# An Exploratory Study of Novice Programming Experiences and Errors

by

Suzanne Marie Thompson
B.Sc., University of Victoria, 2004

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

An Exploratory Study of Novice Programming Experiences and Errors

by

Suzanne Marie Thompson
B.Sc., University of Victoria, 2004

**Supervisory Committee**

Dr. Margaret-Anne Storey, (Department of Computer Science)
Supervisor

Dr. Daniel German, (Department of Computer Science)
Departmental Member

Dr. Allyson Hadwin, (Department of Education Psychology and Leadership Studies)
Outside Member

**Supervisory Committee**

Dr. Margaret-Anne Storey, (Department of Computer Science)
Supervisor

Dr. Daniel German, (Department of Computer Science)
Departmental Member

Dr. Allyson Hadwin, (Department of Education Psychology and Leadership Studies)
Outside Member

## ABSTRACT

Learning how to program is a difficult task: students must learn programming concepts, a language's syntax, and a software environment that will assist their programming activities. In this work we attempt to learn more about novice programming errors so that we can provide better tool support and information for instructors. We discuss our study where, with a software monitor, we tracked students' errors and their usage of the Gild integrated development environment. Based on our log file data, we describe student interactions with Gild: their first use, the features they use, and how they used them. It was found that a small number of error types accounted for the majority of errors made. Although feedback to Gild's extra error help feature was largely positive, improvements to this feature are required as it was found that the errors that students take the longest to fix are also less frequently made.

# Table of Contents

**Supervisory Committee** ii

**Abstract** iii

**Table of Contents** iv

**List of Tables** x

**List of Figures** xii

**Acknowledgements** xiii

**Dedication** xiv

**1 Introduction** 1

1.1 The Difficulties of Learning How to Program . . . . . 1

1.2 Why We Need to Know the Common Novice Errors . . . . . 3

1.3 Research Questions . . . . . 3

1.4 Thesis Outline . . . . . 4

**2 Common Novice Errors and Misconceptions** 5

2.1 Java . . . . . 6

2.1.1 Syntax Errors . . . . . 6

2.1.2 Misconceptions . . . . . 10

2.2 Novice Errors in Other Languages . . . . . 11

2.2.1 Pascal . . . . . 11

2.2.2 Algorithms and Data Structures . . . . . 12
2.2.3 Eiffel . . . . . 13
2.2.4 Blue . . . . . 13
2.2.5 C . . . . . 14
2.3 Summary . . . . . 15

**3 Gild and Other Software for Novice Programmers 16**
3.1 Requirements for a Novice Development Environment . . . . . 17
3.2 Gild . . . . . 18
3.2.1 Eclipse . . . . . 18
3.2.2 Features . . . . . 19
3.2.2.1 Import and Export Archived Project . . . . . 19
3.2.2.2 Simplified Java Editor . . . . . 21
3.2.2.3 Separated Save/Compile/Run process . . . . . 21
3.2.2.4 Simplified Debugger . . . . . 21
3.2.2.5 Gild Web Browser . . . . . 22
3.2.2.6 Problem View with Extra Error Support . . . . . 23
3.3 Other Development Environments for Novices . . . . . 25
3.3.1 BlueJ . . . . . 25
3.3.2 DrJava . . . . . 27
3.3.3 Penumbra . . . . . 27
3.3.4 Kenya and KenyaEclipse . . . . . 28
3.4 Java Compiler Enhancements and Similar Programs . . . . . 28
3.4.1 Gauntlet . . . . . 29
3.4.2 University of St. Andrews and University of Abertay Dundee . . . 29
3.4.3 Expresso . . . . . 30
3.4.4 University of the West of England Tool Set . . . . . 30
3.5 Intelligent Tutoring Systems . . . . . 30

3.6 Summary . . . . . 31

**4 Methodology** **33**

4.1 Data Collection Technique . . . . . 33

4.2 Mylar Monitor . . . . . 34

4.2.1 Changes to the Mylar Monitor . . . . . 36

4.3 Participants . . . . . 38

4.4 Procedure . . . . . 38

4.5 Research Questions and how the Data was Used . . . . . 40

4.5.1 What are the Most Common Errors (Compiler and Runtime) that Students Make? . . . . . 40

4.5.2 How Long does it Take for Students to Fix Their Errors? . . . . . 42

4.5.3 Do Students' Errors Change over the Course of the Semester? . . . 42

4.5.4 Where do Students go for Extra Help for an Error they do not Understand? . . . . . 42

4.5.5 Does the Extra Error Help Feature Help Students? . . . . . 42

4.5.6 What further insights about novice programmers and Gild can we gain through the analysis of the log file data? . . . . . 43

4.6 Limitations . . . . . 43

4.7 Summary . . . . . 44

**5 Qualitative Results** **45**

5.1 Participants . . . . . 45

5.2 Course structure . . . . . 47

5.3 Participant Summaries . . . . . 47

5.3.1 Participant 68 . . . . . 49

5.3.1.1 Work Habits . . . . . 50

5.3.1.2 Extra Error Help . . . . . 50

5.3.2 Participant 85 . . . . . 51

5.3.2.1 Work Habits . . . . . 52
5.3.2.2 Extra Error Help . . . . . 52
5.3.3 Participant 102 . . . . . 52
5.3.3.1 Work Habits . . . . . 54
5.3.3.2 Extra Error Help . . . . . 54
5.3.4 Participant 119 . . . . . 55
5.3.4.1 Work Habits . . . . . 56
5.3.4.2 Extra Error Help . . . . . 56
5.3.5 Participant 136 . . . . . 56
5.3.5.1 Work Habits . . . . . 58
5.3.5.2 Extra Error Help . . . . . 58
5.3.6 Participant 153 . . . . . 58
5.3.6.1 Work Habits . . . . . 60
5.3.6.2 Extra Error Help . . . . . 60
5.3.7 Participant 170 . . . . . 60
5.3.7.1 Work Habits . . . . . 62
5.3.7.2 Extra Error Help . . . . . 62
5.3.8 Participant 204 . . . . . 62
5.3.8.1 Work Habits . . . . . 64
5.3.8.2 Extra Error Help . . . . . 64
5.3.9 Participant 211 . . . . . 64
5.3.10 Participant 238 . . . . . 64
5.3.10.1 Work Habits . . . . . 66
5.3.10.2 Extra Error Help . . . . . 66
5.4 Summary . . . . . 66

**6 Quantitative Results** **68**

6.1 What are the Most Common Errors that Students Make? . . . . . 68

6.1.1 Compilation Errors . . . . . 68
6.1.2 Runtime Exceptions . . . . . 75
6.2 How Long Does it Take to Fix an Error? . . . . . 79
6.3 Do the Errors Change over Time? . . . . . 85

**7 Synthesis of Results and Discussion** **90**
7.1 What are the Most Common Errors that Students Make? . . . . . 90
7.1.1 Compilation Errors . . . . . 90
7.1.2 Runtime Exceptions . . . . . 92
7.2 How Long Does it Take for Students to Fix their Errors? . . . . . 93
7.3 Do Students' Errors Change over the Course of the Semester? . . . . . 94
7.4 Where do Students go for Extra Help for an Error they do not Understand? 94
7.5 Does Gild's Extra Error Help Feature Aid Students? . . . . . 97
7.5.1 Match between Gild Extra Help and Errors Made by Students . . . 98

**8 Future Work and Conclusions** **99**
8.1 Future Work . . . . . 99
8.1.1 Tool Improvements . . . . . 99
8.1.2 Future Studies and Study Improvements . . . . . 100
8.2 Conclusions . . . . . 101
8.2.1 Contributions . . . . . 101

**Bibliography** **103**

**Appendix A Gild Extra Help** **108**
A.1 Fifty-one Supplemental Error Messages . . . . . 108

**Appendix B Gild 2006 Questionnaire** **122**

**Appendix C Gild 2006 Survey Responses** **125**
C.1 Student Responses: Where do Students go for Extra Help? . . . . . 125

C.2 Student Responses: Was the Extra Error Help Feature Useful? . . . . . 131

**Appendix D Compiler Errors for all Participants** **134**

# List of Tables


5.1 Participants in the Gild logging study . . . . . 46
5.2 Participant log file overview . . . . . 46
5.3 Important course dates for CSC 115, spring 2006 . . . . . 47
5.4 Assignment and lab assignment overview . . . . . 48

6.1 Error information for all participants . . . . . 69
6.2 Ten most frequent errors and their mean fix times for all participants . . . . 69
6.3 Participant 68 - Most frequent errors and their mean fix times . . . . . 70
6.4 Participant 85 - Most frequent errors and their mean fix times . . . . . 70
6.5 Participant 102 - Most frequent errors and their mean fix times . . . . . 71
6.6 Participant 119 - Most frequent errors and their mean fix times . . . . . 71
6.7 Participant 136 - Most frequent errors and their mean fix times . . . . . 72
6.8 Participant 153 - Most frequent errors and their mean fix times . . . . . 72
6.9 Participant 170 - Most frequent errors and their mean fix times . . . . . 73
6.10 Participant 204 - Most frequent errors and their mean fix times . . . . . 73
6.11 Participant 238 - Most frequent errors and their mean fix times . . . . . 74
6.12 Runtime exceptions for all participants . . . . . 75
6.13 Participant 68 - Runtime exceptions . . . . . 75
6.14 Participant 85 - Runtime exceptions . . . . . 76
6.15 Participant 102 - Runtime exceptions . . . . . 76
6.16 Participant 119 - Runtime exceptions . . . . . 77
6.17 Participant 136 - Runtime exceptions . . . . . 77
6.18 Participant 153 - Runtime exceptions . . . . . 77

6.19 Participant 170 - Runtime exceptions . . . . . 78
6.20 Participant 204 - Runtime exceptions . . . . . 78
6.21 Participant 238 - Runtime exceptions . . . . . 78
6.22 Errors with the longest mean fix times, for all participants . . . . . 79
6.23 Participant 68 - Longest mean fix times for compiler errors . . . . . 80
6.24 Participant 85 - Longest mean fix times for compiler errors . . . . . 80
6.25 Participant 102 - Longest mean fix times for compiler errors . . . . . 81
6.26 Participant 119 - Longest mean fix times for compiler errors . . . . . 81
6.27 Participant 136 - Longest mean fix times for compiler errors . . . . . 82
6.28 Participant 153 - Longest mean fix times for compiler errors . . . . . 82
6.29 Participant 170 - Longest mean fix times for compiler errors . . . . . 83
6.30 Participant 204 - Longest mean fix times for compiler errors . . . . . 83
6.31 Participant 238 - Longest mean fix times for compiler errors . . . . . 84
6.32 Participant 85 - Changes in errors between assignments . . . . . 85
6.33 Participant 102 - Changes in errors between assignments . . . . . 86
6.34 Participant 170 - Changes in errors between assignments . . . . . 88

C.1 Responses to "What do you do when you encounter an error that you know nothing about? " . . . . . 126
C.2 Responses to "Was the extra error help feature useful?" . . . . . 131

D.1 Compiler errors and mean fix times for all participants . . . . . 134

# List of Figures


3.1 An overview of the Gild IDE . . . . . 20
3.2 Gild debugger . . . . . 23
3.3 Gild extra help . . . . . 24
3.4 Extra error support available in BlueJ . . . . . 26

6.1 Participant 85 - Top 50% of errors per assignment (percentage). . . . . 85
6.2 Participant 85 - Top 50% of errors per assignment (count). . . . . 86
6.3 Participant 102 - Top 50% of errors per assignment (percentage). . . . . 87
6.4 Participant 102 - Top 50% of errors per assignment (count). . . . . 87
6.5 Participant 170 - Top 50% of errors per assignment (percentage). . . . . 88
6.6 Participant 170 - Top 50% of errors per assignment (count). . . . . 89

# *Acknowledgements*

I would like to thank my supervisor, Dr. Margaret-Anne Storey, for all of her support, enthusiasm, and feedback. I would also like to thank the other members of my committee, Dr. Daniel German and Dr. Allyson Hadwin, for their time and input.

Additional thanks goes to the members of the CHISEL group, and to the GILD research group. Their assistance and feedback have been valuable to my research. It has been a pleasure to work with you all. Thank you also to the members of the Learning Kit/gStudy project [1] for allowing me to modify the Log Analyzer software for my study. I would also like to thank Laura Young and Wendy Wiggins for their help with the data presentation, and editing of this thesis, respectively.

# *Dedication*

For my family and Bjørn.

Thank you for your support.

# Chapter 1

# Introduction

> "Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution." - Jakob Nielsen, 1994 [2]
>
> "better error messages... error on string tokenizer :/#%@^ expected, what the hell does that mean, honestly" - CSC 115 student, Gild questionnaire, February 2004

## 1.1 The Difficulties of Learning How to Program

In 2001, a multi-national, multi-institutional study was carried out by an ITiCSE working group (the "McCracken group") to answer the following question: "Do students in introductory computing courses know how to program at the expected skill level?" [3]. There were 216 participants from universities in four countries. In the study, students had to complete a short programming assignment that the authors felt was in keeping with the expected programming skill of an introductory programming student; the assignment could be completed in a language of the student's choice, which ended up being either C++ or Java. Assignments were evaluated on a predefined criteria that included: correct execution without compilation or run time errors, the correctness given a test case, the correct problem solved, the programming style, and the degree of closeness to the correct solution. The results of the study were greatly disappointing due to the low level of student achievement:

the average score was 22.89 of 100 points. It was found that of the student solutions did not even compile, many of the problems were due to syntax errors. The fact that there were so many syntactically incorrect programs was worrisome. The authors found that the students had the greatest difficulty with "abstracting the problem to be solved from the exercise description" [3]. If a student could not submit a syntactically correct program, what are the chances that they were able to effectively use their programming knowledge to abstract the given problem?

As a follow-up to the 2001 ITiCSE working group's study, an ITiCSE 2004 working group was formed to address the question: "to what degree did students perform poorly in the McCracken study because of poor problem solving skills, or because of fragile knowledge and skills that are a precursor to problem-solving?" [4]. The participants in this study were from multiple institutions, in multiple countries. The students, most of whom had just completed their first semester of Java or C++ programming, completed a 12 question multiple choice survey in an exam condition. From the results of this study the authors concluded that many novices were indeed lacking in the programming knowledge and skills that precede problem solving. They believed that these problems were more associated with reading code, as the students had difficulty with comprehending the short pieces of code that were given in the exam. Their conclusions were based on the premise that all other programming tasks are based on comprehension. The findings of the 2004 ITiCSE working group agree with some of the findings from Widenbeck *et al.* [5]. Widenbeck *et al.* found that novice programmers (first year programming students) lacked the ability to connect the high-level program goals to the associated program code segments.

The results of the ITiCSE working groups, as well as the work discussed in the following chapter, show that the problems introductory programming students are encountering are not limited to a single teaching institution, a teaching methodology or style, or even a particular language. Further to this, novices' problems cannot necessarily be blamed on undeveloped problem solving skills. By helping novice programmers to better develop their knowledge of syntax, we should be able to shift their focus to the more abstract problem

solving and pattern recognition skills that are needed to become a good programmer.

## 1.2 Why We Need to Know the Common Novice Errors

When a student programmer makes an error, it may be due to many reasons, from carelessness to genuine misunderstandings about a programming concept. While a simple mistype is easily corrected, the more fundamental programming errors can result in a student spending hours in frustration trying to fix it. By knowing what errors novice programmers (typically first year students) are making, we can improve tool support to help novices better understand programming. We can also then develop better tools for assisting with the grading of student assignments [6].

There is a discrepancy between what instructors think are the most common errors, and what actually are the most common errors. In a study by Jackson *et al.* [7], it was found that of the top nine compilation errors reported by instructors, only five of the errors were present in the list of top 10 errors made by students. The most common errors are only part of the information that we need to know: additional studies are required to determine what errors students are struggling with the most. The time it takes for students to fix a particular error type can better indicate these more difficult errors. By determining not only the most common novice errors, but also the average time to fix an error, we can provide educators with information on the problems their students may encounter, and their lessons can address these programming concepts.

## 1.3 Research Questions

The research questions I ask in this study are intended to provide a better understanding of not only the most common errors that students are making, but also what errors they are struggling with the most, as evidenced by the fix times for the errors. Changes in student errors over a semester offer further insight to the errors that students make as they

gain experience with programming. Additional questions, intended to provide information for improvements to Gild, focus on where students go for help when they encounter an unknown error, and their opinion of Gild's extra error help feature. I have six primary research questions that I want to address:

- What are the most common errors (compiler and runtime) that students make?
- How long does it take for students to fix their errors?
- Do students' errors change over the course of the semester?
- Where do students go for extra help for an error they do not understand?
- Does Gild's extra error help feature aid students?
- What further insights about novice programmers and Gild can we gain through the analysis of the log file data?

## 1.4 Thesis Outline

In this work I first present some background information on what is known about common novice programming errors. In Chapter 3, the features of the Gild integrated development environment, and other development environments for novice Java programmers, are presented. In Chapter 4 the methodology for my logging study of novice compilation and run time errors is discussed, and from this study the user stories and most common errors are presented in Chapters 5 and 6, respectively. In Chapter 7 I discuss the results of the study. This is followed by future work to improve tool support for novice programmers, and a discussion of my conclusions and contributions.

# Chapter 2

# Common Novice Errors and Misconceptions

There have been a number of studies that have looked at novice programming errors. First, I discuss research on common novice errors in the Java programming language, followed by errors encountered in other languages. There are several types of errors that one can make when programming. A **syntactic error** is an error due to incorrect grammar: for example, the program code may possibly include invalid language elements, or include proper elements, but in the wrong order. These types of errors are often caught by a program called a compiler, if the language is a compiled language. Java, and many of the other languages discussed in this chapter, are compiled languages. Syntax highlighting provided by a code editor can also indicate syntax errors. A **semantic error** is an error due to misuse of a programming concept, despite correct syntactic structure. An example of a semantic error would be the use of different (incompatible) data types in an assignment. Semantic errors are caught when the program code is compiled. A **run time exception** is an error that occurs when the program is syntactically correct, but some "illegal" operation occurs when the program is executing. Examples of run time errors include a division by zero, or the program using too much of the computer's memory. The final type of error is called a **logic error** or **bug**. A logic error occurs when the program does not solve the problem that the programmer meant for it to solve. This is through no fault of the computer or the programming language, rather, the programmer has failed to program a correct solution.

An example of a logic error would be a program that converts from Celsius to Fahrenheit, when the problem that needed to be solved was a Fahrenheit to Celsius conversion. In this chapter particular attention is paid to common syntax and simple semantic errors, but other common misconceptions that novices have about a language are also discussed.

## 2.1 Java

Identification of the common compiler errors made by novice Java developers is a more recent development in the past two years, despite Java's rise in popularity since the late 1990s as the language taught to beginner programmers. At the University of Victoria, for example, Java has been the language taught in first year computer science programming courses since 1999. The recent studies done by Jackson *et al.* [7] and Ahmadzadeh *et al.* [8], that are discussed in this section, were done concurrently to this work. Although these studies provide valuable information to tool developers about the most common compilation errors, there is no mention of any changes in the errors over time, or the time it took for students to fix the errors. If developers who create tool features for novices base their features this information alone, they may create features that are not useful after a short period of time.

### 2.1.1 Syntax Errors

In 2005, Jackson *et al.* [7] conducted a study of novice Java errors during a first semester course on Java programming. The authors used a logging program to capture the errors every time students and their instructors compiled code. They found that the top ten errors accounted for over 50% of the total errors made by the 583 students and 11 faculty members. The top 20 errors, in order of frequency, were:

- cannot resolve symbol
- ; expected
- illegal start of expression

- class or interface expected
- 'identifier' expected, ) expected
- incompatible types, int
- not a statement
- } expected
- class FinalProject
- illegal start of type
- java.lang.String
- invalid method declaration
- return type required
- boolean
- else without if
- { expected
- double
- ( expected
- possible loss of precision

Many of the errors can be attributed to carelessness, for example, the most frequent error, "cannot resolve symbol", usually occurs when a variable is not declared before it is used or if a misspelling occurs. The "incompatible types", "else without if", and "possible loss of precision" errors suggest that students are indeed confused about some programming constructs. The int, java.lang.String, boolean, and double type errors refer to type casting or loss of precision errors.

An interesting aside to the above study was that instructors at the same institution did not accurately identify the top errors when surveyed by Flowers *et al.* [9]. They missed errors such as incompatible types, and syntactic errors such as missing "class" or "interface" keywords, and illegal expression statements. The authors and instructors felt that the students had a good understanding of the learning objectives; they believed that a lack of programming experience and simple syntax errors were primarily responsible for student errors [9]. The high level of simple syntax errors found in the later study by Jackson *et al.* appears to support that statement.

Another study that obtained information on common novice Java errors was carried out by Ahmadzadeh *et al.* [8]. Students enrolled in a first course on computer programming used the JCreator IDE with a modified Jikes compiler that was enabled to store information on compiler messages encountered by a student, as well as their source code. There were approximately 192 participants enrolled in the course. In this study a differentiation was made between syntax errors, semantic errors, and lexical errors. A syntax error was defined as an error caused by grammar or disordered tokens, while a semantic error occurred "when the meaning of the code [was] not consistent with the language, for instance using a non-static global variable inside the static method." [8] A lexical error was simply an unrecognized token. Syntax errors made up 36% of the errors, while semantic errors were 63%, and lexical errors only 1%. The authors focus their analysis on semantic errors, likely because such errors are less likely to be typos. It was found that a small number of semantic errors (six), accounted for more than 50% of the total semantic errors. The most common semantic errors included:

- file not found
- use of non-static variable inside the static method
- type mismatch
- using a non-initialised variable (the most common error)
- method call with wrong arguments
- method name not found.

In 2003, Jadud [10] carried out a study of student compiler errors during lab classes, where instructors used the BlueJ development environment (BlueJ is further discussed in Section 3.3.1). Data from 63 students was logged every time they compiled their code; this data included any compiler errors as well as the source code. Jadud found that "the five most common errors [accounted] for 58% of all errors generated by students while programming"' [10]:

- missing semicolons (18%)
- unknown symbol : variable (12%)
- bracket expected (12%)

- illegal start of expression (9%)
- unknown symbol : class (7%)

As with [7] and [8], the errors were more simplistic in nature: many of which could be attributed to simple typing errors. Given that the BlueJ development environment is aimed to provide support for an objects-first teaching methodology, it is interesting that students were making very similar errors to students who were likely being taught in a constructs-first approach. Since the specific teaching methodology used is not mentioned in any of the papers, I cannot draw any conclusions from the similarities in the error types.

In an alternative to a study where errors were logged when code was compiled, Coull *et al.* [11] observed weekly lab classes: when students requested help from the tutor, the error and its cause were recorded. Because errors were only recorded when students requested help, we can gain a better understanding of what errors students felt they needed assistance with. Unfortunately, little data has been published from this study; however, an example of the common errors during the second week of a first year course in programming included: files not added, incorrect case, ";" expected, and "}" expected. The classification of the errors differs from the previous studies since the authors were able to directly observe the student, and did not categorize based on the compiler error message. For example, a straightforward "incorrect case" error would not be presented to a student by a Java compiler; the error would be presented to the student as a "variable cannot be resolved" error. It is also interesting that students were requiring the help of their tutor for ";" expected errors. For this error the compiler error message is not terribly misleading, so it is a surprise that students were requesting help with this. Perhaps the presence of the tutor lowered the tolerance students had for spending time to fix the error on their own.

Another source for a listing of common novice Java errors (and solutions) are online resources [12, 13, 14]. These web sites, and many more, are often put together by instructors and teaching assistants who feel that they are seeing students make the same errors over and over. Although these web sites do not provide an accurate resource for the most common errors made, they provide a longer list of common errors than the "top five".

Although many of these studies and resources provide useful lists of the most common novice Java errors, they do not address which errors the students take the longest to correct. As such, we cannot determine which errors the students are struggling with, and which are trivial for them to fix. By determining the time that it takes for students to fix an error, we can better determine which errors students need the most help with.

### 2.1.2 Misconceptions

Student misconceptions about object-oriented programming and programming constructs were investigated by Fleury [15]; Fleury presented 28 students enrolled in an introductory Java programming course with a set of Java programs and variations of those programs, and asked if the variant programs would have the same output as the original programs. From this, Fleury derived some "student constructed rules". It was found that students "constructed incorrect rules by misapplying correct rules" [15]. These erroneous rules included:

- the Java compiler can distinguish between same-named methods only if they have differences in their parameter lists (not if they are located in different classes)
- the only purpose of a constructor is to initialize the instance variables of an object
- numbers or numeric constants are the only appropriate actual parameters corresponding to integer formal parameters
- the dot operator can only be applied to methods

In the work done by the ITiCSE 2004 working group, they found that few comprehension errors were due to misconceptions of programming constructs [4]. The exceptions to this were the semantics of the return statement, which several students had confused; and the concept of arrays, where one student in particular had difficulties differentiating between the position and the contents. This study was limited in that questions focused on arrays, not the object-oriented concepts associated with Java. The work by Fleury found misconceptions because of fragile knowledge of object-oriented concepts [15]. This suggests that student misconceptions are associated with a fragile knowledge of some of the more intermediate programming concepts that are taught in first year programming courses.

Despite the fact that both Fleury and the ITiCSE 2004 working group studies were limited in the types of misconceptions that they explored, it does appear that students have some misconceptions about the Java programming language. In the case of the misconceptions found, many of these would not be apparent in the compiler errors that a student would encounter. A program looking for proper style could possibly catch some of these errors. The above studies highlight the need for tools that can prevent some of these misconceptions from forming. Further studies are required to improve our knowledge of common misconceptions.

## 2.2 Novice Errors in Other Languages

There have been a number of studies that have focused on novice errors and misconceptions in languages other than Java. These studies are relevant, since some fundamental programming concepts transcend a specific language. The Blue study [16], along with the others discussed in this section, demonstrate that the language taught is just one part of teaching how to program, and that instructors should not expect any language to be the silver bullet that will solve all student confusion. Novice programmers are just that: they are learning how to program and are going to make mistakes; it is in our best interest to help students to learn from their mistakes. The studies in this section are presented according to the language or topic of instruction.

### 2.2.1 Pascal

Some of the most influential work done in the area of novice errors and misconceptions was done in 1986 by Spohrer and Soloway [17, 18, 19]. They conducted a study using students' first syntactically correct Pascal programs from an introductory Pascal programming course in the spring semester of 1984. This resulted in 158 programs for three different assignments. Spohrer and Soloway investigated the two common folk wisdoms about bugs: "Just a few types of bugs can account for a majority of the mistakes in students programs", and

"most bugs can be attributed to student misconceptions about language constructs" [18]. They found that, on average, over 55% of bugs in students' programs were caused by 20% of the different error types encountered. While they concluded that a few bug types did account for the majority of student mistakes, they did not conclude that it was misconceptions about language constructs that gave students the most trouble.

The conclusion that misconceptions of language semantics were not responsible for most bugs was based on an analysis of the 10% most common bug types, which accounted for over a third of all bugs found. Common bugs included: incorrect boundary condition (off by one), OR for AND, incorrect constant or formula, need for parentheses, and output fragmentation. Of 11 bugs studied in the three programs, only one was likely due to a misconception (belief that whitespace will be ignored when reading input), while two others (OR for AND, and need for parentheses) could possibly be due to misconceptions. The remainder were unlikely due to confusion of the programming constructs.

This study design would not be appropriate for a study of the Java programming language. The Java compiler can prevent some student misconceptions about programming constructs from becoming bugs by finding them as semantic errors when the code is being compiled. For example, a student who has a misconception about the interface construct, and tries to subclass an interface with a concrete implementation, will encounter a compilation error.

### 2.2.2 Algorithms and Data Structures

Ginat conducted a study [20] which investigated student programming mistakes with respect to a course on algorithms; the author felt that the same mistakes were being made by students in introductory programming courses. Ginat found that students tended to implement "greedy" solutions: the solution that looks the best at the time, as well as solutions that would allow for the greatest gain with each iteration (to reduce the problem size). Students also tended to verify their solutions based on intuition. Such approaches will not get a student very far in a course on algorithms, where careful attention must be paid to the

mathematical correctness of a solution. Ginat suggested that it would be best for instructors to directly challenge the incorrect solutions until the students provided adequate justification. This work suggests that instructors need to ensure that students have well tested and justifiable solutions, since students' abilities in this area are not well developed.

### 2.2.3 Eiffel

A study of 72 novice Eiffel programmers from two unrelated universities, was carried out by Vee *et al.* [21] to better understand what type of errors students were making. Eiffel, like Java, is an object-oriented language. The course was taught using the "Inverted Curriculum" (gradually unveil underlying principles) approach, instead of an objects first or a constructs first approach. The authors modified the compiler used so that student programs were saved at every compilation. The most common errors found were type errors: when the wrong type was used when declaring or assigning variables. Syntax errors were also made, but the authors found that with Eiffel's "simpler" syntax, there are fewer opportunities for omissions; most syntax errors were typos. Other errors included feature call errors, rewrite instead of reusing code, not following hints, language overlap (using syntax from another language), extra variables, assignment errors, unnecessary queries, expressions used as instructions, and inheritance difficulties. This study was unique in that the authors briefly discussed some of the problem solving strategies students used to solve errors in their programs. The authors found that students would continue to use the same problem solving methodology, such as *backtracking*, where small changes were introduced for testing, and then the student would return to the previous answer.

### 2.2.4 Blue

The object-oriented Blue language and development environment, developed at the University of Sydney, was created to provide better support for teaching object-oriented concepts to first year students [16]. The Blue language was designed to use words in lieu of symbols,

and provide only one way of doing the more basic tasks. A web-based survey, to test the knowledge of students enrolled in their second semester of Blue, was taken by 58 students. The results of the survey were somewhat disappointing to the authors, because even with the use of a teaching language, students still encountered problems with object-oriented concepts, and there had not been a "shift away from the stereotyped novice obsession with low-level, syntactic issues" [16]. Specifically, problems with understanding variable scope and object comparison, and a superficial way of reading and commenting code, were found in this study.

### 2.2.5 C

In 1991, Duncan and Robson [22] collected data about common C programming errors though a survey and an interview. The C programming language, unlike Java, is a procedural language: instead of an object paradigm, a program is considered to be a sum of methods, functions, etc. Forty competent C programmers were chosen as subjects. The authors referred to those with less than two years experience as novices, none were what I would refer to as a novice. In the survey, each subject ranked the frequency that they encountered a given fault. Later on, frequently occurring faults were discussed in an interview without reference to how the fault was earlier ranked. There was a fair amount of bias since subjects were providing information based on recollection. The authors also acknowledged that there was additional bias due to the type of work each subject did, their level of experience, and their most recent work. In the study, they found that novices found boolean operators to be "troublesome". Further to this, "missing type definitions occurred frequently for novice C users and less so for the more advanced user" [22].

Another study which focused on syntax errors made by students was carried out by Kummerfield and Kay, at the University of Sydney [23]. The purpose of this study was to evaluate a tool for helping students learn to correct syntax errors. Students who participated in this study of C syntax errors had previously learned Blue, but their C programming skills were varied. The authors had created a web-based reference for students which contained

common C and C++ errors as determined by instructor experience, and some student programs. Users accessed an error explanation and possible correction by searching in the guide for the compiler error message. In the study, seven students were asked to fix eight compiler errors in a program; some students were provided with the guide, while others were not.

Kummerfield and Kay found some differences between the novice and more experienced student programmers in how they solved their errors: experienced programmers had developed strategies to fix a particular type of syntax error, while novices required more attempts before arriving at a solution; when encountering an unfamiliar error message, experts applied several different "generic" strategies and would look at any other error messages, while novices would often panic after reading the first unfamiliar message, and then ask for assistance or refer to the guide; if an error was still not fixed, both novices and the more experienced programmers would start to make random changes in the code [23]. The authors found that use of the guide did help students to fix their errors, and that examples were essential for student understanding. Syntax error correction was found to be a time consuming process: on average students took 30 minutes to correct eight syntax errors.

## 2.3 Summary

In this chapter I discussed several studies that focused on understanding the errors that novice programmers are making. Although some mistakes are likely due to carelessness, novice programmers appear to have a fragile knowledge of syntax and also have some misconceptions about programming constructs. In the next chapter I discuss work that has been done to assist novice programmers with their programming activities through tool support. I focus on work that has been done to help students better understand and correct their errors.

# Chapter 3

# Gild and Other Software for Novice Programmers

Integrated Development Environments (IDEs) are software tools created to assist software developers and programmers with the high cognitive load they must bear. Although IDEs are greatly beneficial to software programmers, there is often a significant amount of time that is spent learning how to fully use an IDE. This time spent learning is likely due not only to the large amount of functionality provided, but also due to the varying nature of software tasks. While a great amount of functionality is useful to an expert user, it is not necessarily so for beginners, who may feel intimidated and suffer a loss of productivity when searching for the more basic functionality.

Those teaching first year computer science courses are often faced with several decisions: whether or not to use an IDE, and should they decide to use an IDE, should it be one developed for novices? Computer science educators have many options available. The simplest solution tends to be to have students use a text editor and then compile and run programs through the command line. At the other end of the spectrum, students may use a professional level IDE.

In this chapter I first discuss some requirements and suggestions that have been made for IDEs for novice programmers. This is followed by a discussion of Gild, an IDE for novice developers that we have developed at the University of Victoria. Finally, I provide a summary of other development environments and tools that have been designed to

help students with their programming tasks. For each development environment, particular attention is paid to its support for novice programming errors.

## 3.1 Requirements for a Novice Development Environment

There are many factors to be considered when choosing a development environment for first year students. In a panel discussion of IDEs for novice Java programmers [24], it was felt that IDEs for students should be simple, stable, and affordable. In addition to those requirements, the following features emerged as desirable in a novice IDE: syntax highlighting, exposed Java compiler, inclusion of teaching aides, resource management, error notification and navigation aides, code completion, debugger, UML support, and the ability to include plug-ins for extensibility.

Pane and Meyers wrote a report on the usability issues that should be taken into account when designing a programming environment for novice programmers [25]. This report, which draws on much of the research on novice programmers, programming environments, and languages, was organized according to Nielsen's usability heuristics [2]. Some suggestions that Pane and Meyers have for novice environments and languages include: highlight important information (such as syntax), code formatting, immediate feedback, support for planning, consistency in notation, minimize working memory load, simple and useful error messages, support for testing and debugging, and documentation that describes everything a user needs to know about a system.

There is no definitive list of requirements for a development environment for novices. One reason is that a programming language, such as Java, can be taught by an instructor in a variety of ways. Objects-first instruction focuses teaching on the higher level principles of the object-oriented language, where much of the focus is on object-oriented design. Constructs-first instruction focuses the initial teaching on the lower-level syntax of a programming language such as if/else statements, loops, and arrays. Some curricula may place an emphasis on testing and test-cases, while others may focus on the creation of programs

that have more sophisticated user interfaces. These different curricula and methodologies will often require different features in an IDE. To add support for all of the varying needs of instructors would likely result in an IDE just as bloated as a professional IDE!

## 3.2 Gild

Gild is an IDE created at the University of Victoria specifically for instructors and novice Java programmers [26]. Like DrJava [27], Penumbra [28], and KenyaEclipse [29], which are discussed later in this chapter, it is a perspective (plug-in) for the Eclipse IDE. Gild was first released in January 2003, and has been used at the University of Victoria for several first year courses, and also at other academic institutions throughout the world [30]. In this section I discuss the features of the Gild IDE.

### 3.2.1 Eclipse

Eclipse is a popular, professional, open-source IDE [31]. Originally developed by IBM, it is now overseen by the non-profit Eclipse Foundation. Eclipse has a community of developers who have created open-source and commercial plug-ins that add functionality to Eclipse. Because of this plug-in based architecture, users can have multiple plug-ins installed in their Eclipse environment. Eclipse developers can create their plug-ins as a view, which is a single window or tabbed pane; plug-ins can also be created as a perspective, which is an entire screen configuration of views. For example, in Figure 3.1, there are three views displayed in the Gild perspective: the Gild editor view, the resource view, and the web browser view. A workspace in Eclipse contains all of the user's files and preferences, including screen layout preferences. A user can have multiple workspaces.

### 3.2.2 Features

Requirements for the Gild IDE were provided by a group of instructors, teaching assistants, and students. Some requirements for Gild included: a simplified file import and save process, syntax highlighting, code completion, help reference for typical errors, code examples, visualizations of classes and concepts, tutorials, an annotation tool for markers, collaborative features such as an instant messenger and code sharing, a separated save/compile/run process, and a simplified debugger. The first release of Gild, in January 2004, included an integrated web browser with linking to the code, simplified and reduced menu and button options, a simplified resource view, support for separate save/compile/run operations, a simplified debugger, a single thread of execution, import and export of archived (zip) files, the DrJava interactions pane [32], and a simplified Java editor. In later versions of Gild, new features included bracket matching, a problem view with extra error support, marking support, and the ability to run a code fragment from an HTML page. Figure 3.1 shows an overview of the Gild IDE.

#### 3.2.2.1 Import and Export Archived Project

The Eclipse IDE manages the user's files, and provides navigation to those files within the IDE. The actual location of the files in the computer's file system is often not considered to be relevant, since many developers work with a file versioning system that stores files at a remote location. A programming assignment in a first year course on programming will often require more than one file, as such a simple method for moving multiple files between home and school was required. Students can import files as a "Gild archived project", which is a zip file: a format commonly used to compress multiple files into a single unit, that can be later uncompressed. The file exporting and saving methods that come with Eclipse are also available to students. Students can also export a single file, a file system, and a jar file, among others. The archived project also enables instructors to bundle their course materials, including html pages, code examples, and PowerPoint slides,

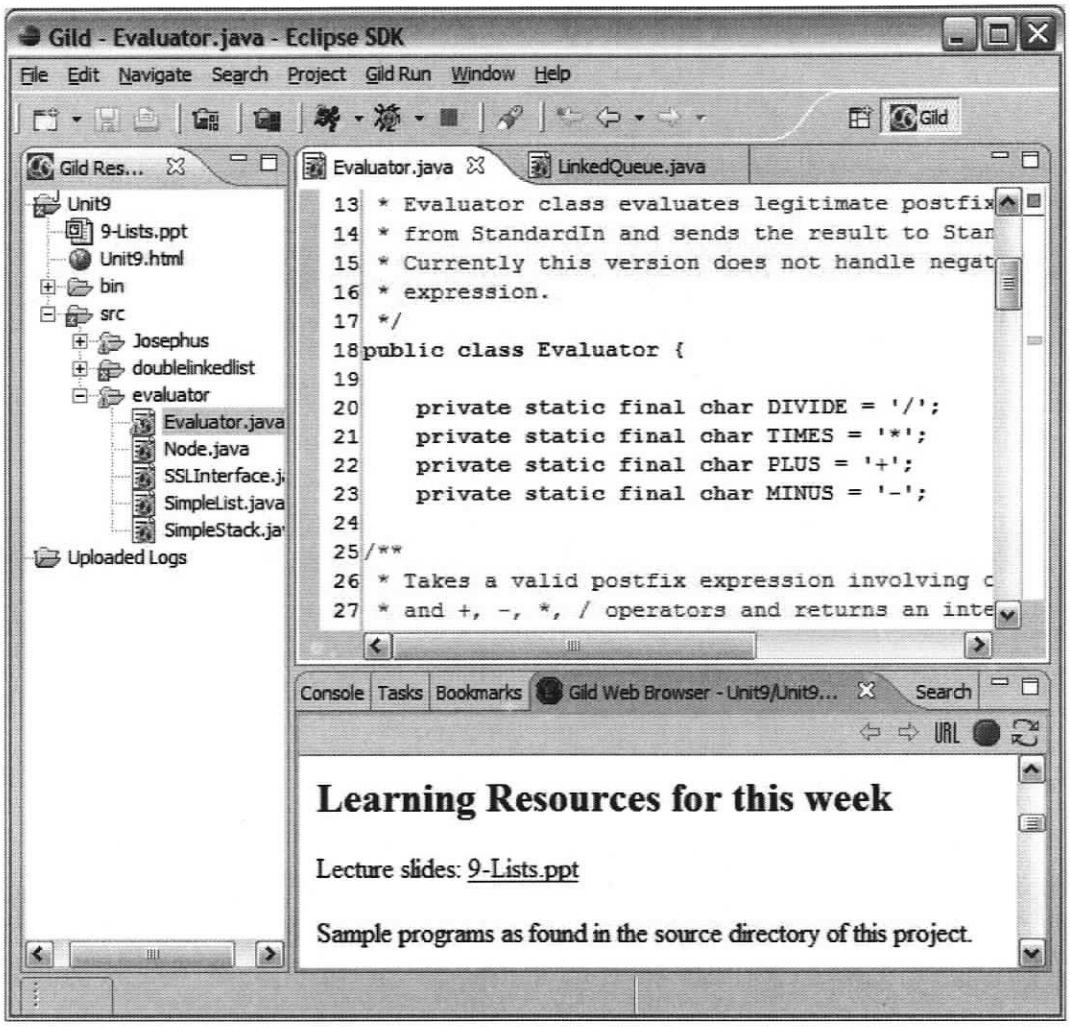



**Figure 3.1.** *An overview of the Gild IDE: simplified code editor, resource view, and web browser*

into one downloadable package.

#### 3.2.2.2 Simplified Java Editor

The simplified Java editor has syntax highlighting, so words that are special in the Java language, as well as comments, are displayed in a different colour. Line numbers are displayed by default. The Eclipse Java editor contains some features such as word completion and templates that have not been included in the Gild editor. Some instructors felt that by having code automatically inserted into student programs, students would be at a disadvantage when they were expected to write code from scratch in an exam situation. Repetition is key to some learning.

#### 3.2.2.3 Separated Save/Compile/Run process

By default, Eclipse enables code compilation (building) when the user saves their code, so no Build button is visible. In Gild, the Build button is made visible. By default, in Gild code must be explicitly built before it can be run but there is the option to enable building on save (automatic building). With Eclipse, when a user first wants to run a Java application, they must first create a run configuration. With Gild students just need to click the Run button to run their code, and the run configuration is created using the defaults. If there are multiple classes that can be run, Gild prompts the user to select a class from a list. When code is running, the background colour of the code editor changes from white to yellow, as a way to indicate this change of state.

#### 3.2.2.4 Simplified Debugger

A debugger is a program that enables users to step through their code one instruction at a time, or run to a specified point. It allows users to inspect variable values, and the flow of execution. Debuggers are useful for assisting programmers with finding logic errors. Eclipse has a debugger that is available as a separate perspective. This perspective displays

a great deal of information, for example, the threads of execution are shown for a Java program. Threading is not a topic discussed in many first year programming courses, and it was thought that this debugger would be overwhelming to first year students.

The Gild debugger was designed to provide students with only the most necessary features. Instead of opening in a separate perspective, it consists of two views that appear right next to the code editor in the Java editing perspective: a variable view, and a breakpoints view. Figure 3.2 shows these two views beside the code editor view. There are only four buttons for using the debugger: Resume (to resume the execution of the code), Terminate (to stop the debugging process), Step Over (to execute a line of code), and Step Into (to execute a line of code, but if there is a method call, go to the method code). A breakpoint provides a mechanism to note a location where the debugger should pause execution. When the Gild debugger is run, a silent breakpoint is inserted where the first line of execution occurs. This way the student can still debug their code if they forget to insert a breakpoint. As with when code is running, debugging code also results in the background colour of the code editor to changed from white to yellow. The line that will be executed next is highlighted in grey.

#### 3.2.2.5 Gild Web Browser

The Gild web browser view was designed to assist teachers with organizing their lecture materials as well as to provide students with an integrated web browser. Since many instructors often demonstrate code samples in class, and include or refer to code snippets in their notes, Gild provided "active HTML" support. In an ordinary web page, instructors can create links to a specific location in a java class (either at the class or method level) or another HTML page. Alternatively, segments of code can be embedded in the HTML file. This reduces the chance of having incorrect copied and pasted code: if the instructor were to update the Java file because of an error, the HTML file would not have to be edited as well.

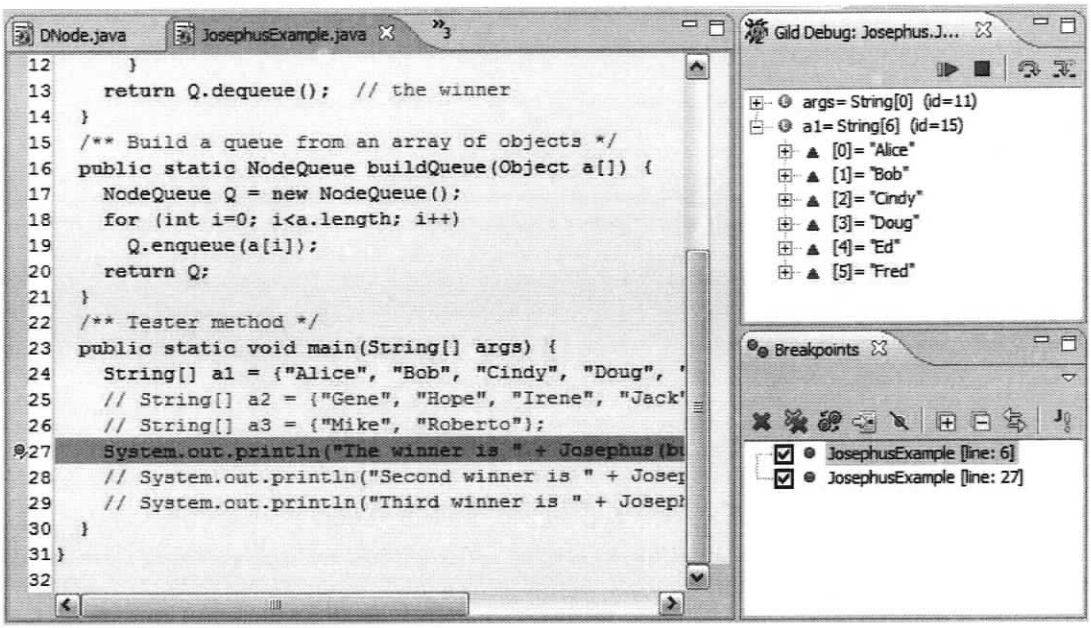



**Figure 3.2.** *Gild debugger*

#### 3.2.2.6 Problem View with Extra Error Support

Eclipse provides several features to assist users to locate and correct their compilation errors. These features are also present in the Gild perspective. When a compilation error is detected, the location of the error in the code view is underlined with a squiggly red line. A problem marker (a white "x" through a red circle) is located beside the line number. Further to this, a red box appears in the file overview located at the other side of the code editor view; this shows the location of the error in the code relative to the entire file.

The problems view lists the details for each of the compilation error(s): the error message, the file, the folder and the line number. By double clicking on an error in this view, the code where the error exists will be opened in the editor view. The problems view was enhanced for the Gild perspective to include an "extra help" feature. Through this feature, specific errors types can be given supplemental information that was predefined in an XML file. The goal was to provide students with help for the more common errors that students struggled with. Help was provided by explaining the errors and possible causes

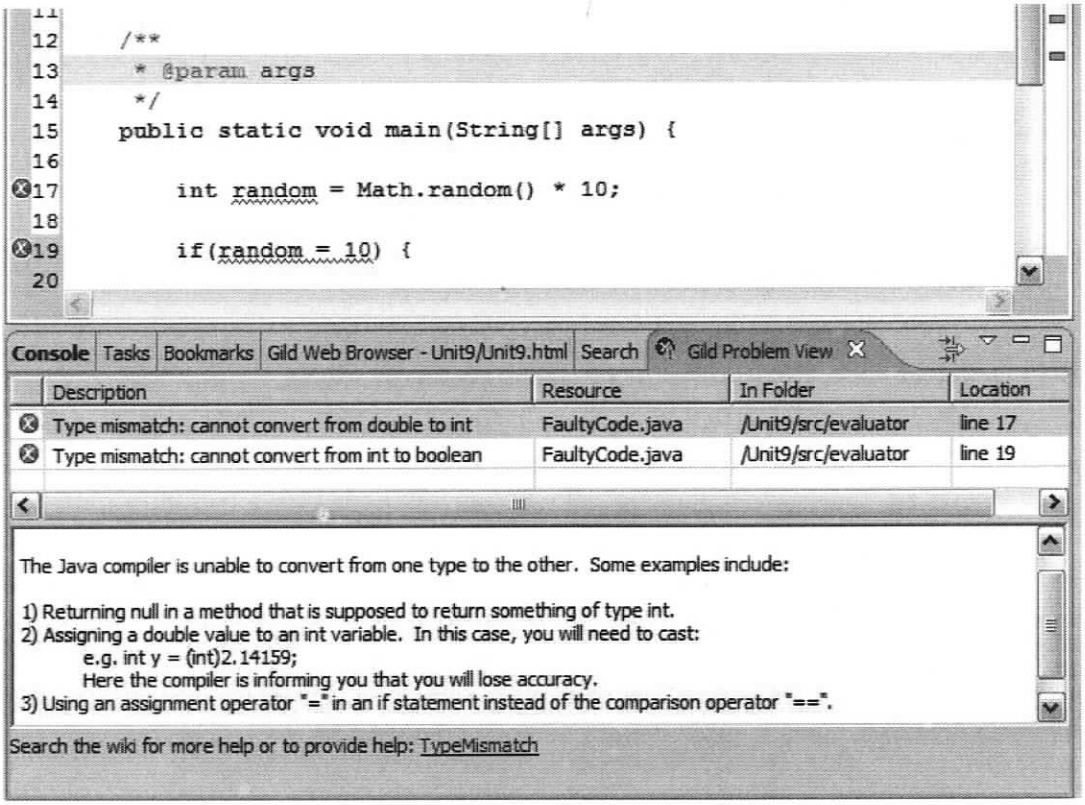



**Figure 3.3.** *Gild extra help*

and solutions in plain English. Messages were kept short, but it was felt that some errors were associated with specific course concepts, and that a longer explanation could benefit students encountering such errors. A wiki site was created to provide further information on a topic. Students could search the wiki for an error type through a link located beneath the "extra help" text area. Figure 3.3 shows the extra help available for a type mismatch error. In the example there are two different errors that both result in similar error messages that are considered to have the same error type. While one error is caused by assigning a double to an int, the second error is caused by using the "=" assignment operator instead of the correct "==" comparison operator.

As of the summer of 2005, there were 347 different compiler error types in Eclipse that could be given supplemental information. To provide supplemental information for

all errors would not have necessarily helped students, since it was likely they were only encountering a fraction of the possible errors. Since Jackson *et al.*'s [7] list of the most common compiler errors was not available at the time, several online resources [12, 13, 14] provided lists from instructors and lab instructors errors that were used to determine what errors would have extra help. As part of this work, a total of 51 extra help messages were created for Gild's Problem View. A list of these errors is available in Appendix A.

## 3.3 Other Development Environments for Novices

The following development environments are popular IDEs aimed at novice Java programmers. Each IDE has a particular teaching methodology or teaching need that it addresses. In the case of DrJava, and to a further extent, KenyaEclipse, the student's interaction with the Java language is changed. With the exception of BlueJ, the IDEs are plug-ins for the professional, open source, Eclipse IDE [31].

### 3.3.1 BlueJ

The BlueJ development environment was created specifically for teaching introductory Java programming in an objects-first manner. BlueJ is a joint project between Deakin University in Australia, and the University of Kent at Canterbury in the UK, and is also supported by Sun Microsystems [33]. The developers of BlueJ believe that development environments are generally flawed because they are not object oriented, are often either overly complex or simple, and tend to use graphics just for designing GUIs [34]. Because BlueJ is intended to support an objects-first introduction to Java programming, UML is used to graphically provide students with a means for interacting with classes: from creating instances and classes to invoking methods.

The developers of the BlueJ IDE recognize the issues that novices have with compilation and run time errors, and try to provide extra support for understanding errors. When discussing Blue, a programming language for novices and the precursor to BlueJ, Kölling

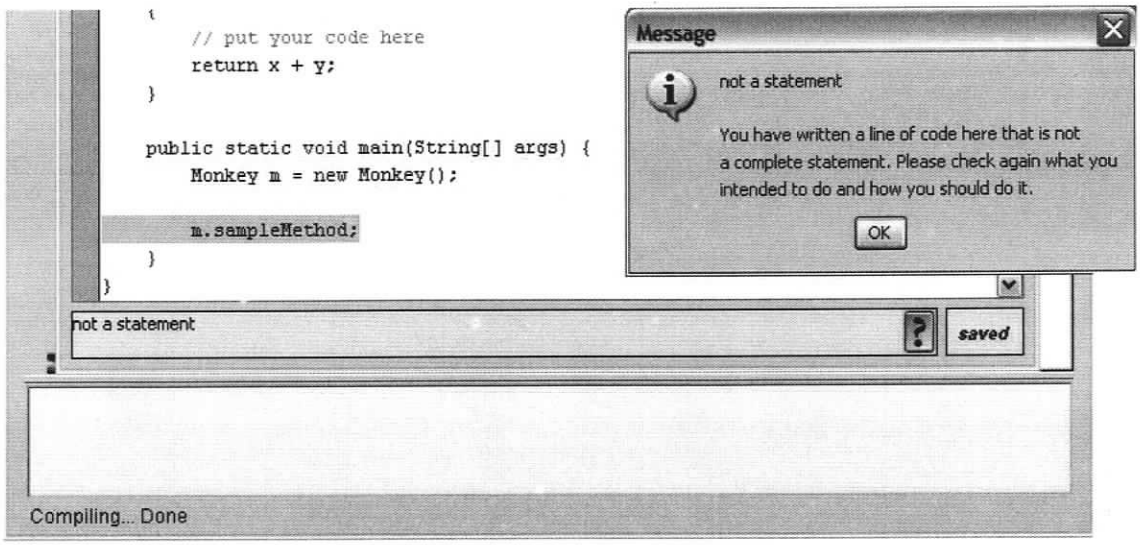



**Figure 3.4.** *Extra error support available in BlueJ*

stresses the large amount of time that novice programmers can spend trying to understand errors:

> Good error messages make a big difference in the usability of a system for beginners. Often the wording of a message alone can make all the difference between a student being unable to solve a problem without help from someone else and a student being able to quickly understand and remove a small error. The first student might be delayed for hours or days if help is not immediately available (and even in a class with a tutor it may take several minutes for the tutor to be able to provide the needed help). [35]

The BlueJ environment presents compilation errors to the user in the editor and in the console: the line in the editor that contains the error is highlighted, and the error message is displayed in an information (console) area. If the file containing an error is not open, BlueJ opens the file, and the line containing the error is displayed. Only one error is displayed at a time for a given file; to get to the next error the code must be compiled again. When the user begins to type, the error message is no longer displayed. For some compilation errors, additional help can be accessed by selecting a question mark icon. The additional

error messages are displayed along with the original compiler message, in a separate pop up window. (See Figure 3.4.) Unfortunately, the user cannot type while the additional help is displayed. As with Gild, all messages are predefined, so error help is specific to an error type.

### 3.3.2 DrJava

DrJava was developed by a group from Rice University in the USA [27]. Like the developers of BlueJ, they observed that traditional IDEs were not well suited for novice programmers. In Java, I/O and the main method syntax are particularly arduous for beginner programmers. DrJava has an Interactions pane that provides a "read-eval-print" loop to shield students from command line I/O, instance creation, method invocation, and main method intricacies [32]. DrJava is available both as a stand alone program, and as a plug-in for the Eclipse IDE. By integrating DrJava with Eclipse, students can switch to Eclipse when they do not require the pedagogic support that DrJava provides. The developers of DrJava provide error assistance to novice developers by highlighting the syntax errors right away, and also by highlighting matching parentheses and braces.

### 3.3.3 Penumbra

Penumbra is another plug-in for the Eclipse IDE that was designed specifically for novice programmers [28]. The developers of Penumbra, from Purdue University in the USA, aimed to simplify the Eclipse interface so that students could better "grow with the environment they first start out with" [28]. Pedagogical support in Penumbra comes in a the form of a simplified Eclipse interface, a simplified CVS interface for managing assignment file distribution and submission, and an object-oriented hierarchy view. The developers of Penumbra spent little attention on helping beginners with their programming errors, and instead relied on Eclipse's error notification and navigation features. The Penumbra project now appears to be inactive.

### 3.3.4 Kenya and KenyaEclipse

Kenya [36], from the Imperial College London, is a teaching language based on Java that is translated into Java code for compilation and running. The student can then view their Kenya code and equivalent Java code side by side. Kenya offers a reduced set of keywords, a similar syntax to Java's, and a simplified I/O. Although the developer of Kenya created a simple IDE for the Kenya language, an Eclipse plug-in was later developed to leverage some of the features of a professional IDE [29], such as the package explorer and the tasks/problems view.

Kenya and KenyaEclipse provides error support primarily by abstracting away some of the complexities of the Java programming language. The KenyaEclipse IDE relies on Eclipse's "as you type" compiler error checking, but also introduces stylistic error checking for things such as convoluted boolean expressions, or missing break statements in switch statements. If a stylistic error is found, a "quick fix" suggestion enables the student to fix the error with a single button click. It could be argued that the "quick fix" option does not enable students to learn by doing, since they may miss the subtlety some of the changes that were automatically made in their code.

## 3.4 Java Compiler Enhancements and Similar Programs

Enhancements to the Java compiler itself enables the creation of more specific error messages. Unlike the static error messages provided by BlueJ and Gild (Section 3), the messages provided by these tools can include the faulty code and thus provide more exact instructions. Many of these tools can also be used independently of an IDE, which allows for more freedom of choice when selecting a development environment to use, but is a less integrated solution. These tools can also look for logical errors that are common among novices. Many of these tools are developed with only that institution's needs in mind, and are not widely available. Some tools are not even given a specific name.

### 3.4.1 Gauntlet

At the United States Military Academy, the instructors in a first course on Java and XHTML programming found that "cadets were not focusing on problem solving but instead were focusing on Java syntax and missing the purpose of the assignments" [9]. As a result, the authors developed Gauntlet [7, 9], a pre-compiler that was intended to better explain the most common novice syntax and semantic programming errors. Gauntlet was initially provided with error explanations for the 50 most common errors. These common errors errors were determined based on instructor experiences at the United States Military Academy, there is no mention in [9] of any other specific resources used or study to determine these errors. Gauntlet provides students with a series of steps to follow to try to solve their problem, and refers to line numbers in the code. The messages are written in a friendly and often humorous manner, and offers praise when code compiles successfully. Gauntlet was eventually integrated into the students' code editor, and instructors have found Gauntlet to be successful as the quality of student code had improved.

### 3.4.2 University of St. Andrews and University of Abertay Dundee

At the University of St. Andrews and the University of Abertay Dundee in Scotland, a Java application was created to provide student Java programmers with additional help and solutions for common compiler error messages [11]. The unnamed application parses compiler error messages from the Java IDE, and using a database that contains common error massages and solutions, information is output to a text file. This application is independent of the IDE. There are plans for evaluation of this program, and the errors will be compared to those collected through observations of a first year lab class. The common compiler error messages were determined through observations of first year programming lab classes, when students requested help from the tutor, thus enabling the authors to note the error's specific cause.

### 3.4.3 Expresso

Expresso is a pre-compiler developed at the Bryn Mawr College that aims to provide students with better compiler messages [37]. To determine what errors they should target, the authors surveyed professors from across the United States and asked them what they thought the five most common programming errors were that were made by novices, and what were the three most difficult errors to fix based on the compiler error message. Survey data was also collected from novice student Java programmers. The authors identified 20 syntactic, semantic and logic errors for which to provide improved messages. Expresso was to be run via the command line. The pre-compiler provided students with a line number for where the error occurred, error messages that were written in plain English, and also suggestions for fixing the error.

### 3.4.4 University of the West of England Tool Set

At the University of the West of England, Bristol, a tool set was developed to work with the Java compiler to produce better quality error messages for beginner programmers [38]. Students used Emacs as their IDE. The command line tool consists of three components: a Pre-Scanner, which runs before the code is compiled, and provides both enhanced syntax error messages and points out common problems; an Instant Feedback component that reports on coding style and use of comments; and a Pretty Printer that reformats and properly indents the code. The results of the pre-scanner from all students were logged, and every day all logs were combined into a single file, but unfortunately no information about the most common errors was included in their paper [38].

## 3.5 Intelligent Tutoring Systems

The development of intelligent tutoring systems became popular in the 1980s, the most notable being the LISP tutor, which was developed to help students achieve their program-

ming goals by keeping them on track toward the correct solution [39]. The Lisp tutoring system was found to improve student performance when compared to a control group.

The Java Intelligent Tutoring System (JITS) was designed to provide beginner Java programmers with an web-based development environment that offers dynamic tutoring as students complete small programming tasks [40]. JITS parses students' code before it is compiled, and is used to assist with compilation errors, not logic errors. JITS provides suggestions to students when their code is different from the correct solution, or if there are syntax errors that JITS can infer a possible correction for. If a student accepts an offered suggestion, the code is automatically updated by JITS. Unfortunately, JITS is computationally expensive: a maximum of 50 lines of code can be parsed in a reasonable amount of time.

## 3.6 Summary

In this chapter I discussed development environments and other tools that have been developed to provide novice programmers with support for learning how to program. These tools are often custom built solutions for a particular institution's requirements. Although some institutions do not make their compiler enhancements available for download, they are sharing their experiences with the tools they have developed. These experiences, along with knowledge of the common errors and misconceptions made by novices, are a valuable resource for other tool developers. In the following chapters of this work I discuss my study of novice errors and student use of the Gild IDE. My aim is to better understand how students use Gild and what errors they are struggling with.

In addition to research questions that focus the most common compiler errors, I also ask what are the most common run time errors. I investigate how long it takes for students to fix their errors, to better determine which errors students are struggling with the most. In order to determine if the extra error support provided by Gild is useful to students throughout the semester, I ask how students' errors change over time. Finally, I also ask for feedback

from students about the Gild extra error help feature, and where they go for help when they encounter an error they do not understand. Answers to these questions is useful for the developers of Gild for the implementation of features, in particular, features to support novices with their programming errors.

# Chapter 4

# Methodology

In this chapter the methodology that was used in an attempt to answer my research questions is discussed. This includes a discussion of the changes made to the software logging program, the target participant group, and the study procedure. I discuss how the data obtained was used to answer each of these questions. Finally, the limitations of the approach taken with this study is addressed.

## 4.1 Data Collection Technique

Questionnaires and short user studies have yielded feedback from students that has been beneficial to the validation of and requirements gathering for Gild. Due to the time commitment required on behalf of participants, these forms of data collection were used sparingly as we wanted to minimize classroom interruption and ensure adequate participation. These forms of information gathering only provide a snapshot of the use of Gild at some point in time. An ethnographic study was also carried out to further observe the use of Gild; however, this also involved a significant time commitment from both participants and researchers. Few participants can be closely monitored by a single researcher, so this was also a limited means for collecting information about the use of Gild. The user studies and the ethnographic study were all likely prone to the Hawthorne effect, where participants under study will behave more positively given the knowledge that they are being observed [41]. The primary method of minimizing the Hawthorne effect is to apply the same level of ob-

servation to the control aspects of the study [41]. Another possible method for minimizing this effect is to limit the presence of observers.

To answer our research questions, I chose to answer the first three questions though the use of a software logger, and the last two questions through a questionnaire. Software loggers are non-intrusive tools that can silently monitor the use of specific software. The advantage of a software logger is that key information such as features used, data typed, and errors encountered by a user can be recorded, while the participant is not reminded consistently that they are involved with a study, thus reducing the Hawthorne effect [42]. An additional benefit is that participants can be monitored over a longer period of time with no additional overhead for separate sessions. Unfortunately, some contextual information is lost when using software loggers, as opposed to directly observing the use of software. For example, I cannot observe what the participant is focused on, what materials they use in addition to the program that is being monitored, and if there are others present.

Log files have been used by many researchers in computer science who study novice errors and programming behaviour [7, 8, 10, 19, 21, 23]. Many of these researchers have logged compilation errors when the code was compiled, and some even obtained a copy of the code that was compiled. None of these studies mention looking at how long students took to fix particular errors.

## 4.2 Mylar Monitor

The Mylar monitor is a component of the Mylar plug-in for the Eclipse IDE; Mylar provides software developers with a task specific view of their workspace [43]. The Mylar plug-in is being developed at the University of British Columbia, but is an open-source project. It was developed to track the user interactions with Eclipse in order to create a degree-of-interest model for a user's task. User interactions such as views selected, preference changes, menu commands and web browser use are tracked. The Mylar monitor contains several monitors that implement an Eclipse listener or monitor. As such, monitors for a specific action can

be added or removed with ease. At the time of our study design, the Mylar monitor was closely coupled with the Mylar project's current research; however, after the requirements of other researchers were discussed with the Mylar team, a more study-neutral version of the Mylar monitor was created. The Mylar monitor was then available as an extension point for Eclipse developers.

The Mylar monitor saves every interaction event in an XML file located on the participant's computer. The XML file is then sent to a remove server. Interaction events are the only type of event logged. For example, an edit interaction event is:

```
<interactionEvent>
<kind>selection</kind>
<date>2006-01-06 07:13:29.269 PST</date>
<endDate>2006-01-06 07:13:29.269 PST</endDate>
<originId>ca.uvic.gild.ui.GildEditor</originId>
<structureKind>file</structureKind>
<structureHandle>Assign1/Calculator.java</structureHandle>
<navigation>null</navigation>
<delta></delta>
<interestContribution>0.0</interestContribution>
</interactionEvent>
```

The kind element refers to the type of interaction. The kind can be one of: selection (user clicks or types on something), propagation (the event is triggered), command (menu option or shortcut), or preference (preference changes). The date and endDate elements generally contain the same data: the time of the event. The originID specifies from what view the event took place in or the menu or shortcut command issued. The structureKind, if applicable, refers to the type of structure, such as a file, or a URL. The structureHandle, if applicable, gives the address of the URL or the name of the file. The navigation element is set to null, and may be deprecated or reserved for future use. The delta element is optional and describes the change if a change took place, e.g. if something was opened or closed. The interestContribution element has a value of 0.0 or 1.0, and is used for Mylar's degree of interest model.

### 4.2.1 Changes to the Mylar Monitor

The original design for the Mylar monitor was to have it send XML logs back to a server that was specified in the plug-in's extension point. The files would be sent when Eclipse was started, and the user would have the option to send their log files either right away or within a fixed amount of time. Since some students work on their assignments while at school, it was necessary to change the Mylar monitor to have the log files sent when the student was exiting Gild. If the log files were not sent on shutdown, they would be lost because Eclipse workspace files are kept on the user's desktop, and all desktop data is deleted every night at midnight.

The Mylar Monitor was intended to be installed only by individuals who were interested in participating in a study of Mylar. It was found that in previous versions of Gild, students had difficulty installing the Java SDK, the Eclipse IDE, and then the Gild plug-in, so a single installation download containing all three components was created. For this study it was decided that the Mylar monitor should be bundled with the Gild plug-in for ease of installation, and to further encourage participation. A version of Gild without the Mylar monitor was also made available for download. Since the Mylar monitor was to be bundled with a version of Gild, and installed in the computer science computer labs on campus, one of the first changes that needed to be made was to alter the Mylar monitor to allow for opting out of the study. Without a means to opt out, students would be prompted to provide information for participation every time Gild was run.

The content of files was not logged by the Mylar monitor, and I decided that this information would not be necessary for answering my research questions. The monitoring program did, however, obfuscate the names and directories of the files that users were working on. For the purposes of our study it was important to know what assignments the students were programming. The Mylar monitor was changed so that file names were logged. Further to this, since a special web browser was developed for Gild, the web browser monitor that was developed as part of the Mylar monitor plug-in had to be adapted for the Gild web browser.

Two additional monitors were also added to the Mylar monitor: one for compilation errors, and the other for run time exceptions. The runtime exception monitor was relatively simple to implement. This monitor implemented a pattern match listener that would check for exceptions written to the console. When a runtime exception occurred, a new "run time exception" interaction event was created. The compilation error monitor that was added to the Mylar monitor program implemented a resource change listener. The specific changes that I was interested in were the changes to the "markers". A marker is an annotation for a resource such as a bookmark or a Java problem. For this study I was interested in problem markers. When the Java problem markers change, all of the *problem marker deltas* (additions, changes, and removals of problem markers) are available through the resource change event. Based on those deltas, the compilation error monitor created interaction events. The problem marker deltas provided were somewhat misleading, however. Consider the following scenario: a user compiles their program and a compilation error results. The problem marker delta for this error will be "added". Now, if the user does not fix the error, but rather edits text below the location of the error, the next time the program is compiled a resource change event will occur and the problem marker delta for the error will be "changed". If the user edits text above the location of the error, but again does not fix the error, a resource change event will again occur, but this time the problem marker delta will be "new". The reason for this difference is that a problem marker is associated with a line number. Further to the previous scenario, when an error is fixed, a resource change event will again occur, and the problem marker delta will be "removed".

Despite testing of the compilation error monitor, I was not aware of the association between a line number and a problem marker. This resulted in an incorrect implementation of the compilation error monitor: If a problem marker delta was "new", a "new error" interaction event was created for the new compilation error, and therefore logged by the monitor program. If a problem marker delta was "changed", the error was added to a hash table if it was not already in the table, and no interaction event was created. If a problem marker delta was "removed", a "removed error" interaction event was created if the marker

existed in the hash table. The flaw lies in "new" compilation errors always being considered new, when they may be previously existing, and not being added to the existing errors hash table. Fortunately, this oversight did not result in erroneous data, but instead meant that analysis of the log files would have to address this problem. This is further discussed in Section 4.5.1.

## 4.3 Participants

The students enrolled in Computer Science 115 at the University of Victoria during the spring 2006 semester were the primary user group targeted for this study. Computer Science 115 is a second course in computer programming, and is a required course for computer science, software engineering, and health and information science students. This course is also taken as an elective by many students who want to gain more programming experience. The topics covered in this course include object-oriented design, data structures: lists, stacks, queues, trees, recursion, and algorithm efficiency. This group was chosen since Gild was the primary IDE used in this course in several previous semesters: in spring 2004 and spring 2005. Gild was positively received by the majority of these students, and it was felt that we could benefit in learning more from this group.

After the last drop date in the spring semester of 2006, there were 112 students enrolled in the two CSC 115 course sections. These sections were taught by different teachers, but the assignments were identical for both course sections. In addition to attending lecture classes, students were also required to attend a lab section, which was one hour every week.

## 4.4 Procedure

At the beginning of the semester Gild was advertised as a potential IDE for students to use in the course. The official development environment supported by the Computer Science Help Desk was TextPad, although students were free to use any IDE for their coursework.

Students were invited to attend one of four "Introduction to Gild" sessions which were held outside of class time on January 10th, 12th, 13th, and 16th. An announcement for these sessions was posted on the course web site. On the first day of class, I was given five minutes of class time to introduce Gild and the user study to the students. The motivation that was given to students for participating was that by learning more about their errors, the Gild IDE could be improved to provide better extra help messages: they would be helping future students, just as previous students had given feedback to improve Gild to its current version. Students were provided with a link to the Gild web site, from the CSC 115 course web site, to download information.

To participate in the study the student needed to download and install the version of Gild with the Mylar monitor included, or access Gild through the computer labs in the Department of Computer Science. When a user first runs Gild, a dialogue box appears that provides information about the goals of the user study and requests user information for enrollment in the study. For the study, the required information included: an e-mail address to uniquely identify a participant, the academic institution they were attending, and the course they were enrolled in. The sex of the participant could optionally be provided. This information would then be sent to the study server, and a study id would be issued to the user. If the e-mail address had already been registered with the study, the student would be given their previously created id, otherwise a new study id would be issued. Once a study id had been obtained for the participant, this dialogue box would no longer appear. The student had the option of closing the dialogue box through a Cancel button, which would result in the dialogue box appearing again the next time Gild was run. Alternatively, another button existed for the student to decline to participate in the study, which would result in the dialogue no longer appearing. In the case of the computer labs on campus, this dialogue box would appear for the student every time Gild was started, due to the workspace being wiped clean every night.

## 4.5 Research Questions and how the Data was Used

In this section I discuss how the data was used to answer each of the six research questions. The research questions are:

- What are the most common errors (compiler and runtime) that students make?
- How long does it take for students to fix their errors?
- Do students' errors change over the course of the semester?
- Where do students go for extra help for an error they do not understand?
- Does Gild's extra error help feature aid students?
- What further insights about novice programmers and Gild can we gain through the analysis of the log file data?

Because the Mylar monitor saved the log data in an XML data format, I was able to adapt the Log Analyzer software that was developed by members of The Learning Kit/gStudy project for viewing the data [44]. The Log Analyzer software displayed the data in a tabular format: XML tags were not visible, and the data could be filtered through the use of queries. This improved the readability of the data. Log files were imported into the Log Analyzer software and the files were split into sessions because some students would leave Gild running for several days, or would take breaks from programming that were longer than 15 minutes. These sessions were then categorized according to the "work unit" that the participant was working on: e.g. Assignment 1, Lab class 5, Midterm Review, etc. The session files were then exported from the Log Analyzer software in a tab delimited format that was usable in Microsoft Excel, to facilitate future analysis.

### 4.5.1 What are the Most Common Errors (Compiler and Runtime) that Students Make?

The log files did not track the location of the error in the file and the file deltas, so I had a limited ability to track specific errors. Even with such a mechanism, errors could not be

accurately pinpointed without having an actual copy of the code. This, compounded with the problem of the "not really a new error, despite the new status", that was discussed in Section 4.2.1, meant that the best I could do was to note when I first encountered an error of a new type, and when all outstanding errors of that type had been fixed. This method reduced the number of error instances I could successfully track, but did reduce the chances of having incorrect data.

Depending on how a student had their Gild environment configured, compilations would occur either on save, or when the code was explicitly built. A perl script was created to count the number of errors of each type, whenever there was an event that would trigger a compilation. Every time a compilation occurred, any new errors were added to a hash map, if that error type did not already exist in the hash map. If there were multiple instances of this error at that time, all instances were added to the hash. The errors were marked as being resolved if fewer appeared on successive compilations, and if a build occurred and there were no compilation errors listed, all compilation errors were considered to be fixed. If a compilation error was not fixed in a session, it was not included in the list of errors. The inclusion of such errors could have resulted in double counting if the same error appeared in a participant's next logged session.

Run time exceptions were also tallied with respect to each programming session. The type of exception and its frequency in a particular work unit were tracked, and an overall list of exceptions for a participant was created. A summary of run time exceptions for all participants was also tallied.

The list of the most common errors was developed from the list of errors determined using the perl scripts. A participant's most common errors were ordered according to their frequency per work project (e.g. assignment), and a total of all errors encountered throughout the semester was also made.

### 4.5.2 How Long does it Take for Students to Fix Their Errors?

Since I only tracked the compilation errors that existed every time the code was compiled, I cannot say for certain when a given error was truly created and when that error was fixed. The error would have been created at some point before the creation time that was recorded, but also fixed sometime before the fix time was recorded. I feel that since the true creation and fix times are both in error in the same direction, the recorded fix times offers an adequate approximation.

### 4.5.3 Do Students' Errors Change over the Course of the Semester?

For this question, I consider the top 50% of errors made by a student for a particular assignment. Only work on assignments, and not labs, are considered because they generally offer the most data. Changes in the occurrences of these most frequent errors are compared across assignments.

### 4.5.4 Where do Students go for Extra Help for an Error they do not Understand?

For this question, a questionnaire was administered to the students enrolled in CSC 115, i.e. the same group of students that were invited to participate in the logging study. The results of the survey are discussed in Section 7.4.

### 4.5.5 Does the Extra Error Help Feature Help Students?

This question was addressed by two different means. First, the log file data was read using the Log Analyzer software, and any instances of the extra error help being used was noted in the participant's summary in Chapter 5. A question was also asked in the survey of CSC 115 students, which is further discussed in Section 7.5.

### 4.5.6 What further insights about novice programmers and Gild can we gain through the analysis of the log file data?

A great amount of data is available through log files, and this question was designed to allow for more qualitative analysis of the data. In Chapter 5 there is a summary for each of the participants in this study. These summaries were written based solely on log file data, and they discuss any notable interactions that the student had with the Gild IDE.

## 4.6 Limitations

There is a limit to the conclusions and generalizations that can be made when analyzing this data. For one, I do not know when a student is actively attempting to fix an error. This can result in fix times for an error that do not reflect the time that the student actually spent fixing the error. Fix times are therefore more of an upper bound, and indicate when the student is first made aware of the error, and when a compilation event occurs and the student can be sure that the error is fixed. Also, some errors can be side effects of other errors, and in this way compiler error messages can be misleading. If the student attempts to fix the first compiler error message, when the underlying error is actually caused by something that is indicated by another compiler error message (reported further down in the list), the student will have "fixed" an error that never really existed, potentially causing an error. It is important to keep these limitations in mind when reviewing any data pertaining to the most common errors made and the fix times for these errors.

Since the content of the files that students were editing was not logged, I am unable to say with certainty if a compilation error is a syntax or a semantic error. Although some compilation errors are more likely one type or the other, the type of compilation error cannot be confirmed in my study. Even with a copy of the student's code at every compilation, unless a clear mental model of the student's knowledge exists, no conclusion can be made. All compilation error types that are referred to in this study are referred to by the name

given by Eclipse's compiler. For example, a "TypeMismatch" compilation error could be due to accidentally referring to a variable of one type when a variable of a different type was intended (confusing variable names), or due to a misconception about the compatibility of different types (e.g. assigning a null value to an int). The first error would be considered a syntax error, while the latter error would be a semantic error.

## 4.7 Summary

In this chapter I discussed my research questions, and my study methodology. I also discussed how the data I obtained was used to answer my research questions, and some limitations of this approach. In addition to answering these questions, I created participant summaries. These qualitative results focus on a participant's first use of Gild, their work habits, and any other interesting aspects of their interactions with Gild's features. The qualitative results follow in Chapter 5, while the quantitative results that are used to answer my research questions follow in Chapter 6.

# Chapter 5

# Qualitative Results

## 5.1 Participants

A total of 10 people participated in this user study. The number of participants was lower than expected. From the results of the survey that was administered to the same group of CSC 115 students, 39 students (66%) stated that they had used or explored the Gild IDE during the semester. Further to this, many more students registered for the study by agreeing to the participation dialogue, but many did not choose to submit their log files when prompted. The participant information is available in Table 5.1. An overview of the log files that were sent for each participant is available in Table 5.2.

**Table 5.1.** *Participants in the Gild logging study; 'N/A' indicates data was not provided by participants*

| ID | Gender | University | Course |
|---|---|---|---|
| 68 | Male | University of Victoria | CSC 115 |
| 85 | Male | University of Victoria | CSC 115 |
| 102 | Female | University of Victoria | CSC 115 |
| 119 | Male | University of Victoria | CSC 115 |
| 136 | Male | University of Victoria | CSC 115 |
| 153 | Female | University of Victoria | CSC 115 |
| 170 | Female | University of Victoria | CSC 115 |
| 204 | N/A | University of Victoria | CSC 115 |
| 221 | N/A | N/A | N/A |
| 238 | Female | University of Victoria | CSC 115 |

**Table 5.2.** *Participant log file overview*

| ID | Files | Total Time hh:mm | Start date | End date | Assignments logged |
|---|---|---|---|---|---|
| 68 | 21 | 13:51 | 2006-01-05 | 2006-02-12 | Assign 1-3; Unit 2, other exploration |
| 85 | 10 | 25:40 | 2006-01-05 | 2006-02-12 | Assign 1-4; SENG 130 Assign 1 |
| 102 | 26 | 34:52 | 2006-01-06 | 2006-03-31 | Assign 1-7; Lab 1, 4, 5 |
| 119 | 8 | 18:47 | 2006-01-07 | 2006-01-22 | Assign: 1, 2 |
| 136 | 7 | 8:19 | 2006-01-08 | 2006-01-25 | Assign 1-3 |
| 153 | 14 | 6:25 | 2006-01-08 | 2006-03-13 | Assign 1; Lab 2, 7; ENGR/Physics 2 assign |
| 170 | 44 | 55:56 | 2006-01-10 | 2006-03-19 | Assign 1-6; Lab 1, 2, 4; Midterm review |
| 204 | 8 | 6:02 | 2006-01-12 | 2006-02-10 | Assign 1, 2, 4; Lab 3 |
| 221 | 1 | 0:14 | 2006-01-09 | 2006-01-09 | n/a |
| 238 | 15 | 12:26 | 2006-01-11 | 2006-03-30 | Assign 1, 3, 4, 7; Lab 2, 4-7; Prelab 2 |

**Table 5.3.** *Important course dates for CSC 115, spring 2006*

| Date | Event |
|---|---|
| 2006-01-05 | First day of class |
| 2006-01-10 | Gild tutorial sessions |
| 2006-01-12 | |
| 2006-01-13 | |
| 2006-01-16 | |
| 2006-01-16 | Assignment 1 due at noon |
| 2006-01-17 | Last day for 100% reduction of term fees |
| 2006-01-23 | Assignment 2 due at noon |
| 2006-01-31 | Assignment 3 due at noon |
| 2006-02-02 | Midterm exam 1 |
| 2006-02-07 | Last day for 50% reduction of tuition fees. 100% of tuition fees are assessed for courses dropped after this date. |
| 2006-02-12 | Assignment 4 due at midnight |
| 2006-02-28 | Last day for withdrawing from courses without penalty of failure |
| 2006-03-05 | Assignment 5 due at midnight |
| 2006-03-09 | Midterm exam 2 |
| 2006-03-19 | Assignment 6 due at midnight |
| 2006-04-03 | Assignment 7 due at 11 a.m. |
| 2006-04-06 | Last day of class |
| 2006-04-29 | Final exam |

## 5.2 Course structure

An overview of important dates for the CSC 115 course is available in Table 5.3. A list of the assignment and lab assignment topics is in Table 5.4.

## 5.3 Participant Summaries

Each student that participated in the study had their own way of interacting with the Gild IDE. In this section I provide short summaries of each participant's log files, where I highlight their first interactions with Gild, their work habits, and their use of the extra error help feature. It is in this chapter that I address the following research question: "What further insights about novice programmers and Gild can we gain through the analysis of the log

**Table 5.4.** *Assignment and lab assignment overview*

| Assignment | Description and learning topics |
|---|---|
| Assignment 1 | Grade calculator: 2-D arrays |
| Assignment 2 | uTunes: object-oriented concepts, 2-D arrays |
| Assignment 3 | uMedia: UML, interfaces, abstract classes |
| Assignment 4 | uSpell: spell checker (linear and binary search), sorting (insertion, bubble), arrays |
| Assignment 5 | uMedia: doubly-linked lists |
| Assignment 6 | Stacks and Train-spur: implement and use a stack |
| Assignment 7 | Binary search tree: test cases, also one of: Maze solver, file system (n-ary tree), Textpad with uSpell |
| Lab 1 | Arrays |
| Lab 2 | Polymorphism |
| Lab 3 | Polymorphism |
| Lab 4 | Secret Decoder: input/output, arrays |
| Lab 5 | Linked lists |
| Lab 6 | Recursion (binary sort) |
| Lab 7 | Merge sort |
| Lab 8 | Binary trees |

file data?" I also discuss any other interactions with Gild that are particular to a participant. While I cannot say for certain if the extra error help is actually read by a student, I can infer if a student selected the error in the problem view in order to navigate to the error's location in the code. Pauses in activity, or in particular, the user clicking on a link to the Gild wiki for an error, suggests that the student is aware of the feature and may have read a help message. Since I do not have access to each participant's grades in the CSC 115 course, or to the code that they wrote, I cannot say which participants had the best grasp on CSC 115 course concepts. A student's work habits cannot be used to speculate a student's standing in the course; Mierle *et al.* [45] found that in their analysis of student CVS repositories, how early a student starts their assignment and how soon it is submitted cannot be used to predict the student's grade in a course on programming. As such, if I comment on the student's likely standing in the course, I do so based on how error-free their assignments are before they are due, and if the student spent time testing their code.

### 5.3.1 Participant 68

This student's first Gild session was 20 seconds long: He created a new file, and then exited the program. On his second logged Gild usage, the student immediately added a new plug-in: net.sourceforge.texlipse.TexEditor, which adds LaTex support for Eclipse. The student also went to the preferences menu and edited the matching bracket colour option. The student then tested the LaTex set up. The next day the user returned to the Gild perspective and in two minutes the student had created a Test.java file, edited it, saved it, built it, and then exited from Gild. From this initial session it appears that this student was familiar with Eclipse and its plug-in architecture, and that he planned on using it for other activities than CSC 115.

In the next session the student imported assignment 1 with no problem. The code was run, and a run time exception resulted from existing compilation errors. The student then selected an error from the Gild problem view to navigate to the code. The student continued to edit code, save it, and then run it. No build happened on save, so the student encountered unresolved compilation problems whenever he ran code. While working on assignment 1 the student tried to use the Gild debugger, but this lasted only a minute, and he never stepped through the code at all. A few days into working on assignment 1, the student went to the Help menu, and then returned to the welcome page. He appears to have gone to some sort of introduction view. After 10 minutes of going through the help the student returned to assignment 1. The student was able to test the code.

One day the student switched to the Java perspective, but went right back to the Gild perspective within seconds.

In addition to assignment work, the student took a look for a few minutes at the unit 2 coding exercises provided by the course instructors. He ran the code, but not much time was spent reading it and he did not do any coding. The student mostly just looked through it to see what was there. The student also spent about 20 minutes playing with a Swing application called "Celsius Converter". I am unsure if he could properly run the Swing application in Gild. In the next session logged, the student switched to the Java perspective

and generated Javadoc for some code.

While working on assignment 2 the student encountered the following error: "The code for the static initializer is exceeding the 65535 bytes limit." He selected the problem in the Gild problem view, and eventually went to Google in the internal web browser and then to the Help Search, possibly for help with this error. While working on assignment 2 the student returned to unit 2 and coded and ran the "Complex Constructor", "Shapes", "Music", and the "Pirates" examples.

After 10 hours of using Gild the student discovered the "Rebuild" button, enabling him to check for compiler errors without running code and generating a run time error due to unresolved compiler errors. After several minutes of using the Rebuild button, the student ran his code again while compiler errors existed. The student then continued to use the Rebuild button, but also used the Run button. He did not follow the Save → Build → Run steps.

At one point while not working on assignment code, the student encountered package naming errors. The package naming errors did not appear to be a result of an improper import; rather, the errors cropped up later, and were fixed within the hour.

#### 5.3.1.1 Work Habits

Although log files were submitted from this student nearly every day, they were often short. Even when the student had a longer coding session, there were frequent breaks from coding. This student spent a fair amount of time looking at and running example code, specifically from unit 2. He also spent time programming in Java Swing. I do not believe that I have all of the student's log files for assignment 2 and 3.

#### 5.3.1.2 Extra Error Help

The student used the problem view for navigation to the error's location in the code, but did appear to view the extra help for an undefined method compilation error. After pausing for a few seconds, the student clicked on the link to the Gild wiki. There was no wiki entry

or extra error help available for that error. The student continued to use the problem view frequently for navigating to errors.

### 5.3.2 Participant 85

The first time the student entered the Gild IDE, he imported an archived project: assignment 1, 20 minutes after obtaining a study id. The student edited a file for a minute or two, and then exited. In the student's second logged usage, very little editing activity occurred, and then Gild was exited.

It was not until the student's third logged session, quite early on January 14th, that any major work occurred. The student did a fair amount of editing before building code. The student also occasionally ran code that did not compile. While working on assignment 1, the student would refer to a CSC 110 assignment: the "Penny Jar" assignment. The pattern of long periods of editing, followed by builds and runs was continued for the rest of assignment 1. During the student's last session on assignment 1, much of his time was spent with the tester file. While working on assignment 2 the student again opened the "Penny Jar" assignment from CSC 110 for reference. While this student did refer to existing code for assistance, he was never observed referring to example code made available from the course instructors.

The student started work on assignment 3 the night before it was due. While working on assignment 3, he referred to code from assignment 2. Also, when the student was working on assignment 3, he had Gild compiling on save, instead of having to explicitly build code. By the point the student was working on assignment 3 and 4 he was using Gild well. There were fewer instances of running code when there was a compile error. This student tended to use Gild's more basic "New File", for file creation.

This student also used Gild for his SENG 130 assignment 1, which was also done just before it was due. For assignments 1 through 4, I am fairly certain that I have log files for all of this student's activities.

#### 5.3.2.1 Work Habits

This student would often not stop work on assignments until just before the assignment was due, and would work late into the night in the days before the assignment was due. He did some testing when finishing assignments, but in the case of assignment 3, this was only approximately 15 minutes. The student did not begin work on assignment 4 until less than 10 hours before it was due. Despite a marathon coding session, his code was likely not complete to the best of standards: while the last build was successful, many of the previous builds and runs were not.

#### 5.3.2.2 Extra Error Help

It is difficult to tell if the student was indeed reading the extra error help, or if he was thinking about the error. I believe that the student did read some of the extra error help and was familiar with the feature. When the student encountered the following error: Syntax error on token(s), misplaced construct(s), and there was no extra help available, the student went to the Gild wiki and tried to access the page several times. There was no associated help on the Gild wiki for that error. The student also paused when selecting some other errors which did have extra help available, but also paused after selecting some errors that did not have extra help.

### 5.3.3 Participant 102

The first time the student entered the Gild program she imported a file and viewed the first assignment in the Gild web browser; this was a short first session. After a few short sessions, the student did her first significant work on assignment 1, on January 10th. She successfully imported assignment 1 as an archived project. At one point the student imported an assignment from CSC 110, "PolyMult", for reference. The student also went to the Preferences menu, and edited some of the highlighting colours. The student knew to use the Build or Rebuild button to compile her code, though she did run her code several

times when compilation errors existed.

While working on assignment 1 this student appeared to have many conceptual problems with arrays. The student tried to use the + operator with an int[] and an int[][] or an int. The student also tried to convert between these types as well. The student's difficulties with arrays extend to having problems with array boundaries: she encountered many ArrayIndexOutOfBounds exceptions.

This student worked both at home and at school on assignments. She had some problems with importing her work. Although the student did try to first import her assignment as an archived project, she was not successful and then had to create a new project and then import files to the project. This was probably because she was not exporting her work as an archived project. It is confirmed later, while the student was working on assignment 2, that although she did try to export her work as an archived project, she also exported her files to disk.

While working on assignment 1, the student created a new file but forgot the .java extension. She tried to run the file, but another file with a main method ran instead. The student went to the Help menu for 4 minutes, returned, and did some editing on the file, and then saved the code. After a few seconds she did a Save As, and gave the file a .java extension.

A month after first working on lab 4 (secret decoder), this student revisited the lab and continued work on it.

Log files for this student significantly decreased, and I do not have much for the student's work on assignments 5, 6, and 7. The student did return to Gild in assignment 7 to use the debugger. After the student imported code, she enabled automatic building of the code on save, and then debugged the assignment without first setting a breakpoint, which is perfectly fine since Gild will start the debugging from the first line of execution. The student used the Step Into and Step Over buttons as needed. After terminating the debugging of the code, the student again immediately debugged the code. The debugging lasted approximately five minutes. The student then continued to edit the code and run it. In the

final session logged for this student, the student again opened Gild for the explicit purpose of debugging her code. After several debugging runs, the code was edited and Gild was exited.

In this student's log files some of the known strengths and weakness of Gild are highlighted. The student encountered problems with importing and exporting files, which is something that should be made simple for students who work between home and school. Despite the fact that the student was moving away from Gild as the course progressed, the Gild debugger was the exact feature that brought the student back to Gild. The success of the simplified Gild debugger should not be understated.

#### 5.3.3.1 Work Habits

This student started major work on assignment 1 six days before it was due, but the student spent a fair amount of time working on it. It is likely that she finished assignment 1 at school before handing it in since the student was still encountering some run time errors due to the array index being out of bounds. I also did not see the final assignment 2 that was handed in. Assignment 3 was finished the day it was due, with many clean builds before work was stopped. I have little data for assignments after assignment 3. This student likely had a difficult start to the course, but as the assignments progressed I think the student was having fewer errors, and more success with programming.

#### 5.3.3.2 Extra Error Help

This student did not make extensive use of the extra help feature. The student never visited the Gild wiki. There were times that the student paused on an error after selecting it in the problem view, but many of these errors did not have extra error help associated with them. There was one error, a "NoMessageSendOnArrayType" error, that it appears that the student read the extra error help for: "Cannot invoke getArtist() on the array type String[]". This error was fixed immediately after the student paused after selecting this message, and changed code as per the suggestions, resulting on a getArtist() method call on a String

object, which resulted in a new error.

### 5.3.4 Participant 119

This student used Eclipse prior to enabling the logging of his data. This can be inferred because the first log files show selections of files that were never opened or imported, and the Gild perspective was explicitly opened by the student. The extent of the student's expertise is not known, and I was unable to observe his initial exploration of Gild and Eclipse. Before starting on assignment 1, the student was working with some Matrix and MatrixSizeException classes. Switching to the Gild perspective and importing the first assignment was done correctly. He used the internal web browser to view the assignment's html file. Code was often first built, and then run using the Run button. Code was rarely run when a compile error existed. This student also saved work on a regular basis, especially when trying to fix a bug, since a compilation occurred on save. The debugger was never used by this student. The student had many clean builds, and was able to run code to test it.

This student made use of find/replace. He also made use of short cuts when editing, such as Ctrl+Shift+Right or Left to highlight a word or section of text instead of using the mouse to highlight. In the user's second session that was logged, he immediately went and viewed and edited some of the preferences related to the UI. The student then went to the Help view for a couple of minutes and then went to Software Updates → Manage Configuration. He then switched workspaces, and continued work on assignment 1.

This student would often know what he wanted to do; it was just a matter of figuring out the proper mechanism for doing it. For example, when he wanted to determine the length of a String Buffer: first he tried to use `.length`, then he tried `getLength()`, then he tried `.size`, all of which were incorrect. What he needed to use was the length method for the StringBuffer class.

Although I have many log files from this participant for assignment 1 and 2, he ceased to send log files after January 22nd.

#### 5.3.4.1 Work Habits

Assignment 1 was begun two days after the first day of class and was completed the day before it was due. Most of the student's time programming assignment 1 occurred the day it was due. While working on assignment 1 the participant started work on a project that indicated that he were taking a course on numerical analysis or mathematics. The student began work on assignment 2 on January 17th, the same day that assignment 1 was due. Most of January 17th was spent on assignment 2, but after that day it was not picked up again until the day before it was due, where another busy day was spent working on it. Assignment 2 was not complete as of the last log file received at 11:30 the night before it was due: run time errors were encountered on many of the recent runs and a syntax error remained. This student worked on assignments in bursts, and would leave much of the work for the day before the assignment was due.

#### 5.3.4.2 Extra Error Help

Eight problems in the problem view were selected by this user: 6 of which were type mismatch errors. This user only appeared to read the extra error help once, suggesting that the other selections were for navigation to the error's location in the code. This student never went to the wiki.

### 5.3.5 Participant 136

There is little evidence that this student did any exploration when first using Gild. The student twice tried to open assignment 1 as an external file, but then successfully imported assignment 1 as an archived project. The student did some edits in files, viewed the assignment submission page in the internal web browser, and then saved his code. The student's first session lasted nine minutes. The next time the student entered Gild was the next morning when he first tried to run his assignment as a java application, and then as a JUnit test. The student did some more editing and then used the Gild Debug button, which gener-

ated several compilation errors. The student continued to use the Debug button to generate compilation errors.

Most surprising about the usage logs for this user was how long he spent fixing compiler errors. The student rarely had a program that compiled without errors. He would sometimes go for hours without ever being able to run his program. In fact, I can see little evidence that he was able to run and test working code before handing it in. The student could, of course, have done this at school and not have had that session logged.

Although this is not a very thorough analysis of the errors, the errors made on the first two assignments were more simplistic. The student would often forget to initialize a variable, but could fix this error quickly. Syntax errors (e.g., missing a semi-colon at the end of a statement) were also quite common, as were type conversion errors. The student's errors while working on assignment 3 indicated a lack of knowledge of polymorphism concepts. He did not appear to understand how calls to a super constructor work, and he also demonstrated a lack of knowledge of how to subclass classes and interfaces. The student did try to run code that contained compile errors.

Another interesting thing was his switch to the Java perspective. The first time he appeared to switch at Eclipse's suggestion, after creating a new Java project. Unfortunately, this switch created some package naming errors that the student did not fix while working in the Java perspective - I suspect that the student was rather confused. The student eventually found his way back to the Gild perspective. The student never did discover that he should be running code with the Run button. Later on, in the last few sessions logged, the switch to the Java perspective appears to be intentional. I suspect that the compilation that occurred on save was beneficial to this student. I do not believe that he was aware of the functionality of the Build button in Gild.

For this student, like many others, if something worked well enough, he just stuck with it: e.g. using the debugger for compiling code. The student did not use the debugger for debugging code, though he did make use of the integrated web browser.

#### 5.3.5.1 Work Habits

This student would start assignments early enough, though I do not have a very complete picture of this student's programming activities. There is little data for assignment 2, there are no log files for assignment 1 after January 8th, and there are no log files for assignment 3 after January 25th. It appears that log files were only received when the participant was starting the first few assignments.

#### 5.3.5.2 Extra Error Help

The student did select the extra error help at one point, and he went as far as to click on the link to the wiki. The compilation error that the student had encountered was for a length method call on a 2-dimensional int array. While there was extra error help available for this error, and it did correctly translate the general error, it did not address this specific error of trying to get the length of a 2-dimensional array. Unfortunately, there was no link to detailed extra information for this type of error on the wiki. The Gild Problem View is used later, on only one occasion, for navigation.

### 5.3.6 Participant 153

When this student first started work in Gild she immediately imported an archived project: assignment 1. She viewed the assignment in the internal web browser, and then began editing. When she wanted to compile/run code, this student clicked in order: the Run button, the Debug button, and then the Create Class button. The Create Class button may seem a logical choice, since compiling does create class files, but it is incorrect. Creating a class means that a new java class will be created in a java file. After selecting the Create Class button, the student then selected the Build Project button. After a few minutes she again cycled though the same buttons to compile her code. In the second, third and fourth sessions logged for this student, she spent 10 minutes trying to make some new files and projects, and then switched workspaces and exited Gild. The student then re-entered Gild, created

a new project, and then switched to the Java perspective, likely at Eclipse's suggestion. A new file was created, and then Eclipse was exited. The student then reentered Gild, tried to open an external file, went to create a new file, and then exited Gild.

The next log file received from this participant was from January 20th, when she worked on a physics assignment on Verlet integration. The student immediately switched to the Java perspective, and imported a project. The student got some package errors, which were fixed in five minutes. The student did not have build on save enabled, as such when she went to run code, she then got a list of compilation errors. This student figured out how to open files in the Java editor as opposed to using the default Gild editor. The student used the content assist feature. The student eventually enabled the showing of line numbers. She also spent a minute toggling, enabling and disabling breakpoints, but never debugged code. She did not export code from Eclipse at the end of the session. By the 23rd of January, when working on lab 2 at home, the student used Gild, and selected the Build All button to properly compile code.

Log files were not received from this participant again until March 13th. The student entered Eclipse and switched to the Java perspective, and then exited. The student then re-entered Eclipse, switched to the Java perspective, viewed her usage analysis through the Mylar reporting feature, switched workspaces, and then exited for a grand total of one minute of activity. The student then re-entered Eclipse and again switched to the Java perspective, and worked on her assignment. The student used the Rename function under the Refactor menu. The student then revisited the Mylar monitoring usage analysis button and viewed the report for a minute, and then continued work on her assignment. After this there were repeated log files from the student showing her entering Eclipse and switching workspaces and then exiting. The student then continued work on her physics assignment, but also spent time working on her MergeSort lab. After getting some ArrayIndexOutOfBounds exceptions, the student used the debugger to debug code. She terminated the debugger, and then debugged again several times. The student alternated debugging and running the code while the run time exception remained. After a break, the student resumed

work on her physics assignment, likely for an engineering class.

#### 5.3.6.1 Work Habits

I have very few log files from this participant. This was the only participant to find and use the Mylar log analyzer feature that gives basic statistics on the views used. I am unsure of this student's motivation for participating in this study. I have very little data from CSC 115 for this student. It should be noted that by switching workspaces the student would again encounter the study enrollment dialogue, but from the log files it appears that she did not enable logging for the new workspace.

#### 5.3.6.2 Extra Error Help

This student rarely used the Gild problem view for navigation, and never used it for extra error help.

### 5.3.7 Participant 170

In this student's first session she successfully imported lab 1 as an archived project, and immediately began editing the lab assignment. The student used the Rebuild button to compile code, and the Run button only after the code compiled successfully. The student exited without exporting any files.

While working on assignment 1, on January 11th, this student used the debugger for the first time after encountering a "null pointer" exception. The student set a breakpoint, and then pressed the Debug button. She stepped over one line of code, inspected a variable, and then the debug view was closed. The student then repeated the previous actions. She debugged again, and pressed the Resume button. A new breakpoint was added, the debug process was terminated, then the code was debugged again. The student eventually selected the "Step into" button, which resulted in a longer debugging session. The student also successfully located the Terminate button. It took almost an hour for the student to resolve

the "null pointer" exception. The student continued to use the Gild debugger while working on the assignment. The student also used the Gild web browser to view the assignment 1 instructions.

When the student worked on assignment 2, and created a new class, she switched to the Java perspective. This was likely through an Eclipse prompt suggesting the switch. In the student's next session she immediately switched to the Java perspective, though her files continued to be opened in the Gild perspective. After editing for a while and performing a build, the student then switched back to the Gild perspective. After one particular build, the student did a few small edits and then ran code, which failed because of an unresolved compilation error. The debugger was used in assignment 2 when the student encountered a "null pointer" run time exception.

While at school on the 24th of January, the student switched to the Java perspective, but during the next session on the 26th of January, the student remained with Gild. The student exported files to disk and exited, but then restarted Gild to export files again. While working on assignment 3, the student began using the debugger after she encountered a "null pointer" exception. Later, while working on lab 4, when the student went to create a new project, she again switched to the Java perspective, but switched right back to Gild. After importing the lab 4 files, a package naming error was encountered, but the student fixed this by creating a new project. This again triggered the Java perspective, after which the student switched back to the Gild perspective. The student continued to use the debugger as needed throughout the course.

When this student imported work she would first create a new project. This resulted in projects that had unneeded extra directories in the package structure, due to package naming conflicts: e.g. "New Folder/New Folder/Lab_Material/Secret Decoder/SecretDecoder.java", it is the "New Folder" directories that were undesirable. On February 8th, while working on assignment 4, the student enabled automatic building on save. The student still occasionally used the Build button, however. When the student worked at school she remembered to enable automatic building. During this session the student encountered a compilation

error after running her program, which prompted the student to select errors in the problem view. The student also used the debugger, except this time a run time exception did not trigger its use.

The night/early morning before the second midterm this student spent time reviewing code. She walked through example code using the debugger, and also reviewed assignment 5. Linked lists and recursion were the focus of her study.

#### 5.3.7.1 Work Habits

This student would start work on assignments well in advance, but would often still be working on the assignments shortly before they were due. This student rarely did long programming sessions: she would work in small sessions. Sometimes she would touch files for less than a minute, and then return to the work hours later. Late night work that continued into the morning was not limited to shortly before an assignment was due. This student appears to have finished assignments 1 through 6 without errors on many of the final compilations and runs; there is no data for assignment 7, which could be done in pairs. The student also did testing on the assignments.

#### 5.3.7.2 Extra Error Help

This student used the extra error help occasionally for navigation; there is no evidence to suggest that the extra error help was ever read.

### 5.3.8 Participant 204

When first using Gild, this student first attempted to open an external file, and then tried several times to open an archived project. The student then opened the log file as an external file, and then soon after opened files for Assignment 1, one at a time, as external files. The student did several "Save To" actions, went to the Help contents, and within a minute built the working set, went to the file search, and opened a new perspective (the

Gild perspective); shortly after that, Gild was exited.

The second logging session is from when the student attended one of the Gild introduction tutorials, held on January 16th. The student was able to build and run files, view extra help for an error, create bookmarks, export projects and files, and also experimented with the debugger. The student then imported a file to assignment 1, and then exported it to disk.

The next time the student returned to work on an assignment in Gild, the student again tried to import an archived project, but then opened files as external files. A new Gild project made, and the student then exited the program. A few minutes later the student re-entered the program and again tried to import an archived project, but then created a new Gild project.

The next log file that was received from the student was when assignment 2 code was imported. The student tried to import an archived project, to import files into a project called Assignment 2, and also tried to open external files. Eventually the student exited from Gild, having done no coding: just multiple attempts to import assignment code. This student was struggling with the import process, and it was not until January 19th that the student again imported files (not an archived project) and worked on an assignment in Gild.

When the student first ran code, it was done when compile errors existed. The student did this the second time as well, but was directed to the problem view, where the student did select the problem and spent enough time that it was likely read. The student did not have Gild set up to build on save, despite the fact that the code was saved before it was run, the student be informed of compiler errors only after selecting the Run button. After 30 minutes the student started to use the Build button to compile code. The student saved and built code frequently when fixing problems, but would also have longer editing periods of more than 10 minutes between builds. Despite the long time between builds at times, the student's error count was low.

The student did use the internal web browser to view Unit2.html. The student would also export files to a disk at the end of a session. The student enabled monitoring during the third lab class, where the student switched to the Java perspective, though the code

remained open in the Gild editor. In the student's last logged session, the student opened Assignment 4 and encountered package naming errors after importing files to a project called Assignment 4. The student spent three minutes and three build attempts before finally exiting.

#### 5.3.8.1 Work Habits

It is difficult to evaluate this student's work habits since the student obviously use some other editor for assignments and only use Gild sporadically.

#### 5.3.8.2 Extra Error Help

Despite being shown the extra error help feature in the introduction to Gild session held at the beginning of the semester, this student only selected an error in that view once, for a "method undefined" compilation error. Extra help existed for this error, and the student spent 12 seconds of inactivity after selecting the error. However, the error was not fixed for 29 minutes.

### 5.3.9 Participant 211

This study participant provided little log data to the study, just 14 minutes, but the student highlighted the difficulties some students have with importing assignments. The student tried Gild on January 9th, and first created a new project. The student then tried to import an archived project five times, and also tried to open an external file. There are no other activities during this time listed for this participant.

### 5.3.10 Participant 238

In this participant's first session she immediately imported an archived project, assignment 1, and saved it. She then switched to the Java perspective, though the Gild editor remained the editor for files. The student spent time reading code, but was not actively editing. After

22 minutes she went to debug, but then immediately went to the Help menu to search for something. The student then returned to the Debug menu, and then to the Run menu. Soon after this, the program was exited.

In the second log file from this student, she again began by switching to the Java perspective, and spent some time working on the lab 2 assignment. She then imported assignment 2, but continued to work on lab 2. Another time, while working on prelab 2, the student got an error: "cannot instantiate the type Shape", where Shape is an abstract class. The student referred to her notes in the web browser for a few minutes, and then exited the program. She returned to the prelab a few hours later, and there was no sign of the instantiation error. The student did use the problem view, and often selected the "The field Square.fillColour is never read locally" warning, possibly for navigation, but the sheer number of times that it was selected suggests that the error was not understood.

When working on assignment 3, this student first imported unit 2, the example on polymorphism (Music.java), which she looked at for a few minutes. This student also made use of the rename function, an option in the "Refactor" menu. Assignments were imported and exported as zip files. This student's first logged use of the debugger occurred on January 30th. She used the debugger when she encountered NullPointerExceptions while running code. The student used the variable view, and did one step, but then resumed execution. The student knew how to return to the Java perspective. In the Java Perspective, the student used the "Generate Javadoc" functionality. When continuing work on assignment 7, after importing a project, the student encountered some package name errors. The student used the QuickFix suggestion provided by Eclipse to fix the errors.

The student did not appear to understand the difference between invoking a method on something and passing a variable as an argument to a method. The student tried to invoke a method on an int[]. Syntax errors were frequently discovered. The student knew how to use the Eclipse IDE, and she compiled code frequently in order to fix problems soon after they arose.

#### 5.3.10.1 Work Habits

The log files for this student are from when she worked on assignments at school, probably between classes. Because of this, some of the log files show the student's work in lab classes. In the Feb 6th lab class the student did not finish the assignment, but stayed right until the next class started. I am unable to tell how well she completed the assignments, and when.

#### 5.3.10.2 Extra Error Help

Since this student would always immediately switch to the Java perspective when starting up Gild/Eclipse in the labs, I did not expect to see the extra error help used by this student. However, while working on assignment 3 on January 30th, after some time spent editing, the student explicitly opened the Gild Problem View. The student had previously tried to run code when the following error existed: "The method getMediaLibrary() is undefined for the type MediaLibraryInterface." Extra help was available for this error. The error was fixed by the next save and build. Later, the student again referred to the Gild Problem View for an "Unreachable Code" error for seven seconds, which also had extra help available, and the student fixed it shortly thereafter. The student paused long enough after selecting that error that she likely read some of the extra help. The problem view was used for navigation to the error's location in the code after the student had run the code when a compiler error existed. The student never used the Gild problem view again. The student did continue to use the Eclipse problem viewer to navigate to an error's location in the code.

## 5.4 Summary

In this chapter, a summary for each study participant was created to provide a qualitative analysis of the log file data in order to address the following research question: "What further insights about novice programmers and Gild can we gain through the analysis of the

log file data?". The remaining research questions are discussed in Chapter 6. Qualitative data from log file analysis is also included in Chapter 6, as appropriate.

# Chapter 6

# Quantitative Results

In this chapter the data obtained through the methods discussed in Section 4.5 is displayed. In each section, I address each of my research questions. A synthesis and discussion of the data follows in Chapter 7.

## 6.1 What are the Most Common Errors that Students Make?

### 6.1.1 Compilation Errors

In this section I present the most common compiler errors. First, Table 6.1 presents an overview of the amount of compiler error data that was acquired for each participant, and the percentage of error types that account for the most common errors. This is followed by a summary of the 50% and 70% most common compiler errors for each participant, and the mean fix times for these errors. The error types that account for 50% of the errors are bolded in the tables. In the case of extreme fix time values, a median fix time is supplied in addition to the mean fix time. A complete listing of the compilation errors made by all participants in this study can be found in Appendix D.

**Table 6.1.** *Error information for all participants - total number of errors and error types, and number of error types that account for 50% and 70% of total errors for a participant; mean excludes data from participant 153*

| | | | Top 50% of errors | | Top 70% of errors | |
|---|---|---|---|---|---|---|
| **User ID** | **Total errors** | **Total types** | **# Types** | **Actual %** | **# Types** | **Actual %** |
| 68 | 185 | 43 | 9 | 57% | 14 | 72% |
| 85 | 450 | 38 | 3 | 55% | 6 | 70% |
| 102 | 1108 | 49 | 4 | 53% | 8 | 70% |
| 119 | 485 | 51 | 5 | 52% | 8 | 72% |
| 136 | 122 | 26 | 4 | 58% | 8 | 77% |
| 153 | 16 | 8 | 3 | 63% | 4 | 75% |
| 170 | 811 | 50 | 4 | 53% | 9 | 72% |
| 204 | 60 | 19 | 5 | 52% | 8 | 72% |
| 238 | 360 | 41 | 5 | 50% | 8 | 75% |
| mean | | | 4.9 | | 8.6 | |
| stdev | | | 1.7 | | 2.2 | |

**Table 6.2.** *Ten most frequent errors and their mean fix times for all participants*

| **Error Type** | **Errors (%)** | **Errors (#)** | **Mean h:m:s** | **Extra help?** |
|---|---|---|---|---|
| **UndefinedName** | 20% | 726 | 0:02:46 | yes |
| **TypeMismatch** | 9% | 317 | 0:02:21 | yes |
| **UndefinedMethod** | 8% | 305 | 0:03:59 | yes |
| **ParsingErrorInsertToComplete** | 8% | 303 | 0:02:03 | yes |
| **ShouldReturnValue** | 5% | 117 | 0:06:25 | yes |
| UndefinedType | 5% | 173 | 0:02:16 | yes |
| ParsingErrorDeleteToken | 4% | 146 | 0:01:19 | yes |
| PackageIsNotExpectedPackage | 3% | 115 | 0:04:11 | yes |
| UndefinedConstructor | 3% | 103 | 0:02:12 | yes |
| ParameterMismatch | 3% | 95 | 0:02:23 | no |

**Table 6.3.** *Participant 68 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **UndefinedName** | 15% | 28 | 0:00:22 | 0:00:23 | |
| **UndefinedType** | 6% | 12 | 0:03:08 | 0:03:49 | |
| **UndefinedMethod** | 6% | 11 | 0:02:46 | 0:02:35 | |
| **ParsingErrorInsertToComplete** | 6% | 11 | 0:01:15 | 0:01:21 | |
| **TypeMismatch** | 5% | 10 | 0:01:39 | 0:01:36 | 0:00:55 |
| **NonStaticAccessToStaticField** | 5% | 9 | 0:07:56 | 0:01:38 | |
| **PackageIsNotExpectedPackage** | 5% | 9 | 0:00:55 | 0:03:20 | |
| **AbstractMethodMustBeImplemented** | 4% | 8 | 0:02:53 | 0:02:08 | 0:01:32 |
| **ArrayReferenceRequired** | 4% | 8 | 0:02:04 | 0:00:39 | |
| ParsingErrorDeleteTokens | 4% | 7 | 0:01:23 | 0:01:23 | |
| ParsingErrorDeleteToken | 4% | 7 | 0:01:04 | 0:01:04 | 0:00:35 |
| UndefinedConstructor | 3% | 6 | 0:00:48 | 0:00:22 | |
| ShouldReturnValue | 2% | 4 | 0:03:29 | 0:02:22 | |
| MissingReturnType | 2% | 4 | 0:00:35 | 0:00:19 | |

**Table 6.4.** *Participant 85 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **UndefinedName** | 31% | 141 | 0:05:26 | 0:08:52 | 0:01:25 |
| **ShouldReturnValue** | 12% | 55 | 0:11:52 | 0:12:44 | |
| **UndefinedMethod** | 12% | 53 | 0:03:38 | 0:05:38 | 0:03:04 |
| ParsingErrorInsertToComplete | 6% | 27 | 0:04:11 | 0:08:41 | 0:01:22 |
| ParsingErrorDeleteToken | 4% | 20 | 0:02:51 | 0:08:38 | 0:00:25 |
| UndefinedField | 4% | 18 | 0:00:33 | 0:00:19 | |

Many of the mean fix times for Participant 85 had outlier times. In the summary for this participant's log files it was noted that he would spend a fair amount of time editing code, which was then followed by a series of code building and running. This pattern of editing and then fixing problems may be the cause for the outlier times. This student had few TypeMismatch errors (2% of the total errors made) compared to the average for all participants. For this group of students, TypeMismatch errors are the second most popular error type, and accounted for 9% of the total errors.

**Table 6.5.** *Participant 102 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **UndefinedName** | 23% | 258 | 0:02:11 | 0:03:51 | |
| **TypeMismatch** | 16% | 177 | 0:03:00 | 0:04:38 | 0:01:02 |
| **UndefinedMethod** | 7% | 81 | 0:03:33 | 0:05:18 | 0:01:00 |
| **ParsingErrorInsertToComplete** | 7% | 73 | 0:01:12 | 0:01:28 | |
| InvalidOperator | 5% | 57 | 0:02:01 | 0:02:36 | 0:00:50 |
| ShouldReturnValue | 5% | 54 | 0:04:12 | 0:04:42 | 0:03:38 |
| UndefinedConstructor | 4% | 39 | 0:02:37 | 0:05:05 | 0:01:11 |
| ParsingErrorDeleteToken | 3% | 38 | 0:00:46 | 0:01:05 | |

**Table 6.6.** *Participant 119 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **UndefinedName** | 16% | 77 | 0:03:20 | 0:05:07 | 0:01:05 |
| **TypeMismatch** | 13% | 61 | 0:00:59 | 0:01:28 | |
| **UndefinedMethod** | 12% | 57 | 0:02:17 | 0:03:03 | 0:00:48 |
| **ParsingErrorDeleteToken** | 6% | 30 | 0:01:43 | 0:02:58 | 0:00:18 |
| **ArrayReferenceRequired** | 5% | 26 | 0:01:17 | 0:01:04 | |
| ParsingErrorInsertToComplete | 5% | 24 | 0:02:18 | 0:03:24 | |
| UndefinedConstructor | 5% | 22 | 0:01:17 | 0:01:03 | |
| ShouldReturnValue | 4% | 19 | 0:05:11 | 0:04:32 | |

"TypeMismatch" errors were higher for Participant 102 than most of the other participants. This error occurred frequently in the first three assignments done by this student (Table 6.33). This error often occurred because the student assigned an object type to an array of the object type, or vice versa. The student also assigned objects to strings. The errors generated by this student suggested that they had difficulties with array concepts.

The high number of "ArrayReferenceRequired" errors encountered by Participant 119 is unusual when compared to the most frequent errors of the other participants. Upon further examination, it was found that the majority of these errors occurred, and were later fixed, at the same time. These errors may therefore be the result of one or two errors.

**Table 6.7.** *Participant 136 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **UndefinedName** | 18% | 22 | 0:02:36 | 0:03:22 | 0:00:40 |
| **TypeMismatch** | 16% | 20 | 0:00:35 | 0:00:35 | |
| **ParsingErrorInsertToComplete** | 14% | 17 | 0:00:54 | 0:00:59 | |
| **UndefinedConstructorInImplicit ConstructorCall** | 10% | 12 | 0:11:12 | 0:09:23 | |
| ParsingErrorDeleteToken | 6% | 7 | 0:01:17 | 0:01:54 | |
| ShouldReturnValue | 5% | 6 | 0:00:48 | 0:00:26 | |
| InvalidClassInstantiation | 4% | 5 | 0:10:36 | 0:10:42 | |
| NoMessageSendOnArrayType | 4% | 5 | 0:02:25 | 0:05:06 | |

**Table 6.8.** *Participant 153 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **ParsingError** | 25% | 4 | 0:02:35 | 0:03:59 | 0:00:17 |
| **UndefinedMethod** | 19% | 3 | 0:00:33 | 0:00:12 | |
| **PackageIsNotExpectedPackage** | 19% | 3 | 0:05:30 | 0:00:00 | |
| ParsingErrorInsertToComplete | 13% | 2 | 0:00:17 | 0:00:00 | |

There is nothing unusual about most frequent errors for Participant 136 when compared to the other students; however, in the participant summary it was noted that this student rarely had code that compiled without error. The UndefinedConstructorInImplicitConstructorCall and InvalidClassInstantiation errors occured while the student was working on assignment 3. As noted in the participant summary for this student, his work on assignment 3 indicated a lack of knowledge of polymorphism concepts and constructor chaining. For this student, the UndefinedConstructorInImplicitConstructorCall and InvalidClassInstantiation errors are likely semantic in nature.

**Table 6.9.** *Participant 170 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **UndefinedName** | 19% | 153 | 0:01:42 | 0:02:31 | 0:00:28 |
| **ParsingErrorInsertToComplete** | 13% | 103 | 0:02:51 | 0:05:01 | 0:01:03 |
| **UndefinedType** | 12% | 99 | 0:01:14 | 0:02:20 | |
| **PackageIsNotExpectedPackage** | 9% | 71 | 0:01:49 | 0:02:23 | |
| UndefinedMethod | 6% | 52 | 0:03:40 | 0:06:37 | 0:01:20 |
| ParsingErrorInsertTokenAfter | 4% | 29 | 0:01:45 | 0:02:23 | |
| ParsingError | 3% | 27 | 0:04:44 | 0:13:06 | |
| ParsingErrorDeleteToken | 3% | 27 | 0:01:01 | 0:01:21 | |
| ShouldReturnValue | 3% | 21 | 0:05:29 | 0:05:43 | |

**Table 6.10.** *Participant 204 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **ShouldReturnValue** | 13% | 8 | 0:01:00 | 0:00:11 | |
| **UndefinedMethod** | 12% | 7 | 0:10:18 | 0:12:06 | 0:05:18 |
| **UndefinedConstructor** | 12% | 7 | 0:01:21 | 0:01:13 | |
| **PackageIsNotExpectedPackage** | 8% | 5 | 0:00:41 | 0:00:00 | |
| **ParsingErrorDeleteToken** | 7% | 4 | 0:02:02 | 0:00:48 | |
| ParsingError | 7% | 4 | 0:04:25 | 0:06:19 | 0:00:57 |
| TypeMismatch | 7% | 4 | 0:03:20 | 0:02:52 | |
| ParsingErrorInsertToComplete | 7% | 4 | 0:00:18 | 0:00:11 | |

Like Participant 85, Participant 170 had few TypeMismatch errors (only 2% of errors were TypeMismatch errors.) The summary for Participant 170 suggests that she was a diligent worker who knew how to use Gild and Eclipse well, and was likely one of the more competent students in the course. Participant 85 also appears to be more competent than some of the other participants, though he left assignments too close to the time that they were due. There may be a correlation between a low percentage of TypeMismatch errors and programming ability or understanding. There is not enough information in this study to support a stronger statement than the suggestion that this be further investigated in a future study.

There are only five PackageIsNotExpectedPackage errors for Participant 204, but this

**Table 6.11.** *Participant 238 - Most frequent errors and their mean fix times*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| **UndefinedName** | 12% | 43 | 0:02:02 | 0:02:48 | 0:00:53 |
| **ParsingErrorInsertToComplete** | 12% | 42 | 0:01:02 | 0:01:17 | |
| **UndefinedMethod** | 11% | 39 | 0:01:07 | 0:01:51 | |
| **AbstractMethodMustBeImplemented** | 10% | 35 | 0:03:35 | 0:02:58 | |
| **ParameterMismatch** | 6% | 22 | 0:01:41 | 0:02:35 | |
| NonStaticAccessToStaticField | 5% | 18 | 0:01:09 | 0:02:20 | |
| UndefinedType | 4% | 15 | 0:00:44 | 0:00:44 | |
| TypeMismatch | 4% | 15 | 0:02:05 | 0:02:39 | |

student struggled repeatedly with importing files or archived projects into Gild. There are few errors recorded for this student since use of Gild was sporadic.

In the user story for Participant 238 it was noted that there were many simple syntax errors made, which agrees with this list of the most common errors. The NonStaticAccessToStaticField error and the TypeMismatch errors are the only two errors in this list that are more likely semantic errors.

### 6.1.2 Runtime Exceptions

In this section, run time exceptions are first presented as an aggregation of all participant run time exceptions. This is followed by a listing for each participant, according to the frequency of the exception's occurrence. The assignment or work project where the majority of the errors occurred is noted.

**Table 6.12.** *Runtime exceptions for all participants*

| Count | % | Error type | Notes |
|---|---|---|---|
| 184 | 40% | Null pointer | Assignment 1-4, Lab 5 |
| 124 | 27% | Array index out of bounds | Assignment 1-4, Lab 4, 7 |
| 117 | 26% | Unresolved compilation problem(s) | Usually decreases in occurance |
| 8 | 2% | Class cast | Assignment 2, 4, 6 |
| 7 | 2% | String index out of bounds | Assignment 4, 5 |
| 7 | 2% | No such element | Assignment 4 |
| 4 | 1% | Stack overflow | Assignment 2, Lab 7 |
| 1 | <1% | Out of memory error: Java heap space | Assignment 3 |
| 1 | <1% | Regular expression: pattern syntax | Assignment 4 |
| 1 | <1% | Number format: for input string | Assignment 6 |
| 1 | <1% | Test: PASSED numeric java.lang.NullPointerException | Assignment 1, test case |

**Table 6.13.** *Participant 68 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 87 | 60% | Unresolved compilation problem(s) | Occurs throughout, no decrease |
| 34 | 23% | Null pointer | Assignment 3, Misc exploration |
| 23 | 16% | Array index out of bounds | Assignment 1, most in assignment 3 |
| 1 | 1% | Out of memory error: Java heap space | Assignment 3 |

Participant 68 continued to encounter many "Unresolved compilation problem(s)" runtime exceptions because he would press the "Run" button in Gild without first building his code. Although this student eventually built his code with the "Rebuild" button, this did not occur until he had been using Gild for at least 10 hours. After the discovery of the

"Rebuild" button this student would occasionally build his code, but he continued to run his code without first building it.

**Table 6.14.** *Participant 85 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 10 | 38% | Array index out of bounds | Most in assignment 4, some in assignment 1 and 2 |
| 8 | 31% | Null pointer | Assignment 1, some in assignments 3 and 4, and SENG 130 work |
| 7 | 27% | Unresolved compilation problem(s) | Assignment 1, then drops off over time, none in assignment 4 |
| 1 | 4% | Regular expression: pattern syntax | Assignment 4 |

Participant 85 encountered fewer "Unresolved compilation problem(s)" after the first assignment because he learned to build code before he ran it, and by assignment 3 he had enabled compilation on save.

**Table 6.15.** *Participant 102 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 66 | 66% | Array index out of bounds | Assignment 1, some from assignment 4 |
| 22 | 22% | Null pointer | Most from lab 5 |
| 12 | 12% | Unresolved compilation problem(s) | Most from assignment 1, lab 1 |

Participant 102 struggled with the syntax of arrays: e.g. assignment, accessing elements. She also encountered many more run time exceptions due to accessing array elements than the other participants. The high number of array index out of bounds exceptions may be a good indicator that a student is struggling with the concept of arrays.

**Table 6.16.** *Participant 119 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 39 | 78% | Null pointer | Most from assignment 2, some from assignment 1 |
| 6 | 12% | Class cast | Assignment 2 |
| 3 | 6% | Stack overflow | Assignment 2 |
| 2 | 4% | Array index out of bounds | Assignment 2 |

**Table 6.17.** *Participant 136 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 3 | 100% | Unresolved compilation problem(s) | Assignment 1 |

Participant 136 occasionally used the debug button to run code that had compiler errors, which led to the unresolved compilation problem(s). The student eventually switched to the Java perspective where compilation occurred on save, and this error did not occur again.

**Table 6.18.** *Participant 153 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 8 | 89% | Array index out of bounds | Lab 7 |
| 1 | 11% | Stack overflow | Lab 7 |

Few unresolved compilation problem(s) runtime exceptions were encountered by Participant 170 as she was able to use Gild effectively from early on in the semester. The runtime exceptions were more varied for this student, likely because she provided the most complete set of interaction log files for her coursework. The no such element exception is another example of a boundary condition error, as it was thrown when there were no more tokens in the StringTokenizer object.

**Table 6.19.** *Participant 170 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 43 | 60% | Null pointer | Some in assignment 2, 3, and 4 |
| 7 | 10% | Array index out of bounds | Assignment 2 |
| 7 | 10% | String index out of bounds | Assignment 4, 5 |
| 7 | 10% | No such element | Assignment 4 |
| 4 | 6% | Unresolved compilation problem(s) | Assignment 2, 4 |
| 2 | 3% | Class cast | Assignment 4, 6 |
| 1 | 1% | Number format: for input string | Assignment 6 |
| 1 | 1% | Test: PASSED numeric java.lang.NullPointerException | Assignment 1, test case |

**Table 6.20.** *Participant 204 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 1 | 50% | Unresolved compilation problem(s) | Assignment 2 |
| 1 | 50% | Null pointer | Assignment 2 |

Participant 204 encountered the resolved compilation problem(s) runtime exception because this student had run code before compiling it. Approximately 30 minutes after using run to compile code, this student started to first build code.

**Table 6.21.** *Participant 238 - Runtime exceptions*

| Count | % | Error type | Notes |
|---|---|---|---|
| 37 | 77% | Null pointer | Most assignment 3, lab 5 |
| 8 | 17% | Array index out of bounds | Lab 4 |
| 3 | 6% | Unresolved compilation problem(s) | Assignment 3 |

Unresolved compilation problem errors are noted for Participant 238 while work was done on assignment 3. Despite having used the build button, this student used the Run button a few times instead. This student did not use Gild very much for the first two assignments, so most initial learning of Gild likely took place while working on assignment 3.

## 6.2 How Long Does it Take to Fix an Error?

In addition to the mean fix times for the compilation errors that are listed in Tables 6.3 to 6.11, I provide a listing of the five compilation errors with the longest mean fix time for each participant. I also list the error types of the five compilation errors that took the participant the longest to fix. Median fix times are listed if large outlier times affect the mean.

**Table 6.22.** *Errors with the longest mean fix times, for all participants*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | Extra help? |
|---|---|---|---|---|
| BytecodeExceeds64KLimitForClinit | <0.1% | 2 | 1:23:04 | no |
| DuplicateCase | 0.2% | 8 | 0:13:39 | no |
| DuplicateModifierForMethod | <0.1% | 1 | 0:12:06 | no |
| VoidMethodReturnsValue | 0.2% | 7 | 0:09:13 | no |
| UndefinedConstructorInImplicitConstructorCall | 0.6% | 20 | 0:07:52 | yes |
| InvalidClassInstantiation | 0.3% | 11 | 0:06:45 | no |
| ShouldReturnValue | 4.9% | 177 | 0:06:25 | no |
| Undefined: The project cannot be built until build path errors are resolved, or type already defined | 0.2% | 6 | 0:06:08 | no |
| NonStaticAccessToStaticMethod | 0.1% | 5 | 0:06:03 | yes |
| IncompatibleTypesInEqualityOperator | 0.5% | 18 | 0:05:13 | no |
| UnhandledException | 0.4% | 13 | 0:04:34 | no |
| NotVisibleField | 0.8% | 27 | 0:04:15 | yes |
| PackageIsNotExpectedPackage | 3.2% | 115 | 0:04:11 | no |
| UndefinedMethod | 8.5% | 355 | 0:03:59 | no |
| StaticMethodRequested | 0.5% | 19 | 0:03:51 | yes |
| ParsingErrorMisplacedConstruct | 0.9% | 33 | 0:03:40 | no |
| DuplicateBlankFinalFieldInitialization | <0.1% | 1 | 0:03:35 | no |
| InvalidOperator | 2.0% | 72 | 0:03:26 | no |
| ExpressionShouldBeAVariable | 0.2% | 6 | 0:03:21 | no |
| AbstractMethodMustBeImplemented | 2.5% | 89 | 0:03:10 | yes |

**Table 6.23.** *Participant 68 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| BytecodeExceeds64KLimitForClinit | 1% | 1 | 1:23:04 | 0:00:00 | |
| UninitializedLocalVariable | 1% | 2 | 0:17:58 | 0:17:22 | |
| NonStaticAccessToStaticField | 5% | 9 | 0:07:56 | 0:03:20 | |
| NonStaticAccessToStaticMethod | 2% | 3 | 0:07:56 | 0:03:20 | |
| IllegalModifierForClass | 1% | 1 | 0:07:45 | 0:00:00 | |

The five error types that Participant 68 took the longest to fix were: BytecodeExceeds64K-LimitForClinit (1:25:04), UninitializedLocalVariable (0:35:20), PackageIsNotExpectedPack-age (0:12:08), NonStaticAccessToStaticMethod (3 errors, 0:12:03 each), and NonStaticAc-cessToStaticField (0:12:03).

It was while working on assignment 2 that participant 68 encountered the "BytecodeExceeds64K-LimitForClinit" error, which was presented to the student as: "The code for the static initial-izer is exceeding the 65535 bytes limit." The student spent some time trying to determine the cause of the error. He selected the problem in the Gild problem view, and eventually went to Google in the internal web browser, likely to search for help. The participant even went to the Help menu's search functionality, though the error was not fixed until sometime later. This student was visibly struggling to understand and fix this uncommon error.

**Table 6.24.** *Participant 85 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| PackageIsNotExpectedPackage | <1% | 1 | 0:12:02 | 0:00:00 | |
| ShouldReturnValue | 12% | 55 | 0:11:52 | 0:12:44 | |
| IncompatibleTypesInEqualityOperator | <1% | 1 | 0:08:42 | 0:00:00 | |
| UndefinedName | 31% | 141 | 0:05:26 | 0:08:52 | 0:01:25 |
| ParsingErrorInvalidToken | 2% | 8 | 0:04:58 | 0:10:05 | 0:00:51 |

The five error types that Participant 85 took the longest to fix were: ShouldReturnValue (3 errors, 0:46:33 each), ParsingErrorDeleteToken (0:40:13), TypeMismatch (0:34:22), ParsingErrorInsertToComplete (2 errors, 0:33:58 each), and ParsingErrorInvalidToken (0:31:31).

The "PackageIsNotExpectedPackage" error which had the longest mean fix time only occurred once, while the student was working on assignment 4, and the fix time noted is likely longer than it really was. The file in question was edited and saved shortly after the error message appeared, but code was not built for 12 minutes. The "ShouldReturnValue" errors, which have the longest fix times of any errors, occurred at the same time as several other syntax errors (parse errors, misplaced constructs). These parsing errors may have been the cause of the "ShouldReturnValue" errors, or they may have been seen as more important to fix first.

**Table 6.25.** *Participant 102 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| InvalidClassInstantiation | <1% | 1 | 0:11:30 | 0:00:00 | |
| ExpressionShouldBeAVariable | <1% | 2 | 0:08:58 | 0:00:00 | |
| NoMessageSendOnBaseType | <1% | 1 | 0:06:17 | 0:00:00 | |
| MiscBuildErrors (path or type already defined) | 1% | 6 | 0:06:08 | 0:05:39 | |
| UndefinedType | 3% | 32 | 0:05:42 | 0:05:27 | |

The five error types that Participant 102 took the longest to fix were: TypeMismatch (2 errors: 0:31:51, 0:25:46), UndefinedConstructor (0:31:51), UndefinedMethod (0:25:12), ParameterMismatch (2 errors, 0:25:12 each), and UndefinedName (0:22:30).

**Table 6.26.** *Participant 119 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| UnhandledException | <1% | 1 | 0:15:42 | 0:00:00 | |
| DuplicateCase | 2% | 8 | 0:13:39 | 0:00:00 | |
| NotVisibleField | 1% | 5 | 0:12:25 | 0:03:25 | |
| DuplicateModifierForMethod | <1% | 1 | 0:12:06 | 0:00:00 | |
| UndefinedType | 1% | 4 | 0:06:46 | 0:07:22 | |

The five error types that Participant 119 took the longest to fix were: UndefinedType (0:19:14), UndefinedName (8 errors: 7 at 0:17:41, 1 at 16:39), UnhandledException (0:15:42), UndefinedMethod (0:15:29), and ShouldReturnValue (0:14:34).

**Table 6.27.** *Participant 136 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| PackageIsNotExpectedPackage | 2% | 2 | 2:24:39 | 0:00:00 | |
| UndefinedMethod | 2% | 2 | 2:13:34 | 0:00:00 | |
| VoidMethodReturnsValue | 1% | 1 | 0:28:40 | 0:00:00 | |
| MethodRequiresBody | 1% | 1 | 0:28:40 | 0:00:00 | |
| UndefinedConstructorInImplicit ConstructorCall | 10% | 12 | 0:11:12 | 0:10:42 | |

The five error types that Participant 136 took the longest to fix were: PackageIsNotExpectedPackage (2 errors, 2:24:39 each), UndefinedMethod (2:13:34 - 2 errors), UndefinedConstructorInImplicitConstructorCall (2 errors, 0:28:40 each), VoidMethodReturnsValue (0:28:40), and MethodRequiresBody (0:28:40).

This student had many compiler errors that took a while to fix, and appeared to struggle to fix his errors more than other participants in this study. The PackageIsNotExpectedPackage error and the UndefinedMethod with the longest fix times occurred while the student was working on assignment 1. The student built their code fairly often during this time, and the PackageIsNotExpectedPackage error was usually not the only error present.

**Table 6.28.** *Participant 153 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| PackageIsNotExpectedPackage | 19% | 3 | 0:05:30 | 0:00:00 | |
| ArgumentHidingField | 6% | 1 | 0:05:30 | 0:00:00 | |
| ParsingError | 24% | 4 | 0:02:35 | 0:03:59 | 0:00:17 |
| InvalidClassInstantiation | 6% | 1 | 0:01:34 | 0:00:00 | |
| AbstractMethodMustBeImplemented | 6% | 1 | 0:01:34 | 0:00:00 | |

The five error types that Participant 153 took the longest to fix were: ParsingError (0:09:28), ArgumentHidingField (0:05:30), PackageIsNotExpectedPackage (0:05:30), InvalidClassInstantiation (0:01:34), and AbstractMethodMustBeImplemented (0:01:34).

The ParsingError with the longest fix time was due to a package error. There was very little data for this participant, but package errors are the problems that took this student the

longest to solve.

**Table 6.29.** *Participant 170 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| ParsingErrorMisplacedConstruct | 1% | 8 | 0:09:14 | 0:16:56 | 0:01:05 |
| IncompatibleTypesInEqualityOperator | 1% | 11 | 0:06:59 | 0:07:25 | |
| NotVisibleField | 1% | 5 | 0:06:53 | 0:02:56 | |
| StaticMethodRequested | 1% | 8 | 0:06:41 | 0:08:31 | |
| ArrayReferenceRequired | <1% | 3 | 0:05:58 | 0:03:28 | |

The five error types that Participant 170 took the longest to fix were: ParsingErrorMisplacedConstruct (0:52:51), ParsingError (0:51:22, 0:50:22), UndefinedMethod (2 errors, 0:34:09 each), and ParsingErrorInsertToComplete (2 errors, 0:31:39), StaticMethodRequested (0:24:19).

The 11 IncompatibleTypesInEqualityOperator errors occurred while this student was working on assignment 6, and was providing an Object where and int was expected. This error appears on Participant 170's longest mean fix time list, despite the low number of type errors that this participant made. This error occurred while the student had many errors referring to static references for non-static fields and methods, which may have taken precedence.

**Table 6.30.** *Participant 204 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| VoidMethodReturnsValue | 2% | 1 | 0:28:35 | 0:00:00 | |
| UndefinedMethod | 12% | 7 | 0:10:18 | 0:12:06 | 0:05:18 |
| ParsingError | 7% | 4 | 0:04:25 | 0:06:19 | 0:00:57 |
| TypeMismatch | 7% | 4 | 0:03:20 | 0:02:52 | |
| ParsingErrorInsertTokenAfter | 3% | 2 | 0:02:51 | 0:02:02 | |

The five error types that Participant 204 took the longest to fix were: UndefinedMethod (0:29:14 - 2 errors, 0:05:18 - 2 errors), VoidMethodReturnsValue (0:28:35), ParsingError(0:15:21, 0:07:51), ParsingErrorInsertTokenAfter (0:04:53), and UndefinedConstructor (0:03:58).

The UndefinedMethod error was the one error that this student viewed the extra error help for, but this student did not fix the error until approximately 29 minutes later. The VoidMethodReturnsValue error is more likely a semantic error, but may also have been due to insufficient planning. For example, the student may have wanted a method to return a value, but neglected to specify that in the method signature, so the student left the return value as was until the signature was finalized.

**Table 6.31.** *Participant 238 - Longest mean fix times for compiler errors*

| Error Type | Errors (%) | Errors (#) | Mean h:m:s | stdev h:m:s | Median h:m:s |
|---|---|---|---|---|---|
| UndefinedConstructorInImplicit ConstructorCall | 1% | 3 | 0:05:48 | 0:03:44 | |
| AssignmentHasNoEffect | <1% | 1 | 0:05:34 | 0:00:00 | |
| UndefinedConstructor | 2% | 6 | 0:04:05 | 0:03:27 | |
| AbstractMethodMustBeImplemented | 10% | 35 | 0:03:35 | 0:02:58 | |
| NonStaticAccessToStaticMethod | 1% | 2 | 0:03:14 | 0:03:08 | |

The five error types that participant 238 took the longest to fix were: ParameterMismatch (0:11:39), AbstractMethodMustBeImplemented (0:11:30), UndefinedMethod (0:10:35), UndefinedConstructorInImplicitConstructorCall (0:09:39), and UndefinedName (5 errors, 0:08:47 each).

It was noted in the participant summary for this student that code was compiled frequently and problems were often fixed soon after they arose. This may have had an effect on the computed fix times for this student. Longest mean fix times for compiler errors for this student are shorter than for most other participants in this study.

## 6.3 Do the Errors Change over Time?

To answer this question, I considered the top 50% of errors made on each assignment for the three participants who had sufficient data from at least four assignments. Data from participants 85, 102 and 170 is found in this section. Errors that comprise the top 50% of the errors for an assignment are shown in bold in the table. In addition to the data summarized in the tables, there is also a graph representation of the data.

**Table 6.32.** *Participant 85 - Changes in errors between assignments*

| | Assign 1 | | Assign 2 | | Assign 3 | | Assign 4 | |
|---|---|---|---|---|---|---|---|---|
| **Error type** | **%** | **#** | **%** | **#** | **%** | **#** | **%** | **#** |
| UndefinedName | **35%** | **13** | **50%** | **87** | **13%** | **12** | **17%** | **20** |
| ParsingErrorInsertToComplete | **24%** | **9** | 4% | 7 | 2% | 2 | 6% | 7 |
| ShouldReturnValue | 5% | 3 | 10% | 17 | **32%** | **31** | 4% | 5 |
| ParameterMismatch | n/a | n/a | n/a | n/a | **9%** | **9** | 1% | 1 |
| UndefinedMethod | 8% | 3 | 9% | 15 | 1% | 1 | **30%** | **34** |
| **Total for all errors** | | 35 | | 175 | | 96 | | 115 |

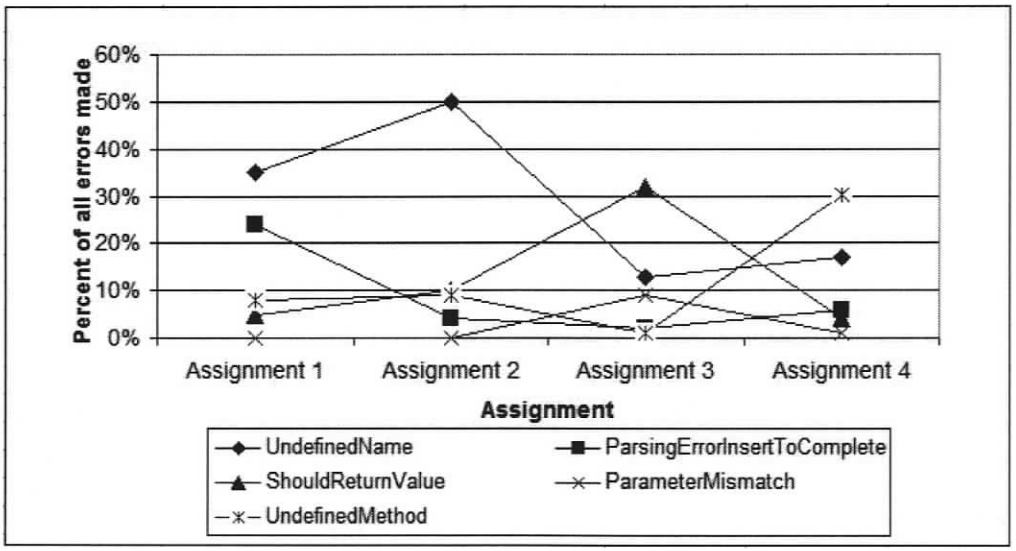



**Figure 6.1.** *Participant 85 - Top 50% of errors per assignment (percentage).*

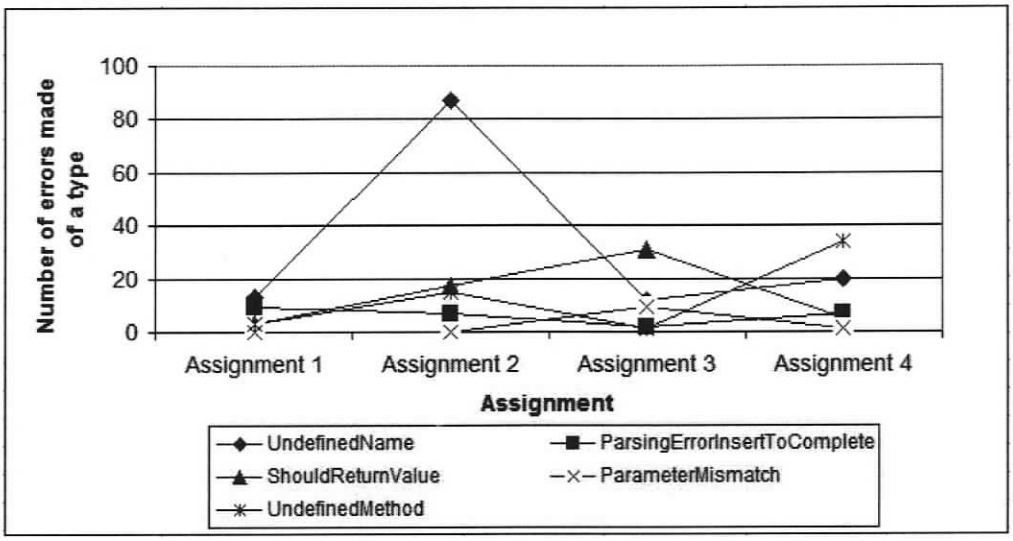



**Figure 6.2.** *Participant 85 - Top 50% of errors per assignment (count).*

**Table 6.33.** *Participant 102 - Changes in errors between assignments*

| | Assign 1 | | Assign 2 | | Assign 3 | | Assign 4 | |
|---|---|---|---|---|---|---|---|---|
| **Error type** | **%** | **#** | **%** | **#** | **%** | **#** | **%** | **#** |
| UndefinedName | **44%** | **147** | **13%** | **45** | **17%** | **37** | **14%** | **12** |
| TypeMismatch | **14%** | **45** | **22%** | **76** | **13%** | **28** | 4% | 3 |
| ParsingErrorInsertToComplete | 3% | 11 | **10%** | **33** | 7% | 15 | 5% | 4 |
| UndefinedConstructor | n/a | n/a | **8%** | **27** | 6% | 12 | n/a | n/a |
| UndefinedMethod | 1% | 2 | 6% | 19 | **19%** | **40** | 6% | 5 |
| AbstractMethodMustBeImplemented | n/a | n/a | n/a | n/a | **12%** | **25** | n/a | n/a |
| ShouldReturnValue | 2% | 6 | 6% | 19 | 2% | 4 | **30%** | **25** |
| PackageIsNotExpectedPackage | n/a | n/a | n/a | n/a | n/a | n/a | **11%** | **9** |
| UndefinedType | <1% | 1 | n/a | n/a | 2% | 5 | **11%** | **9** |
| **Total for all errors** | | 332 | | 339 | | 214 | | 83 |

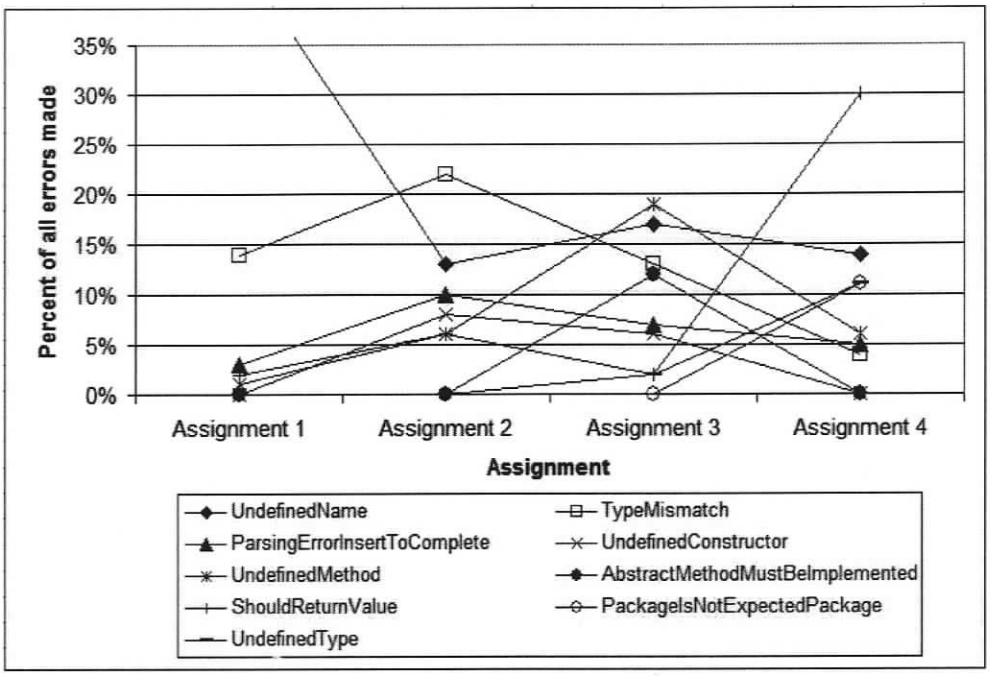



**Figure 6.3.** *Participant 102 - Top 50% of errors per assignment (percentage).*

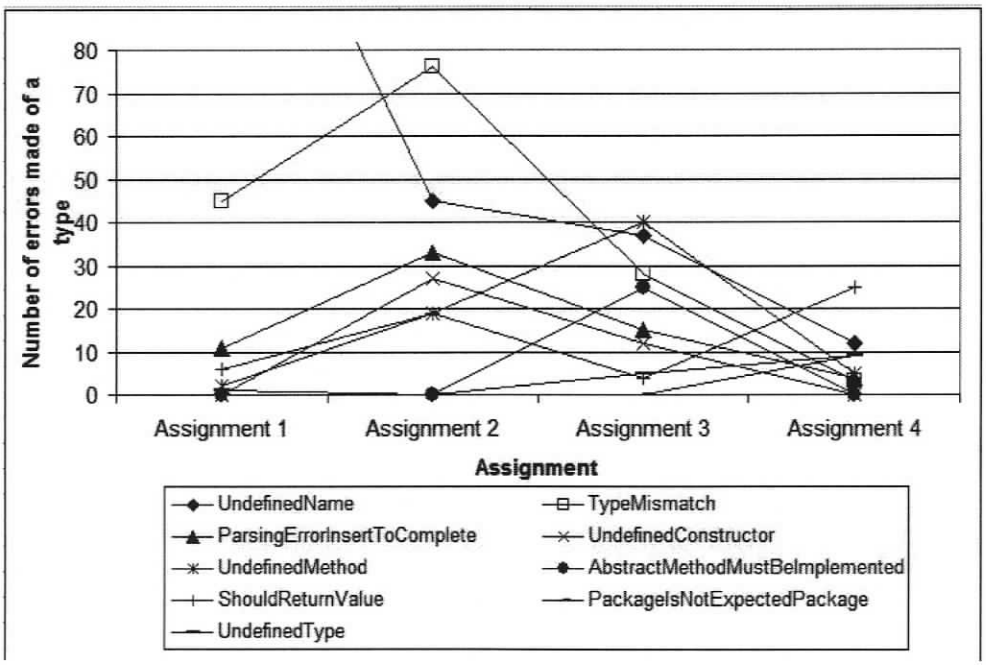



**Figure 6.4.** *Participant 102 - Top 50% of errors per assignment (count).*

**Table 6.34.** *Participant 170 - Changes in errors between assignments*

| | Assign 2 | | Assign 3 | | Assign 4 | | Assign 5 | | Assign 6 | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Error type** | **%** | **#** | **%** | **#** | **%** | **#** | **%** | **#** | **%** | **#** |
| UndefinedName | **38%** | **21** | 3% | 3 | **22%** | **38** | **27%** | **44** | **15%** | **30** |
| UndefinedMethod | **18%** | **10** | 1% | 1 | 1% | 1 | 10% | 16 | **12%** | **24** |
| ParsingErrorInsertToComplete | 9% | 5 | **28%** | **28** | **12%** | **21** | **12%** | **19** | **14%** | **27** |
| UndefinedConstructor | n/a | n/a | **10%** | **10** | n/a | n/a | 2% | 4 | 3% | 6 |
| ParsingError | n/a | n/a | **9%** | **9** | **9%** | **16** | n/a | n/a | n/a | n/a |
| ParsingErrorInsertTokenAfter | 2% | 1 | **8%** | **8** | 4% | 7 | 2% | 4 | 3% | 5 |
| UndefinedType | 9% | 5 | n/a | n/a | **10%** | **17** | **13%** | **21** | **13%** | **25** |
| **Total for all errors** | | 55 | | 101 | | 175 | | 161 | | 198 |

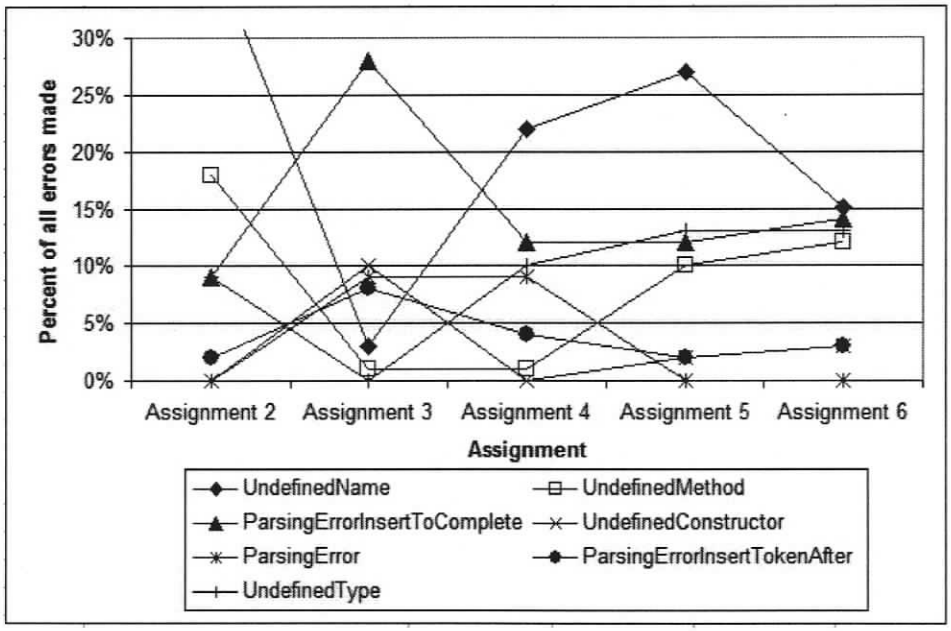



**Figure 6.5.** *Participant 170 - Top 50% of errors per assignment (percentage).*

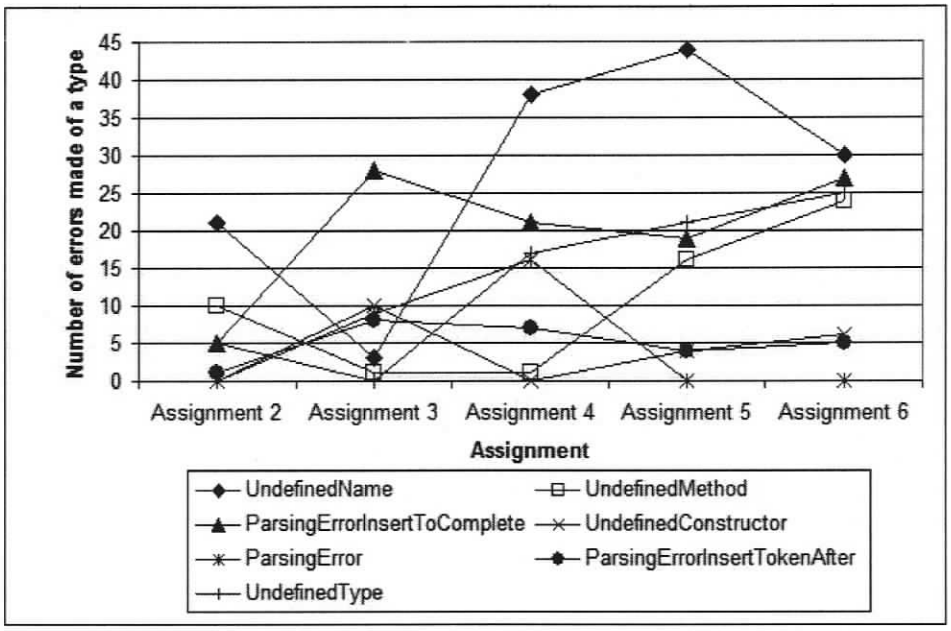



**Figure 6.6.** *Participant 170 - Top 50% of errors per assignment (count).*

# Chapter 7

# Synthesis of Results and Discussion

In this chapter I provide a synthesis of some of the results from the previous chapter, and discuss some of the implications of the qualitative and quantitative data. Each section in this chapter addresses one of my research questions.

## 7.1 What are the Most Common Errors that Students Make?

Despite the small sample size of my study, results indicate that a small number of error types accounted for most of the errors made by students is consistent with the findings of [7, 8, 10, 19]. On average, across all students, five errors accounted for at least 50% of the errors made, while closer to 9 errors accounted for at least 70% of the total errors.

### 7.1.1 Compilation Errors

The compilation errors made by the students appear to be primarily syntactic errors, though errors more likely to be semantic errors are also encountered by participants. The "UndefinedName" error was the most common error for seven of the participants. This syntactic error occurs when a variable cannot be resolved, which can be due to forgetting to declare the type of a variable, or misspelling the name of a variable that has been declared. The "UndefinedType" and "UndefinedMethod" errors that occur frequently can also be due to a mistyping, which is also a syntax error. The "ParsingErrorInsertToComplete" error was one of the most common parsing errors, and was a frequent error for all nine participants.

This error can occur due to a missing semi-colon at the end of a statement, or unbalanced parentheses or braces. The other parsing errors can also be due to unbalanced parentheses or braces, or omitting an operator, again suggesting a syntax error.

The "ShouldReturnValue" error was one of the more frequent errors for seven of the participants, and occurs when a method they have defined does not return a value despite the signature of the method specifying that it does return a value. This error is also a syntactic error; students likely forgot to return something because the method implementation was incomplete when the program was compiled. The "AbstractMethodMustBeImplemented" error is similar to the 'ShouldReturnValue" error, since the implementation for a particular method was simply lacking when the program was built.

"TypeMismatch" errors, which were frequent errors for six participants, occur when a value of one type is assigned to a variable of an incompatible type, or if the assignment operator, "=", is used in an conditional statement instead of the equality operator: "==". Such an error would be a semantic error if the student made the mistake of assigning two incompatible types. The incorrect use of the assignment operator in a conditional statement is a syntax error, but it can be indicative of a misconception about the differences between the assignment and equality operators.

The "UndefinedConstructor", and 'UndefinedMethod" errors can occur due to a misspelling of the method or constructor name, and is a syntax error. The "ParameterMismatch" error is due to an incorrect number of arguments specified when calling a method, which is also a syntax error.

The "NoMessageSendOnArrayType" error is an example of a semantic error. It occurs when a programmer attempts to call a method on an array of some object, except arrays are not an object, but rather a container of objects, and method calls are not applicable to arrays. The log files show an instance where Participant 102 called a method on a String array, which resulted in a "NoMessageSendOnArryType" error. The student then attempted to call the same method on a String object, though the method was not defined for the String object, but rather the MusicFile object. This particular error demonstrates the

confusion that some students have with objects.

The "NonStaticAccessToAStaticField" error is in fact a warning from the compiler, and will not prevent a programmer from running their code. It is a semantic error, and occurs when a programmer attempts to access a field through the instance of an object instead of through the class itself. Such a warning indicates a lack of knowledge of the purpose of the static modifier.

The package error "PackageIsNotExpectedPackage" was a frequent error for four participants. Since students in CSC 115 are not explicitly taught the concept of packages, this error was likely frustrating for those who encountered it. The students who encountered this error often did so due to importing and exporting their files incorrectly into a new project, therefore leading to incorrectly named packages. This error was likely exasperated by the use of the Gild IDE, since the underlying concept of packages that exists in Eclipse is not removed in Gild. Students opting to edit their files in a more simple editor such as TextPad likely did not experience package errors, but I cannot say for certain if that was the case.

### 7.1.2 Runtime Exceptions

Three runtime exceptions accounted for over 93% of the exceptions made by all participants. These included: null pointer, array index out of bounds, and unresolved compilation problem(s). These results are not suprising, as they are recognised as common run time exceptions. These exceptions can often occur due to problems with boundary conditions; boundary conditions are important for both novices and experts alike to test. The instances of a particular type of run time exception did not appear to be specific to an assignment. For most students who encountered unresolved compilation problems, these exceptions occurred more frequently in earlier assignments, although Participant 68 continued to encounter that error due to his use of the Run button instead of building his code first.

The runtime exceptions encountered by the participants were varied, partially due to the different work units that students allowed to be logged. An interesting result was the use of

the debugger by participants in order to determine the exact cause of their runtime exceptions. Such debugging techniques are important for programmers, and it is promising that participants 153, 170, and 238 used the debugger to pinpoint the cause of their exceptions.

## 7.2 How Long Does it Take for Students to Fix their Errors?

Despite the fact that I only have approximate error fix times, these times provide some insights about where students may be struggling. The error fix times that I am working with are the creation and fix times from the participant's perspective: until the participant has compiled, they do not know that the error exists or has been fixed. Only direct observation or a more detailed screen recording or logging could show the true error creation and fix times. Jadud observed that "when students have just encountered a syntax error, they are likely to recompile quickly" [10]. Based on readings of the participants' log files, this is likely the case with the students that were monitored in my study. Periods of many rapid, consecutive compilations are observed as the participant would attempt to fix a series of errors. While this was not the focus of my research questions, it is a finding that I agree with. An implication of this behaviour is that the error fix times that were recorded are more likely to be accurate, as the observed fix time would be close to the actual fix time.

The most frequent compilation errors were fixed fairly quickly, usually within a few minutes or seconds. It is interesting that the errors that took the longest for students to fix were also less frequently made, and were usually semantic errors. Many of these errors demonstrate a misunderstanding or a difficulty that the student is having with a programming concept or construct. It is unfortunate that the "PackageIsNotExpectedPackage" error was the error that took three students, on average, the longest to resolve, since this error was likely exasperated due to the use of Gild/Eclipse. This suggests that Gild does not provide sufficient support for helping students to understand packages, and package naming.

## 7.3 Do Students' Errors Change over the Course of the Semester?

Error patterns over time are unique to each of the three participants. Most error types spiked for a particular assignment, while a few error types increased or decreased in later assignments. For a given assignment, the errors that were most frequent for one participant were often not the same as the errors for another participant. Participants do appear to avoid making the same error type after a spike in the occurrences of that error.

The perennial favourite syntax error: "UndefinedName", continued to be a frequent error for all participants. It was only for participant 102 that the occurrences of this error decreased. For participant 170, "ParsingErrorInsertToComplete" was a common error over most assignments. As for errors that increase, both participants 102 and 170 began to encounter more "UndefinedType" errors around assignment 4. An implication of the personal uniqueness of the error types is that designers of error support tools should not necessarily assume that a particular course topic is associated with a particular frequent error type.

It is difficult to conclude if students became faster at fixing their errors as the semester progressed. The average error fix times between assignments showed no obvious trends. Although one would assume that students would become faster at fixing their most common errors, other factors may have caused later fix times to remain similar to earlier fix times.

## 7.4 Where do Students go for Extra Help for an Error they do not Understand?

We have used many different methods for evaluating the success of the Gild IDE and to gain further insight into students' programming habits. Every semester that Gild was used at the University of Victoria, we surveyed the students through a questionnaire. Interviews, user observations, and an ethnographic study were also conducted.

In this section and the following section two questions from a questionnaire administered at the end of the spring semester of 2006 are discussed. Participants were enrolled in Computer Science (CSC) 115, a second course of Java programming. This group of participants was the same group that was invited to participate in the logging study discussed in the following chapters. The students in CSC 115 were separated into two classes. The response rate from one class was 25 of 42 students registered, (60% response rate), while the response rate for the other class was 34 of 70 students registered (49% response rate). Both questions discussed in this thesis are related to programming errors, and in the case of the second question, specifically to the Gild extra error support feature. A copy of the questionnaire is available in Appendix B.

The specific question posed to students was: "When programming, what do you do when you encounter an error that you know nothing about? (Where do you go for help?)" The motivation for this question was to learn more about the problem solving resources that students turn to, and to determine if Gild provided features that did assist students with the difficult programming situation of an unknown error.

This question is similar to a question asked in the Gild requirements gathering survey of first and second year students that was done in April 2003. In that survey, students were asked to rank a list of 9 problem solving strategies in terms of their usefulness: I ask my friends, I visit my TA, I visit my professor, I go to the Help Desk, I use the course web site, I email my TA, I email my professor, I use the lecture material, and I look at similar code. It was found that looking at similar code and asking friends was ranked very highly, followed by the help desk (consultants). The less useful problem solving strategies were: to visit the TA, go to the help desk (consultants), email the TA, and email the professor.

In this survey the question was more specifically directed toward unknown errors, but it was designed for open-ended responses. Encountering an error that a student does not know anything about is arguably one of the most frustrating situations to be in when programming. The question was intentionally stated vaguely, since I wanted students to consider any error, whether it is a compiler, run time, or logic error. Fifty-eight students responded

to this question. For a list of the student responses, and the response classifications, refer to Appendix C.

Google or the Internet (which includes the Java API) was by far the most popular place for students to go for help (31 students). The course web board, where students could post questions and answers, was specifically mentioned by eight students. The Gild web browser is a feature that supports these forms of problem solving. Talking to the computer science help desk consultants and to other students was each mentioned by 10 and nine students respectively. These forms of problem solving usually involve face to face communication. Talking to friends, unlike talking to the consultants at the help desk, is something that could be done remotely. Gild does not yet include features for remote communication with other students, though an instant messaging component has been suggested. The ability to share code in a distributed manner may also be beneficial to these students. Fourteen students responded that they would persevere, or just look at the code. Finally, debugging was mentioned by 7 students, which is a feature that has been implemented in Gild since its first release.

This question cannot be directly compared to the question asked in the 2003 requirements gathering survey, though it does bring up the possibility that when students said they favoured looking at code, it could be that this code was sometimes found online. It could also be that in the last 3 years, the growth of the Internet in conjunction with Google's search abilities has made online resources more numerous and attainable, thus changing students' problem solving techniques. Although Google and the Internet was the most popular choice to go to for help, the fact that no particular site except for the course web site and the Java API was mentioned, indicates that there was no single good source for help for many of their problems.

Another interesting result from this question was that only one person mentioned going to the program they were using for help. The common style of help menus for most software is for the help menu to contain information about how to use the software, not help for what the software is used for. In the case of Gild, the help menu would ideally supply

students with information on Java and programming principles. This would be no small effort, given the large amount of (varying) material that is taught to first year programmers.

Unfortunately, I cannot determine how useful these sources were for students. One can anticipate that their very use indicates that they were helpful to some extent.

## 7.5 Does Gild's Extra Error Help Feature Aid Students?

This question: "Was the extra error help feature useful? Why or why not?" was asked to students who had used the Gild IDE at some point in the semester. There were 28 students who provided responses to this question. For a list of the student responses, and the response classifications, refer to Appendix C.

Eleven students agreed that the feature was useful, while a further three students found that although useful, the feature was more useful at the beginning of the semester. Two students felt that it was only sometimes useful. Four students responded that they were not aware of the extra error help feature, while five more never used it. Two students felt that the feature was not useful at all. Despite having a picture supplied of the feature beside the question, some of the students may have been including the problem view navigation when forming their responses. For example, one student responded: "Yes! Sometimes errors are hard to fix + this pinpointed where they were." It is unclear if the "pinpointing" is a reference to the location of the error or to the cause. Since the extra error help was static in nature, the former seems most likely. It is also likely that some students had switched to the Java perspective, in which case the Gild problem view would not be in these students' workspaces.

The student response to the extra error help feature was positive, with 57% of respondents finding the feature useful at least some of the time. Thirty-two percent of respondents were either not aware of the feature or had not used it, while only 7% found it not useful. The students who found the feature not useful felt that the feature didn't provide any "valuable information about the errors", or that "most (99%) of the time the extra help didn't

show up or was completely unhelpful". These comments indicate a need for more specific error messages, and better coverage of the errors that are most encountered by students.

### 7.5.1 Match between Gild Extra Help and Errors Made by Students

Of the 10 error types that accounted for over 70% of all student errors made, nine of the errors had an extra error help message defined in the Gild extra error help feature (See Table 6.2.) A complete list of all errors and extra help messages that were created for Gild are available in Appendix A. The 10th most common error, "ParameterMismatch", was the only error in the top 10 without extra error support. Of the 16 error types that accounted for over 80% of all student errors made, only two errors in addition to the "ParameterMismatch" error did not have an extra error help message: InvalidOperator, and UndefinedField.

Although many of the most frequent errors had extra help available, there was little extra error help for the errors that took the students, on average, the longest to fix. Of the 20 error types with the longest average fix times, over all participants, only 5 errors had extra help available (See Table 6.2.) The 20 error types with the longest fix times accounted for 26% of all errors. Based on some of the student feedback from the survey, the students who were disappointed with the extra error help may have benefitted from help for these types of errors.

# Chapter 8

# Future Work and Conclusions

## 8.1 Future Work

This work provides information about some of the programming errors that novice programmers are encountering. User stories provide further details about student interaction with the Gild IDE. This information is important not only for the developers of the Gild IDE, but also for developers of other software for helping novices programmers with their programming tasks. Future work is focused on better understanding the errors that programmers are struggling with, and ways to improve tool support.

### 8.1.1 Tool Improvements

Some of the first work to be done will be to improve the extra error help messages in Gild to include messages for the errors that students take the longest to fix. Since it is likely that these errors are also the ones that students struggle with the most, this could increase the usefulness of the feature. It may also be useful to make the error messages more specific, by, for example, referring to problem variables or objects by name.

Based on the initial usages of the Gild IDE by participants, we will consider ways to improve the initial user experience, so that more guidance is available to students. There is little evidence that students referred to the Gild tutorial available in the Help menu. Since some students incorrectly compiled their code by using the Run or Debug button, and some features, such as the extra error help, were not obvious to students, some type of initial tour

may be useful. Initial, temporary, pop-up information for the different features could be an alternative solution.

### 8.1.2 Future Studies and Study Improvements

A larger scale study would provide more statistical significance to the research questions that are addressed with frequency-based results. Although many similarities were found in the most common errors, further investigation to undercover any themes in the errors that take students the longest to fix, would be beneficial. Detailed analysis like that done in Chapter 5 would be difficult in a larger study, however. Although many CSC 115 students downloaded and used the Gild IDE during the semester, only 10 provided log files. In order to obtain more information on errors, future logging-based studies may need to offer some form of compensation. Another useful study would be to determine if the time to fix the more difficult errors is reduced due to Gild's extra error help feature.

The Mylar Monitor logging plug-in and the Log Analyzer program were both useful for this study. The additional functionality that was added to the Mylar Monitor should be further improved to better determine the current compile errors. A list of all errors displayed at every compilation would be simple to implement and less prone to error than the current version. It would also be useful to have a snapshot of the lines of code surrounding a particular error, or the files in their entirety, at each compilation. This would result in a very large amount of data, but would give more context to the errors that students are struggling with. The data analysis process should be further automated. The use of a database instead of perl scripts for analyzing data would allow for more varied queries.

There may be a correlation between a low percentage of TypeMismatch compilation errors and programming ability or understanding. A future study should investigate any possible link by obtaining some metric for student understanding and ability to program, and the occurance of TypeMismatch errors.

Future work could also involve extending this study to more experienced programmers. By learning how quickly more experienced programmers fix their errors we can gain a

better idea of how much novices are struggling with their errors relative to experienced programmers. Tools for helping first year students understand their compiler errors may also be beneficial for programmers who are experienced, but new to the Java programming language.

## 8.2 Conclusions

This work demonstrates that software logging can be an effective method for acquiring data that goes beyond frequency counts. The user stories provide useful information about student experiences: what features students used, and how they used them.

Results from this work suggest that errors due to misconceptions about programming concepts are occurring; these errors, which are semantic in nature, are taking students longer to fix than the more frequently made syntax errors. Design issues and a lack of knowledge about an IDE can also be an obstacle for students. Concepts such as package structures and the save/build/run process challenged several students. Surveys indicate that providing extra error help is beneficial to student understanding of errors, though monitored student usage does not show that this feature was extensively used.

### 8.2.1 Contributions

This work is intended to be a resource for instructors of introductory computer programming courses, and developers of tools for novice programmers. The contributions of this work include:

1. A listing of the common compilation and runtime errors that students encounter.
2. Approximate fix times for compilation errors, which provides insight into which types of errors students take the longest to fix.
3. Descriptions of students' first use of Gild, and interesting aspects of their usage of Gild.

4. Suggestions for developers of tools for novice programmers.
5. Suggestions for improvements to Gild features, in particular the extra error support.
6. Fifty-one extra help messages for novice compiler errors (Appendix A)

# Bibliography


[1] SFU, "Learning kit/gstudy educational technology research project," 2006. [Online]. Available: http://www.learningkit.sfu.ca

[2] J. Nielsen and R. L. Mack, *Usability Inspection Methods*. Wiley, April 1994. [Online]. Available: http://www.amazon.fr/exec/obidos/ASIN/0471018775/citeulike04-21

[3] M. Mccracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz, "A multi-national, multi-institutional study of assessment of programming skills of first-year cs students," in *ITiCSE-WGR 2001: Working group reports from ITiCSE on Innovation and technology in computer science education*. New York, NY, USA: ACM Press, 2001, pp. 125–180. [Online]. Available: http://dx.doi.org/10.1145/572133.572137

[4] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hamer, M. Lindholm, R. Mccartney, J. E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas, "A multi-national study of reading and tracing skills in novice programmers," in *ITiCSE-WGR 2004: Working group reports from ITiCSE on Innovation and technology in computer science education*, vol. 36, no. 4. New York, NY, USA: ACM Press, December 2004, pp. 119–150. [Online]. Available: http://dx.doi.org/10.1145/1044550.1041673

[5] S. Wiedenbeck, V. Fix, and J. Scholtz, "Characteristics of the mental representations of novice and expert programmers: an empirical study," *Int. J. Man-Mach. Stud.*, vol. 39, no. 5, pp. 793–812, November 1993. [Online]. Available: http://dx.doi.org/10.1006/imms.1993.1084

[6] N. Truong, P. Roe, and P. Bancroft, "Static analysis of students' java programs," in *CRPIT '30: Proceedings of the sixth conference on Australian computing education*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, pp. 317–325. [Online]. Available: http://portal.acm.org/citation.cfm?id=980011

[7] J. Jackson, M. Cobb, and C. Carver, "Identifying top java errors for novice programmers," vol. 1, October 2005, pp. T4C–24–T4C–27. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1611967

[8] M. Ahmadzadeh, D. Elliman, and C. Higgins, "An analysis of patterns of debugging

among novice computer science students," in *ITiCSE 2005: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. New York, NY, USA: ACM Press, 2005, pp. 84–88. [Online]. Available: http://dx.doi.org/10.1145/1067445.1067472

[9] T. Flowers, C. A. Carver, and J. Jackson, "Empowering students and building confidence in novice programmers through gauntlet," in *Frontiers in Education Conference*, vol. 1. ASEE/IEEE, October 2004, pp. T3H/10–T3H/13. [Online]. Available: http://ieeexplore.ieee.org/search/selected.jsp?qry=%28%28empowering+students+and+building+confidence+in%29%3Cin%3Emetadata%29\&srch=1\&resset=1293212\&chklist=1408551%40ieeecnfs\&submit=View+Selected+Items

[10] M. C. Jadud, "A first look at novice compilation behavior using bluej," in *16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004)*. Carlow, Ireland: Institute of Technology, April 2004. [Online]. Available: http://www.jadud.com/people/mcj/files/2004-PPIG-flcbBlueJ.pdf

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

[12] U. Brown, "Cs15 online ta: Compiler error explanations," 2006. [Online]. Available: http://www.cs.brown.edu/courses/cs015/ref/CompilerErrors.shtml

[13] N. Ziring, "Novice java programmers' favourite mistakes," 2001. [Online]. Available: http://users.erols.com/ziring/java-npm.html

[14] R. W. Topor, "Cit1104 programming ii: Common (java) programming errors," 2002. [Online]. Available: http://www.cit.gu.edu.au/~rwt/p2.02.1/errors.html

[15] A. E. Fleury, "Programming in java: student-constructed rules," in *SIGCSE 2000: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, vol. 32, no. 1. New York, NY, USA: ACM Press, March 2000, pp. 197–201. [Online]. Available: http://portal.acm.org/citation.cfm?id=331854

[16] M. Barr, S. Holden, D. Phillips, and T. Greening, "An exploration of novice programming errors in an object-oriented environment," in *ITiCSE-WGR 1999: Working group reports from ITiCSE on Innovation and technology in computer science education*, vol. 31, no. 4. New York, NY, USA: ACM Press, December 1999, pp. 42–46. [Online]. Available: http://dx.doi.org/10.1145/349316.349392

[17] J. C. Spohrer and E. Soloway, "Alternatives to construct-based programming misconceptions," in *CHI 1986: Proceedings of the SIGCHI conference on Human factors in computing systems*, vol. 17, no. 4. New York, NY, USA: ACM Press, April 1986, pp. 183–191. [Online]. Available: http://dx.doi.org/10.1145/22627.22369

[18] J. C. Spohrer and E. Soloway, "Novice mistakes: are the folk wisdoms correct?" *Commun. ACM*, vol. 29, no. 7, pp. 624–632, July 1986. [Online]. Available: http://dx.doi.org/10.1145/6138.6145

[19] J. C. Spohrer and E. Soloway, "Analyzing the high frequency bugs in novice programs," in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. Norwood, NJ, USA: Ablex Publishing Corp., 1986, pp. 230–251. [Online]. Available: http://portal.acm.org/citation.cfm?id=28897

[20] D. Ginat, "The greedy trap and learning from mistakes," in *SIGCSE 2003: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, vol. 35, no. 1. New York, NY, USA: ACM Press, January 2003, pp. 11–15. [Online]. Available: http://dx.doi.org/10.1145/611892.611920

[21] M.-H. N. C. Vee, B. Meyer, and K. L. Mannock, "Empirical study of novice errors and error paths," August 2005. [Online]. Available: http://se.ethz.ch/~meyer/publications/teaching/novices.pdf

[22] I. M. M. Duncan and D. J. Robson, "An exploratory study of common coding faults in c programs," University of Durham, Tech. Rep., 1991.

[23] S. K. Kummerfeld and J. Kay, "The neglected battle fields of syntax errors," in *ACE 2003: Proceedings of the fifth Australasian conference on Computing education*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 105–111. [Online]. Available: http://portal.acm.org/citation.cfm?id=858416

[24] K. A. Murray, J. M. Heines, M. Kölling, T. Moore, P. J. Wagner, N. C. Schaller, and J. A. Trono, "Experiences with ides and java teaching: what works and what doesn't," in *ITiCSE 2003: Proceedings of the 8th annual conference on Innovation and technology in computer science education*, vol. 35, no. 3. New York, NY, USA: ACM Press, September 2003, pp. 215–216. [Online]. Available: http://dx.doi.org/10.1145/961511.961571

[25] J. F. Pane and B. A. Myers, "Usability issues in the design of novice programming systems," Carnegie Mellon University, Tech. Rep., August 1996. [Online]. Available: http://www.cs.cmu.edu/~pane/cmu-cs-96-132.html

[26] M.-A. Storey, D. Damian, J. Michaud, D. Myers, M. Mindel, D. German, M. Sanseverino, and E. Hargreaves, "Improving the usability of eclipse for novice programmers," in *eclipse 2003: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA: ACM Press, 2003, pp. 35–39. [Online]. Available: http://dx.doi.org/10.1145/965660.965668

[27] E. Allen, R. Cartwright, and B. Stoler, "Drjava: a lightweight pedagogic environment

for java," in *SIGCSE 2002: Proceedings of the 33rd SIGCSE technical symposium on Computer science education.* New York, NY, USA: ACM Press, 2002, pp. 137–141. [Online]. Available: http://dx.doi.org/10.1145/563340.563395

[28] F. Mueller and A. L. Hosking, "Penumbra: an eclipse plugin for introductory programming," in *eclipse 2003: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange.* New York, NY, USA: ACM Press, 2003, pp. 65–68. [Online]. Available: http://dx.doi.org/10.1145/965660.965674

[29] R. Chatley and T. Timbul, "Kenyaeclipse: learning to program in eclipse," in *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering.* New York, NY, USA: ACM Press, 2005, pp. 245–248. [Online]. Available: http://dx.doi.org/10.1145/1081706.1081746

[30] P. C. Rigby and S. Thompson, "Study of novice programmers using eclipse and gild," in *eclipse 2005: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange.* New York, NY, USA: ACM Press, 2005, pp. 105–109. [Online]. Available: http://dx.doi.org/10.1145/1117696.1117718

[31] Eclipse, "Eclipse.org home," 2006. [Online]. Available: http://www.eclipse.org

[32] C. Reis and R. Cartwright, "Taming a professional ide for the classroom," in *SIGCSE 2004: Proceedings of the 35th SIGCSE technical symposium on Computer science education.* New York, NY, USA: ACM Press, 2004, pp. 156–160.

[33] M. Kölling, "http://www.bluej.org/," 2006. [Online]. Available: http://www.eclipse.org

[34] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, "The bluej system and its pedagogy," *Journal of Computer Science Education*, vol. 13, no. 4, December 2003. [Online]. Available: http://www.bluej.org/papers/2003-12-CSEd-bluej.pdf

[35] M. Kolling, "The design of an object-oriented environment and language for teaching," PhD Thesis, Basser Department of Computer Science, University of Sydney, February 1999. [Online]. Available: http://www.cs.kent.ac.uk/pubs/1999/2171

[36] R. Chatley, "Java for beginners," Imperial College London, Tech. Rep., 2001. [Online]. Available: http://chatley.com/kenya/thesis

[37] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting java programming errors for introductory computer science students," in *SIGCSE 2003: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, vol. 35, no. 1. New York, NY, USA: ACM Press, January 2003, pp. 153–156. [Online]. Available: http://dx.doi.org/10.1145/611892.611956

[38] B. Lang, "Teaching new programmers: a java tool set as a student teaching aid," in *PPPJ 2002/IRE 2002: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*. Maynooth, County Kildare, Ireland, Ireland: National University of Ireland, 2002, pp. 95–100. [Online]. Available: http://portal.acm.org/citation.cfm?id=638496

[39] J. R. Anderson, A. T. Corbett, K. R. Koedinger, and R. Pelletier, "Cognitive tutors: Lessons learned," *The Journal of the Learning Sciences*, vol. 4, no. 2, pp. 167–207, 1995.

[40] E. R. Sykes and F. Franek, "An intelligent tutoring system prototype for learning to program java," in *The 3rd IEEE International Conference on Advanced Learning Technologies*. IEEE, July 2003, pp. 485+.

[41] C. Sadler and B. A. Kitchenham, "Evaluating software engineering methods and tool—part 4: the influence of human factors," *SIGSOFT Softw. Eng. Notes*, vol. 21, no. 5, pp. 11–13, September 1996. [Online]. Available: http://dx.doi.org/10.1145/235969.235972

[42] B. W. Liffick and L. K. Yohe, "Using surveillance software as an hci tool," in *Information Systems Education Conference*, Cincinnati, Ohio, November 2001. [Online]. Available: http://cs.millersv.edu/~liffick/hci/hci.html

[43] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *AOSD 2005: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA: ACM Press, 2005, pp. 159–168. [Online]. Available: http://dx.doi.org/10.1145/1052898.1052912

[44] A. F. Hadwin, P. H. Winnie, J. C. Nesbit, and C. Murphy, "Loganalyzer: A toolkit for analyzing gstudy log data and computing transition metrics (version 1.0)," [computer program]. Simon Fraser University, Burnaby, BC, Canada, 2005.

[45] K. Mierle, K. Laven, S. Roweis, and G. Wilson, "Mining student cvs repositories for performance indicators," in *MSR 2005: Proceedings of the 2005 international workshop on Mining software repositories*. New York, NY, USA: ACM Press, 2005, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1145/1083142.1083150

# Appendix A

# Gild Extra Help

## A.1 Fifty-one Supplemental Error Messages

**AbstractMethodMustBeImplemented** The class you are creating either extends an abstract class, or implements an interface. Abstract classes and interfaces both may declare methods that they do not implement. It is up to the children of the interface or abstract class to implement these methods. This error is caused by one of the following conditions:

1) You don't intend for your class to be abstract and you have forgotten to implement all of the (abstract) methods of the parent class or interface.

2) You intend your class to be abstract but have forgotten to include the 'abstract' modifier in your class declaration.

**ArrayReferenceRequired** You are referring to some variable as something of type array when it isn't. An example of an array creation (type int, size 3): int[] pointsArray = new int[3];

**BodyForAbstractMethod** An abstract method should not have a body, rather, you define such a method like the following example: public abstract methodName(int arg1);

**CannotHideAnInstanceMethodWithAStaticMethod** A method with the same name and parameters exists in a parent of your class, but it is declared with the "static" modifier in the subclass.

For example, if a class called Parent declared an instance method called aMethod(),

aMethod() can only be called if an instance of Parent exists. In a subclass of Parent, called Child, the static modifier is used for aMethod(), which means that no instance of Child is required. You can call aMethod() the following way: Child.aMethod() The superclass and the subclass must have the same requirement for aMethod(). Either both are static, or neither one is.

You probably don't want to hide the parent's instance method. Try giving your method a different name or remove the static modifier.

**CannotOverrideAStaticMethodWithAnInstanceMethod** A method with the same name and parameters exists in a parent of your class, but it is declared with the "static" modifier. You can't override a static method in a subclass unless the subclass's method is also static. This is because static methods can be accessed without the need of an instance of the class that declares it.

For example, if a class called Parent declared a static method called aMethod(), aMethod() can be called like so: Parent.aMethod(); no instance of Parent is required. Now, if a subclass called Child extended Parent, then Child should inheret the static method aMethod(), and we should be able to call it in a static way: Child.aMethod(). But, if Child declares an instance method called aMethod() (i.e. "public aMethod()", instead of "public static aMethod()"), then it would be impossible to call aMethod() in a static way (instance methods require instances of the classes that declare them). There is a contradiction: aMethod() must be declared as static in Child.

More than likely, you don't want to hide the parent's static method. Try giving your method a different name.

**CannotReturnInInitializer** You have tried to return a value from the initialization code of your class (i.e. the constructor). This is impossible: constructors do not return values; they create instances.

You might want to think of it this way: when a constructor is called, it always follows the "new" operator (for example: "new Object()") which always returns a new instance of a class. If the constructor returned a value (an int, for example), what

should be assigned to your variable when "new" is used–the new instance, or the value returned by the constructor? Only one can be returned, and what you really want is the new instance of the class. So constructors cannot return values.

**CodeCannotBeReached** Some condition is making it impossible to reach this code. This is usually caused by misplaced or unaccounted for "return" statements in methods or code that you forgot to delete after a return statement.

First, check to make sure that you actually want this code to run. If you do, look for some return statement(s) that might be in the wrong place: inside a "finally" block, for example.

If you have a series of "if/else" blocks that have "return" statements in them, check to see if those "if/else's" exhaust all possible conditions. If they do, this might be the root of your problem.

Also, if you have a "switch" block with a "return" statement somewhere near the end, make sure that you have all the necessary "breaks" in the switch conditions.

**DuplicateField** You have more than one field with the same name. You can only declare a field (variable) once within its scope. You likely just need to rename one of your variables so that all names are unique.

**DuplicateMethod** You have two methods with the same signature. Even though each method may return different types, you cannot overload a method by changing the return type alone. Possible solutions:

1) Change the name of the method

2) Change the parameter type(s) or the number of parameters

3) Delete the duplicate method

**FinalFieldAssignment** You are assigning a value to a field (variable) that was declared as final. If a field is final, this means that the value it is given when it is declared is fixed and cannot be changed (even if it is null!).

E.g. final int MAX_RABBITS = 100;

In Java, the convention is to CAPITALIZE the name of any final field. This is an additional reminder that the field is final.

**IllegalCast** You have tried to cast an object or a simple data type to another type, which is not allowed. For example, "int i = (int)"ABCD";" is not allowed because Strings cannot be represented as integers. Neither will "Integer i = (Integer)new String("ABCD");" work because Integer isn't a sub-type of String, nor is String a sub-type of Integer.

**IllegalVisibilityModifierCombinationForField** You can only specify one visibility modifier for a variable:

1) public - accessible by any class

2) protected - only accessible within its package, subclasses, or by other files with which it is compiled

3) private - only accessible within its class

**IllegalVisibilityModifierCombinationForMethod** You can only specify one visibility modifier for a method:

1) public - accessible by any class

2) protected - only accessible within its package, subclasses, or by other files with which it is compiled

3) private - only accessible within its class

**IncompatibleReturnType** A method with the same name and parameters is declared in a parent class of your class but it has a different return type.

Consider what would happen if a parent class called Parent declared a method called "int aMethod()", and you extend Parent with a class called Child which declares the same method name but with a different return type: "String aMethod()". Following the rules of inheretance, what would happen if someone tried to create an instance of Child like this: "Parent child = new Child();". If "child.aMethod()" is called, what should be returned? Child says that a String should be returned, but Parent says that an int should be returned!

If you absolutely have to return a different type than the parent does, change the name of your method. Otherwise, if your method is meant to extend the functionality of the parent class's method, change the return type on your method to match the parent.

**IncorrectSwitchType** A switch statement can only compare simple primitive values such as int, char, byte and short. Either use one of these types or consider using a series of if/else statements.

**InterfaceCannotHaveConstructors** You are trying to make an instance of an interface. Interfaces cannot be instantiated because an interface is more of an outline that says what you want to do, but not how to do it. To use the interface you will need to make a class that implements the interface. e.g.In file Shape.java: public interface Shape{...} In file Square.java: public class Square inplements Shape{...}

**InvalidCharacterConstant** A char can only hold a single character. By using single quotes ' ' you are indicating that you are refering to a char. A String can contain multiple characters. Use double quotes " " when refering to a String.
e.g. char c = 'a'; String s = "abcdefghi";

**InvalidClassInstantiation** You are trying to make an instance of an abstract class or of an interface. Abstract classes and interfaces cannot be instantiated. To use an abstract class you must create a subclass of the abstract superclass. e.g. In file Shape.java: public abstract class Shape{...} In file Square.java: public class Square extends Shape{...}

An interface, on the other hand, would need to be implemented by a class. e.g. In file Animal.java: public interface Animal{...} In file Rabbit.java: public class Rabbit implements Animal{...}

What is the difference between abstract classes and interfaces? 1. A class can only extend one abstract class, but can implement more than one interface. 2. An abstract class can implement some methods, but an interface contains no implementation code.

**InvalidEscape** When you use a backslash in a String or char, this is a special character called an escape sequence. This escape sequence allows for special "character" like a tab or line return. If you want the backslash to be a backslash you need to type: \\. Some common escape sequences includes: \\t - tab \\n - newline \\r - line feed (use

n instead of this one, it is more reliable on different platforms) \\' - single quote \\" - double quote

e.g. System.out.println("Name:\\tBob");

**InvalidExplicitConstructorCall** In a constructor, if you want to call the superconstructor, you need to do this on the first line. Also, make sure that you aren't calling a superconstructor from a method.

**MethodReducesVisibility** A method with the same name and parameters exists in a parent of your class, but it has greater visibility than what you have defined (i.e. your method is declared "private" whereas the parent's method is declared "public"). This causes a contradiction: the parent declares that the method should be visible where your class says that it shouldn't be. You probably don't want to hide the parent's visibility. Solution: Either give the methods the same visibility or give your method a different name.

**MissingReturnType** You are specifying a method that does not have a return type.

1) If you are making a constructor, check that you have the correct spelling and capitalization. 2) If you are writing a method, make sure you specify a return type. (void is used to specify that there is nothing returned)

**MissingSemiColon** You are missing a semicolon at the end of a statement. All java statements end with semicolons; most likely, you forgot to include one at the end of a line. Check the lines surrounding this error

**NoMessageSendOnArrayType** You are attempting to call a method on an array. Arrays are not associated with methods. You probably want to call the method on some

element in your array.

For example, the following code will produce an error:

String[] s = new String[3];

s.charAt(2);

The following code works fine because the method is being called by a String object (in location 0 if the array), not on the array itself:

String[] s = new String[3];

s[0].charAt(2);

**NoMessageSendOnBaseType** You are attempting to call a method on a primitive type. Primitive types (e.g. int, boolean, double, etc) are not associated with methods. They are limited to simple operations. You probably want to use the method from a wrapper class. Every primitive type has a wrapper class that has many methods associated with it.

e.g. for int there is Integer, boolean there is Boolean, etc. For more information, see http://java.sun.com/j2se/1.5.0/docs/api/java/lang/package-summary.html

**NonStaticAccessToStaticField** A static field (variable) is a class variable. That is, no mater what instance of a class you use to access this variable, the value of the variable (field) will be the same. So, logically, we do not access this field through an instance of the class.

For example, say the field count in the Rabbit class was declared to be static: public static int count = 0;

To access this field, you could do the following: int current_count = Rabbit.count;

You didn't have to create an instance of the Rabbit class to access this field!

**NonStaticAccessToStaticMethod** A static method is a class method. That is, no mater what instance of a class you use to call this method, the behaviour of the method will be the same. So, logically, we do not call this method through an instance of the class.

For example, say the method getCount() in the Rabbit class was declared to be static:

public static int getCount() {...}

To call this method, you do the following: Rabbit.getCount();

You didn't have to create an instance of the Rabbit class to use the method!

**NonStaticFieldFromStaticInvocation** You are trying to access a non-static variable from within a static context. Some methods and fields can be defined as "static" for a class. This means that an instance of the class does not have to be created before the method or field is accessed. Common examples are the main(String[] args) method, and constant variables like MAX_SIZE, or ID. You cannot access non-static fields from within a static method because no instance of the class has been created for the use of the static method, and the non-static fields have not been allocated. A very common error is to try and reference a non-static field in your class from inside the main(String[] args) method. Try making the field static, or try creating a new instance of your class.

**NotVisibleField** In Java the visibility of classes, methods, and instances is one of: public, protected or private. When something is private, this means that it is only visible (usable) within its class, while something that is public is visible (usable) in any class. You are trying to access something that has been specified to be private. You can do the following:

1. Change the visibility of what you are trying to access to public or protected

2. Access the private functionality through a public method, etc. For class variables it is proper style to update private variables though public getter and setter methods.

e.g. in Shape.java:

private int height;

public int getHeight() { return height; }

public void setHeight(int height) { this.height = height; }

**PackageIsNotExpectedPackage** In Java, code is organized into packages. Any references to a file's location in a package are always located in the top line of the file.

For example: package ca.uvic.gild;

If you're getting this error double check that you have the correct spelling and package name.

**ParsingError** The Java compiler encountered a syntax error. There are many possible mistakes that could result in theis error. Some examples are: - a catch block with no preceeding try block - mispelling a class definition e.g. public clas Rabbit (clas should be class) - an operation missing a term e.g. int i = 3 - ; (either missing a number or - should be –)

The Java compiler will likely give you a suggestion to fix this error.

**ParsingErrorDeleteToken** You have placed an extra character in a statement around this line of code. Most likely, you have unbalanced brackets [ ], braces { }, or parentheses ( ). Check for an extra one of these in the code surrounding the error.

Another possible error is that you are trying to instantiate an (create a new) object and you have forgotten to include a parameter list. e.g. Integer i = new Integer; //incorrect Integer i = new Integer(); //correct - even if the parameter list is empty, it is still required

The Java compiler sometimes likes to tell you to delete tokens when you really just need to add something. Do not delete things with reckless abandon.

**ParsingErrorDeleteTokens** You have placed one or more extra characters in a statement around this line of code. Most likely, you have unbalanced brackets [ ] or parentheses ( ). Check for an extra one of these in the code surrounding the error.

**ParsingErrorInsertToComplete** The compiler has detected that you are missing a character. Most likely, you have unbalanced brackets [ ], braces { }, parentheses ( ), quotes, or you are missing a semicolon. You may also be missing an assignment = operator.

A common error is forgetting to put parentheses ()'s after a method call, even though the method takes no arguments.

Consider the following incorrect code where sleep is a method: rabbit.sleep;

To fix this, do the following: rabbit.sleep();

**ParsingErrorInsertToCompletePhrase** The compiler has detected that you are missing a character that is required to complete a phrase. Most likely, you are missing a semicolon or a brace '}'.

**ParsingErrorInsertToCompleteScope** You are missing a character that is required to finish a block of code. Most likely, you are missing a closing brace '}'.

**ParsingErrorInsertTokenAfter** The compiler is expecting a character immediately after the position given. Some common mistakes are: 1) You forgot to put parentheses around the condition of a while, for, or if statement. 2) You forgot to open parentheses for a method declaration or call. 3) You forgot to open a brace '{' at the beginning of a new method's block of code.

**PublicClassMustMatchFileName** The name of your file does not match the name of the public class or interface that you are trying to define. Java is case-sensitive, so make sure you didn't forget to capitalize properly. For example, in the file named Rabbit.java your class or interface should be defined as: public class Rabbit {...} public interface Rabbit{...}

**ShouldReturnValue** In your method signature you have indicated that your method will return a value of a specified type. You do not currently return a value from your method.

e.g. In the following code something of type int is returned:

```
public int getCurrentCount() {
//a bunch of code
return 42; //on the last line of your method!
}
```

**RedefinedLocal** You are redefining a local variable. Check in your method to see if you want to either:

1) Rename one of the variables

2) Remove the second variable's type declaration

**StaticMethodRequested** You are calling a non-static method in a static context. What is the difference between the two?

1. A static method is a class method. That is, no mater what instance of a class you use to call this method, the behaviour of the method will be the same. So, logically, we do not call this method through an instance of the class. e.g. The following method in java.lang.String: public static String valueOf(boolean b) {...}

You can use this method like this:

String result = String.valueOf(true);

2. A non-static method has behaviour that is specific to some instance of a class. You must create an instance of a class. e.g. The following method in java.lang.String: public int length() {...}

You can use the method like this:

String s = new String("hello!");

int result = s.length();

**SuperclassMustBeAClass** You are trying to made a subclass of an interface. A subclass must extend a class.

e.g. In file Shape.java:

public abstract class Shape{...}

In file Square.java:

public class Square extends Shape{...}

(Note: a superclass does not have to be an abstract class)

An interface, on the other hand, would need to be implemented by a class.

e.g. In file Animal.java:

public interface Animal{...}

In file Rabbit.java:

public class Rabbit implements Animal{...}

What is the difference between abstract classes and interfaces?

1. A class can only extend one abstract class, but can implement more than one interface.

2. An abstract class can implement some methods, but an interface contains no implementation code.

**SuperclassNotFound** You are trying to extend a superclass that does not exist. Check that you are using the proper spelling and capitalization for the superclass.

For example, you should have something like: public class Rabbit extends Animal {

**SuperInterfaceMustBeAnInterface** An interface is essentially an outline that says what you want to do, but not how to do it. When you create a class that implements an interface, you are agreeing to implement code for all methods in the interface.

e.g. in the file Animal.java: public interface Animal {...}

in the file Rabbit.java: public class Rabbit implements Animal {...}

You should only list method signatures in an interface (Animal.java), in a class that implements an interface (Rabbit.java) you will supply method bodies (functionality!)

**TypeMismatch** The Java compiler is unable to convert from one type to the other. Some examples include:

1) Returning null in a method that is supposed to return something of type int.

2) Assigning a double value to an int variable. In this case, you will need to cast:

e.g. int y = (int)2.14159;

Here the compiler is informing you that you will lose accuracy.

3) Using an assignment operator "=" in an if statement instead of the comparison operator "==".

**UndefinedConstructor** The constructor that you are calling has not been implemented. Double check that you are passing the correct number and type of arguments.

**UndefinedConstructorInImplicitConstructorCall** You are missing the default (no argument) constructor in your class' superclass. The default constructor will be automatically provided by the runtime system as long as no other constructors have been

specified. You can fix this by doing either of the following: 1. Write a no argument constructor in your superclass 2. Call a specific superconstructor in your super class by using: super(arg1, arg2, etc); Note: this must be done on the first line in your constructor.

**UndefinedMethod** You are trying to access a method that hasn't been defined for an object. Some common mistakes are: you misspelled the method; you are trying to call a method that hasn't been implemented on the object you are referencing

**UndefinedName** You are using a vaiable that cannot be resolved. It is likely that you have not declared the variable properly. All variables have a type and a name.

Consider the following code where we have a variable named "number_of_carrots" being assigned a value of 2: number_of_carrots = 2;

You will get an error if you never declare number_of_carrots to be of type int. A possible fix: int number_of_carrots; number_of_carrots = 2;

**UndefinedType** You are referring to a class that the Java compiler cannot locate.

e.g. Rabbit r = new Rabbit(); could trigger this error if you do not have a Rabbit class. Make sure that you are importing the correct packages and double check your spelling and capitalization.

Also, this error can occur if the class you are trying to use did not compile correctly.

**UninitializedBlankFinalField** You have a final field (variable) that should be given some value when you declare it. This is the only time a final field can be given a value.

**UninitializedLocalVariable** The Java compiler has detected that you have a local variable that has not necessarily been initialied. You can fix this by giving the variable some default null value.

e.g. int i = 0; or String s = "";

**VoidMethodReturnsValue** Void means that a method does not return any value. Either change your signature from void to some type, or remove your return statement.

You may have tried to return a value from the initialization code of your class (i.e.

the constructor). Constructors do not return values; they create instances of some class. Remove any return statements from a constructor.

# Appendix B

# Gild 2006 Questionnaire

This appendix includes the questionnaire that students attending CSC 115 were asked to fill out for more information about their use of Gild and any other programming environments. This questionnaire was given to students during a class in the last week of the semester.

# CSc 115 Gild Survey

Thank you for your interest in teaching and learning issues in first year Computer Science courses. Prior to completing this survey, the Human Research Ethics Committee of the University of Victoria requires that you be advised of the following:

Your participation in this research is voluntary. It is not in any way associated with your performance in the course, and your instructors cannot identify you with any responses you might give. If you do decide to participate, you may withdraw at any time without any consequences or any explanation. The questionnaire process is completely anonymous. At no time will anyone be able to identify you with your responses. The purpose of this study is to develop technology to support education in Computer Science. Your responses will only be used for the purpose of this study.

Results from this questionnaire will be published in computer science and software engineering journals, presented at scholarly meetings, and will form part of several theses and dissertations. Results will only ever be presented in anonymous form. Results will be available through the Gild website at: http://gild.cs.uvic.ca/

Please contact Mary Sanseverino if you have further questions. Phone Ms. Sanseverino at 250-721-8753 or email at msanseve@uvic.ca. In addition to being able to contact the study researchers, you may verify the ethical approval of this study, or raise any concerns you might have, by contacting the Associate Vice-President, Research at UVic: 250-472-4545.

Consent is given by filling out this survey.

---

1) When programming, what do you do when you encounter an error that you know nothing about? (Where do you go for help?)

2) What development environments have you used or explored for programming in CSc 115? (Check all that apply.)

| | | |
|---|---|---|
| ❑ Gild | ❑ Eclipse Java Perspective | ❑ Visual Studio |
| ❑ Textpad | ❑ Xcode | ❑ UltraEdit |
| ❑ JGrasp | ❑ BlueJ | ❑ Emacs |
| ❑ CodeWarrior | ❑ Project Builder | ❑ Other: _____ |

3) What development environment do you use the most frequently in CSc 115?

4) Why do you use these development environments for CSc 115?

5) Did the development environment you primarily use change over the semester? If so, why?

6) If you are not using Gild, why not?

**If you used the Gild development environment, please continue with this survey. If you did not use Gild, we thank you for taking the time to fill out this survey!**

---

For questions 7 and 8, select the response that reflects your experience.

7. Gild is easy to use.

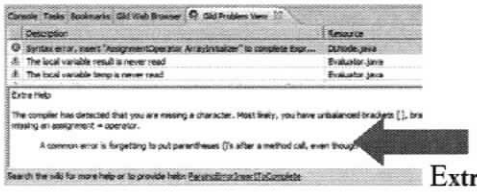
❑ Strongly Disagree ❑ Disagree ❑ Neutral ❑ Agree ❑ Strongly Agree ❑ N/A

8. Gild was easy to install.

❑ Strongly Disagree ❑ Disagree ❑ Neutral ❑ Agree ❑ Strongly Agree ❑ N/A

9. Which features in Gild do you find the most useful?

10. Was the extra error help feature useful? Why or why not?



**Extra error help**

11. If you were to change anything about Gild, what would it be?

If you have any other comments, suggestions or feature requests please write below.

**Thank you for taking the time to fill out our survey!**

# Appendix C

# Gild 2006 Survey Responses

In this appendix we present all student responses for two questions from a questionnaire that was distributed at the end of the spring semester in 2006, to students enrolled in CSC 115, at the University of Victoria. A copy of the survey is available in Appendix B.

## C.1 Student Responses: Where do Students go for Extra Help?

The specific question posed to students was: "When programming, what do you do when you encounter an error that you know nothing about? (Where do you go for help?)" In the table below, we present all student responses. These responses are classified according to common answers given to the question.

Table C.1: Responses to "When programming, what do you do when you encounter an error that you know nothing about? (Where do you go for help?)"

| Comments | Google/Internet | CSC 115 web site | Help desk | Friends/Students | Persevere (Look at code) | Debug |
|---|---|---|---|---|---|---|
| investigate, Debug, ask nearby people, post on web board, go to ELW | | • | | • | | • |
| I find ways of tracking down the error by tracking down what is causing it and seeing if there is a synonymous line I can use instead. I might post an explanation on the message board for our course, if it was an assignment. | | • | | | • | |
| I look at the line number that the error occurred on and try to track it. | | | | | • | |
| I've never really had that problem. | | | | | | |
| consultants | | | • | | | |
| consultants | | | • | | | |
| The Internet | • | | | | | |
| google | • | | | | | |
| I seek help from other comp sci students who are not in first year. | | | | • | | |
| google, friends, teacher (in that order) | • | | | • | | |

Continued on Next Page...

Table C.1 – Continued

| Comments | Google/Internet | CSC 115 web site | Help desk | Friends/Students | Persevere | Debug |
|---|---|---|---|---|---|---|
| Attempted to fix it | | | | | • | |
| My old 105 TA | | | | | | |
| I don't | | | | | | |
| Google, Java site | • | | | | | |
| try a few different thing myself and if I still can't figure it out I would either ask the consultants or my tutor | | | • | | • | |
| help desk, other students, prof, my tutor | | | • | • | | |
| tutor | | | | | | |
| look it up | | | | | | |
| Google it | • | | | | | |
| if have no idea, would just google it or find the line number of the error and determine the prob | • | | | | • | |
| I ask my smarter friends or completely change the code | | | | • | | |
| Java API? Read compile error and stack trace for run-time errors. Check out suggestions given by eclipse interface | • | | | | • | • |
| uvic csc 115 web, google | • | • | | | | |

Continued on Next Page...

Table C.1 – Continued

| Comments | Google/Internet | CSC 115 web site | Help desk | Friends/Students | Persevere | Debug |
|---|---|---|---|---|---|---|
| The csc 115 forum, or occasionally guild when I need a debugger | | • | | | | • |
| I follow the indicated line number and investigate | | | | | • | |
| Google | • | | | | | |
| If giving up is not an option, I use the internet or phone a friend. | • | | | • | | |
| debugging, consultant's office | | | • | | | • |
| look up where it originated and what kind of error. I go to Google or to friends for help. | • | | | • | • | |
| The java API, or just sit and think about it, usually the second option. | • | | | | • | |
| I go to the consultants office | | | • | | | |
| google it or message board | • | • | | | | |
| search online for information about the problem and go to the Helpdesk (Eng. Building) | • | | • | | | |
| walk through the debugger | | | | | | • |
| I know all the errors! | | | | | | |
| Not to the Gild wiki since it does not work. → Google | • | | | | | |

Continued on Next Page...

Table C.1 – Continued

| Comments | Google/Internet | CSC 115 web site | Help desk | Friends/Students | Persevere | Debug |
|---|---|---|---|---|---|---|
| Try to figure it out myself (stubborn), consultant's office occasionally | | | • | | • | |
| look online | • | | | | | |
| Internet | • | | | | | |
| Ask someone who might know or look for what is causing the error | | | | • | • | |
| Java API Google | • | | | | | |
| Derek, the senior Lab Guy or the help desk (consultant's office) | | | • | | | |
| Either post on the web board or go to API specs | • | • | | | | |
| Debug and look @ code | | | | | • | • |
| Press cancel or shift_ALT+DELETE | | | | | | |
| I google it it usually works. | • | | | | | |
| The program's help menu. Or, if that doesn't work, the program creator's site. | • | | | | | |
| the net (Google search) | • | | | | | |
| google / javadocs | • | | | | | |
| the internet, friends | • | | | • | | |
| webboard for class, or google! | • | • | | | | |

Continued on Next Page...

Table C.1 – Continued

| Comments | Google/Internet | CSC 115 web site | Help desk | Friends/Students | Persevere | Debug |
|---|---|---|---|---|---|---|
| I google it, I use a lot of System.out.println() statement, I check the java.sun.com J2E APIs, I bang my head on the keyboard. | • | | | | | • |
| search the net | • | | | | | |
| Google | • | | | | | |
| Internet or try to figure out what may have gone wrong from the code | • | | | | • | |
| Google! Doesn't usually happen, because most of the errors in Eclipse (I don't use GILD) are self explanatory. | • | | | | | |
| The textbooks. If I still find nothing, then I go to consultants. | | | • | | | |
| Google it, post on the web board, look at my code closely, etc | • | • | | | • | |

## C.2 Student Responses: Was the Extra Error Help Feature Useful?

The specific question posed to students was: "Was the extra error help feature useful? Why or why not?" In the table below, we present all student responses. These responses are classified according to common answers given to the question.

Table C.2: Responses to "Was the extra error help feature useful? Why or why not?"

| Comments | No, never noticed | No, never used it | No, it was useless | Yes, only at first | Yes | Sometimes |
|---|---|---|---|---|---|---|
| I think | | | | | | |
| didn't notice, maybe used eclipse? | • | | | | | |
| yes, easy to understand errors | | | | | • | |
| No. I usually recognized the errors. | | • | | | | |
| yes, it explains errors in detail so I may know what to fix | | | | | • | |
| very! Most java errors are so cryptic. | | | | | • | |
| yes!!! | | | | | • | |
| sure, at the beginning | | | | • | | |
| not really. If I knew the error I knew how to fix it. | | • | | | | |
| sometimes, I don't know what is he talking about | | | | | | • |
| Didn't use | | • | | | | |

Continued on Next Page...

Table C.2 – Continued

| Comments | No, never noticed | No, never used it | No, it was useless | Yes, only at first | Yes | Sometimes |
|---|---|---|---|---|---|---|
| omg, I was just talking about you! Yes, at the beginning w/ the easier errors. It sucks at hard ones and wiki has nothing usefull | | | | • | | |
| Not very helpful since they usually don't give any valuable information about the errors | | | • | | | |
| no, don't know about it | • | | | | | |
| Was helpful | | | | | • | |
| Yes, sometimes I needed them to fix the error because the standard message made no sense. | | | | | • | |
| Didn't really use [note: student indicated that they had only used Eclipse, not Gild!] | | • | | | | |
| Very helpful tells you where and a possible fix | | | | | • | |
| Yes | | | | | • | |
| I never used it. | | • | | | | |
| yes - told me if I was missing ; or wrong variables | | | | | • | |
| Definitely, though there is still room for improvement in regards to better explanations of the problem. | | | | | • | |
| yes, but only for the first couple of months | | | | • | | |

Continued on Next Page...

Table C.2 – Continued

| Comments | No, never noticed | No, never used it | No, it was useless | Yes, only at first | Yes | Sometimes |
|---|---|---|---|---|---|---|
| most (99%) of the time the extra help didn't show up or was completely unhelpful | | | • | | | |
| I never got an extra help feature with my errors (or never noticed it). | • | | | | | |
| Yes! Sometimes errors are hard to fix + this pinpointed where they were. | | | | | • | |
| sometimes. I had so much experience with the straight error messages I usually knew what they meant by the time I started using GILD. | | | | | | • |
| Not really, I barely noticed it. | • | | | | | |

# Appendix D

# Compiler Errors for all Participants

A total of 88 different compiler error and warning types were encountered by the participants, excluding warnings for unused variables methods.

Table D.1: Compiler errors and mean fix times for all participants

| **Error Type** | **Errors (%)** | **Errors (#)** | **Mean** | **Extra help?** |
|---|---|---|---|---|
| UndefinedName | 20.2% | 726 | 0:02:46 | yes |
| TypeMismatch | 8.8% | 317 | 0:02:21 | yes |
| UndefinedMethod | 8.5% | 305 | 0:03:59 | yes |
| ParsingErrorInsertToComplete | 8.4% | 303 | 0:02:03 | yes |
| ShouldReturnValue | 4.9% | 117 | 0:06:25 | yes |
| UndefinedType | 4.8% | 173 | 0:02:16 | yes |
| ParsingErrorDeleteToken | 4.1% | 146 | 0:01:19 | yes |
| PackageIsNotExpectedPackage | 3.2% | 115 | 0:04:11 | yes |
| UndefinedConstructor | 2.9% | 103 | 0:02:12 | yes |
| ParameterMismatch | 2.6% | 95 | 0:02:23 | no |
| AbstractMethodMustBeImplemented | 2.5% | 89 | 0:03:10 | yes |
| ParsingErrorInsterTokenAfter | 2.4% | 88 | 0:01:21 | yes |

Continued on Next Page...

Table D.1 – Continued

| Error Type | Errors (%) | Errors (#) | Mean | Extra help? |
|---|---|---|---|---|
| ParsingError | 2.3% | 83 | 0:02:55 | yes |
| InvalidOperator | 2.0% | 72 | 0:03:26 | no |
| UndefinedField | 1.6% | 57 | 0:01:06 | no |
| ArrayReferenceRequired | 1.6% | 56 | 0:01:36 | yes |
| IllegalModifierForArgument | 1.1% | 39 | 0:01:32 | no |
| NoMessageSendOnArrayType | 1.0% | 37 | 0:02:18 | yes |
| ParsingErrorMisplacedConstruct | 0.9% | 33 | 0:03:40 | no |
| ParsingErrorInvalidToken | 0.9% | 33 | 0:02:29 | no |
| UninitializedLocalVariable | 0.9% | 32 | 0:01:50 | yes |
| ArgumentHidingField | 0.9% | 31 | 0:01:34 | no |
| NonStaticAccessToStaticField | 0.8% | 30 | 0:03:09 | no |
| PublicClassMustMatchFileName | 0.8% | 28 | 0:02:41 | yes |
| NotVisibleField | 0.8% | 27 | 0:04:15 | yes |
| NonStaticFieldFromStaticInvocation | 0.8% | 27 | 0:00:56 | yes |
| ParsingErrorDeleteTokens | 0.6% | 20 | 0:01:14 | yes |
| UndefinedConstructorInImplicitConstructorCall | 0.6% | 20 | 0:07:52 | yes |
| NotVisibleType | 0.5% | 19 | 0:00:57 | no |
| StaticMethodRequested | 0.5% | 19 | 0:03:51 | yes |
| IncompatibleTypesInEqualityOperator | 0.5% | 18 | 0:05:13 | no |
| MissingReturnType | 0.4% | 16 | 0:00:50 | yes |
| MustDefineEitherDimensionExpressionsOrInitializer | 0.4% | 15 | 0:01:34 | no |
| MethodRequiresBody | 0.4% | 14 | 0:02:42 | no |

Continued on Next Page...

Table D.1 – Continued

| **Error Type** | **Errors (%)** | **Errors (#)** | **Mean** | **Extra help?** |
|---|---|---|---|---|
| UnhandledException | 0.4% | 13 | 0:04:34 | no |
| ParsingErrorReplaceTokens | 0.3% | 12 | 0:01:04 | no |
| DuplicateField | 0.3% | 12 | 0:00:30 | yes |
| InvalidClassInstantiation | 0.3% | 11 | 0:06:45 | yes |
| RedefinedLocal | 0.3% | 11 | 0:00:45 | yes |
| UsingDeprecatedMethod | 0.3% | 11 | 0:01:06 | no |
| IllegalCast | 0.3% | 10 | 0:00:46 | yes |
| ArrayConstantsOnlyInArrayInitializers | 0.2% | 8 | 0:00:47 | no |
| DuplicateCase | 0.2% | 8 | 0:13:39 | no |
| DuplicateMethod | 0.2% | 7 | 0:00:36 | yes |
| VoidMethodReturnsValue | 0.2% | 7 | 0:09:13 | yes |
| NoMessageSendOnBaseType | 0.2% | 7 | 0:01:37 | yes |
| CodeCannotBeReached | 0.2% | 6 | 0:01:00 | yes |
| InvalidEscape | 0.2% | 6 | 0:02:18 | yes |
| IncompatibleReturnType | 0.2% | 6 | 0:00:37 | yes |
| ExpressionShouldBeAVariable | 0.2% | 6 | 0:03:21 | no |
| Undefined: The project cannot be built until build path errors are resolved, or type already defined | 0.2% | 6 | 0:06:08 | no |
| InvalidExplicitConstructorCall | 0.1% | 5 | 0:01:04 | yes |
| NonStaticAccessToStaticMethod | 0.1% | 5 | 0:06:03 | yes |
| AssignmentHasNoEffect | 0.1% | 5 | 0:02:59 | no |
| MethodReducesVisibility | 0.1% | 5 | 0:00:52 | yes |
| BodyForAbstractMethod | 0.1% | 5 | 0:01:48 | yes |

Continued on Next Page...

Table D.1 – Continued

| **Error Type** | **Errors (%)** | **Errors (#)** | **Mean** | **Extra help?** |
|---|---|---|---|---|
| NotVisibleMethod | 0.1% | 4 | 0:00:21 | no |
| ImportNotFound | 0.1% | 4 | 0:00:40 | no |
| SuperInterfaceMustBeAnInterface | <0.1% | 3 | 0:00:12 | yes |
| UnterminatedString | <0.1% | 3 | 0:00:18 | no |
| FinalFieldAssignment | <0.1% | 3 | 0:00:18 | yes |
| UnreachableCatch | <0.1% | 3 | 0:02:05 | no |
| IllegalModifierForClass | <0.1% | 3 | 0:03:06 | no |
| UndefinedConstructorInDefaultConstructor | <0.1% | 2 | 0:01:04 | no |
| InvalidBreak | <0.1% | 2 | 0:03:05 | no |
| BytecodeExceeds64KLimitForClinit | <0.1% | 2 | 0:42:49 | no |
| UninitializedBlankFinalField | <0.1% | 2 | 0:00:57 | yes |
| AbstractMethodInAbstractClass | <0.1% | 2 | 0:00:17 | no |
| DirectInvocationOfAbstractMethod | <0.1% | 2 | 0:02:45 | no |
| MethodReturnsVoid | <0.1% | 2 | 0:01:15 | no |
| ClassExtendFinalClass | <0.1% | 2 | 0:02:17 | no |
| InvalidCharacterConstant | <0.1% | 2 | 0:00:37 | yes |
| NoFieldOnBaseType | <0.1% | 2 | 0:00:15 | no |
| InstanceFieldDuringConstructorInvocation | <0.1% | 2 | 0:00:27 | no |
| AmbiguousMethod | <0.1% | 2 | 0:01:05 | no |
| IncorrectSwitchType | <0.1% | 1 | 0:00:47 | yes |
| InterfaceCannotHaveConstructors | <0.1% | 1 | 0:00:16 | yes |
| DuplicateModifierForMethod | <0.1% | 1 | 0:12:06 | no |
| DuplicateBlankFinalFieldInitialization | <0.1% | 1 | 0:03:35 | no |
| UnterminatedComment | <0.1% | 1 | 0:00:51 | no |

Continued on Next Page...

Table D.1 – Continued

| **Error Type** | **Errors (%)** | **Errors (#)** | **Mean** | **Extra help?** |
|---|---|---|---|---|
| IncompatibleTypesInConditionalOperator | <0.1% | 1 | 0:00:07 | no |
| HierarchyHasProblems | <0.1% | 1 | 0:00:33 | no |
| DuplicateTypes | <0.1% | 1 | 0:00:36 | no |
| IsClassPathCorrect | <0.1% | 1 | 0:00:57 | no |
| ParsingErrorMergeTokens | <0.1% | 1 | 0:00:04 | no |
| CannotImportPackage | <0.1% | 1 | 0:00:25 | no |
| SuperclassMustBeAClass | <0.1% | 1 | 0:01:11 | yes |
| ParsingErrorInsertTokenBefore | <0.1% | 1 | 0:00:53 | no |