher precision than the prototype-produced diagnostic, there was a syntactical error in the statement declaring the relevant variable. This caused a parse error and led to the declaration not being recognised by the prototype, so that it diagnosed an undeclared variable rather than an issue with the declaration, which is strictly accurate (since a statement containing a syntax error cannot declare anything) although clearly imprecise. This suggests that a fully developed diagnosis tool should perhaps not perform diagnoses involving symbol resolution upon detection of a syntax error, or be better able to process declarations with incorrect syntax.

Summary:
Compiler accuracy: 10/10
Tool accuracy: 5/5, with no diagnosis in 5 cases.
Precision: The tool produced a higher-precision diagnostic in 4 cases, lower-precision in 1 case.

7.5.4 Method call targeting wrong type

Nine sessions usable for evaluation were categorised with this error; several sessions which were otherwise usable had code which appeared to use external libraries that were not available to us, which could interfere with symbol resolution.

The tool did not detect or diagnose one case, which on inspection appeared to be a miscategorisation. In 7 of the remaining 8 cases, it reported the possibility of an attempt to call a method declared in a type different to that of the receiver; in the final case, it reported an undeclared method. In all 9 cases where the tool provided a diagnosis, the diagnosis was judged to be more precise than that of the compiler.

Summar:
Compiler accuracy: 9/9
Tool accuracy: 8/9 (one case of no diagnosis).
Precision: The tool produced a higher-precision diagnostic in all 8 cases.

7.5.5 Method not declared

Six sessions usable for evaluation were categorised with this error; a single snapshot from each of these sessions was used for evaluation. In all cases, both the compiler diagnostic and the tool diagnostic were accurate, and followed the same form. For the compiler diagnostic, the form of each diagnostic was (where “xyz” is the name of the relevant method):

cannot find symbol - method xyz()

If the method lookup requires a method with certain argument types, these are included in the method signature. For the prototype tool, diagnostic messages took the following form:

> *Line 73: Call to method 'xyz' on a receiver of type Student,*
> *which does not have a method with that name:*
> *That method does not seem to be declared.*

The latter form provides some additional information over that of the equivalent compiler diagnostic, though it does not include method parameter types; for this reason, the two forms were judged as equivalently precise. However, it should be noted that the compiler form is almost invariably used when the method is declared in another class (but not in the receiver), whereas the diagnostic tool uses a different form of diagnostic in that case. In this sense, the diagnosis of the tool is more precise than that of the compiler.

Summary:
Compiler accuracy: 6/6
Tool accuracy: 6/6
Precision: Equivalent in all cases.

7.5.6 Overall Summary of Evaluation

The prototype was able to accurately diagnose the errors in the majority of code snapshots used for evaluation, in many cases with a higher-precision diagnostic message than that produced the standard compiler, and with equivalent precision in the clear majority of all other cases. This is an impressive result given the limitations in the design of the prototype, and strongly suggests that the approach taken to its design may be worth further exploration.

7.5.7 Risks and Limits to Validity

The evaluation relied on personal judgements regarding accuracy and precision of diagnostics for errors. While we have strived for objectivity, and provided reasoning where appropriate, this should be considered when interpreting the results of the evaluation.

The evaluation was performed on a limited number of code samples, which had already manually been categorised as containing errors relevant to the diagnostic capabilities of the tool. This is suitable for assessing the feasibility of approach, but the results should not be considered for determining the suitability of using the prototype tool for diagnosis of arbitrary errors.

8 Future Work and Conclusions

8.1 Summary

This thesis has presented work to develop a methodology for categorisation of programming errors, providing categorisation more accurate and precise than previous attempts found in literature. The ability of such a categorisation to properly distinguish novice errors has been established, and the importance of measuring error category severity rather than merely frequency has been demonstrated. In the introduction to this work, several hypotheses were proposed:

Hypothesis 1 : *Measuring the frequency of logical error categories will yield different results than diagnostic message frequency*.

Sub-hypothesis 1.1 : *A suitably experienced human can reliably recognise the cause of an error in many cases, even when the compiler gives a misleading error message*.

Hypothesis 2 : *Error frequency tables are misleading in the sense that they do not accurately represent the most difficult to handle errors; with a different measure, where a severity calculation is made, a different result will ensue*.

Hypothesis 3 : *in some cases reasoning that humans make to decide error cause can be implemented as a heuristic in an automated tool to produce more useful diagnostic messages*.

In Chapter 4.1 we described a methodology for categorising errors using logically distinct error categories rather than by the diagnostic message produced by a compiler, as had been done in previous work; the results, in Chapter 5, affirm Hypothesis 1: categorisation by suitably refined logical categories provides a more accurate and precise determination of the frequency with which different errors are encountered. Further, the validation of the category scheme also described in Chapter 4.1 affirms the sub-hypothesis, by showing that multiple human coders who were experienced programmers were able to manually categorise errors with a high level of agreement.

In Chapter 4.2, we extended the methodology devised in Chapter 4.1 to measure the severity of errors in different categories. The results (Chapter 6) affirm the second hypothesis: a ranking of error categories by severity results in a markedly different ordering than using frequency alone, which as a measure of the impact of an error category can therefore be misleading.

Finally, in Chapter 7, we presented the design of a prototype-level, proof-of-concept tool for performing sophisticated error diagnosis with heuristic-based reasoning derived from human reasoning regarding several high-severity errors. We evaluated the tool by comparing its diagnoses of a number of code snapshots, that had formed part of the dataset for the research described in Chapter 4.2, with the diagnoses produced by the *javac* compiler for those same snapshots. The results of evaluation affirm the final hypothesis, showing that this approach is indeed viable.

8.2 Recommendations for Future Work

8.2.1 Longitudinal Study of Error Severity

The error severity study (Chapter 4.2) relied on data selected at random from the *Blackbox* project (Brown et al. 2014). Blackbox collects data including anonymised code snapshots from users of BlueJ, who are expected for the most part to be novice programmers. However, the data selected for the study would have represented programmers at various stages of learning and experience. While this was useful for capturing an overall representation of error severity across different stages, it would also be beneficial to (for example) the development of course material and structure if educators could obtain information regarding the severity of different categories of error at different stages of development.

A longitudinal study would require a large sample of program errors (in the form of code snapshots) taken over a period of time, preferably a complete year or more, from participants at a known stage of learning. This would preclude using data sampled completely at random from the Blackbox database, although Blackbox supports a mechanism to allow identification of groups of participants which could enable such a study using a known cohort.

Collecting a sample of errors from different known stages of learning would make it possible to make a determination of the severity of different error categories at different stages. The most problematic aspect of such a study would be the error categorisation, which is a tedious manual process. We have shown that using compiler diagnostic messages for categorisation is not suitable for developing an accurate picture of error severity distribution, however, meaning that manual categorisation is essential at this point in time; however, it remains an open question whether automated error diagnosis could be improved to the point where this was no longer necessary.

8.2.2 Further Development and Evaluation of Diagnostic Tool

The tool developed as described in Chapter 7 is a prototype, intended for assessing the feasibility of applying reasoning to develop heuristic techniques for error diagnosis. With a simple self-conducted evaluation, we showed that it was possible to improve on the precision of diagnosis of a number of errors that had been classified under high-severity categories. The approach to the design of the tool was to develop a number of heuristic techniques for refinement of error diagnosis once an error had been detected.

The techniques developed rely in some cases on using a scoring system to decide between two possibilities forming part of a diagnosis. In Chapter 7.3.1 we outlined the concept of a *declaration-use match score*, and described how it could be used to decide on a suitable match between an unresolved variable identifier and any number of potentially matching declarations. The formula derived for the declaration-use match score using straight-forward reasoning but its suitability for choosing the best match has not been evaluated; a future study could perform such an evaluation and assess the effects of modifying the formula in various ways, such by differently weighting the difference in scope level between declaration and usage compared to string edit distance.

Since the prototype tool relies on post-parse analysis, it is not as reliable in the presence of syntactical errors in the code. It is possible that the error recovery in the parser could be improved in order to allow it to recognise, for instance, variable declarations with minor syntax issues.

More generally, the tool could be further developed in order to allow it to be suitable for diagnosing a larger range of errors. Although it could diagnosis at least some cases of five out of the ten most severe error categories, the remaining five were broad and considered likely to be too difficult to diagnose for this feasibility study. However, this certainly warrants further investigation; broad error categories may have specific cases for which it is possible to automate diagnosis, and the severity of these error categories would justify such efforts. It may also be straight-forward to diagnose some of the error categories not within the top-ten severity list.

While better automated error diagnosis has been shown to be possible, there remains the question of whether and how much a tool designed around the techniques we have presented would actually benefit novice programmers. A study to assess this would ideally have two groups of participants with equal ability and prior experience, one of which (the control group) would use a conventional compiler, and the other of which would use an improved compiler or additional tool with enhanced diagnostic capability, to complete a number of programming tasks. Performance during these tasks would be analysed to compare the effectiveness of the tool.

8.2.3 Extension to Other Programming Languages

The research presented in this thesis applies to the Java programming language, however the methodology used should be equally applicable to other programming languages. A comparison between the distribution of error severity amongst the error categories applicable to two or more different programming languages could give insights into particular aspects of the different languages that make them more or less difficult for novices to learn and understand. This, in turn, could guide the design of new programming languages, especially those designed specifically for the teaching of programming.

8.3 Conclusion

This work has affirmed the hypotheses set out in the introduction. A methodology for the development of logical error categories, and for approximating the difficulty of resolution of errors in the different categories, has been developed and applied; the notion of error severity as a product of frequency and difficulty has been formulated, and a ranking of error categories by severity has been produced relevant for novices using the Java programming language. Error severity has been show both in tabular form and using plots showing severity as a combination of the two factors, frequency and difficulty.

We have produced a prototype tool for automated diagnosis of a number of high severity errors, and shown via an evaluation that it can typically diagnose errors with equal or better precision than a standard compiler, demonstrating the feasibility of the design approach used for production of the tool and establishing a basis for further development and study of the techniques employed. A number of possible areas for future work on this topic have been outlined.

References

Ahmadzadeh, M., Elliman, D. and Higgins, C. (2005). An Analysis of Patterns of Debugging among Novice Computer Science Students. *ACM SIGCSE Bulletin*, vol. 37, no. 3, pp. 84-88

Amabile, T. and Kramer, S. (2011). *The Progress Principle: Using Small Wins to Ignite Joy, Engagement and Creativity at Work*. Harvard Business Press.

Attride-Stirling, J. (2001). Thematic networks: an analytic tool for qualitative research. *Qualitative Research*, vol. 1, no. 3, pp. 385-405

Becker, B.A. (2016). An Effective Approach to Enhancing Compiler Error Messages. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 126-131

Ben-Ari, M. (1998). Constructivism in Computer Science Education. *ACM SIGCSE Bulletin*, vol. 30, no. 1, pp. 257-261

Bloch, S.A. (2000). Scheme and Java in the First Year. *Journal of Computing Sciences in Colleges*, vol. 15, no. 5, pp. 157-165

du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Computing Education Research*, vol. 2, no. 1, pp. 57-73

Braun, V. and Clarke, V. (2006). Using Thematic Analysis in Psychology. *Qualitative Research in Psychology*, vol. 3, no. 2, pp. 77-101

Brown, N.C., Kölling, M., McCall, D. and Utting, I. (2014). Blackbox: A Large Scale Repository of Novice Programmer's Activity. *Proceedings of the 45th ACM technical symposium on Computer Science Education*, pp. 223-228

Brown, P.J. (1983). Error Messages: The Neglected Area of the Man/Machine Interface. *Communications of the ACM*, vol. 26, no. 4, pp. 246-249

Cooper, S., Dann, W. and Pausch, R. (2003). Teaching Objects-first in Introductory Computer Science. *ACM SIGCSE Bulletin*, vol. 35, no. 1, pp. 191-195

Dann, W., Cosgrove, D., Slater, D., Culyba, D. and Cooper, S. (2012). Mediated Transfer: Alice 3 to Java. *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pp.

141-146

DePasquale, P., Lee, J.A. and Pérez-Quiñones, M.A. (2004). Evaluation of Subsetting Programming Language Elements in a Novice's Programming Environment. *ACM SIGCSE Bulletin*, vol. 36, no. 1, pp. 260-264

Denny, P., Luxton-Reilly, A. and Carpenter, D. (2014). Enhancing Syntax Error Message Appears Ineffectual. *Proceedings of the 2014 conference on Innovation & Technology in Computer Science Education*, pp. 273-278

Denny, P., Luxton-Reilly, A., Tempero, E. and Hendrickx, J. (2011). Understanding the Syntax Barrier for Novices. *Proceedings of the 16th annual joint conference on Innovation and Technology in Computer Science Education*, pp. 208-212

Denny, P., Luxton-Reilly, A., Tempero, E. and Hendrickx, J. (2011). CodeWrite: Supporting Student-Driven Practice of Java. *Proceedings of the 42nd ACM technical symposium on Computer Science Education*, pp. 471-476

Dillon, E., Anderson, M. and Brown, M. (2012). Comparing Feature Assistance Between Programming Environments and their Effect on Novice Programmers. *Journal of Computing Sciences in Colleges*, vol. 27, no. 5, pp. 69-77

Dy, T. and Rodrigo, M. (2010). A Detector for Non-Literal Java Errors. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pp. 118-122

Eclipse. (n.d.). [Online], Available from: <http://www.eclipse.org/> [September 2016]

Felleisen, M., Findler, R.B., Flatt, M. and Krishnamurthi, S. (1998). The DrScheme Project: An Overview. *ACM Sigplan Notices*, vol. 33, no. 6, pp. 17-23

Fidge, C. and Teague, D. (2009). Losing Their Marbles: Syntax-Free Programming for Assessing Problem-Solving Skills. *Proceedings of the Eleventh Australasian Conference on Computing Education*, vol. 95, pp. 75-82

Fleury, A.E. (2000). Programming in Java: Student-constructed Rules. *ACM SIGCSE Bulletin*, vol. 32, no. 1, pp. 197-201

Flowers, T., Carver, C.A. and Jackson, J. (2004). Empowering Students and Building Confidence in Novice Programmers through Gauntlet. *FIE 2004*

GCC, the GNU Compiler Collection. (n.d.). [Online], Available from: <https://gcc.gnu.org> [September 2016]

Gaspar, A., Langevin, S. and Boyer, N. (2008). Redundancy and Syntax-late Approaches in Introductory Programming Courses. *Journal of Computing Sciences in Colleges*, vol. 24, no. 2, pp. 204-212

Grandell, L., Peltomäki, M., Back, R. and Salakoski, T. (2006). Why Complicate Things? Introducing Programming in High School using Python. *Proceedings of the 8th Australasian Conference on Computing Education*, vol. 52, pp. 71-80

Gray, K.E. and Flatt, M. (2003). ProfessorJ: A Gradual Introduction to Java through Language Levels. *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, pp. 170-177

Grune, D., van Reeuwijk, K., Bal, H.E., Jacobs, C.J. and Langendeon, K. (2012). *Modern Compiler Design*. Springer.

Hartmann, B., MacDougall, D., Brandt, J. and Klemmer, S.R. (2010). What Would Other Programmers Do? Suggesting Solutions to Error Messages. *Processings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1019-1028

Hasker, R.W. (2002). HiC: A C++ Compiler for CS1. *Journal of Computing Sciences in Colleges*, vol. 18, no. 1, pp. 56-64

Holland, S., Griffiths, R. and Woodman, M. (1997). Avoiding Object Misconceptions. *ACM SIGCSE Bulletin*, vol. 29, no. 1, pp. 131-134

Hovemeyer, D. and Pugh, W. (2004). Finding Bugs is Easy. *ACM Sigplan Notices*, vol. 39, no. 12, pp. 92-106

Hristova, M., Misra, A., Rutter, M. and Mercuri, R. (2003). Identifying and Correcting Java Programming Errors for Intro CS Students. *ACM SIGCSE Bulletin*, vol. 35, no. 1, pp. 153-156

Jackson, J., Cobb, M. and Carver, C. (2005). Identifying Top Java Errors for Novice Programmers. *Frontiers in Education Conference*, vol. 35, no. 1, p. 0

Jadud, M.C. (2005). A First Look at Novice Programmer Behaviour using BlueJ. *Computer Science Education*, vol. 15, no. 1, pp. 25-40

Jeffery, C.L. (2003). Generating LR Syntax Error Messages from Examples. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 5, pp. 631-640

Jikes. (n.d.). [Online], Available from: <http://jikes.sourceforge.net/> [September 2016]

Johnson, S.C. (n.d.). yacc. [Online], Available from: <http://dinosaur.compilertools.net/yacc/> [September 2016]

Johnson, W.L. (1990). Understanding and Debugging Novice Programs. *Artificial Intelligence*, vol. 42, no. 1, pp. 51-97

Kummerfeld, S.K. and Kay, J. (2003). The Neglected Battlefield of Syntax Errors. *Proceedings of the fifth Australasian Conference on Computing Education*, vol. 20, pp. 105-111

Kölling, M. (2010). The Greenfoot Programming Environment. *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 14

Kölling, M., Brown, N.C. and Altadmri, A. (2015). Frame-Based Editing: Easing the Transition from Blocks to Text-Based Programming. *Proceedings of the Workshop in Primary and Secondary Computing Education*, pp. 29-38

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003). The BlueJ System and its Pedagogy. *Journal of Computer Science Education*, vol. 13, no. 4, p. 0

Lang, B. (1986). On the Usefulness of Syntax Directed Editors. *Advanced Programming Environments*, pp. 47-51

Lang, B. (2002). Teaching New Programmers: A Java Tool Set as a Student Teaching Aid. *Proceedings of the inaugural conference on the Principles and Practice of programming*, pp. 95-100

Lee, M.J. and Ko, A.J. (2011). Personifying Programming Tool Feedback Improves Novice Programmers' Learning. *Proceedings of the seventh international workshop on Computing Education Research*, pp. 109-116

Lombard, M., Snyder-Duch, J. and Bracken, C.C. (2002). Content Analysis in Mass Communication: Assessment and Reporting of Intercoder Reliability. *Human Communication Research*, vol. 28, no. 4, pp. 587-604

Madison, S.K. (1995). A Study of College Students' Construct of Parameter Passing Implications for Instruction. *Doctoral Dissertation, University of Wisconsin*

Maloney, J., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education (TOCE)*, vol. 10, no. 4, p. 16

Marceau, G., Fisler, K. and Krishnamurthi, S. (2011). Mind Your Language: On Novices' Interactions with Error Messages. *Proceedings of the 10th SIGPLAN symposium on new ideas, new paradigms, and reflections on programming and software*, pp. 3-18

McIver, L. and Conway, D. (1996). Seven Deadly Sins of Introductory Programming Language Design. *Software Engineering: Education and Practice, 1996. Proceedings. International Conference.*, pp. 309-316

Molich, R. and Nielsen, J. (1990). Improving a Human-Computer Dialogue: What Designers Know about Traditional Interface Design. *Communications of the ACM*, vol. 33, no. 3, pp. 338-348

Moth, A.L., Villadsen, J. and Ben-Ari, M. (2011). SyntaxTrain: relieving the pain oof learning syntax. *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, p. 387

NetBeans IDE. (n.d.). [Online], Available from: <https://netbeans.org/> [September 2013]

Nienaltowski, M., Michael, P. and Bertrand, M. (2008). Compiler Error Messages: What Can Help Novices. *ACM SIGCSE Bulletin*, vol. 40, no. 1, pp. 168-172

Parr, T. (n.d.). ANTLR. [Online], Available from: <http://www.antlr.org/> [September 2016]

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M. and Paterson, J. (2007). A Survey of Literature on the Teaching of Introductory Programming. *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 204-223

Ragonis, N. and Ben-Ari, M. (2005). A Long-term Investigation of the Comprehension of OOP Concepts by Novices. *Computer Science Education*, vol. 15, no. 3, pp. 203-221

Schneiderman, B. (1982). Designing Computer System Messages. *Communications of the ACM*, vol. 25, no. 9, pp. 610-611

Schorsch, T. (1995). CAP: An Automated Self-Assessment Tool to Check Pascal Programs for Syntax, Logic and Style Errors. *ACM SIGCSE Bulletin*, vol. 27, no. 1, pp. 168-172

Shackelford, R.L. and Badre, A.N. (1993). Why Can't Smart Students Solve Simple Programming Problems?. *International Journal of Man-Machine Studies*, vol. 38, no. 6, pp. 985-997

Sorva, J. (2013). Notional Machines and Introductory Programming Education. *ACM Transactions on Computing Education (TOCE)*, vol. 12, no. 2

Sorva, J., Karavirta, V. and Malmi, L. (2013). A Review of Generic Program Visualization Systems for Introductory Programming Education. *ACM Transactions on Computing Education (TOCE)*, vol. 13, no. 4, pp. 1-64

Spohrer, J.C., Soloway, E. and Pope, E. (1985). A Goal/Plan Analysis of Buggy Pascal Programs. *Human Computer Interactions*, vol. 1, no. 2, pp. 163-207

Tabanao, E.S., Rodrigo, M.T. and Jadud, M.C. (2011). Predicting At-Risk Novice Java Programmers Through the Analysis of Online Protocols. *Proceedings of the seventh international workshop on Computing Education Researech*, pp. 85-92

Traver, V.J. (2010). On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction*

Vee, M.N., Meyer, B. and Mannock, K.L. (2006). Empirical Study of Novice Errors and Error Paths in Object-Oriented Programming. *7th Annual HEA-ICS conference*, p. 0

Warren, P. (2001). Teaching Programming using Scripting Languages. *Journal of Computing*

Sciences in Colleges, vol. 17, no. 2, pp. 205-216

Appendix A List of Error Categories

In this appendix, the error categories created during the initial category development phase of the research presented in this thesis (Chapter 4.1) are listed in full, with category name, long description (not present for some categories), and number of occurrences found in the data set after categorisation.

Error category	**Long description**	**#**
Variable not declared	A variable was used or assigned to, but not declared at all (see also: "Variable name written incorrectly", "Use of variable from another class")	41
; missing	semicolon missing at end of statement	38
Variable name written incorrectly	A name of an existing variable was either mis-spelt, or the wrong word was used (slip of the mind, not misunderstanding about variables). The student seemed to know what the variable was, and it was declared. See also "Wrong variable used".	31
Invalid syntax	An invalid syntactical construct that we cannot otherwise categorise	29
Method name written incorrectly	simple typo, or wrong word used	18
Missing parentheses for method call	A method call was missing the required parentheses.	15
Unhandled exception	A method is called which throws an exception; that exception is neither caught nor specified as thrown by the calling method.	11
Class name written incorrectly	Class name misspelled or otherwise not written correctly.	10
Method call: parameter type mismatch	A value is used as a method call parameter; a value of another type is needed.	9
Missing parentheses for constructor call	a "new XYZ()" expression without the required parentheses.	9
Type mismatch in assignment	A value was assigned to a variable, but the value type and variable type are not compatible. (See also "Value from raw collection must be cast")	9
Mismatched parentheses in or around expression	One two many or one two few parentheses in an expression (or around it, eg in 'if' statement condition).	7
Array used in place of	An array itself was used where an element of the array was	6

array element	probably needed. Essentially '[x]' is missing from the expression, for some unknown x.	
Missing '.' between names in qualified name		6
Variable needs a unique name	A variable was declared twice, however the intent appears to be declaration of two separate variables (rather than re-assignment to an existing variable).	6
Extraneous closing curly brace	Extraneous '}'. See also "misplaced curly brace"	5
Keyword written incorrectly	A language keyword was mistyped or misspelled	5
Missing closing curly brace after control stmt body	Missing curly brace at end of statement block after "if" or "while" etc.	5
Uncategorised	Errors that - for whatever reason - were not categorised. Most likely undecidable.	5
Class does not implement required method	The class is declared to implement an interface, but doesn't appear to implement the required method at all.	4
Constructor call to non-existent copy constructor	A constructor call with a single parameter matching the type being constructed, where it appears the intention was to create a copy of the original object, however no such constructor is defined.	4
Method call: parameter number mismatch	A method call to an existing method has too many or two few parameters.	4
Method call: parameter number mismatch; call incorrect	A method call to an existing method has too many or two few parameters, and it seems that the call is wrong rather than the declaration.	4
Method declaration: no type specified for parameter	In a method declaration, no type is specified for one or more parameters (the type specifier is omitted).	4
Misplaced closing curly brace	A closing curly brace '}' should be in a different location than where it was placed.	4
Assignment to a	Assignment to an element of a collection retrieved by get(...) or	3

collection element	similar method, with the apparent intention of storing to the collection.	
Class should be declared to implement interface	A class should be declared to implement some interface. (implements ...).	3
Extra closing parenthesis	An extra closing parenthesis - ')' - at the end or in the middle of an expression or other location requiring parentheses.	3
Method call: Parameter types included	types were included in a method call (as in method definition). See also "Method call: parameter doubles as declaration"	3
Method declaration: Semicolon at end of method header	spurious semicolon at the end of a non-abstract method header	3
Missing 'return' statement	A method declared to return a value does not have a return statement at all.	3
Missing closing curly brace at end of method		3
Use of non-static method from static context		3
Wrong variable used	A variable name (identifying an existing variable) was used in some context, but a different variable was intended.	3
: in place of ;	colon instead of semicolon	2
Method call targeting wrong type	A method call was made that does not exist in the class of the method call target, but exists in another class. (eg: in expression "xyz.abc()", "xyz" is of type String, but the "abc" method is defined in another class).	2
Missing closing parenthesis in constructor call	missing or misplaced closing parenthesis	2
Missing operator between expression elements	A missing operator between two values in an expression	2
Missing space after 'new'	In a constructor call, the whitespace between the 'new' keyword and the class name is missing.	2
Type mismatch in	The 'return' statement returns the wrong type (not the declared	2

'return' statement	return type for the method).	
Use of variable in context requiring a type	In a context requiring a type (class or interface name), a variable name was used instead. See also "Constructor call attempted using variable name"	2
'=' used in place of '=='	Equality check using wrong operator	1
Call to 'super' in constructor not first statement	There is a 'super' call in a constructor, but it is not the first statement in the constructor.	1
Class not defined	A class was referenced by name, but not defined.	1
Collection used in place of element	A collection was used when an element of the collection was most likely intended.	1
Comment incorrectly written	It appears that a comment was intended, but it is not written in the correct syntax.	1
Constructor call attempted using variable name	A constructor call was intended, but the intended destination variable name was used instead of the appropriate class name.	1
Constructor call without using 'new'	An attempted constructor call (object creation) lacks the 'new' keyword	1
Constructor is declared to return 'void'	A constructor was incorrectly specified to return 'void'.	1
Constructor parameter type/number mismatch		1
Extraneous '(' between 'else' and 'if'		1
Extraneous opening curly brace	There was an extraneous opening curly brace	1
Local variable declaration with illegal modifier	A declaration for a local variable includes a modifier such as 'private' or 'static' which is not allowed for local variables.	1
Method call attempted using variable name	A variable name was treated as a method name (eg. argument list was applied). It appears that a method call was intended.	1
Method call: missing comma between parameters		1
Method call:	A method parameter type is included in the method call parameters,	1

parameter doubles as declaration	together with a name, in such a way that it appears a declaration of a variable may be intended.	
Method call: parameter number mismatch; declaration incorrect	A method call where the supplied arguments don't match the declaration, and it appears that the declaration is wrong. More specific than: "Method call: parameter number mismatch"	1
Method call: semicolon in place of comma	A semicolon separates method call parameters	1
Method declaration: Missing comma in parameters	A missing comma between parameters in a method declaration	1
Method declaration: missing method body	The method body is missing for a method declaration (and the method is not abstract). See also "Method declaration: Semicolon at end of method header"	1
Method declaration: missing return type		1
Method declaration: return type should be void	A method is declared with a return type, but should be declared to return void.	1
Method duplicated exactly	A method is declared twice, exactly duplicated.	1
Missing closing curly brace - ambiguous	There is a missing closing curly brace, and no clear indication of where the brace is really missing.	1
Missing closing curly brace at end of class		1
Missing opening curly brace after method header		1
Object instantiation uses variable name	An apparent object instantiation is attempted, but using a variable name rather than a class name as the type to instantiate.	1
Object instantiation uses variable name; intends variable access	A "new xyz()"-type expression where 'xyz' is actually a variable, and it appears that the intention is to access the value of the variable rather than construct a new object.	1
Parenthesis in wrong	A parenthesis is in the wrong location.	1

location		
Statement outside method/block	A statement appears in an invalid context outside of a method body or initialiser block.	1
Type doesn't exist	A reference to a type (class or interface) was made, but the named type does not appear to exist.	1
Type mismatch: arithmetic performed on String; use length	Arithmetic operator applied to a String, most likely the string length should be taken.	1
Use of non-static variable from static context		1
Use of variable from another class	A field or local variable declared in another class was used.	1
Value from raw collection must be cast	A value is pulled from a 'raw' collection (List, Map etc) and not cast to the appropriate type.	1
Variable access used as statement	An expression accessing a variable value is being used as a standalone statement.	1
Variable declaration: name and type in wrong order		1
Variable declaration: name missing	A variable declaration occurs without a variable name	1
Variable declaration: variable name contains whitespace		1
Class from other package used without 'import'		0
Constructor call: invalid syntax	A constructor call was intended, but the syntax used was wrong. (There are more specific categories which may be more appropriate for some errors).	0
Invalid type cast	A type cast was used but is illegal due to type hiearchy or eg casting a primitive type to a reference type	0
Method declaration: semicolon in place of comma	A method declaration has a semicolon (;) in place of comma (,) in the parameter list.	0

Method not declared		0

Appendix B List of Revised 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), as created during the category development prior to severity calculation and ranking (4.2.2) . 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 “(*)”.

Category	Occurrences
Uncategorized (*)	94
Variable not declared	85
Variable name written incorrectly	75
; missing	74
Simple syntactical error (*)	66
Semantic error (*)	49
Variable: Incorrect variable declaration (*)	48
Variable: Incorrect attempt to use variable (*)	47
Method name written incorrectly	47
Method: Incorrect method declaration (*)	46
Class or type name written incorrectly	44
Type error	38
Method not declared	29
Method call: parameter number mismatch	27
Method call targeting wrong type	25
Missing 'return' statement	23
Extraneous closing curly brace	21
Missing parentheses for method call	21
Method: Incorrect method call (*)	20
Use of non-static method from static context	16
Wrong return type declared	15
Class from other package used without 'import'	13
Missing closing curly brace at end of class	12
Missing operator between expression elements	12
Type mismatch in assignment	12
Method call: parameter type mismatch	10
Method declaration: missing return type	10
Mismatched parentheses in or around expression	10
Variable needs a unique name	9
Array used in place of array element	8
Missing closing curly brace at end of method	7
Unhandled exception	7
'=' used in place of '=='	6
Missing closing curly brace after control stmt body	6
Incorrect/attempted use of class or type (*)	5

Constructor parameter number mismatch	5
Method declaration: Wrong number of parameters	5
Use of non-static variable from static context	5
Constructor: Incorrect constructor declaration (*)	4
Class not defined	4
Keyword written incorrectly	4
Method call: Parameter types included	4
Use of variable from another class	4
Wrong variable used	4
Constructor: Incorrect constructor call (*)	2
Class does not implement required method	2
Extra opening curly brace '{'	2
Extraneous or misplaced ')'	2
Method declaration: missing method body	2
Method declaration: no type specified for parameter	2
Method declaration: return type should be void	2
Method duplicated exactly	2
Missing closing curly brace	2
Missing opening curly brace after method header	2
Statement outside method/block (*)	2
: in place of ;	1
Comment incorrectly written	1
Constructor parameter type mismatch	1
Invalid type cast	1
Local variable declaration with illegal modifier	1
Method declaration: Semicolon at end of method header	1
Missing parentheses for constructor call	1
Assignment to a collection element	0
Call to 'super' in constructor not first statement	0
Class should be declared to implement interface	0
Collection used in place of element	0
Constructor call attempted using variable name	0
Constructor call to non-existent copy constructor	0
Constructor call without using 'new'	0
Constructor is declared to return 'void'	0
Extra closing parenthesis	0
Extraneous '(' between 'else' and 'if'	0
Method call attempted using variable name	0
Method call: missing comma between parameters	0
Method call: parameter doubles as declaration	0
Method declaration: Missing comma in parameters	0
Method declaration: semicolon in place of comma	0
Missing '.' between names in qualified name	0
Missing closing parenthesis in constructor call	0
Missing space after 'new'	0
Object instantiation uses variable name	0
Object instantiation uses variable name; intends variable access	0

Type mismatch: arithmetic performed on String; use length	0
Use of variable in context requiring a type	0
Value from raw collection must be cast	0
Variable access used as statement	0
Variable declaration: name and type in wrong order	0
Variable declaration: name missing	0
Variable declaration: variable name contains whitespace	0

Appendix C Category Hierarchy

This appendix contains the complete category hierarchy developed prior to the severity calculations (Chapter 4.2). 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)

Appendix D   Evaluation of Diagnostic Tool Prototype

This appendix contains a log of judgements made during evaluation of the diagnostic tool prototype.

For each of the five error types that can be diagnosed by the tool, ten different code snapshots corresponding to compilation events with errors which had been categorised with the error type in question were selected for purpose of evaluating the diagnostic tool prototype; these corresponded to the first such event from each of the first ten suitable recorded sessions (ordered by time). However for some error types there were less than ten (suitable) sessions, and so fewer snapshots were used for evaluation of the tool's ability to diagnose these error types.

For each selected snapshot, both the compiler diagnostic and tool diagnostic is judged as either accurate or inaccurate; if both the compiler diagnostic and tool are diagnostic are judged as accurate, then the tool diagnostic's precision is judged against the compiler diagnostic (as higher, equivalent, or lower).

Accuracy is judged according to whether the text of the diagnostic is correct, rather than whether it matches the selected categorisation for the error. Therefore a compiler diagnostic such as "cannot resolve symbol" is nearly always considered accurate, though it may be judged to have low precision compared to the tool.

The following log is divided into five sections (one for each error type that the tool was designed to diagnose). In each section, the snapshots used for evaluation are presented: first, the corresponding event number from the database is listed, a sample of the source code surrounding the error location identified by the compiler diagnostic is presented, followed by the diagnostic produced by the compiler and the prototype. The two diagnostics are separately judged as accurate or not, and if both are considered accurate, a judgement regarding the precision of the tool's diagnostic compared to that of the compiler is made and recorded.

D.1 Variable Not Declared

Event #25

This snapshot was skipped due to the code using an external library that was not available to us.

Event #75

Code sample:

```
80     {
81      if ( ptr == 0 )
82        return true;
83      return false;
84     }
85
86    // Test de pile pleine
87    public boolean estPleine()
88     {
89        assertEquals(true, p.estPleine());
90              System.out.println(this.ptr);
91              if ( this.ptr==this.zone.length )
92        return true;
93
94      return false;
95     }
96
97    // Taille de la pile
98    public int taille()
```

Compiler diagnostic: cannot find symbol - variable p

Accurate: Yes

Tool diagnostic:

```
Line 89: Call to method 'assertEquals' on a receiver of type question1.Pile,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: (N/A)

Comment: The compiler and diagnostic tool have each correctly identified a different error. A comparison of precision does not therefore make sense.

Event #180

Code sample:

```
1 public class main
2 {
3     public static void main(String[]args)
4     {
5         Eingabe text = new Eingabe();
6         ergebnis = text.eigeben("Wie heißt du?");
7     }
8 }
```

Compiler diagnostic: cannot find symbol - variable ergebnis

Accurate: yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a variable called "ergebnis", but the variable is not
declared.

The name is used on line 6 (in the "main" method).

I suggest that a declaration should be inserted
in the "main" method.
```

Accurate: yes

Precision: Higher

Comment: the tool diagnostic is more precise and informative.

Event #305

Code sample:

```
74              result = result * (i + 1);
75          }
76          return result;
77      }
78      public static void fib(int n)
79      {
80          int f1, f2 =0, f3=1;
81          for ( int i = 1; i<=n; i++)
82          {
83              f = f + g;
84              g = f - g;
85          }
86          System.out.println(n);
87      }
88  }
```

Compiler diagnostic: cannot find symbol - variable f

Accurate: Yes

Tool diagnostic:

```
There seems to be uses of a
variable called "f", but the variable is not declared.

The name is used:
- on line 84 (in the "fib" method)
- on line 83 (in the "fib" method)
- on line 83 (in the "fib" method)

I suggest that a declaration should be inserted
in the "fib" method.
```

Accurate: Yes

Precision: Higher

Comment: Again the tool's diagnostic is more precise and informative.

Event #329

Code sample:

```
29  //devuelve cierto si la pila está vacía o falso en caso contrario (empty).
30  public boolean vacia(){
31   return this.isEmpty();
32  }
33
34  public void imprime()
35  {
36      for(int i=0;i<=2;i++)
37      {
38          dato=this.get(i);
39       System.out.println();
40      }
41  }
42 }
```

Compiler diagnostic: cannot find symbol - variable dato

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a
variable called "dato", but the variable is not declared.

The name is used on line 38 (in the "imprime" method).

I suggest that a declaration should be inserted
in the "imprime" method.
```

Accurate: Yes

Precision: Higher

Event #609

Code sample:

```
24     public void addChoice(String choice, boolean correct)
25     {
26         // TODO: Add choice to choices.
27         // Ignore the "correct" parameter for now
28     }
29
30     public void display()
31     {
32         // For now, just print the choices
33         System.out.println(choices);
34     }
35 }
36
```

Compiler diagnostic: cannot find symbol - variable choices

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a
variable called "choices", but the variable is not declared.

The name is used on line 33 (in the "display" method).

I suggest that a declaration should be inserted
in the "display" method.
```

Accurate: Yes

Precision: Higher

Event #756

Code sample:

```

        JPanel pn3 = new JPanel(new BorderLayout());
        c.add(pn3, "Center");



        JPanel pn1 = new JPanel();// 运算界面

        pn1.setLayout(new GridLayout(4, 2, 1, 1));
   pn2.setLayout(new GridLayout(4, 2, 1, 1));
        pn3.add(pn1);

        // 设置按钮
        JButton b = new JButton("1");

        b = new JButton("1");// add butten 1
        b.addActionListener(this);
        pn1.add(b);
        b = new JButton("2");// add butten 2
```

Compiler diagnostic (recorded): 找不到符号 符号：变量pn2 位置：类Calculator

(reproduced): cannot find symbol – variable pn2

Accurate: Yes

Tool diagnostic: (none).

Accurate: No

Precision: -

Comment: The undeclared “pn2” is used in a receiver context, meaning that it could theoretically be either a value or type. The diagnostic tool does not attempt to resolve such symbols.

Event #818

Code sample:

```
332
333     public Dienblad getDienblad()
334     {
335         return dienblad;
336     }
337
338     public Iterator<Artikel> getIteratorArtikel()
339     {
340         getIteratorArtikel();
341         return it;
342     }
343 }
344
345
```

Compiler diagnostic: cannot find symbol - variable it

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a variable called "it", but the variable is not
declared.

The name is used on line 341 (in the "getIteratorArtikel" method).

I suggest that a declaration should be inserted
in the "getIteratorArtikel" method.
```

Accurate: Yes

Precision: Higher

Event #865

This code snapshot was not used for evaluation because it uses an external library which is not available to us.

Event #890

Code sample:

```
3   {
4       private ArrayList<Integer> I;
5       public Stats()
6       {
7           I = new ArrayList<Integer>();
8       }
9
10      public Stats(int[] nums)
11      {
12          list = new ArrayList<Integer>();
13          for(int i = 0; i < I.size(); i++)
14          {
15              list.add(nums[i]);
16          }
17          I = list;
18      }
19
20      public void insert(int num)
21      {
```

Compiler diagnostic: cannot find symbol - variable list

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be uses of a variable called "list", but the variable is not
declared.

The name is used:
- on line 17 (in the "Stats" method)
- on line 12 (in the "Stats" method)

I suggest that a declaration should be inserted
in the "Stats" method.
```

Accurate: Yes

Precision: Higher

Event #943

Code sample:

```
8  #### ## # ####### ######## ## ##### # #### ##### ## ### ###### ## ##########
9   */
10 public class Foolin
11 {
12    public static void main(String[] args){
13        Scanner scanner = new Scanner(System.in);
14        System.out.print("Enter your word ");
15        String word = scanner.nextLine();
16        System.out.println("Your word is " + word.length() + " characters");
17        for(i = word.length(); i >= 0; i--){
18            System.out.println(word);
19         }
20
21     }
22 }
```

Compiler diagnostic: cannot find symbol - variable i

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be uses of a variable called "i", but the variable is not
declared.

The name is used:
- on line 17 (in the "main" method)
- on line 17 (in the "main" method)
- on line 17 (in the "main" method)

I suggest that a declaration should be inserted
in the "main" method.
```

Accurate: Yes

Precision: Higher

Event #980

Code sample:

```
117
118     public void makeGreyscale() {
119         int width = red.length;
120         int height = red[0].length;
121
122         // Average the red channel.
123         for (int h=0;h<height;h++) {
124             for (int w=0;w<width;w++) {
125                 int average = (red[w][h] + green[w][h] + blue[w][h])/3;
126                 temp[w][h] = red[w][height-h-1];
127             }
128         }
129     }
130 }
```

Compiler diagnostic: cannot find symbol - variable temp

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a
variable called "temp", but the variable is not declared.

The name is used on line 126 (in the "makeGreyscale" method).

I suggest that a declaration should be inserted
in the "makeGreyscale" method.
```

Accurate: Yes

Precision: Higher

D.2 Variable Name Written Incorrectly

Event #2.

Code sample:

```
        File lear = new File("lear.txt");
        Scanner learScan = new Scanner(lear);
        String current;
        int wordCount = 0;
        int charCount = 0;
        while (nameScan.hasNext())
        {
            current = nameScan.next();
            charCount += current.length();
            Word Count ++;
        }
    }
    catch (FileNotFoundException ex)
    {
        System.out.println("File not found, dawg");
    }
}

public String wordCounter(File f)
```

Compiler diagnostic: **';' expected**

Accurate: No

Tool diagnostic: (none).

Accurate: No

Precision: -

Comment: This is a case of “variable name written incorrectly” that the tool is not designed to handle (the name is written with internal whitespace). Parse failure prevents “Word” and “Count” from being recognised as variable references.

Event #123

Code sample:

```java
/**
 * Constructor for objects of class Asterix
 */
public Asterix()
{
    System.out.println("Height Of Table : ");
    Scanner in = new Scanner(System.in);
    int x = in.nextInt();
    System.out.print("Width Of Table : ");
    Scanner in1 = new Scanner(System.in1);
    int y = in.nextInt();
    System.out.println();
    for(int i=1; i<=x; i++)
        {
            for(int j=1; j<=x-1; j++)
            {
                System.out.print("\t" + "*");
            }
            System.out.println("\t" + "*");
```

Compiler diagnostic: cannot find symbol - variable in1

Accurate: Yes

Tool diagnostic: (none)

Accurate: No

Precision: -

Comment: the tool does not attempt to resolve qualified variables, and so misses this case of "System.in" being written as "System.in1".

Event #270

Code sample:

```
}

public void setId_destino(int id_destino)
{
    this.id_destino=id_destino;
}

public int getId_destino()
{
return Id.destino;
}

public void setDestino(String Destino)
{
    this.destino = destino;
}

public String getDestino()
{
```

Compiler diagnostic: cannot find symbol - variable Id

Accurate: Yes

Tool diagnostic: (None)

Accurate: No

Precision: -

Comment: The tool does not attempt to resolve the qualified “Id.destino” value.

Event #295

Code sample:

```
557     int area;
558     int perimeter;
559     public void input(int l,int b)
560     {
561         length=l;
562         breadth=b;
563     }
564     public int calc_area()
565     {
566         area=l*b;
567         return area;
568     }
569     public int calc_perimeter()
570     {
571         perimeter=2*(length+breadth);
572         return perimeter;
573     }
574     public static void main(String args[])
575     {
```

Compiler diagnostic: cannot find symbol - variable l

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a
variable called "b", but the variable is not declared.

The name is used on line 566 (in the "calc_area" method).

I suggest that a declaration should be inserted
in the "calc_area" method.
```

Accurate: Yes

Precision: Equivalent.

Comment: the variable names ‘b’ and ‘breadth’ differ too much to be considered as a match by the tool’s edit-distance based heuristic.

Event #301

Code sample:

```
18      }
19
20      public void Naam(String biersoort)
21      {
22          System.out.println("De naam van de biersoort is: " + biersoort);
23      }
24
25          public   void   Prijs(double   volume,   double   statiegeld,   double
    prijsPerLiter)
26      {
27          double prijs = volume * statiegeld * prijsperLiter;
28          System.out.println("De prijs van dit vat bier bedraagt: " + prijs +
    " euro");
29      }
30  }
```

Compiler diagnostic: cannot find symbol - variable prijsperLiter

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a variable called "prijsperLiter", but the variable is
not declared.

The name is used on line 26 (in the "Prijs" method).

I suggest that a declaration should be inserted
in the "Prijs" method.
```

Accurate: Yes

Precision: Equivalent.

Comment: The tool does not recognise method parameter declarations, and so does not in this case match “prijsperLiter” with the “prijsPerLiter” paramter.

Event #370

Code sample:

```
8    /**if(dato != null){
9     this.add(dato);
10   }else{
11    System.out.println("Introduzca un dato no nulo");
12   } **/
13   Object datoAuxiliar1 = null;
14   Object datoAuxiliar2 = null;
15        for(int i=0;i<=10;i++)
16        {
17            datoAuxiliar = c1.get(i);
18        }
19        for(int i=0;i<=10;i++)
20        {
21            datoAuxiliar2 = c2.get(i);
22        }
23        for(int i=0;i<=10;i++)
24        {
25            this.add(datoAuxiliar, datoAuxiliar2);
26        }
```

Compiler diagnostic: cannot find symbol - variable datoAuxiliar

Accurate: Yes

Tool diagnostic:

`line 17: Possible misspelling of datoAuxiliar2 as datoAuxiliar`

Accurate: Yes

Precision: Higher

Comment: There are two equally likely misspellings in this sample; the tool correctly identifies one of them.

Event #420

Code sample:

```
public class PilaP1 extends ArrayList{
 public String arr[]=new String[10];


 //se añade un elemento a la pila.(push)
 public PilaP1(){}


 public void apilar(Object dato){
   for(int i=0;i<arr.legth;i++)
   {
       arr.add(dato);
   }
 }

 //se elimina el elemento frontal de la pila.(pop)
 public boolean desapilar(){
  if(size() > 0){
   //Object dato= this.get(this.size()-1);
```

Compiler diagnostic: cannot find symbol - variable legth

Accurate: yes

Tool diagnostic: (none)

Accurate: no

Precision: -

Comment: Another case of the tool not resolving qualified names.

Event #573

This event was not considered for evaluation, since it used classes from a library which were not available to us.

Event #600

Code sample:

```
12          }
13          media=(numeros[4] +numeros[5]);
14          for(i=0; i<=10; i++){
15              if(numeros[i]>mayor){
16                  mayor=numeros[i];
17              }
18          }
19          for(i=0; i<10; i++){
20              if(numeros[i]<menor){
21                  menor=numero[i];
22              }
23          }
24           System.out.println("La suma de datos es: "+ suma +" la media es:"+
   media +" el numero mayoer es: "+ mayor +" el numero es: "+ menor);
25      }
26  }
```

Compiler diagnostic: cannot find symbol - variable numero

Accurate: yes

Tool diagnostic:

`line 21: Possible misspelling of numeros as numero`

Accurate: yes

Precision: Higher

Event #744

Code sample:

```
8       double d1, d2;
9       JFrame jf = new JFrame("Calculator");
10      JLabel jl = new JLabel("Answer");
11      JTextField tf = new JTextField();
12
13      public void init() { // 实现计算器界面
14          Container c = jf.getContentPane();
15          tf.setHorizontalAlignment(JTextField.RIGHT);// 文本框
16          c.add(tf, "North");
17          tl.setHorizontalAlignment(JLabel.RIGHT);// 文本框
18          c.add(tl, "North");
19          JPanel pn3 = new JPanel(new BorderLayout());
20          c.add(pn3, "Center");
21
22
23
24          JPanel pn1 = new JPanel();// 运算界面
25
26          pn1.setLayout(new GridLayout(4, 2, 1, 1))
```

Compiler diagnostic: cannot find symbol – variable tl

(reproduced in English; recorded diagnostic: 找不到符号 符号：变量tl 位置：类Calculator)

Accurate: yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be use of a
variable called "tl", but the variable is not declared.

The name is used on line 18 (in the "init" method).

I suggest that a declaration should be inserted
in the "init" method.
```

Accurate: yes

Precision: Equivalent.

Comment: The variable name “tl” is too short to be considered by the prototype as a match for the declaration of “jl”, which is a likely candidate.

Event #902

Code sample:

```
6       {
7           I = new ArrayList<Integer>();
8       }
9
10      public Stats(int[] nums)
11      {
12          I = new ArrayList<Integer>();
13          for(int i = 0; i < I.size(); i++)
14          {
15              this.add(nums[i]);
16          }
17      }
18
19      public void insert(int num)
20      {
21          int index = 0;
22          for(int i = 0; i < I.size(); i++)
23          {
24              if(I.get(index).compareTo(num) < 0)
```

Compiler diagnostic: cannot find symbol - method add(int)

Accurate: yes

Tool diagnostic:

```
Line 15: Call to method 'add' on a receiver of type Stats,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: yes

Precision: Higher

Comment: This is arguably a miscategorisation; the variable name is not written correctly, but rather, the wrong value is used. The prototype gives a slightly more precise diagnostic by virtue of explicating the receiver type.

D.3 Incorrect Variable Declaration

Event #10

Code sample:

```
29
30     public static String lineCounter(File f)
31     {
32         Scanner Scan = new Scanner(f);
33         int lineCount = 0;
34         String currrent;
35         while(Scan.hasNextLine())
36         {
37             lineCount++;
38             current = Scan.nextLine;
39         }
40         return ("Number of Lines: " + lineCount-1);
41     }
42 }
```

Compiler diagnostic: cannot find symbol - variable current

Accurate: yes

Tool diagnostic:

```
line 37: Possible misspelling of currrent as current (or vice-versa)
```

Accurate: yes

Precision: Higher

Comment: The tool recognizes “current” and “currrent” (line 34) as likely matches and gives an appropriate diagnostic.

Event #125

Code sample:

```
/**
 * Constructor for objects of class Asterix
 */
public Asterix()
{
    System.out.println("Height Of Table : ");
    Scanner in = new Scanner(System.in);
    int x = in.nextInt();
    System.out.print("Width Of Table : ");
    Scanner in = new Scanner(System.in);
    int y = in.nextInt();
    System.out.println();
    for(int i=1; i<=x; i++)
        {
            for(int j=1; j<=x-1; j++)
            {
                System.out.print("\t" + "*");
            }
            System.out.println("\t" + "*");
```

Compiler diagnostic: variable in is already defined in constructor Asterix()

Accurate: yes

Tool diagnostic: (none given)

Accurate: no

Precision: -

Comment: Arguably a miscategorisation. In any case, re-declaration of a variable with the same name is not a form of the error category that the tool is designed to detect.

Event #206

Code sample:

```
51          sun.changeColor("yellow");
52          sun.moveHorizontal(180);
53          sun.moveVertical(-10);
54          sun.changeSize(60);
55          sun.makeVisible();
56      }
57
58      public void sunset()
59      {
60          sunny = 100;
61
62          while(sunny > 1){
63              sun.slowMoveVertical(100);
64              draw();
65              sunny--;
66          }
67
68      }
69
```

Compiler diagnostic: cannot find symbol - variable sunny

Accurate: yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be uses of a variable called "sunny", but the variable is not
declared.

The name is used:
- on line 64 (in the "sunset" method)
- on line 61 (in the "sunset" method)
- on line 59 (in the "sunset" method)

I suggest that a declaration should be inserted
in the "sunset" method.
```

Accurate: yes

Precision: Higher

Comment: tool diagnosis is clearly more precise and informative.

Event #238

Code sample:

```
7   */
8  public class Factorial
9  {
10
11     /**
12      * Constructor for objects of class Factorial
13      */
14     public void getFactorial(int number)
15     {
16         tempTerm = 1;
17         for(int i = number; i >= 1; i--)
18         {
19            tempTerm = tempTerm*i;
20         }
21         system.out.println(tempTerm);
22     }
23
24     public static void main(String[] args)
25     {
```

Compiler diagnostic: cannot find symbol - variable tempTerm

Accurate: yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be uses of a variable called "tempTerm", but the variable is not
declared.

The name is used:
- on line 19 (in the "getFactorial" method)
- on line 16 (in the "getFactorial" method)
- on line 21 (in the "getFactorial" method)
- on line 19 (in the "getFactorial" method)

I suggest that a declaration should be inserted
in the "getFactorial" method.
```

Accurate: yes

Precision: Higher

Event #442

Code sample:

```
    private Team teamName2;
    private boolean secondPlayer;
    private String choicePlayer1;
    private String choicePlayer2;

    public Play(Team team1, Team team2)
    {
        player1 = team1.getPlayer();
        player2 = team2.getPlayer();
        teamName1 = team1.getTeamName();
        teamName2 = team2.getTeamName();
        secondPlayer = false;
    }

    /**
     *
     */
    private void figure(String choice,String teamColor,boolean whichPlayer){
        int xPos;
```

Compiler diagnostic: incompatible types

Accurate: yes

Tool diagnosis: (none)

Accurate: no

Precision: -

Comment: categorisation notes explain that teamName1 and teamName2 should be defined as String; their declarations are incorrect. The tool is not designed to detect the resulting error.

Notes: *The variable type for teamName1 and teamName2 should probably be 'String'. This would match the right-hand-side of the assigment. Also 'Name' within the variable name suggests a string.*

Event #468

Code sample:

```
/**
 * ##### ###### ######### ### #################.
 */
public class Pruefungsergebnis {
  private String student;
  private double note;
  private boolean pruefungBestanden;
  private HashMap <String,double,String,boolean> studentMitNote;

  /**
   * Erzeugt ein Pruefungsergebnis.
   *
   * @param student Name des Studierenden.
   * @param note Note des Studierenden.
   */
  public Pruefungsergebnis(String student, double note) {
    this.student = student;
    this.note = note;
    istPruefungBestanden(note);
```

Compiler diagnostic: cannot find symbol - class HashMap

Accurate: yes

Tool diagnostic: (none)

Accurate: no

Precision: -

Comment: the HashMap class is not imported, which results in the “cannot find symbol” diagnostic from the compiler. Also, too many type parameters are required (HashMap needs two and there are four given). The tool is not designed to diagnose either form of error.

Event #628

Code sample:

```
import java.util.*;
public class Grades{
    private double myGPA;
    public Grades(){
    }
    public void input()
    {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter a letter grade: ");
        char grade = in.nextLine();
        grade.toLowerCase();
         while ( grade.equals("a") || grade.equals("b") || grade.equals("c") || grade.equals("d") || grade.equals("f"))
        {
            System.out.print("Enter a letter grade: ");
            grade = in.nextLine();
            grade.toLowerCase();
            int letter = grade;
        }
    }
```

Compiler diagnostic: incompatible types

Accurate: yes

Tool diagnostic: (none)

Accurate: no

Precision: -

Comment: The type in the declaration is incorrect; the tool is not designed to diagnose this form of error.

Event #822

Code sample:

```
    }

    public Dienblad getDienblad()
    {
        return dienblad;
    }

    public Iterator<Artikel> getIteratorArtikel()
    {
        Artikel it = getIteratorArtikel();
        return it;
    }
}
```

Compiler diagnostic: incompatible types

Accurate: yes

Tool diagnostic: (none)

Precision: -

Comment: similar to case in #628

Event #953

Code sample:

```
1  import java.io.*;
2  class kaprekar
3  {
4      public static void main(String args[])throws IOException
5      {
6          InputStreamReader read=new InputStreamReader(System.in);
7          BufferedReader br=new BufferedReader(read);
8          System.out.println("Enter a number");
9          int n=Integer.parseInt(br.readLine());
10         int a=n*n,c=0,int e,f;
11         for(int i=a;i>0;i=i/10)
12         {
13             int d=i%10;
14             c=c+1;
15         }
16         if(c%2==0)
17         {
18             int e=c%((int)(Math.pow(10,c/2)));
19             int f=c/((int)(Math.pow(10,c/2)));
```

Compiler diagnostic: <identifier> expected

Accurate: yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be uses of a
variable called "e", but the variable is not declared.

The name is used:
- on line 10 (in the "main" method)
- on line 26 (in the "main" method)

I suggest that a declaration should be inserted
in the "main" method.
```

Accurate: yes

Precision: Lower

Comment: syntax error means that declaration of 'e' is invalid, therefore the tool diagnostic is strictly accurate. However, it does not give a precise description of the error.

Event #1106

Code sample:

```
140         }
141     }
142
143 public String DvdList()
144     {
145         String DvdLis = "";
146         if(isEmpty())
147         {
148
149                 dvd = "The System is empty nothing to display";
150
151         }
152
153         else
154         {
155
                        dvd = "Dvd Id: " + list[i].getDvdId() + "Dvd Title: " +
    list[i].getDvdTitle() + "Dvd Age Class: " + list[i].getAgeClass() + "Dvd
156 Category: " + list[i].getCategory() + "Dvd Rating: " + list[i].getRating()
    + " Dvd Id: " + list[i].getnumMins() + "Dvd Length of Time in Shop: " +
    getLengthOfTime();
157
158         }
```

Compiler diagnostic: cannot find symbol - variable dvd

Accurate: Yes

Tool diagnostic:

```
Reference to undeclared variable:
There seems to be uses of a variable called "dvd", but the variable is not
declared.

The name is used:
- on line 159 (in the "DvdList" method)
- on line 156 (in the "DvdList" method)
- on line 149 (in the "DvdList" method)

I suggest that a declaration should be inserted
in the "DvdList" method.
```

Accurate: Yes

Precision: Higher

Comment: The tool does not match the declaration of “DvdLis” with its various apparent uses as “dvd”, however it still gives a diagnostic that is more precise and information than the compiler.

D.4 Method call targeting wrong type

Event #46

This event will not be used for evaluation because it uses an external library to which we do not have access.

Event #372

Code sample:

```
16        {
17            datoAuxiliar1 = c1.get(i);
18        }
19        for(int i=0;i<=10;i++)
20        {
21            datoAuxiliar2 = c2.get(i);
22        }
23        for(int i=0;i<=10;i++)
24        {
25            this.add(datoAuxiliar1.concat(datoAuxiliar2));
26        }
27
28  }
29
30   public Object frente(){
31   Object datoAuxiliar = null;
32   if(this.size() > 0){
33    datoAuxiliar = this.get(0);
34   }
```

Compiler diagnostic: cannot find symbol - method concat(java.lang.Object)

Accurate: Yes

Tool diagnostic:

```
Line 25: Call to method 'concat' on a receiver of type java.lang.Object,
which does not have a method with that name:
A method with that name is declared in the following types: java.lang.String
Perhaps the method should be called on a different receiver, or the receiver's
type should be declared to be one of the above types.
```

Accurate: Yes

Precision: Higher

Event #537

Code sample:

```
31           /**
              * affichage d'un message dans la zone de texte ce message est de la
32 forme
              * observateur this.nom : clic du bouton nom_du_bouton exemple :
33 observateur
34            * jbo1 : clic du bouton A, voir la méthode getActionCommand()
35            *
36            * @param à
37            *             compléter
38            */
39           public void actionPerformed(ActionEvent e) {
                     String message = "observateur " +
40 e.getSource().getText(); // à compléter, inspirez-vous de l'applette de
   l'énoncé
41                   contenu.append(message + "\n");
42           }
43
44 }
45
```

Compiler diagnostic: cannot find symbol - method getText()

Accurate: Yes

Tool diagnostic:

```
Line 40: Call to method 'getText' on a receiver of type java.lang.Object,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: Higher

Comment: prototype tool does not categorise as “method call on wrong target type”, but the message is accurate and more precise than that of the compiler

Event #1173

Code sample:

```
208         if(newDvd == null)
209         {
210 
211             System.out.println("No such Dvd id exists");
212 
213         }
214         else
215         {
216 
217             System.out.println("Dvd Id: " + getDvdId());
218             System.out.println("Dvd Name: " + getDvdTitle());
219             System.out.println("Dvd Age Clasification: " + getAgeClass());
220             System.out.println("Dvd Category: " + getCategory());
221             System.out.println("Dvd Rating: " + getRating());
222             System.out.println("Dvd Run Time: " + getNumMins());
223             System.out.println();
224         }
225     }
226 
```

Compiler diagnostic: cannot find symbol - method getDvdId()

    Accurate: Yes

Tool diagnostic:

```
Line 217: Call to method 'getDvdId' on a receiver of type DvdTester,
which does not have a method with that name:
A method with that name is declared in the following types: Dvd, DVD
Perhaps the method should be called on a different receiver, or the receiver's
type should be declared to be one of the above types.
```

    Accurate: Yes

Precision: Higher

Event #1505

Code sample:

```
12      JLabel results;
13      public DeleteRecordPanel(){
14         dSearch = new JTextField(4);
15         results = new JLabel(".....");
16         dSearch.addActionListener(new TempListener());
17         add(dSearch);
18      }
19      private class TempListener implements ActionListener{
20          public void actionPerformed (ActionEvent event){
21              if ((deleteRecord(dSearch.getText()))==1){
22                  results.setText("Records Deleted!");
23              }else{
24                  results.setText("No Records Found!");
25              }
26          }
27      }
28  }
```

Compiler diagnostic: cannot find symbol - method deleteRecord(java.lang.String)

Accurate: yes

Tool diagnostic:

```
Line 21: Call to method 'deleteRecord' on a receiver of type
DeleteRecordPanel.TempListener,
which does not have a method with that name:
A method with that name is declared in the following types: OrgCRUD,
AbstractCRUD
Perhaps the method should be called on a different receiver, or the receiver's
type should be declared to be one of the above types.
```

Accurate: yes

Precision: Higher

Event #1690

Code sample:

```
27          return first + " " + middle + " " + last;
28      }
29      public String lastFirstMiddle()
30      {
31          return last + ", " + first + " " + middle;
32      }
33      public boolean equals(Name objectname)
34      {
35          String object = objectname.firstMiddleLast();
36          if(this.equalsIgnoreCase(object))
37          {
38              return true;
39          }
40          else
41          {
42              return false;
43          }
44      }
45      public String initials()
```

Compiler diagnostic: cannot find symbol - method equalsIgnoreCase(java.lang.String)

Accurate: Yes

Tool diagnostic:

```
Line 36: Call to method 'equalsIgnoreCase' on a receiver of type Name,
which does not have a method with that name:
A method with that name is declared in the following types: java.lang.String
Perhaps the method should be called on a different receiver, or the receiver's
type should be declared to be one of the above types.
```

Accurate: Yes

Precision: Higher

Event #1721

Code sample:

```
13              int side2 = input.nextInt();
14      //Ask user for hypotenuse and store it
15          System.out.println("Input hypotenuse :");
16              double hypot = input.nextDouble();
17      //Check if it is a right triangle
18      if(side1 * side1 + side2 * side2 == hypot * hypot){
19              System.out.println("This is a right triangle!");
20          //Find the angles of this triangle
21              System.out.println("The first angle is 90 degrees");
22               System.out.println("The second angle is " + tan(side1/side2) +
   " degrees!");
23               System.out.println("The third angle is " + tan(side2/side1) + "
   degrees!");
24
25      }  else{
26          System.out.println ("Not a right triangle");
27          System.out.print ("End of Program.");
28      }
29      }
30  }
```

Compiler diagnostic: cannot find symbol - method tan(int)

Accurate: Yes

Tool diagnostic:

```
Line 22: Call to method 'tan' on a receiver of type Triangle,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: Higher

Comment: The prototype tool does not search java.lang.Math for matching method declarations and so does not the “tan” method declared there. However, it provides a more precise and informative diagnostic than the compiler.

Event #2075

Code sample:

```
12         gallery.add(new Picture("monet1.jpg"));
13         gallery.add(new Picture("monet2.jpg"));
14         gallery.add(new Picture("renoir1.jpg"));
15 
16         for (Picture pic : gallery)
17         {
18             // TODO: Move the first picture to offset 10,
19             // the second picture ten pixels to the right of the first one,
20             // and so on
21             pic.translate(getMaxX()+10,0);
22         }
23 
24         for (Picture pic : gallery)
25         {
26             pic.draw();
27         }
28     }
29 }
```

Compiler diagnostic: cannot find symbol - method getMaxX()

Accurate: Yes

Tool diagnostic:

```
Line 21: Call to method 'getMaxX' on a receiver of type ListOfPictures,
which does not have a method with that name:
A method with that name is declared in the following types: Picture
Perhaps the method should be called on a different receiver, or the receiver's
type should be declared to be one of the above types.
```

Accurate: Yes

Precision: Higher

Event #2122

This event is not used for evaluation as the code appears to use an external library which is not available to us.

Event #2203

Code sample:

```
234                 case "7":
235                             clearScreen();
236                             luckyDipMachine.addNewPrize();
237                             System.out.print("\nPlease press enter to continue!");
238                             userInput.nextLine();
239                             break;
240
241                 case "8":
242                             clearScreen();
243                             writePrizeFile();
244                             System.out.println("Thank you for playing!!");
245                             System.out.println("\nSee you next time!!");
246                             break;
247
248                 default:
249                             System.out.println("\nInvalid option selected, \nplease select an option between 1 and 8!");
250                             System.out.print("\nPlease press enter to continue!");
251                             userInput.nextLine();
252                             break;
```

Compiler diagnostic: cannot find symbol - method writePrizeFile()

Accurate: Yes

Tool diagnostic:

```
Line 243: Call to method 'writePrizeFile' on a receiver of type Game,
which does not have a method with that name:
A method with that name is declared in the following types: LuckyDipMachine
Perhaps the method should be called on a different receiver, or the receiver's
type should be declared to be one of the above types.
```

Accurate: Yes

Precision: Higher

Event #2818

Code sample:

```
    {
        BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("Enter a string");
        String n = br.readLine();
        int l = n.length();
        String g = "";
        for(int i = 0;i<l;i++)
        {
            char ch = n.charAt(i);
            if(ch.isUpperCase.equals(true))
            {
                ch = ch.toLowerCase();
            }
            else if(ch.isLowerCase.equals(true))
            {
                ch = ch.toUpperCase();

            }
            g  = g+ch;
```

Compiler diagnostic: char cannot be dereferenced

Accurate: Yes

Tool diagnostic: (none)

Precision: -

Comment: this is a miscategorisation; the recorded categorisation comment indicates that the '==' operator should be used instead of a method call in this instance.

Note: there are no more sessions with the "method call targeting wrong type" categorisation.

D.5 Method not declared

Event #729

This event snapshot is unsuitable for evaluation, as the code appears to refer to an external library that is not available to us.

Event #1004

Code sample:

```
108     else{
109
110         return list[i];
111
112     }
113 }
114
115 public String upDatelengthOfTime()
116     {
117         Dvd.setLengthOfTime(1);
118
119         return "DVD information updated";
120
121     }
122
123
```

Compiler diagnostic: cannot find symbol - method setLengthOfTime(int)

Accurate: Yes

Tool diagnostic:

```
Line 117: Call to method 'setLengthOfTime' on a receiver of type Dvd,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: Equivalent

Event #1435

Code sample:

```
20       * @param  y    ein Beispielparameter für eine Methode
21       * @return        die Summe aus x und y
22       */
23      private int einloggen()
24      {
25          int y = -1;
26          int ergebnis = 0;
27          for(int x = 0; x > y; ++x)
28          {
29              ergebnis += eimloggen();
30          }
31          return ergebis;
32      }
33  }
```

Compiler diagnostic: cannot find symbol - method eimloggen()

Accurate: Yes

Tool diagnostic:

```
Line 29: Call to method 'eimloggen' on a receiver of type Passwort,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: Equivalent

Event #1517

Code sample:

```
12     JLabel results;
13     public DeleteRecordPanel(){
14        dSearch = new JTextField(4);
15        results = new JLabel(".....");
16        dSearch.addActionListener(new TempListener());
17        add(dSearch);
18     }
19     private class TempListener implements ActionListener{
20         public void actionPerformed (ActionEvent event){
21             interdelete(dSearch.getText());
22         }
23     }
24
```

Compiler diagnostic: cannot find symbol - method interdelete(java.lang.String)

Accurate: Yes

Tool diagnostic:

```
Line 21: Call to method 'interdelete' on a receiver of type
DeleteRecordPanel.TempListener,
which does not have a method with that name:
That method does not seem to be declared.\
```

Accurate: Yes

Precision: Equivalent

Event #1863

Code sample:

```
58              switch(zufall-20)
59              {
                    case  '0': mKarte4= new Karte(5,"Herz"); mKarte1 = new
60 Karte(5,"Kreuz"); mKarte3 = new Karte(9,"Herz"); mKarte2 = new
   Karte(9,"Kreuz"); break;
                    case  '1': mKarte4= new Karte(5,"Herz"); mKarte2 = new
61 Karte(5,"Kreuz"); mKarte1 = new Karte(9,"Herz"); mKarte3 = new
   Karte(9,"Kreuz"); break;
                    case  '2': mKarte4= new Karte(5,"Herz"); mKarte2 = new
62 Karte(5,"Kreuz"); mKarte3 = new Karte(9,"Herz"); mKarte1 = new
   Karte(9,"Kreuz"); break;
                    case  '3': mKarte4= new Karte(5,"Herz"); mKarte3 = new
63 Karte(5,"Kreuz"); mKarte1 = new Karte(9,"Herz"); mKarte2 = new
   Karte(9,"Kreuz"); break;
                    case  '4': mKarte4= new Karte(5,"Herz"); mKarte3 = new
64 Karte(5,"Kreuz"); mKarte2 = new Karte(9,"Herz"); mKarte1 = new
   Karte(9,"Kreuz"); break;
65              }
66          }
67          mKarte4.getZeichen();
            // return mKarte1(int pZahl, String pMuster); mKarte2(int pZahl,
68 String pMuster); mKarte3(int pZahl, String pMuster); mKarte4(int pZahl,
   String pMuster);
            // mKarte1 = new Karte(5,"Herz");    mKarte2 = new Karte(5,"Kreuz");
69
   mKarte3 = new Karte(9,"Herz");    mKarte4 = new Karte(9,"Kreuz");
70      }
71      //public AufnahmeStapel(byte pAnzahl)
72      {
73        //  zAnzahl = pAnzahl;
74      }
75      public void mischeStapel()
76      {
```

Compiler diagnosis: cannot find symbol - method getZeichen()

Accurate: Yes

Tool diagnosis:

```
Line 67: Call to method 'getZeichen' on a receiver of type Karte,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: Equivalent

Event #1990

Code sample:

```
72                          break;
73                 case 4 : System.out.print ("Sorting Records");
74                          break;
75                 case 5 : System.out.println ("Searching Records");
76                          break;
77
78                 case 6 : System.out.println ("Displaying Records");
79                          System.out.println ("------ --   ----");
80                          System.out.println ();
81                          DisplayDetails(Record);
82                          break;
83
84             }
85         }
86     }
87
```

Compile diagnosis: cannot find symbol - method DisplayDetails(java.util.ArrayList<SetGet>)

Accurate: Yes

Tool diagnosis:

```
Line 81: Call to method 'DisplayDetails' on a receiver of type MainMenu,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: Equivalent

Event #2543

Code sample:

```
64
65
66
67
68    //no work required below this point
69
70    //just assigned the partner for the current lab
71    public void setPartnerID(int partnerID)
72    {
73       int labID = getNumLabs();
74       labs.setPartnerID(labID, partnerID);
75       hasPartner = true;
76    }
77
78    public char computeFinalGrade(double[] grade_constants)  //the constants
   to achieve each letter grade
79    {
80       if (!active)
81       {
82          return 'W';  //student has withdrawn from the class
```

Compiler diagnostic: cannot find symbol - method getNumLabs()

Accurate: Yes

Tool diagnostic:

```
Line 73: Call to method 'getNumLabs' on a receiver of type Student,
which does not have a method with that name:
That method does not seem to be declared.
```

Accurate: Yes

Precision: Equivalent

Event #20392

This event's code snapshot was not suitable for analysis as the code appeared to use an external library which was not available to us.

Appendix E Publications

Brown, N. C. C., Kölling, M., McCall, D., & Utting, I. (2014, March). Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 223-228). ACM.

McCall, D., & Kölling, M. (2014, October). Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings* (pp. 1-8). IEEE.

Submitted for publication and awaiting response:

McCall, D., & Kölling, M. (n.d.). Frequency is not Severity – A New Look at Novice Programmer Errors. Submitted to ACM Transactions on Computing Education (*TOCE).*