HintC — extending HiC with runtime hint-like feedback for CS1 students

Master’s Thesis

Kristopher K. Kruger
University of Wisconsin - Platteville
HintC — extending HiC with runtime hint-like feedback for CS1 students

Approved: [Signature]
Committee Chair
Date: 2009-09-14
Year-Month-Day

Approved: [Signature]
Committee Member
Date: Sep. 13, 2009
Year-Month-Day

Approved: [Signature]
Committee Member
Date: 2009-09-14
Year-Month-Day
HINTC — EXTENDING HIC WITH RUNTIME
HINT-LIKE FEEDBACK FOR CS1 STUDENTS

A Thesis
Presented to
The Graduate Faculty
University of Wisconsin - Platteville

In Partial Fulfillment
Of the Requirement for the Degree
Master of Science in Computer Science
in the Joint International Master program
in the Graduate College of the
University of Applied Sciences - Darmstadt
and the
University of Wisconsin - Platteville

By
Kristopher K. Kruger
B. Sc. (Computer Science) University of Wisconsin - Platteville
B. Sc. (Psychology) University of Wisconsin - Platteville
2009
Abstract
This thesis examines how students in introductory computer science courses at a public university have been using a programming environment explicitly targeted for use in the introductory courses. From an analysis of data spanning a few years, this thesis suggests that students spent, in relationship to the main themes of the assignments, inversely proportionate amounts of resources on trivial matters. The terms non-computed output, resubmits, and thrashing are introduced to classify, respectively, string constants as a source of the beginner’s attention, submissions to the automated grading server with no apparent change to the code, and situations where the student seems stuck and submitting out of frustration.

A system called HintC is introduced, developed in this thesis to reduce thrashing by providing (at least) a starting point for debugging through a hint-like feedback system as part of an automated grading system and to increase student learning by removing some of the trickery and enigma surrounding output differences. An informal questionnaire indicated a positive response by the students to HintC and the continued use of HintC by the CS faculty at the university indicate a positive response by the professors. An analysis of data collected through its use indicated a reduction in the amount of thrashing and an increase in the amount of attention spent on issues other than non-computed output.
Contents

Cover Page a
Approval Page b
Title Page c
Abstract d
Table of Contents i
List of Figures iv
List of Tables vi

1 Statement of the Problem 1

2 HiC as a Basis for HintC 2
   2.1 HiC - a C++ Compiler for CS1 ........................................... 2
   2.2 Submitting and Submission Logs ........................................... 3

3 Discussing Observations 6
   3.1 Observing a Lab Session .................................................... 6
   3.2 Submission Log Data ......................................................... 8
      3.2.1 Assignment Types ....................................................... 8
      3.2.2 Submission Characteristics ......................................... 9
   3.3 Analysis of Data from Labs ............................................. 10
      3.3.1 Lab 6 - Watermelon Seeds .......................................... 10
CONTENTS

3.3.2 Lab 9 - The Faster Quilter ............................................. 12
3.3.3 Lab 10 - Draw a Box .................................................. 16
3.3.4 Lab 11 - Calculate Compound Interest ............................ 18
3.3.5 Lab 12 - Draw a Square .............................................. 19

3.4 Analysis of Data from Programs ........................................ 19
3.4.1 Program 3 from the Fall of 2004 .................................... 20
3.4.2 Program 3 - Roman to Arabic Numeral Conversion ............. 20
3.4.3 Program 4 - Correcting Tests ....................................... 20
3.4.4 Program 5 - Table Tennis Anyone .................................. 21
3.4.5 Program 6 - Table Tennis Tournament ............................ 23

3.5 Non-computed Output .................................................. 25

3.6 Putting it into Perspective ............................................. 26

4 Presenting a Solution .................................................... 28

4.1 Requirements .......................................................... 28
4.1.1 No Configuration Required ......................................... 29
4.1.2 Focusing on the First Error ......................................... 29
4.1.3 Mistakes are Inevitable: Learn from Them ....................... 30
4.1.4 Changes expected from using HintC .............................. 31

4.2 Implementation ........................................................ 31
4.2.1 Into the Submissions .................................................. 33
4.2.2 Detecting Differences in Output .................................... 35
4.2.3 Detecting Loop Errors ............................................... 35

4.3 HintC’s Method: Hint-like Feedback ................................ 36
4.3.1 The Error Cases ....................................................... 37
4.3.2 The Context of an Error ............................................. 38
4.3.3 Sample Hints ......................................................... 41

5 Analysis and Evaluation ................................................. 46

5.1 Analyzing Submission Logs generated by HiC ...................... 46
5.1.1 Number of Submissions by Assignment Type .................... 47
# List of Figures

2.1 A CS1-Friendly Error Message in HiC ........................................ 4
2.2 Sequence Diagram of HiC Client and Server ............................... 5

3.1 Number of Lines changed in Lab 9 (only with changes) ................. 13
3.2 Number of Lines changed in Lab 9 (all submissions) .................... 14
3.3 Types of Changes using HiC on Lab 9 ...................................... 15
3.4 An Example of Thrashing from Lab 9 ....................................... 17
3.5 Number of Submissions per Student from Lab 10 ....................... 18
3.6 Number of Submissions per Student from Lab 11 ....................... 19
3.7 Types of Changes on Program 5 ............................................. 22
3.8 Number of Lines changed on Program 5 (all submissions) ............. 24
3.9 Number of Lines changed on Program 5 (only with changes) ........ 24

4.1 Class Diagram of an Abstract Syntax Tree: Hello World! ............. 32
4.2 Activity Diagram of Submitting an Assignment with HintC ............ 34

5.1 Number of Submissions by Assignment Type ............................. 47
5.2 Number of Submissions by Lab ............................................. 49
5.3 Number of Submissions by Program ...................................... 50
5.4 Number of Zero-Difference Submissions by Assignment Type .......... 51
5.5 Number of Zero-Difference Submissions by Lab .......................... 52
5.6 Number of Zero-Difference Submissions by Program .................... 53
5.7 Number of Differences in First Submissions by Assignment Type .... 54
5.8 Number of Differences in First Submissions by Lab .................... 55
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.9</td>
<td>Number of Differences on First Submissions by Program</td>
<td>56</td>
</tr>
<tr>
<td>5.10</td>
<td>First Submission Differences and Number of Submissions in Labs</td>
<td>57</td>
</tr>
<tr>
<td>5.11</td>
<td>First Submission Differences and Number of Submissions in Programs</td>
<td>58</td>
</tr>
<tr>
<td>5.12</td>
<td>First Submission Differences and Number of Submissions (all data sets)</td>
<td>59</td>
</tr>
<tr>
<td>5.13</td>
<td>Number of Lines changed using HintC (only with changes)</td>
<td>61</td>
</tr>
<tr>
<td>5.14</td>
<td>Number of Lines changed using HintC (all submissions)</td>
<td>62</td>
</tr>
<tr>
<td>5.15</td>
<td>Types of Changes between Submissions using HintC</td>
<td>62</td>
</tr>
<tr>
<td>5.16</td>
<td>Correlation: First Submission Differences and Number of Submissions (Pro-</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>grams)</td>
<td></td>
</tr>
<tr>
<td>5.17</td>
<td>Number of Differences on First Submissions (all data sets, grouped)</td>
<td>68</td>
</tr>
<tr>
<td>5.18</td>
<td>Number of Differences on First Submissions (programs, grouped)</td>
<td>68</td>
</tr>
<tr>
<td>5.19</td>
<td>Number of Differences on First Submissions (programs)</td>
<td>69</td>
</tr>
<tr>
<td>5.20</td>
<td>Number of Submissions by Development Environment</td>
<td>70</td>
</tr>
<tr>
<td>5.21</td>
<td>Number of Submissions by Development Environment (Programs only)</td>
<td>71</td>
</tr>
<tr>
<td>5.22</td>
<td>Number of Zero-Difference Submissions by Program</td>
<td>72</td>
</tr>
<tr>
<td>5.23</td>
<td>Types of Changes between Submissions</td>
<td>77</td>
</tr>
<tr>
<td>5.24</td>
<td>Types of Changes between Submissions</td>
<td>78</td>
</tr>
<tr>
<td>5.25</td>
<td>Number of Differences in first Submissions with Water Everywhere</td>
<td>79</td>
</tr>
<tr>
<td>5.26</td>
<td>Number of Matching-Output Submissions with Water Everywhere</td>
<td>80</td>
</tr>
<tr>
<td>5.27</td>
<td>Number of Submissions with Water Everywhere</td>
<td>81</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Example of non-computed output ................................................. 26
3.2 Example of computed output ......................................................... 26

4.1 Excerpt from a Submission Log showing Suggestions ......................... 30
4.2 Example of a nested Loop .............................................................. 36
4.3 HintC on a Substitution of Characters (Code: Table A.3) ........................ 41
4.4 HintC on a Variable with Wrong Value (Code: Table A.2) ..................... 42
4.5 HintC on an Extra Space (Code: Table A.4) ....................................... 43
4.6 HintC on an Infinite Loop (Code: Table A.5) ..................................... 43
4.7 HintC on an Uninitialized Variable (Code: Table A.1) .......................... 44

5.1 ANOVA : Number of Submissions by Lab ........................................... 48
5.2 ANOVA : Number of Submissions by Program ..................................... 49
5.3 ANOVA : Number of Zero-Difference Submissions by Lab ....................... 52
5.4 ANOVA : Number of Zero-Difference Submissions by Program ................ 52
5.5 ANOVA : Number of Differences in First Submissions by Lab .................. 55
5.6 ANOVA : Number of Differences on First Submissions by Program ............. 56
5.7 HiC-style Suggestions for Program 3’s Floats ...................................... 61
5.8 ANOVA : Number of Matching-Output Submissions by IDE ..................... 73
5.9 ANOVA : Number of Matching-Output Submissions by IDE and Data set ...... 73
5.10 ANOVA : Number of changed Lines by IDE .................................... 76

A.1 Source Code Segment Referred to by Figure 4.7 ................................ 89
A.2 Source Code Segment Referred to by Figure 4.4 ................................ 90
| A.3 | Source Code Segment Referred to by Figure 4.3 | 90 |
| A.4 | Source Code Segment Referred to by Figure 4.5 | 91 |
| A.5 | Source Code Segment Referred to by Figure 4.6 | 91 |
Chapter 1

Statement of the Problem

Programming languages and integrated development environments specifically aimed at students in introductory programming courses have changed the entry requirements for computer science. Their use allows introductory programming courses to ease students into problem solving through programming and program design (Dierbach, Taylor, Zhou, & Zimdand, 2005) rather than to require a deep level of understanding of the syntax of a particular programming language and underlying concepts of computer science (Stein, 1998). Despite best efforts with the specially designed tools, however, students still are found struggling with some of the core concepts (Bergin & Reilly, 2005), basic debugging tactics and rationales (Etheredge, 2004), and language syntaxes and development environments (Reed, 2001). This thesis analyzes and assesses data provided by students using such a tool. It highlights trouble areas and suggests improvements to the tool. It also analyzes the results from having used the tool with the proposed changes.

Data collected through the use of the compiler/programming environment HiC at the University of Wisconsin - Platteville (UWP) was analyzed. Analysis from the data suggested that students needlessly struggle and that improvements to programming performance could be made. A survey of the research in the field provided only a small amount of material specifically related to improving performance on programming while yielding a lot of other, general work on introductory programming environments and student learning. This thesis in itself provides some of the lacking material on improving programming performance.

The system presented in this thesis, HintC, specifically addresses some of the key problems noticed throughout the data. The new system should be no more difficult to use than existing solutions and should improve upon existing tools. The new tool could be used as a drop-in replacement for the tool HiC, from which it was built.

The goal of this thesis is to research programming performance in introductory courses and then to design and implement a solution for improving that performance. The solution is then to be evaluated and conclusions are to be drawn about the effectiveness of the solution.
Chapter 2

HiC as a Basis for HintC

The C++ compiler/programming environment HiC was used as the basis for this development environment HintC, proposed in this thesis. It provided a convenient base because its source code would be made available and because a pool of data on its use by CS1 students was already available for analysis. Data collected through its use at the University of Wisconsin - Platteville would be analyzed, looking for metrics for measuring programming performance and areas where improvements could be made.

The idea for such a system, the system that became HiC, started with the notion that students are struggling in CS1\textsuperscript{1}, introductory computer science (CS) courses for CS majors. Generally, introductory level languages or development environments are seen as helpful for CS1 students because of the lightweight nature of the tools. Using professional development environments can be difficult even if a textbook or a professor mentions only some portion of the language because the tool uses a language targeted for professionals. Research (Hasker, 2002) has shown inconclusively that students seemed less frustrated when using the CS1-targeted development environment called HiC than when using the professional tool Turbo C++ for DOS. These results together suggest that simply improving the compiler and language may not be enough to improve CS1 success rates.

2.1 HiC - a C++ Compiler for CS1

HiC is a CS1-targeted compiler/programming environment that accepts a subset of C++ and utilizes CS1-targeted error messages. The subset of C++ was chosen to allow for the generation of error messages that would be more meaningful to CS1 students than messages generated by professional compilers. As such, elements of the language that were neither needed nor usually taught at the CS1 level were removed. Also removed were programming language features which most instructors would not want CS1 students to use such as \textit{GOTO}

\textsuperscript{1}CS0 on the other hand refers to the “survey of computer science” courses typically designed for non-majors.
CHAPTER 2. HIC AS A BASIS FOR HINTC

statements and global variables\(^2\). Data collected generated by HiC would show the types of issues students have (or still have) while using a CS1-targeted development tool.

Motivation for HiC came from the idea that students had difficulty with the terms used in the compile time error messages of full-fledged, professional compilers (Hasker, 2002). In HiC, error messages at runtime indicate the type of error, via the name of the error exception, and provide a reference to the relevant section. For example, the HiC runtime error message for accessing a variable that has yet to have been assigned a value is the following:

\textit{Error: HicUndefException: Accessing uninitialized value from TotalStudents on line 84.}

The HiC integrated development environment (IDE) consists of several components: an interpreter, a parser, a debugger, and a submission client. At compile time, the common programming mistakes made by CS1 students can be identified and responded to with helpful messages. Within the IDE, syntax errors are highlighted in source code with explanations appearing in the error message window. As depicted in Figure 2.1, HiC displays the message

\textit{Parse error on line 30: Expected a semicolon (;) at this point. Hint: Check for a missing semicolon at the end of the previous line}

in response to an error encountered at compile time. At the CS1 level, execution speed is of little concern so the performance hit by using runtime checking is offset by the pedagogical usefulness to the students. As the interpreter of HiC runs the student’s program, HiC catches a selection of potential runtime errors (i.e. attempting to divide by zero, accessing an array out of bounds, and indications of an infinite loop).

2.2 Submitting and Submission Logs

Data generated by HiC comes in the form of submission logs. Students may provide submissions when they want and as often and they want.

During a submission, the HiC client (the part running on the student’s workstation) connects to the automated grader, a server set up specifically for use with the HiC client application.

\(^2\)When asked if he said something to the effect of “C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off”, Bjarne Stroustrup, the inventor and implementer of the original C++ language, replied that he did say something like that circa 1986. He then continues with the explanation of how avoiding simple problems may be hiding more complex ones: “…what I said there about C++ is to a varying extent true for all powerful languages. As you protect people from simple dangers, they get themselves into new and less obvious problems. Someone who avoids the simple problems may simply be heading for a not-so-simple one. One problem with very supporting and protective environments is that the hard problems may be discovered too late or be too hard to remedy once discovered. Also, a rare problem is harder to find than a frequent one because you don’t suspect it.”


With respect to HiC, the focus was never on reducing errors but rather on making it easier to deal with them when they surface.
The student is authenticated by entering a username and password and authorized by entering the course. HiC obtains from the server a series of tests for the particular assignment. These tests are sets of input and expected output. For each test, the data is provided to the interpreted program as input. The interpreter continues interpreting the student’s program until either it has terminated without errors or has terminated upon encountering a runtime error. After all of the tests have been run, a log of the activities is presented to the student. The interactions between the HiC client and the HiC server are illustrated in Figure 2.2.

The log generated from each submission is kept on the server. Each log is stored in a directory based on the assignment and in its own file with a name that indicates which student provided the submission and when the submission was provided. The files followed the naming convention of student274.20051202-192300 where student274 is the submitter identifier, 20051202 is the date in the format YYYYMMDD, and 192300 is the time in 24-hour notation. Available as Appendix B is a sample of the details found in the HiC submission log. The report contains the following key items:

- a number labeled “differences”, which indicates the number of lines in the actual output differing from the expected output
- the student’s source code, augmented with line numbers
- the output from the student’s program and expected output, both augmented with line numbers, for each test

![Figure 2.1 – A CS1-Friendly Error Message in HiC](image)
• a list of differences between the actual and expected outputs, following each set of output
• a list describing how to turn the actual output into the expected output by deleting unnecessary lines and adding missing lines

Figure 2.2 – Sequence Diagram of HiC Client and Server
Chapter 3

Discussing Observations

Data collected from nine assignments was provided for analysis for this thesis. Of the nine sets, five were lab assignments and the remaining four were program assignments. Lab assignments are small programming tasks and are expected to be completed by all students within the 52 minute period allotted for the class period. Generally, students are provided with a skeleton of the solution for labs, needing to complete just a few specific tasks within the provided framework. An example lab assignment might be a program to convert units (temperature, weight, distance, or similar). Program assignments, on the other hand, are expected to take longer (with several days or weeks between being assigned and being due) and deal with more complex problems, thereby requiring more effort.

3.1 Observing a Lab Session

The author of this thesis observed a lab session in the Fall of 2006. Students sat themselves one student per computer with each student facing toward the front of the room. The instructor indicated their purpose there, where they could find the materials, and that he would be asking the students approximately every fifteen minutes how they are doing. With that the students’ focus moved from the front of the room in the direction of the professor to their screens.

The lab session observed was that of lab assignment four, an assignment about computing factorials. HiC was to be used for submitting their solutions (and could be used as an IDE as well though no students were observed using any other editor). The class started at 11 AM. About ten minutes after 11 AM, most students were finished reading through the assignment description.

There were a few sporadic questions, indicated by students raising their hands (and arms) to capture the professor’s attention. From student to student, the professor made his way around the room, engaging in a dialog with each student, answering the questions of that student to that student. One student was assisted by being reminded of the `cin.eof` function and its syntax. It was observed that that student then submitted the student’s template
with only that one change. Some students were seen seeking help from other students, asking questions of his/her neighbor. Students were allowed to work together on labs but each student is required to submit his or her own working solution.

It seemed that students were quick to use HiC’s error messages and highlighting (example in Figure 2.1) to fix missing semi-colons and misspelled variable names. It also seemed that the students were relatively quick to fix missing or extra newline statements in this particular lab. In contrast, students did not seem so good at catching missing parentheses on function calls despite HiC providing compiler error message about expecting a parenthesis.

Also observed was that students generally would not use the HiC debugging tools — watches, step-through, or the status window — until instructed to use them and shown how they can be used. The extent to which the debugging facilities were actually used was not determined but the observed students at least left the windows opened after having used them when shown how.

In this particular lab session, students were asked to use the input method by which HiC would receive input from a file. This meant that students would not need to manually enter input with each run of the program. They still would need to interpret the output to understand if their programs were doing what they expected them to do.

After submitting a solution, HiC responds with the count of the number of differences in the output along with text showing the actual differences. Students typically focus on the number of differences. When questioned by the observer, students apparently understood that this number did not represent the actual number of errors or bugs in the program. They did mention that they liked the number as they felt it provided feedback that was both visual and quick and one that can be seen to decrease as they get closer to a correct solution. Despite this, they also conveyed that the number labeled differences did not necessarily indicate progress or regress. They said they understood that any amount other than 0 meant that there were still errors within their solutions. Students also expressed an interest in keeping a number displayed next to “Differences” instead of, perhaps, a general indicator such as “None”, “a Few”, “Some”, or “Many” or a binary indicator (no differences / more than no differences).

Some students suggested displaying the expected output and actual output side-by-side so that they could use the mouse to navigate the text. Others suggested providing an indicator in the student’s output to mark the first difference in a long output, such as the case with output-producing infinite loops. This suggestion seems to coincide with research (McIver, 2000) which has shown a CS1 student could easily get lost amid a lot of data.

All students finished the lab within the allotted time. Many finished with time to spare. Some left when finished, while others were observed checking email and surfing the web. At least one student used the remaining time to continue work on a nearly-completed program assignment.
3.2 Submission Log Data

Students using HiC are generally required to submit their finished programs to a server where the solutions are collected for evaluation by the instructors. This log data was made available for analysis after individually-identifying content (usually just the student’s name in a comment at the top of the file) had been removed. Some of the data provided contained the entire log for each submission while others contained just a summary of the number of output differences per submission per student.

Students run the client portion of HiC, only accessing the server when submitting their solutions. As can be seen in Diagram 2.2, the server provides both authentication and authorization and supplies the test input and expected output. The client performs the rest of the tasks, such as changing the student’s code into a form that the interpreter can execute, running the program on the test data, and constructing the submission log.

The submission process with HiC is not merely a way to upload the assignment to a central server for retrieval by the professor but also way to test the submitted program. Though testing and debugging facilities exist within the client portion of HiC, the tests available during submission allow for executing the student’s program against data provided by the instructor and for automating the process by providing input and validating output. Results of the test are available in the submission log, which the student sees upon completion of the submission.

For those assignments where each and every submission was saved on the server, the submission log data contains both students’ final, usually working, solutions and all submissions up to that point. The final solutions could show how differently or similarly the solution is approached by different students. Having each submission allows for the inspection of which changes each student made from one submission to the next. The starting point of the project was to analyze this data to assess programming performance and look for patterns illustrating trouble spots or students’ tendencies.

3.2.1 Assignment Types

Two different types of assignments were included in the data sets and analysis. The two types differentiate themselves in their complexity and their intended use as determined by the Computer Science department. The easier labs seem focused on first getting students accustomed to the development environment and submission process and then later introducing students to particular programming concepts. The harder programs seem focused on assessing if students can create a complete solution on their own.

Labs A typical semester consists of around 13 lab assignments. Lab assignments are relatively small programming assignments and constructed so as to be completed by all students within the lab session. The first half of the labs focus on very basic structured programming concepts through the use of “Karel the Robot”. The first few of
the second half introduce a submission process and the tools to be used. The last few introduce or reiterate loops, functions, and data types.

**Programs** Briefly, programs are bigger, harder, and take longer than the lab assignments. *Program* assignments are relatively complex programming assignments, requiring more effort and planning from the students. A typical semester contains around five programs. Other than just being more demanding than the labs, programs are also much more important because a student cannot pass the class without having turned in a running solution for each of the program assignments. Programs are also usually assigned with both an earlier deadline (due date) but also a later deadline (grace date). No points will be awarded after the grace date, full points can be awarded between the due date and the grace date, and working solutions turned in by the due date can receive up to two extra points toward style and documentation errors.

### 3.2.2 Submission Characteristics

Several different characteristics of the data are used to describe the submission log data. Some describe the log data in a general sense (such as the number of submissions, number of matching-output submissions, and number of differences per submission) whereas others take a closer look inside the data (such as the number of lines changed and types of changes) in an attempt to quantify the quality of the submissions.

**Number of Lines Changed** The number of lines changed between submission is determined by counting how many lines of code were changed between consecutive submissions. It is likely related to the number of *issues* addressed between submissions. Whereas one student may submit after changing a single line another may address all occurrences of an issue then submit. This statistic is interesting because it could indicate how students approach the various assignments, which problems they encounter within the assignments, and how they use the development environment itself.

**Types of Changes** The changes made from submission to submission could be sorted into categories. Some of the types of changes occurred often enough to appear in a category all their own for a particular assignment. Most of the others could be classified into one of the main groups. The changes made between submissions are interesting because students needed to manually perform the submit. Considering students receive feedback for their submissions (results from the automated grader), submitting after making a change indicates that the students were actively trying to figure out which effects their changes had on their solution.

**Number of Submissions** A basic piece of data is the number of submissions. It is given either as a total for the assignment or in terms of averages. It is a relatively easy statistic to obtain considering there is a file for each submission and the files are named by student. Its significance is not exactly clear. The numbers of submissions per student can be high because an upper limit on the number of submissions allowed was not set.
Differences per Submission One of the basic pieces of data present with each submission is the number of differences. Any number larger than zero means that the submission does not provide matching output as determined by the automated grader. It may or may not be an indicator of the student’s progress toward a working solution.

The data on the differences on first submissions represents a special case. It may be a way to assess the relative difficulty of an assignment. It may also be a way to compare two assignments or two groups of students.

Submissions with Zero Differences The automated grader reports when a submitted program produces output that matches the expected output. A student’s output that matches expected output also has zero output differences and would be shown as such in the submission log. Generally a zero-difference solution is a correct, working solution but often there are other requirements for the assignment. These other requirements concern themselves with adhering to the style guidelines or other implementation specific details as stated in the description of the assignment. Therefore, matched output is not in itself the ultimate goal but could be used as an indication of progress during development. Once reaching a solution that works, students may then add comments and make other changes (such as using specific data types) as needed, submitting again to verify that the solution still works. If so, the student has provided two zero-difference solutions. Some students provide many more than just one or a few.

3.3 Analysis of Data from Labs

Submission log data from Labs 6, 9, 10, 11, and 12 from the CS1 course in the Fall of 2005 at UWP was provided for analysis. The lab log data included over 1500 submissions from 300 submitters. The type of analysis varied from one data set to another. One type of analysis involved finding a predictor for the number of submissions, such as one based on the number of differences in the first submission. Another type involved determining how plausible it would be to use the first line of mismatched output to indicate which error the student had encountered. Some of the other analysis dealt tracking students’ intents in either changing several items in one submission or making changes over several submissions.

3.3.1 Lab 6 - Watermelon Seeds

The earliest dated piece of lab data provided for analysis for this thesis and one of the first analyzed is that of the submission data from Lab 6. For the assignment, students needed to input some values and calculate averages. They were also to output a list of players who performed well enough to receive certain rewards. Inputs were the reward thresholds and the scores (given in distances).

Lab 6 is a typical UWP CS1 lab assignment in that students were provided with a source file which contained more or less a completed assignment, albeit with sections missing. These
missing sections were labeled for them to complete. In Lab 6, they were to complete the following:

- enter their name
- enter their class section
- provide an algorithm to compute an average that when given 0 items would return an average of 0
- complete a function called `updateAverageStats` by filling in its header (which meant addressing which parameters needed to be `pass-by-reference` and which could be `pass-by-value`)
- complete a function called `updateThresholdStats` by filling it its header and its body (writing the function)
- update the function prototypes for `updateAverageStats` and `updateThresholdStats`
- use the correct variables as parameters when making a call to `updateAverageStats`
- use the correct variables as parameters when making a call to `updatePinWinners`

Analysis of the data in Lab 6 proceeded by examining the data after having removed submissions following the first submission that produced matching output as determined by the automated grader. The intent was to concentrate on students’ development and learning styles up until the point of providing a functionally correct solution. The changes to code between submissions were analyzed with the main differences and the time between submissions being noted.

There were 153 total submissions made by 76 submitters. Thirteen of those were provided after providing a first solution with matching output, leaving 140 submissions by all students to get to the first matching output solution. On average, each submission had just under one difference and, on average, each student provided just over 2 submissions. Every student provided a functionally correct solution, a solution that produced matching output. Actually, for this assignment, 67% of the students provided a working solution in the first submission.

As could be expected, some of the more prevalent changes between submissions dealt with exactly those points students needed to complete for the assignment:

- Several students produced incorrect results due to using an integer (`int`) rather than a decimal-carrying type (`float`) to store their calculations.
- Several also used `pass-by-value` parameters where `pass-by-reference` ones were required.
- Several changes addressed fixing the `updateAverageStats` function to handle the case of a list with 0 items, avoiding a divide-by-zero runtime error.
- Several changes were concerned with adjusting the algorithm to determine group placement (changing the test order, changing the test to use `if else` expressions, and changing the test to use data from the input rather than hard-coded values).
The assignment seemed to be simple in that so many students provided working solutions on their first submission and that there were so few differences per submission (less than one, on average). For those students who provided more than one submission up to and including a first solution with matching output, 72% provided a total of either 2 or 3 submissions. In all, 91% provided a first matching solution in either the first, second, or third submission.

### 3.3.2 Lab 9 - The Faster Quilter

The assignment from Lab 9 was to determine which of two contestants would win a race. Input was given as the rate of performing the task and the amount of the task to be performed (the assignment described quilters, how fast they could quilt, and how much material they had to quilt) (Clifton et al., 2005).

One of the interesting twists with this particular assignment was that a smaller, rather than a bigger, result indicates the winner (the faster one required less time to complete the task). Several students initially referred to the one with the bigger number as the winner before flipping an `if` statement to label the one who did it in less time as the winner. This seems to support the notion that problems and their solutions are seen in a superficial manner (Barr, Holden, Phillips, & Greening, 1999).

There were 480 submissions provided by 58 students, or 8 submissions per student. In all, 422 comparisons were made as part of the analysis of the changes between submissions. Given the quantity of submissions and the types and numbers of changes between submissions, it seemed students were using the submission process as something other than just a method through which to turn in their completed solutions for Lab 9.

#### Submissions with Zero Output Differences

In this set, 115 submissions, or 24% of all submissions, were submissions with matching output. Although three students did not provide zero-difference solutions, there were enough matching-output submissions for each student to have provided two each. Of the 115 total submissions with matching output, 25 students had just one submission of zero differences, three students had none, and 30 students had more than one zero-difference submission. The most zero-difference submissions by any one student was twelve and occurred once.

#### Number of Lines changed between Submissions

The frequency of the number of lines changed between consecutive submissions is shown in Figure 3.1 and Figure 3.2. The data is shown in two graphs to differentiate between submissions containing changes and those that did not. Not every submission followed a change to the code. Figure 3.1 shows the frequency of number of lines changed when changes were made while Figure 3.2 shows also those in which no change had taken place. As can
be seen from the graph in Figure 3.1 of submissions with changes, just one or two lines were changed in approximately 65% of the submissions.

The frequencies of submitting with various amounts of lines changed including those submissions without any changes to the lines of code are shown in Figure 3.2. Clearly the majority of the submissions followed changes to any of two lines, one line, or no lines of code. These three account for around 75% (30 + 27 + 17) of the alterations between two consecutive submissions. Resubmits, submissions following no changes to the code from what was provided in the previous submission, are interesting because nothing in the code had changed and submitting is a way for the student to use the submission process to observe the effects of the changes or to confirm that nothing had changed.

**Types of Changes between Submissions**

Considering that students were allowed to submit as often as they liked and considering that it seemed students were using the submission process to quickly test their solutions, the sheer number of submissions may not be such a good indicator of programming performance. Instead, it may be more interesting to examine the changes between submissions and the types of errors students seemed to be having.

Figure 3.3 shows the types of changes students were making between submissions in Lab 9. The changes from Lab 9 were categorized into the following four groups:

- changes to the non-computed output
- changes to the main algorithm
• changes to how a variable is calculated
• changes to none of lines containing code

Non-computed Output Changes to non-computed output refer to changes generally made within strings appearing between the quotation marks of `cout` statements and to changes to `endl` statements. Changes to some `endl` statements had more to do with an understanding to the procedural nature of the program solution than merely adjusting non-computed output and were counted as such.

Main Algorithm Changes to the main algorithm of the program included all changes that modified the general logic for solving the problem. Also included were those changes where the student’s previous submission was rather bare, containing perhaps a comment header and an empty `main` function. Changes to the program’s main logic usually coincided with the changing of many lines of code.

Variable Calculation Several of the changes affected how variables are calculated. In some cases this meant using different variables or constants or different constant values. In others it meant changing arithmetic precedence.

Resubmits Some submissions contained either changes just to lines other than those containing source code or simply no changes at all. These submissions are referred to as resubmits. In order to be considered a resubmit, a submission only needed to have been void of changes to its lines of code. Resubmits include both those occurring just
CHAPTER 3. DISCUSSING OBSERVATIONS

after a few seconds or minutes as well as those occurring over a span of several days\(^1\). The majority of resubmits seem to have been provided with only a small delay from one to the next and all within a small timespan.

As can be seen from the Figure 3.3, most submissions (64\%) followed changes to the non-computed output. The second most submissions (23\%) followed no changes to the code. Though most of these mere resubmits were of no apparent changes whatsoever, a few had changes to the comments or style\(^2\). These two issues (changing nothing and changing the non-computed output) account for well over 80\% of the variations between submissions.

Research (Barr et al., 1999) has shown that CS1 students have a tendency to spot style errors easily when reviewing code but are completely oblivious to even simple logic errors. This could help explain the distribution of submissions focused on certain, minor issues.

Fortunately, spacing and empty lines were not the only items receiving students’ attention between submissions. Around ten percent of the submissions followed changes made to the main approach to solving the programming assignment. These changes included modifying loops and tests, either adding or removing loops or tests themselves, as well as changing

---

\(^1\)It seemed that one of the first things some students would do after taking a few days off was to submit the solution just as it had been days before.

\(^2\)Perhaps interesting to note is that changes to the comments and style of the source could affect neither the program’s performance during runtime nor its syntax’s validity during parsing. Nonetheless, students would make minor changes to the style, perhaps adding braces around a single statement, adding parentheses around equations which were already evaluated in the desired order, changing indentation, or changing comments, then resubmit the code. One may be curious to know how many of those students were expecting that the changes made would affect the number of output errors for their programs.
the conditions of those items. A few resubmissions followed changes to the calculation of a variable, either introducing that variable or changing how it is calculated.

**Thrashing: The Pattern of a Particular Phenomenon**

While analyzing the submission log data, it was noticed that some series of submissions contained a sequence of submissions in which each submission quickly followed the one before it and during which there had been little or no change. This phenomena, henceforth referred to as *thrashing*, does not simply include all submissions made within a short timespan as it could have been the case that a student submitted after quickly realizing and fixing a mistake. Instead, thrashing is meant to refer to students submitting out of frustration.

An example of what was labeled as thrashing is shown as Figure 3.4 (Page 17). The graph illustrates the number of submissions, the number of differences per submission, and the time elapsed between submissions. Each data point in the graph represents a submission. The vertical axis corresponds to the number of differences per submission while the horizontal axis corresponds to elapsed time.

The data includes all submissions by one student for this particular assignment up to and including the first submission with zero differences. From the first submission until the last submission shown on the graph, just over 96 minutes had elapsed. In this set, there were 55 submissions with an average of 30 differences per submission. Of the 55 submissions shown here, 29 of the submissions were with 39 differences. On the graph, these data points are seen as a straight horizontal line segment.

Between the first and the last of the submissions with exactly 39 differences, there were also a few submissions with more than 39 differences. During this period, the student provided 34 submissions, leaving only an average of 102 seconds between two consecutive submissions. There is nothing magical about the two minute wait between submissions, other than that there were several submissions one after another between which the student changed little or nothing.

### 3.3.3 Lab 10 - Draw a Box

With Lab 10, students were asked to calculate and output a shape, called a box, based on input provided by the user. The student needed to provide a message if the input amount did not pass the requirements (an integer greater than 1), ask for valid input until some was provided, then calculate and output the shape accordingly. The box was to be two rows tall. Both lines would use the | character as the sides. The first row would then use the - character to fill the box while the second row would use a space character (Clifton et al., 2005).

The data set consisted of 334 submissions from 56 submitters. On average there were just over 13 submissions per student and just over 21 differences per submission. At the same time, 82 of the submissions were zero-difference submissions and one submission contained
Figure 3.4 – An Example of Thrashing from Lab 9
384 differences. There were still 13 students who provided a zero-difference submission with the first submit and three that did not provide a submission with matching output (zero differences). Figure 3.5 illustrates the distribution of the number of submissions by student on Lab 10.

### 3.3.4 Lab 11 - Calculate Compound Interest

Data from Lab 11 included 375 submissions from 60 submitters. A skeleton structure for Lab 11 was provided with students needing to just fill in a few pieces in order to calculate compound interest. Students were to a) read the initial deposit, the interest rate, and the number of months, b) check the input data according to some validation criteria, c) compute and output the result, and d) repeat until the quit condition (Clifton et al., 2005).

On average, there were six submissions per student. One student submitted 95 times, another 39 times, and the rest all submitted fewer than 15 times. The average number of differences per submission was just under 43 with the median being 20 differences per submission. Twenty submitters made just one submission. Twelve of those twenty first and only submissions produced matching output. This means that there were eight students who both submitted only once and failed to submit a solution of zero differences. There were also two other students in the set who did not provide solutions with zero differences. The breakdown of number of submissions per student for Lab 11 is shown in Figure 3.6.
3.3.5 Lab 12 - Draw a Square

The focus of Lab 12 was on loops as students were asked to provide a solution which drew a square. The size of the square was to be provided by the user. Students were to gain experience with loops and uncomplicated functions (Clifton et al., 2005).

The data set included 220 submissions from 50 students with 21 students submitting just once, 20 of which were of a solution with zero differences. Of these 21, the remaining submission had 91 output differences and was the only student in Lab 12 to not submit a solution with zero output differences.

Consecutive submissions in Lab 12 were compared with the main differences being noted. These notes provide a synopsis of the types of difficulties students were having with Lab 12 and would lead the way on how the logs could be analyzed to obtain indicators of performance.

3.4 Analysis of Data from Programs

Submission log data for program assignments consists of data from four program assignments from two semesters. Assignment Program 3 is from the CS1 course during the Fall 2004 semester and Programs 4, 5, and 6 are from the CS1 course in Fall 2005. The program log data included over 1500 submissions from 300 submitters.
3.4.1 Program 3 from the Fall of 2004

Only metadata about the submissions from the assignment was available. The actual submissions themselves were not available. The assignment was apparently the same or similar to the assignment discussed in Section 5.2.1. Students were to create a small game in which users would enter a distance to the target and the angle and velocity for a burst of water. If the burst of water gets close enough to the target, the user wins (Clifton et al., 2004).

The data set contains 81 submissions from 12 students. There were 19 zero-difference submissions. Three-fourths of the students provided at least one matching-output solution.

3.4.2 Program 3 - Roman to Arabic Numeral Conversion

For this assignment, Roman numerals and mathematical operations were used as inputs. Students were to translate the entered Roman numerals into Arabic numerals and to perform the requested operation on the numerals. The program was to echo back the read values and operation as well as provide the result of the translation and calculation (Clifton et al., 2005).

The data set contains a total of 891 submissions from 69 submitters. Two hundred seventy-five submissions contained matching output. The submission with the most differences had 838 differences. Students used functions to handle the different tasks of the program and generally used pass-by-value parameters. Some utilized switch statements and if-else statements.

3.4.3 Program 4 - Correcting Tests

The task for Program 4 from the CS1 class in the Fall of 2005 was to build a program which would grade tests for a given class. The tests contained multiple-choice questions. Valid choices included numbers between 1 and 5 (inclusive) or any of the following characters: T, F, and the letters A through F. A class could have more than one section but no exam could have more than 30 questions. The answer key is valid for all sections within a class for a particular exam. The program then was to compare the provided responses with the correct answers and present the user with the number of correct replies (Clifton et al., 2005).

In completing their solutions for Program 4, students utilized functional programming, pass-by-reference parameters, array indexing, and loops. The data set contains 1301 submissions from 65 students. This corresponds to just over an average of 20 submissions per student. On average each submission contained between 2 and 3 differences. The data set contains a total of 279 matching-output submissions.
3.4.4 Program 5 - Table Tennis Anyone

The purpose of Program 5 was to read a list of players, print this original list, process a list of transactions on those players, and then print the updated list. Students were allowed to assume a maximum number of players for the list but the number of transactions was not specified and as such students’ solutions needed to handle an any number of transactions. The transactions were not to be stored but were to be processed in the order received and processed when read. Proper feedback was expected when attempting to add a player to an already full list. Several of the basic structures were specified in the assignment description (Clifton et al., 2005).

The data from Program 5 consisted of 1337 submissions, which were provided by 56 submitters. Of the 1337 total submissions, 1110 were submissions with some output differences. As a program assignment toward the end of the semester, Program 5 allowed for more varied solutions.

Submissions with Zero Output Differences

In this data set, 227 of the 1337 submissions were submissions with zero differences. That is, 17% of all submissions were submissions containing no differences. Assuming that each student would have submitted at least once with no differences (typically seen as having provided a correct solution), then 13% of the remaining submissions were of zero differences. Two hundred twenty-seven submissions are enough for each student to have provided four zero-difference submissions. However, the median number of submissions with zero differences per student was three. The most zero-difference submissions by any one student was 19. Two students did not provide any zero-difference submissions whereas 6 provided just one zero-difference submission and 15 others provided just two zero-difference submissions.

It seemed some students would submit multiple times with zero differences in the process of documenting and cleaning up their code. This usually happened toward the end of the assignment as the student had successfully creating a working solution but had not yet commented and styled the document as required by the professors. It is generally presumed that the students would hack away at their program until they arrived to one with zero differences and then would begin to clean the code and add the appropriate comments. Also, some students seemed to be extra careful when making changes in that they would submit several times during the cleanup just to make sure no bugs were introduced.

Types of Changes between Submissions

The types of changes made between submissions were categorized into just a few groups. As with the lab type of assignment (mentioned in Section 3.3.2), it seems that students were using the submission process with program type of assignment not just to hand in the assignment when finished but also as a way to compile and also debug the program.
The groups are as follows: main algorithm, non-computed output, no apparent changes, refactoring, debugging, and miscellaneous.

As with the changes appearing in labs, changes in the program assignments can be classified into several groups:

- changes to the main algorithm
- changes to the non-computed output
- changes to something other to the code or to nothing at all
- changes that seemed to reflect refactoring
- changes that seemed to reflect debugging
- changes that could not be easily categorized elsewhere

These groups are ordered by the frequency shown in Figure 3.7. Groups refactoring, debugging, and miscellaneous were not as prevalent in the lab assignments.

Changes to the main logic (main algorithm) of the programming solution, in this assignment, included maintaining a list of players (adding to, removing from, and searching the list), calculating ratings, and processing transactions. Non-computed output changes included adjusting statements with \texttt{endl}, changing case or addressing misspellings and miswordings, changing \texttt{setw} statements (replacing tabs with them or altering their values), adjusting periods, and adjusting spaces.
The resubmits, as introduced in Section 3.3.2, in this assignment consisted of the usual cases of nothing being changed or only comments being changed. Special to this assignment, it seemed, was that of making the lines narrower. These too were labeled as resubmits as the code itself was not changed but was just reformatted to fit in a smaller width.

Some of the submissions seemed to come after changes which were categorized as for debugging use. One such change is that of echoing the values of recently assigned variables, thereby introducing output not in the expected output. A few other changes did not fit well into these categories and were thus placed in miscellaneous, which includes the seven submissions of a completely different assignment altogether.

In contrast with the graph from a lab in Figure 3.3 (on Page 15) where most of the submissions dealt with changes first to the non-computed output and second to nothing at all, most submissions (50%) from the program assignment, as shown in the graph in Figure 3.7, followed changes to the main program logic. From there the order is similar, with the second most (30%) following changes to the non-computed output and third most (10%) following “no apparent change”.

**Number of Lines changed between Submissions**

The number of lines changed between submissions (including submissions made after no lines were changed) is shown in Figure 3.8. From the 1337 submissions, there were 1269 comparisons of consecutive submissions. As can be seen in the chart, 11% of the submissions followed no changes, 44% following changes to just one line, and 17% following changes to two lines of code. Submissions following changes to more than 10 lines of code comprised 8% of the submissions.

**3.4.5 Program 6 - Table Tennis Tournament**

Program 6 from the Fall of 2005 required that students extend their solutions to Program 5. Players were to be assigned to groups for a tournament style play. The intent was to provide the conditions in which students could gain experience with sorting data and maintaining an existing program.

The list of players was to be sorted by player names before being printed. To facilitate with sorting, students were requested to use a new structure called `PlayerName` which contains a player’s first and last name. Solutions also needed to utilize specific functions for the reading and printing of the player names and a particular function header for sorting the list of names. Furthermore, although students could work in groups on Program 6, each student was to provide a submission (Clifton et al., 2005).
Figure 3.8 – Number of Lines changed on Program 5 (all submissions)

Figure 3.9 – Number of Lines changed on Program 5 (only with changes)
3.5 Non-computed Output

While analyzing the data to distinguish among types of errors, some changes were categorized as changes to what was called non-computed output. *Non-computed output* refers to output which was not the result of a calculation and probably is not dependent upon the lesson in logic or procedural programming which the student aught to learn during the assignment. Typically whitespace, spaces and new lines, and the exact wording in strings are non-computed output.

Having inspected the submission logs, it seemed a lot of attention was devoted to addressing changes in the non-computed output. It is not surprising that students spend a fair amount of time cleaning up their code because style and related points can account for a third to a half of the total points for the assignment. However, what might not have been expected is how this seems to have resulted in lots of submissions. Perhaps clicking the button to submit the assignment is such an easy way to check that a new error was not introduced or that, by some chance, the changes resulted in fewer differences that students do it even though they presumably expect things to continue to work as before or that the change would elicit the effects desired. In any case, it seems that there were more reasons for clicking the submit button than just to turn in an assignment.

Despite different assignments with different focal points, a lot of the students’ time\(^3\) seemed to be on adjusting spaces, changing wording, and other types of changes to non-computed output rather than on the core programming concepts the assignment was suppose to help teach. The architects of the assignments may have expected that, on an assignment meant to teach *for* loops, for example, loop concepts and especially those needed for the *for* loop would receive the students’ focus. This seems to suggest a trouble spot not so much because students’ attention was spent on styling issues but because the focus on these issues could vary a lot from assignment to assignment and because the lessons expected to be taught could be outweighed by the nuisances of the process to locate and address differences in the non-computed output.

Non-computed output is determined not by the output itself but in how it is used. For example, the implications of having a dissimilar quantity of * characters is different in cases where the * is a piece of computed-output versus those cases where it is meant as filler in non-computed output. The automated grader, however, is always looking for output that matches the expected output exactly and does not differentiate between non-computed output and computed output. Figure 3.1 and Figure 3.2 depict how the same character, a *, could be used in either computed or non-computed output. In Figure 3.1, the * is used as non-computed output. In Figure 3.2 the * is used as computed output.

\(^3\)The submission log metadata indicates when a solution was submitted but does not directly indicate how much of the time between submissions was spent on any one particular issue. As such, the word *time* is used here somewhat loosely.
cout << "** Please Enter Your Order Number: ";

Table 3.1 – Example of non-computed output

for (int i = 0; i <= width; i++) {
    for (int j = 0; j <= width; j++) {
        cout << '*';
    }
}

Table 3.2 – Example of computed output

3.6 Putting it into Perspective

After having observed a closed lab session and analyzed data from several assignments, patterns describing the process as a whole begin to emerge. For example, it seemed that students were using the submission process not just merely as a way to hand in their completed assignments, but also a means to debug their programs. This could have been because the way in HiC to completely automate the testing process is first through the compile button and second through the submit button. Through submitting their solutions, students receive system-provided feedback on runtime issues. Although students are allowed to submit as often as they like, the current setup might encourage undesired development styles.

In some cases several submissions followed a submission with matching output. It seemed for those submissions following the submission of a solution with zero output differences, adding comments and refactoring were the mostly frequently made types of changes. This may support the notion that students hack away at their code until they produce something that produces matching output then clean it up to adhere to the rest of the requirements for the assignment.

It also seemed that some of the less important parts of the assignments received a lot of the students’ attention. At the University of Wisconsin-Platteville, closed lab assignments are designed to be completed by all students within the 52 minute class period though some students would certainly finish the assignments within minutes. If it were to take 52 minutes to complete the assignment, extrapolating from the data presented, students would be spending an average of 33 of those minutes addressing issues with differences in non-computed output. About twelve minutes would be spent making no changes to the essence of the program (though could be spent making changes required comments and style guidelines).

Actually the literal time, or wallclock time, spent on the various parts of the assignments is not known. The meta data on a submission included just a timestamp describing when it was
submitted but did not include how long a student had been working on the assignment before submitting. However, given consecutive submissions and the changes made, the amount of time between submissions becomes known and the focus of the students’ attention is seen in the changes made. With those pieces of data, the time spent on an issue can be extrapolated.

Continuing with the extrapolated data, of the 52 minute lab session, roughly 45 minutes are spent on two issues: changing the non-computed output and checking that nothing is broken after changing nothing or changing the comments and style. Even if in the end each issue is minor, the amount of resources, such as time and thought, spent dealing with these issues must detract from other, more pressing issues.

Considering that even small errors can have a substantial impact on the experience of programming (McIver & Conway, 1999), one must wonder to which degree these issues are having an impact on the programming experience for these students. One must wonder which lessons the students are learning in the end through their interactions with HiC, the submission process, and each particular assignment. These effects could partially be explained by any of the following:

- as some teaching practices which need improving
- as a lack of training about how to use HiC
- as a lack of instruction on using the submission process
- of assignments which should be rethought and redesigned with pedagogical goals in mind
- of the system’s focus on matching output exactly and students’ difficulty in finding small differences amid lots of data
Chapter 4

Presenting a Solution

It seems that students were spending a disproportionate amount of time on small, trivial issues, especially those related to non-computed output. It seems that students’ programming performance (and perhaps learning experience) suffered accordingly. This thesis proposes a system which should increase students’ programming performance (and perhaps learning experience) by addressing output differences and by improving runtime assistance.

Typically syntax errors are caught by HiC during parsing because both the system was designed that way and because the parser is usually unable to proceed upon encountering such errors. On the other hand, logic errors, by their very nature, are embedded deeper into the code. Anyone who has had to debug someone else’s code can attest to this: syntax errors can be found quickly but the why behind the existing code takes much longer to understand, sometimes even with decent comments. Locating logic errors that remain after the syntax errors have been corrected is the most frustrating, and not necessarily the most pedagogically rewarding, chore of debugging (Freund & Roberts, 1996).

Some of the logic errors could result in runtime errors. Runtime checks can be used to perform some debugging on behalf of the student. Doing so allows the system to provide more CS1-friendly error messages but at the same time leaves only the more difficult errors to be discovered by the student.

The analysis discussed in Chapter 3 provides a glimpse into the role of the submission process in the CS1 students’ development cycle. The proposed solution, called HintC, is an extended version of HiC. HintC enhances HiC by providing improved feedback for general runtime errors and by providing direct feedback for output differences. To the student, these features are depicted as hint-like feedback and are available at runtime.

4.1 Requirements

The proposed system should to able to be used as a drop-in replacement for HiC, should provide better assistance during runtime than that provided by HiC, and should not encourage
dependency, abuse, or over-reliance. The proposed solution should not attempt to fix, create, or maintain the student’s programming solution nor should it be meant as a replacement for instructors and students’ interactions with instructors.

4.1.1 No Configuration Required

Since it was observed that students would not use the debugging facilities of HiC unless explicitly instructed to do so (discussed in Section 3.1), HintC should also require no configuration on the part of the student. The feedback provided by HintC should appear without specific provocation on the behalf of the student but the system should also provide a mechanism by which the new features can be disabled. The system need not be efficient nor does it need to be exceptionally fast, but the new system should at least not be noticeably slower to the point of disrupting students’ development cycle.

4.1.2 Focusing on the First Error

Noted in Section 3.1, at least one student mentioned an interest in locating the first error in a submission with several errors. To facilitate in this matter, HintC draws attention to just the error occurring first in a given submission, providing its detailed feedback on only that one issue. The normal HiC “suggestions” remain in the submission log so that they can still be accessed by the student if desired. With this approach, HintC provides feedback that is both manageable and easy to find while at the same time comprehensive by providing access to the full list of “suggestions”.

In the example of a submission log shown in Table 4.1, the output from the student’s program matches closely with the expected output. All non-computed output is correct. There are, however, two items that do not match the corresponding items in the expected output. One of those items occurs on Line 6 and the other on Line 7. The expected output in Table 4.1 shows that three players placed at the gold level with another three placing at the silver level. The student’s program instead produced seven players placing at the gold level and zero at the silver level. The suggestions about which lines to delete and add seem to indicate that lines in their entirety are wrong. The suggestions also seem to indicate that something is wrong with Line 5, since Line 5 is referred to twice in the suggestions.

This short example submission log only contained a few differences. Introducing more differences results in more insert and delete suggestions, which only serve to reduce readability of the suggestions. Introducing more suggestions may also make it proportionally more difficult to extract patterns or infer causes of the errors.

Introductory computer science students (introduced in Section 2 on Page 2) may find themselves dealing with the results of a particular logic error without understanding the logic error itself. Not yet aware of the cause of the problem, they make changes, perhaps random changes, to their code before compiling it again to see the new results. Although this debugging/learning technique using the edit-compile-run loop is not new, its effectiveness
CHAPTER 4. PRESENTING A SOLUTION

Output generated by your program from test test2 (line numbers added):
1: Score needed for Gold Pin: 15.0
2: Score needed for Silver Pin: 12.0
3: 16.4 11.2 9.6 14.0 13.4 11.8 16.0 15.9 14.3 11.3
4:
5: Average: 13.390
6: Gold Level 15.000: 7
7: Silver Level 12.000: 0

The CORRECT output (line numbers added for reference):
1: Score needed for Gold Pin: 15.0
2: Score needed for Silver Pin: 12.0
3: 16.4 11.2 9.6 14.0 13.4 11.8 16.0 15.9 14.3 11.3
4:
5: Average: 13.390
6: Gold Level 15.000: 3
7: Silver Level 12.000: 3

The following describes how to turn YOUR output into the CORRECT output by inserting and deleting lines. Use it to help find the errors:
Insert after Line 5: Gold Level 15.000: 3
Insert after Line 5: Silver Level 12.000: 3
Delete Line 6: Gold Level 15.000: 7
Delete Line 7: Silver Level 12.000: 0

Table 4.1 – Excerpt from a Submission Log showing Suggestions

in modern learning settings is questionable. The speed of modern processors (Bailey, 2005) and the ease of submitting in HiC shorten this loop drastically thereby enabling students to perform the loop much more frequently and after the slightest change. In the end it may be that students have spent less time understanding the logic that the solution should follow and more time symptom chasing or more time thrashing. Regarding this issue HintC will not lengthen the edit-compile-run loop but will highlight an error, providing the student with a place on which to focus.

4.1.3 Mistakes are Inevitable: Learn from Them

Ideally students would build a connection between how incorrect output looks and what needs to be done to fix it. HintC should facilitate this process by providing the feedback on just one issue so that the student can focus his/her attention on addressing that issue. In this way, students can make use of their mistakes, which all programmers make (Bennedsen & Caspersen, 2005, p. 188), by learning from them (Ginat, 2003, p. 11). HintC should help keep students involved in the development process by constantly providing a first step for debugging. This way, students should not feel helpless or unduly frustrated.
4.1.4 Changes expected from using HintC

As was seen in Section 3.3.2 and Section 3.4.4, a lot of changes between submissions were categorized as resubmits (no changes), as changes to non-computed output, and changes to the main algorithm of the program (especially in the program type of assignment). The following improvements are expected through using HintC in place of HiC:

- reduce the number of resubmits (submissions following no change)
- reduce the total number of submissions (though still allowing students to submit as often as they like)
- reduce the occurrences of thrashing (by providing a start point for debugging)
- reduce the amount of changes to the non-computed output (thereby increasing concentration on the other changes)

Traversing one’s self through a problem space in a timely, accurate, and meaningful way has been shown to be a more effective way to learn than otherwise random attempts by a student to stumble upon a way out (McIver & Conway, 1996). Simply put, eventually a student may stumble across a solution but the journey in doing so would have been rewarding had the student been actively involved in the process.

HintC should help students become more actively involved by removing some of the causes of frustration when encountering output differences. It was thought that some of the resubmits were because students had difficulty seeing the difference between the two, especially in the case of missing or extra spaces or character substitutions such as a semi-colon (;) in place of colon (:). HintC facilitates a more meaningful and timely traversal of the problem space by providing the students with starting points in the form of hints.

4.2 Implementation

Instead of creating machine code, HiC interprets the student’s code within itself. Whereas a compiler generates machine instructions to be executed later as part of a separate program, an interpreter simulates the execution of a program by combining the phases of compilation and execution (Freund & Roberts, 1996). The HiC interpreter has been implemented using the continuation model. A continuation is a function, inclusive with its context (Friedman, Wand, & Haynes, 2001), that merely describes which action to perform next, providing a system through which control flow can be expressed effectively (Appel, 2007).

A student’s code is edited within the source code editor of HiC. Running the program begins with a parse of the student’s code, creating the abstract syntax tree (AST). Abstract syntax is charged with the task of describing the structural essence of a language (Donzeau-Gouge, Kahn, Lang, & Mélèse, 1984). An abstract syntax tree is then a representation of operators
— the trees — as important concepts in a language where the operands — the leaves — are the important elements associated with those concepts (Visser, 2002). Figure 4.1 illustrates the abstract syntax tree of a simple “Hello World” program as generated by HiC.

Running the program continues with the interpreter walking the abstract syntax tree, evaluating the nodes. As with any interpreted environment, state of the environment must be maintained. In HiC, a Processor object maintains the state of the running program. HiC also provides access to the abstract syntax tree during interpretation (or what would be called runtime if the program had been compiled and were executed). This proves helpful, for example, in providing context-rich feedback in response to runtime errors.

HiC executes one instruction at a time, looping until the either the user has canceled the execution, a runtime error has occurred, or the program has completed. Being able to stop after each instruction allows for runtime debugging features in HiC. It also provided a system through which the features of HintC could more easily be realized.

The basic execution model of HiC is to check before executing each operation, throwing an error when an operation cannot be executed completely. The reasons that an operation could not be fully executed are known as runtime errors. These include operations such as performing division with zero as the divisor. HiC utilizes exceptions to handle the error.

The HintC implementation introduces a runtime exception for differing output to the set of exceptions defined in HiC. The new exception is used to indicate when the generated output
and expected output do not match, essentially turning unexpected output into a runtime error. This is desired in that it allows HintC to provide feedback to that specific error case.

In order for HintC to recognize when the generated output does not match the expected output, HintC needs access to the expected output as well as the generated output. HiC already provides a mechanism for comparing output (to create the number of labeled as “differences” in the submission log) and, although the student could run the program without submitting, providing the input manually (or from a file, as mentioned in Section 3.1), HiC provides no mechanism for the student to enter expected output. HintC interacts with the submission process, the process from which a) the data mentioned in Chapter 3 came and b) a process in which both input and expected output would be known.

4.2.1 Into the Submissions

Figure 4.2 (Page 34) illustrates the submission process. The process goes something like this: The student starts HiC and loads the source code. The submission process is started by clicking the appropriate button. A connection is made to the HiC server. If the student has not already logged in this session, the student is asked for a username and a passphrase as well as asked to select the course and assignment.

Having successfully logged in, the test data for the selected assignment is downloaded to the HiC client. The source code is parsed into an abstract syntax tree (AST). The interpreter is then initialized with the AST.

If there are no more tests to run (test data from the server may include one or more tests), the submission log is copied to the server and displayed to the student. If there are more tests to run, interpretation through the continuation model begins. As long as there are more instructions to interpret and the interpreting environment is still active (the student could abort the process, for example), the interpretation continues, “executing” the student’s program.

If an error, such as attempting to divide by zero or attempting to access an array outside of its bounds, occurs during runtime, an exception is raised. An exception is also raised when the produced output does not match the expected output (shown in Figure 4.2). In a submission with multiple errors, the exception raised would normally be for the first runtime error encountered. In HintC, however, higher precedence is given to the so-called real runtime errors. That is, even if the output differed several lines before the real runtime error (the attempt to divide by zero, for example), the exception for dividing by zero rather than the one named OutputDiffersException for differing output would be raised.

Execution of the student’s program is suspended when an exception is raised. The exception is caught and handled. During the handling of the raised exception, the feedback, in the form of a hint, is generated.

After processing all of the available test cases, the logged data is both saved on the server and presented to the student in the client as the submission log.
CHAPTER 4. PRESENTING A SOLUTION

Figure 4.2 – Activity Diagram of Submitting an Assignment with HintC
4.2.2 Detecting Differences in Output

The HintC implementation attempts to raise output difference exceptions when they would make the most sense or when they would be the most useful. When the student’s source code calls for some item to be printed, typically through a `cout` statement, the item to be printed is sent through an output stream, namely a class called `HicTestOutput` or another called `HintcTestOutput`. The print method of either is called for each and every argument of the `cout` statement. Arguments of the `cout` are those items following the double less than signs (`<<`).

Although HintC performs a character-by-character comparison, HintC also makes use of the segments, each segment being an argument to the `cout`. It does this by considering them as whole, independent units. Doing so provides for greater flexibility in determining the context of the error and ultimately influencing the feedback. Feedback is formulated to indicate which segment (not just which character or which line in its entirety) does not match the expected output.

The process of comparing character-by-character while utilizing segments for hints is as follows: If the student’s code contains the statement `cout << "Learning " << "is " << "great.";` but the expected output is `Learning is great!`, an exception would be raised. These two vary only in the punctuation used at the end of the sentence. In this example, a method called `print` would be called three times, one for each argument of the `cout` statement. A character-by-character comparison is performed one segment at a time until the difference is spotted. While processing the third item an output difference is detected. HintC has compared the characters of `great.` with the characters of `great!` and has raised an exception upon comparing the `.` to the expected `!`. HintC produces feedback utilizing the segments `great.` and `great!` rather than just using the mismatched characters.

4.2.3 Detecting Loop Errors

Loop errors represent a special case of errors. Unlike the other errors where the item being executed at the time of the error is the one blamed for the error, with loop errors the item being executed at the time of the error may not have been the instruction responsible for the error. With loop errors, the blame can be placed on the loop containing the instruction being executed when the loop error occurs. Because the type of error is known to be a loop error, it is also known that the loop operation is where attention should be focused and not necessarily the operation being executed at the time.

When looking for the cause of a loop error, HintC retraces the steps from where the error occurred until the first loop command it encounters. To explain through an example, the code snippet found in Figure 4.2 contains nested loops and a logic error. Eventually a timeout, called a loop error, will occur because the program has not finished within the set amount of time. The timeout could occur while processing any of the instructions in the test or body of the `while` statement starting on Line 9 but could also occur during the test or body within the inner loop, which is the one starting on Line 11.
2. void draw_square()
3. {
4. float square_number, columns, rows;
5. cout << "Enter the size of the square: ";
6. cin >> square_number;
7. rows = 0;
8. columns = 0;
9. while (columns < square_number)
10. {
11. while (rows < square_number)
12. {
13. cout << "#";
14. rows = rows + 1;
15. }
16. cout << endl;
17. //columns = columns + 1;
18. rows = rows - square_number;
19. }
20. cout << endl;
21. }

Table 4.2 – Example of a nested Loop

If the timeout occurred while, for example, Line 14 was being processed, the current HintC implementation would look to the while loop starting on Line 11. However, in this case, a variable from the loop at Line 9 does not change and so that loop repeats until the timeout occurs. The hint provided by the current HintC implementation would describe a loop error in the context of the inner loop. This may seem less than helpful because the explanation of where the issue occurred is wrong. However, despite this apparent deficiency, this strategy of looking at only the most recently occurring loop on loop errors is probably sufficient at the CS1 level.

4.3 HintC’s Method: Hint-like Feedback

Hints, the feedback provided by HintC to the student, are generated by applying a visitor to the AST given some data about the exception and other useful data about the context, such as the source code line number of the instruction. The visitor pattern is used because it allows new operations to be defined for and executed on an object without requiring any changes to the object itself (Palsberg & Jay, 1998).

In terms of the student’s source code, a hint with specific references to areas of likely causality is displayed to the student. The pieces of code deemed relevant are dependent upon the hint desired and the error encountered. Whenever possible, the hint includes the source code line that appears to contain the cause of the error. This feature is not dissimilar to a CS1 targeted debugger that utilizes the text editor to highlight the offending line in the source (Kölling & Rosenberg, 1996).

Hints are presented toward the top of the submission log to encourage the students to actually make use of them. For those students submitting hundreds of times, scrolling to the bottom
of the log to find out which differences exist can become tedious and tiresome.

When providing background data to illustrate the context of an error, HintC attempts to position the actual and expected elements in a way that lines up the difference, making subtleties less subtle. This should make spotting output differences much easier, much faster, and much less frustrating. Hints are provided for all of the runtime errors including the one when output differs.

In other contexts, some researchers have indicated that direct, explicit feedback can inhibit natural learning processes (Kruger, Harriehausen-Mühlbauer, & Nöth, 2005). To avoid this, hint generation in HintC focuses on what is helpful to the student and learning rather than just on the technical merits of what can be accomplished in the code.

### 4.3.1 The Error Cases

The hints generated by HintC are based on the type of error, the context surrounding when the error occurred, and the datatypes in use. Certain error and context combinations are more probable than others whereas some combinations may never occur. This is due to design of the abstract syntax tree and the checks performed during a given operation. HintC produces hints for all possible combinations where it seems that a hint could be helpful.

HintC discriminates among several errors. The error cases provided by HiC were used, extended, and added upon. The HintC implementation defines the following types of errors:

- **Output Differs** As the name implies, this error indicates that the produced output did not match the expected output. HintC defines this case as a runtime error so feedback on non-computed output errors can be given at runtime. This case does not exist in the original HiC implementation.

- **In Excess** This is a general group for the errors which exceed set limits. The limits for these error cases are meant to be set well enough so as to catch the intended cases while, at the same time, not being triggered too soon. They are generally meant to catch the case of an infinite loop. The cause of the errors given below should be known so those should be used in place of this generic one. Some of the limits can be configured from within the IDE.

- **Too Much Output** This error occurs when the produced amount of output exceeds a set limit. The given limit is arbitrary but set large enough to allow for the output typically generated in CS1 assignments.

- **Too Much Time** This error also attempts to catch infinite loops by being triggered when a set time limit has expired.

- **Too Many Steps** This error occurs when the system has processed more than a set amount of instructions.

- **Arithmetic Error** This is the general error for arithmetic error cases. However, the specific type of arithmetic error generally should be known and should be used when known.
**Divide By Zero** This marks the error case when one is attempting use a divisor of zero.

**Remainder With Wrong Type** This error occurs when the remainder operation is used with non-whole numbers. None of the code examined for this thesis exhibits this error.

**Out Of Range** This error occurs when accessing arrays at an index value above the upper bound or below zero.

**Unassigned Variable** This error occurs when attempting to access the value from a variable which has not yet been assigned a value.

**Other / Unknown** Considering that every check for an error defines the new error to handle, this catch-all case is not needed and should not be used. Nonetheless, it is one of the error cases available.

### 4.3.2 The Context of an Error

Just as the type of error is known due to the specific check for exactly that error, the context where the error occurs is also known. The context of where the error occurs is known because it was while processing that particular operation that the positive error check was performed. HintC uses this valuable data to provide context-sensitive feedback.

The context conditions are defined by the grammar understood by HiC. The HintC implementation does not change this grammar. By design, only certain operations can flag certain errors. HintC provides the following hints to the following situations:

**ConstExpr** The class `ConstExpr` represents a constant expression, such as the use of a number directly. An example of a `ConstExpr` is shown in Table 4.5 as part of the sample hint.

**Output Differs** “The constant [constant value here] was not that which was expected.”

**Divide By Zero** “The constant was zero. In division, the divisor may not be zero.”

**NamedConstantExpr** The class `NamedConstantExpr` represents a named constant, such as one created through the following code: `const int MAXRETRIES = 5;`. The constant can be either global or local in nature and the feedback reflects that.

**Out Of Range** “The global/local constant [constant’s name here] was initialized to [constant’s value here]. [constant’s name here] was declared on line [line of declaration here].”

**VariableLValueExpr** The class `VariableLValueExpr` represents processing a variable expression on the left-hand side of an equals sign. A variable could have been assigned a value when it was declared or any time thereafter. HintC finds where the variable was
last assigned a value to provide that data in the hint. In both the Output Differs case and the Divide By Zero case, the lines generated at this point are similar.

*updated* “The variable [variable name here] was last updated (assigned a value) on line [line number here].”

*initialized* “The variable [variable name here] was declared on line [line number here], where it was initialized to [initial value here]. It has not been assigned a value since.”

**SimpleOutStmt** The class SimpleOutStmt represents an output statement, such as one using a cout statement. Processing this operation could result in the Output Differs error case. The hint is not generated here but further down at the type of item in the output statement.

**ReturnStmt** The class ReturnStmt represents the return statement, typically found as the last instruction in a function and as a function’s only exit point. HintC constructs a hint for this case with the Output Differs error. However, the hint generation is pushed down to the expression in the return statement rather than here at the return statement itself so as to describe the error’s context better.

**ValueExpr** The class ValueExpr represents a container class for other value expressions. Hints for both Output Differs and Divide By Zero are generated where this error occurs but details for the feedback come from the class being contained.

**ArraySubscriptLValueExpr** The class ArraySubscriptLValueExpr represents the expression used as the index for an array. The logical error that occurs here is that of accessing an array outside of its bounds.

**OutOfRange** “Remember, the index of an array extends from 0 to the size of the array - 1. For illustration, count to five (5). OK, count again but start with zero (0). Notice you actually use all five fingers with just 0, 1, 2, 3, and 4. The expression [expression as a string here] accessed the array past its bounds.” The rest of the hint comes from the underlying expression (a constant, a variable, etc.) used as the index.

**DivExpr** The class DivExpr represents a division operation. Logically the error that can occur while processing a division operation is that of a Divide By Zero. HintC uses the type of object being processed to provided more details in the hint.

**OutOfRange** “The expression [expression as a string here] evaluated to zero.”

**SubtExpr** The class SubtExpr represents a standard mathematical subtraction operation. The hint contains data from this context when the divisor in a division operation contained a subtraction expression.

**OutOfRange** “The expression [expression as a string here] evaluated to zero.”

**MultExpr** The class MultExpr represents a standard multiplication operation. Its contents are relevant if the case of a Divide By Zero error in that the product of the multiplication was a zero.
Divide By Zero  “The expression [expression as a string here] evaluated to zero.”

RemExpr  The class RemExpr represents the mathematical remainder operation. Similar to the other math operations, its contents are relevant in the case of a Divide By Zero error in that the result of the remainder operation was a zero.

Divide By Zero  “The expression [expression as a string here] evaluated to zero.”

GetValueOp  The class GetValueOp represents the operation for obtaining the value of a variable. The error checked for here is that of accessing the value of a variable before variable has been assigned one.

Unassigned Variable  “The variable [name of variable here] has not yet been assigned a value. One can assign a value to a variable without a value, but one should not request the value of a variable when that variable has not yet been assigned one.”

LoopStmt  The class LoopStmt represents any of the various loops: do while, for, and while. Any of the three loop errors, Too Much Output, To Much Time, or Too Many Steps, may have occurred while processing a loop. In all cases HintC provides a line suggesting that the student’s code may contain an infinite loop. The feedback is then further detailed by HintC for the following two conditions:

*not updated*  In this case, none of the variables within the test condition for the loop were updated within the body of the loop. In a normal CS1 environment this means that the outcome of the test would never change and thus never approach an end case.

Your code may contain an infinite loop. In fact, none of the variables in the loop test were changed within the loop body.

If there was only one statement in the body of the test, the hint is appended with the following:

Did you mean to use { and } for the body of this loop? It appears you have not used {}, which need not mean you meant to and forgot.

The hint then contains data about which variables were contained within the test of the loop:

Your test condition included the following variables: [list of variables in loop test here].

*updated*  In this case at least one of the variables in the test was modified as part of the loop. It could be that the student used a greater than when a less than was required.

At least one of the variables was changed within the loop body, but since there may still be an infinite loop, check that the test condition is what you want and/or that the variables are updated in the way you want.
The following runtime hint-like feedback was produced by HintC.

Incorrect output was found while running your program on test test2. The first difference occurred at (or around) line 22 of your output. This was generated by line 82 of your source code:

82: cout << "<<< You need more practice; you missed after 5 tries."

The expected output where the difference occurred: >>>
The output that your program actually generated: <<<

The constant "<<< You need more practice; you missed after 5 tries." was not expected at this point.

Table 4.3 – HintC on a Substitution of Characters (Code: Table A.3)

The hint then contains data about which variables were contained within the test of the loop:

Your test condition included the following variables: [list of variables in loop test here].

4.3.3 Sample Hints

The feedback HintC provides consists of presenting details about the error encountered and where the program encountered the error. It is not obvious which wording makes for the best hints and which pieces of data are the most useful. Hints describe the context of the runtime error and generally display the source line corresponding to the likely cause of the error. For differences of output, the expected and actual output are lined up in such a way to highlight the difference.

The items chosen were deemed appropriate, at least as starting points, by the author of this thesis. The scope of possible feedback is scaled back in that the system tests for specific, defined error conditions. The context for a given error is determined from the walk along the AST. A challenge of the hint-like feedback is to create a system that walks the fine line of distinguishing unproductive errors, those that would restrict and/or would frustrate the student versus those errors which comprise the important building blocks in the process of learning to program (McIver, 2000). Details of several sample hints follow in this section.

On a Character Substitution

A rather common, trivial output difference is that of character substitution. Seen in Table 4.3, the expected segment is >>> while the student’s program produces <<< at that point. The non-computed output varies from assignment to assignment as do the specific characters being transposed, transcribed, or otherwise entered incorrectly. The amount of time spent looking for the differences though is of concern. Considering students in CS1 are
The following runtime hint-like feedback was produced by HintC.

Incorrect output was found while running your program on test test1. The first difference occurred at (or around) line 4 of your output. This was generated by line 77 of your source code:

77: cout << "Strike " << count << " goes " << comparison << " feet. ";

The expected output where the difference occurred: 1
The output that your program actually generated: 2

The variable count was last updated (assigned a value) on line 69: count = count +1;

Table 4.4 – HintC on a Variable with Wrong Value (Code: Table A.2)

not yet experts, it would make little sense to expect CS1 students know from experience which errors should get their attention.

The hint refers to the first line of the output which differed, the line to which it corresponded in the student’s source code, and the expected and actual output. In this case, the mismatching output is stored in a constant. HintC displays the constant along with an appropriate message.

Wrong Value in Variable

Table 4.3, Table 4.4, and Table 4.5 depict feedback in reference to differences of output. In each case, there were no other runtime errors, only differences between the expected output (available from the test data) and the actual output (as generated by the student’s program). The rest of the standard HiC-style submission log follows the feedback section from HintC. Students could still see in the remainder of the log the very verbose data about which lines were expected (correct) and which were not. The source code snippets to which the sample hints refer are included in Appendix A.

In the case of the difference in Table 4.4, the expected segment of output is 1 while the student’s program produces 2. Inspecting the code reveals that the variable count has been initialized to the value 1 already. Inside a loop, one of the first statements is the increment statement appearing on Line 69.

The specific data available in the hint depends on the particular walk that is made during the visit to the abstract syntax tree. The elements visited and order of the visit affect the hint produced. As seen here, a variable count was used to display the unexpected (wrong) value. As such, the hint in this case references the line of the last update to that variable.
The following runtime hint-like feedback was produced by HintC.

Incorrect output was found while running your program on test test1. The first difference occurred at (or around) line 2 of your output. This was generated by line 97 of your source code:

97: cout << "Enter angle (in degrees) for guess " << guess << " : ";

The expected output where the difference occurred: 1:
The output that your program actually generated: :
The constant " : " was not expected at this point.

Table 4.5 – HintC on an Extra Space (Code: Table A.4)

The following runtime hint-like feedback was produced by HintC.

A runtime error occurred while running your program on test test1. The error occurred while executing line 63 of your program

63: cout <<">>> You need more practice; you missed after 5 tries.";

Your code may contain an infinite loop.
Check the loop starting at line 62 of your source code:
62: while (count >5)

The loop test uses the following variables: count
None of these variables were changed within the loop body.
You may either have an extra ; or have forgotten to use { and } after the loop condition.

Table 4.6 – HintC on an Infinite Loop (Code: Table A.5)

Extra Space in Non-computed Output

A common, trivial output difference is that of an extra or a missing space. Often students fail the first several attempts to place the spaces in the correct positions in the non-computed output. Not always are spaces used exclusively in non-computed output, but in the sample shown in Table 4.5 there is an extra space before the colon. The notion of separate stanzas as arguments to the cout statement was discussed in Section 4.2.2. In Table 4.5, the last stanza contains a space, colon, and space. It was during the processing of this argument that HintC detected an output difference.

Infinite Loop

Not only does HintC generate hints on output differences, but it provides hints on real runtime errors as well. Table 4.6 shows a hint in response to a timeout (loop) error. The hint
CHAPTER 4. PRESENTING A SOLUTION

The following runtime hint-like feedback was produced by HintC.

A runtime error occurred while running your program on test test1. The error occurred while executing line 40 of your program
40: Sdistance = (Velocity * Velocity * sin(2 * Radians))/32.2;

The variable Radians has not yet been assigned a value. A variable needs to be assigned a value before the value in the variable can be accessed.

Table 4.7 – HintC on an Uninitialized Variable (Code: Table A.1)

shows that the error occurs while processing Line 63 but, as was discussed in Section 4.2.3, loop errors are handled differently. In the case show here, the part of the AST generated from source code Line 63 just happened to be the operation being executed when the timeout occurred. The hint shows that the loop started at Line 62 of the source and points out that the variable within the test condition of the loop was never updated within the loop. Inspecting the complete submission log reveals that the student should replace the while statement with an if statement since only a condition is needed at this point.

Uninitialized Variable

One of the detected runtime errors is that of accessing an uninitialized variable. Such a case is shown in Table 4.7. The hint directs the student’s attention to the source code line where the error was detected. The HintC hint also describes the error. This specific error can be avoided by always initializing a variable when declaring it. Some languages have been defined for use within CS1 to utilize declaration and initialization operations but not an assignment operation (Dougherty & Wonnacott, 2005), apparently for the sole purpose of reducing the number of errors that can be avoided by style and syntax changes (such as the one shown in Table 4.7). HiC could do the same, but it does not so as to encourage the student to develop debugging skills.

Tweaking Heuristics to Improve the Results

Generally HintC flags the output difference when it has occurred. There is, though, a special case when HintC flags early what could be an error before actually encountering it and without verifying that it would have been encountered. This was done to raise the exception when the data that seemed most relevant was still available. The special case is when the expected segment is a number with a decimal point and the actual segment is that same number without the decimal point. The assumption is that the student declared the variable carrying that value as an integer rather than as float or another decimal-carrying type.
As an illustration, the logic programmed into HintC when dealing with an expected output of 765.44 inches long and an actual output of 765 inches long is detailed. The actual output will be provided in several stanzas (as was discussed in Section 4.2.2). The first stanza is that of 765, which HintC compares to the first three characters of the expected output. Although the first three characters of both match, HintC raises an exception because it was comparing what looked like a number and the number did not match in its entirety\textsuperscript{1}. Raising the exception at this point allows for displaying the two numbers (“expected 765.44 but received just 765”, for example) in the hint. Had it waited for the next stanza to directly compare the 4\textsuperscript{th} characters, the 765 of the actual output would have no longer been available. The hint may have then read “expected 765.44 but received [a space character]”.

In most cases at the CS1 level this approach should work fine and provide better results. It should provide better results because the hints can contain more context-rich data. It should work fine because it is not likely that students would intentionally split a single number across several variables and HiC does not split a number into more than one stanza.

\textsuperscript{1}HintC receives both the expected and the actual output as large strings of text. A human looking at a printout of the text would easily group a series of characters of numbers together as a single number but HintC normally sees these as just any other character followed by any other character. In this special case, however, HintC does inquire if the current segments could be interpreted as numbers. If they can be, HintC checks if those numbers match.
Chapter 5

Analysis and Evaluation

The setup for evaluating HintC consists of both a more objective evaluation by comparing statistics and a more subjective evaluation made possible via responses to a questionnaire. It also consists of an analysis of HintC and samples of its hints as they showed themselves in the field compared with the ideal.

This chapter starts out by describing the data analyzed thus far (those generated by HiC). Data was presented throughout Chapter 3 from various labs and programs where HiC had been used. In a similar fashion, this chapter presents the statistics and analysis of submission log data generated by HintC. That is followed with a comparison of statistics first from assignments versus assignments, next from labs versus programs, and finally HiC versus HintC. The results from the questionnaire find themselves toward the tail end of this chapter.

‘Bar-and-Whisker’ graphs are used throughout this chapter. They appear as boxes with solid lines cutting through them. The box represents both the upper and lower quartiles. The line in the box represents the median. The ‘whiskers’ illustrate the range of values and small circles are used to indicate outliers. A bar-and-whisker graph is a more comprehensive description of the data within a sample than the average alone.

5.1 Analyzing Submission Logs generated by HiC

The data made available for this thesis contains submission log data from several variables: assignment types, students, assignments, and development environment used, to name just a few. The research question is if any of these groups differs significantly from any of the others. In particular, with this thesis the author wants to describe what can be described about the submission log data and wants to answer the question of the role, if any, played by the development environment.
5.1.1 Number of Submissions by Assignment Type

The analysis starts by investigating if the assignment type (lab or program) affects the number of submissions. It seems logical that there would be more submissions for a program than for a lab. The null hypothesis (H₀) is then that the number of submissions does not differ significantly between the two assignment types or that the number of submissions for labs is higher than for programs. The alternate (Hₐ) calls for the number of submissions for program assignments to be higher than for lab assignments.

Data from the two types of assignments outlined in Section 3 was provided. In all, the data consists of five labs and six programs. One of those six program data sets was generated with HintC while the rest were generated with HiC. As such, the data set generated with HintC will be left out of this comparison so as to avoid one possible source of variation. To test if there is a difference between the means of the two groups, an independent t-test will be used (Khazanie, 1996). A graph of the number of submission means for the two types of assignments is shown in Figure 5.1.

As can be seen in the graph of the mean number of submissions (Figure 5.1), there are more submissions for program assignments than for lab assignments. Also to note is that there is greater variance in the number of submissions for programs than for labs. Because group variance is unequal, the Welch’s t-test is a more appropriate test for analyzing variance to compare the number of submissions per student for programs and labs.

There is a significant difference in the number of submissions per student for program (M = 17.65, SD = 17.94482) and lab (M = 5.163, SD = 9.229333) assignment types: t(351.119) = 9.9114, p – value < 2.2e–16. These results suggest that there is a real difference between
the number of submissions for programs and labs. In other words, one could expect fewer submissions for lab assignment than for a program assignment.

These results are not revolutionary but they do support the claim that there are differences in the number of submissions between the two assignment types. The statistics, however, do not explain why the particular result exists. One possible explanation is that students use the submission process as a means to debug their programs and that program assignments are more difficult than lab assignments, encouraging more debugging. Furthermore, students generally provide submissions for labs only during the lab session (lasting just under an hour) whereas they could be providing program submissions over a span of several days or weeks.

### 5.1.2 Number of Submissions by Assignment

Just as the assignment type (lab or program) seems to be a factor in the number of submissions as shown in Section 5.1.1, the number of submissions may also be affected by the particularities of a given assignment. The problem needed to be solved and the general tools with which one could solve the problem may vary considerably from assignment to assignment. This section analyzes data from the assignments to answer the question of if the assignment itself has an effect on the number of submissions. The null hypothesis ($H_0$) is that the number of submissions does not differ significantly across different assignments. Its opposite, that the number of submissions does differ significantly among different assignments, is the alternate ($H_A$).

Since it was already shown that the type of assignment is a factor in the number of submissions, this section looks at just different assignments within the same assignment type. First data from the lab assignments is analyzed followed by an analysis of the program assignments. The data consists of the number of submissions per student per assignment for five lab assignments.

The graphs illustrating the means for the number of submissions are shown in Figure 5.2 for the labs and in Figure 5.3 for the program assignments. The graphs show that the means within lab assignments vary less than within program assignments. They also show that the means from one lab assignment to the next vary less than from one program assignment to the next.

A one-way ANOVA is used to test if the means of the five in both of the two groups are equal. Table 5.1 shows results of the calculations for the five lab assignments whereas Table 5.2 shows results of the calculations for the five program assignments. In both cases the results are significant at the 0.01 level. The results suggest that for both lab assignments and for
program assignments, the particularities of the assignment affect the number of submissions per student.

5.1.3 Number of Zero-Difference Submissions by Assignment Type

Another easily quantifiable statistic is that of the number of matching-output solutions, or those with zero differences, per student. If every student were to complete the assignment, there should be at least one matching-output submission per student per assignment. The data analysis indicates that some students provide more than one zero-difference solution whereas a few never provide one.

The question to be answered in this section is if the number of zero-difference submissions differs between assignment types (lab vs. program). It seems reasonable to assume that students hack away at the code until they submit something producing matching output and then refactor and address style issues afterward. Assuming they broke nothing, subsequent submits would still yield zero differences.

The null hypothesis (H₀) represents the case when the number of zero-difference submissions
does not differ significantly due to the type of assignment. The alternate ($H_A$) is that there is a difference in the number of zero-difference submissions between the two assignment types.

Again the data from the two types of assignments outlined in Section 3 is used. As in Section 5.1.1, data generated by HiC from the five labs and five programs is used in the analysis. A Welch Two Sample t-test is used to compare the means (Khazanie, 1996). The graph of the number of zero-difference submission means for the two types of assignments is shown in Figure 5.4.

As can be seen in the graph (Figure 5.4), there are more zero-difference submissions for program assignments than for lab assignments. Both groups contained observations (the combination of particular assignment and particular student) in which a student had not provided a matching-output solution. The number of zero-difference submissions per student for lab assignments does not vary much at all, staying close to 1 per student ($1^{\text{st}}$ quartile: 1.000; median: 1.000; mean: 1.337; $3^{\text{rd}}$ quartile: 1.000). With program assignments, on the other hand, the average is just over 4 zero-difference submissions per student ($1^{\text{st}}$ quartile: 2.000; median: 3.000; mean: 4.243; $3^{\text{rd}}$ quartile: 5.000).

The results of the t-test indicate that there is a true difference in the number of zero-difference submissions per student for program ($M = 4.243, SD = 4.905375$) and lab ($M = 1.337, SD = 1.045770$) assignment types: $t(264.45) = 9.1418, p-value < 2.2e-16$. These results suggest that the type of assignment does have an effect on the number of zero-difference submissions and it is appears more likely for there to be more matching-output submissions for a program than for a lab.

Given the aforementioned assumption that students hack away until reaching zero differences
then refine from there, coupled with the fact that students are allowed to submit as often as they like, it seems to make sense that there would be more zero-difference submissions for programs than for labs. After all, programs are bigger and more complex than labs. Therefore students may have more changes to make with programs than with labs and submitting just after a few would result in more submissions of programs than labs. This seems to concur with the explanation that students were using the submission process as a means of debugging their programs.

5.1.4 Number of Zero-Difference Submissions by Assignment

This section analyzes data from the assignments to answer the question of if number of matching-output submissions per student is affected by the assignment itself. The null hypothesis ($H_0$) claims that the number of zero-difference submissions does not differ significantly across different assignments. The alternate ($H_A$) is that there is a statistically significant difference.

Data from assignments of the same assignment type are compared. As with the previous analysis in Section 5.1.2, data from five labs and five programs was included in the analysis. Figure 5.5 shows the means for the number of matching-output submissions for the labs while those for the programs are shown in Figure 5.6. The graph (Figure 5.5) shows that there is little variation in the number of zero-difference submissions among the lab assignments. More variation is found in Figure 5.6 concerning the program assignments.
The one-way ANOVA is used again to test for a difference among the means of the two groups of five. The results from the calculations for the five lab assignments are shown in Table 5.3. Calculation results for the five program assignments are shown in Table 5.4. A significant result, at the 0.001 level, was found for the labs but not for the programs.

For labs, the null hypothesis that the means are equal is rejected. For programs, the alternate hypothesis that the means are not equal is not accepted and the null hypothesis not rejected. It is somewhat surprising that the results were not significant for the programs. In the graph in Figure 5.6, it does appear that the means of zero-difference submissions are not equal.

Table 5.3 – ANOVA : Number of Zero-Difference Submissions by Lab

<table>
<thead>
<tr>
<th></th>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Residuals</td>
<td>295</td>
<td>289.641</td>
<td>0.982</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.5 – Number of Zero-Difference Submissions by Lab

Table 5.4 – ANOVA : Number of Zero-Difference Submissions by Program

<table>
<thead>
<tr>
<th></th>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>4</td>
<td>164</td>
<td>41</td>
<td>1.7239</td>
<td>0.1453</td>
</tr>
<tr>
<td>Residuals</td>
<td>242</td>
<td>5755.4</td>
<td>23.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.1.5 Number of Differences on First Submissions by Assignment Type

Although the number of submissions per student per assignment varies, the submission log data contains at least one submission from each student for each assignment. It is possible that students took the class but did not provide a submission for a particular assignment but there would be no evidence of that in the submission log data.

The first submission is interesting because it is the first point at which the student decided to submit the code. Students may have different motives for submitting the first time. For example, some may submit when it seems the program is ready (complete) while others may submit after having made a minor change.

Despite providing submissions at different intervals or under different pretenses, it seems that there would be more differences, in general, in first submissions from programs than from labs. As such, the null hypothesis (H₀) represents the case in which the number of differences in first submissions does not differ significantly between the assignment types or the case in which there are more first submission differences in labs than programs. The alternate (Hₐ) is the opposite, that there are more differences in first submissions in programs than in labs.

Again the data from the two types of assignments outlined in Section 3 is used. As in Section 5.1.1, data generated by HiC from the five labs and five programs is used in the analysis. A Welch Two Sample t-test is used to compare the means (Khazanie, 1996). The graph of the number of differences in first submission for the two types of assignments is shown in Figure 5.7.
As can be seen in the graph (Figure 5.7), there are a lot more differences in first submissions for programs than for labs. The number of differences in first submissions for labs (1<sup>st</sup> quartile: 0.00; median: 10.50; mean: 21.88; 3<sup>rd</sup> quartile: 32.00) varies less than for programs (1<sup>st</sup> quartile: 49.5; median: 105.0; mean: 120.7; 3<sup>rd</sup> quartile: 167.0).

The t-test results are statistically significant and indicate that there is a real difference in the number of differences in first submissions for program ($M = 120.7$, $SD = 90.53611$) and lab ($M = 21.88$, $SD = 37.66494$) assignment types: $t(315.828) = 16.0522$, $p-value < 2.2e-16$. As such, the $H_0$ is rejected and the $H_A$ accepted. These results suggest that the type of assignment does have an effect on the number of differences in first submissions and that there are more differences for program assignments than for lab assignments.

The result seems rather obvious given what is known about labs and programs. Usually students receive the lab assignments as partially completed solutions and need to just fill in a few parts. With students needing to start on their own, without the template provided for labs, and with the complexity of programs compared to labs, it seems to follow that the first submission of a program would have more differences than the first submission for a lab.

### 5.1.6 Number of Differences on First Submissions by Assignment

This section analyzes data from the assignments to answer the question of if number of differences in first submissions is affected by the assignment itself. Without reason to suspect otherwise, it is assumed that the number of differences in first submissions will be relatively similar in size from assignment to assignment. Therefore for both program and lab
assignments, the null hypothesis \((H_0)\) is that the means of the number of differences in first submissions are equal across assignments (from the same assignment type). \(H_A\), the alternate, is that there is a statistically significant difference among the means from assignment to assignment.

Data from assignments within the same assignment type are compared. As with the previous analysis in Section 5.1.2, data from five labs and five programs was included in the analysis.

Figure 5.8 shows the means for the number of differences in first submissions for the labs while those for the programs are shown in Figure 5.9.

The one-way ANOVA is used again to test if there is a statistically significant difference among the means of the two groups of five. The results from the calculations for the five lab assignments are shown in Table 5.5. Calculation results for the five program assignments are shown in Table 5.6. A significant result, at the 0.001 level, was found for both the labs and the programs. This means in both cases \(H_0\) is rejected and \(H_A\) accepted.

The results indicate that the content of the assignment itself, be it a lab or a program assignment, affects the number of differences in the first submission. One explanation is that lab and program assignments are designed to be more difficult as the semester progresses.
CHAPTER 5. ANALYSIS AND EVALUATION

<table>
<thead>
<tr>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programs</td>
<td>4</td>
<td>412106</td>
<td>103026</td>
<td>15.541</td>
</tr>
<tr>
<td>Residuals</td>
<td>242</td>
<td>1604304</td>
<td>6629</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6 – ANOVA : Number of Differences on First Submissions by Program

With the programs analyzed, shown in Figure 5.9, this almost appears to be a sufficient explanation. Excluding the first data set listed, which was from a different semester than the others, this seems to be the trend for the remaining data sets. However the mean number of differences in the first submissions drops between the last two data sets listed. It seems likely that there are factors other than the increasing assignment difficulty influencing the number of differences in first submissions.

For the labs analyzed, shown in Figure 5.8, the opposite seems to be true. That is, as the semester progresses and the labs theoretically become more difficult, the mean number of differences in first submissions decreases. Perhaps the explanation then here is that although the labs become more difficult they do not keep pace with the learning students.

5.1.7 Correlation between Number of Differences on First Submissions and Number of Submissions

It seems reasonable that the number of differences on first submissions could be correlated with the number of submissions. The Pearson’s product-moment correlation is used to assess
the linear dependency between the these two variables. Using the data from the two types of assignments outlined in Section 3, correlation tests are detailed here.

The correlation test used yields values in the range from -1 to +1 with 0 referring to no correlation. The closer the result is to ±1, the stronger the correlation. Negative correlations refer to those cases with inverse relationships (as one value goes in one direction, the other goes in the opposite direction). Positive correlations refer to those where both values increase or decrease in the same direction.

The “line of best fit”, also known as the least squares regression line, is superimposed onto each of the correlation plots. The corresponding equations\(^1\) appear in the corresponding sections. The line of best fit helps to visualize the tendency of the data.

Figure 5.10 shows a plot of the data for labs. The equation for its line of best fit is as follows:

\[
NS = 0.04888 \times NFSD + 4.09393
\]

The Pearson’s product-moment correlation test \(t = 3.5139, df = 298, p-value = 0.0005102\) results in a correlation of 0.1994619. That is, there is a positive correlation between the two but a small correlation at that. The results of the test indicate that, in general, as the number of differences in first submissions increases so does the number of submissions. However, with a correlation of 0.1994619, the linear relationship between these two variables for labs is weak.

Similarly, Figure 5.11 shows a plot of the data for programs. The equation to the line of best fit is as follows:

\[
NS = 0.0944 \times NFSD + 6.2559
\]

\(^1\)NS represents the number of submissions and NFSD stands for the number of first submission differences.
The Pearson’s product-moment correlation test \( t = 8.478, df = 245, p-value = 2.220e−15 \) results in a correlation of 0.4762664. The results of the test indicate that as the number of differences in first submissions increases so does the number of submissions. Considering that the correlation is closer to 1 than the same correlation with the labs, the number of submissions follows a stronger linear correlation with the number of differences in the first submission for programs than could be said of the labs. That is, the number of differences in first submissions for programs is a better indicator of the number of submissions than it is for labs.

The plot showing data from both assignment types is shown in Figure 5.12. The plot includes the line of best fit and its equation is as follows:

\[
NS = 0.1005 \times NFSD + 4.1179
\]  
(5.3)

The Pearson’s product-moment correlation test \( t = 15.3629, df = 545, p-value < 2.2e−16 \) results in a correlation of 0.5497202. This correlation is stronger than both the one for labs and the one for programs. Again, there is a positive correlation between the number of differences in first submissions and the number of submissions.

As expected, more submissions are to be expected from students whose first submission has more differences than those with fewer differences. This correlation is stronger for program assignments than for lab assignments. A possible explanation for a weaker correlation within the lab data set and a stronger one within the program data set is that, although the errors encountered in labs are easier to fix than those encountered in programs, each error has more of an effect in the smaller labs than in the larger programs.

Another possible explanation is that students may be more aware of far along they are at completing the assignment with program assignments than with labs. This would at least partially be due to the use of templates for lab assignments, which remove some of the individuality from students’ solutions.
5.2 Analyzing Submission Logs generated by HintC

As was the case with the data generated by HiC and presented in Chapter 3, submission logs generated by HintC were made available in an anonymous form for use in this thesis. The anonymized data made available consisted of only one program assignment and no lab assignments. The data made available was the submission log data from Program 3 from the CS1 course in the Spring of 2007 at UWP. An analysis similar to that performed on the data generated using HiC was performed on the data. The results are discussed in this section.

5.2.1 Analysis of Data from Program 3 - Hitting the Target

The assignment named Program 3 was to create a program that could be used to play a game or practice math skills. Inputs were first the distance of the target and then the angle and velocity required to hit the target. The program would compute if the combination of angle and velocity, when plugged into the provided algorithm, would be sufficient to make contact with the target at the given distance. No more than five attempts were to be allowed for any one execution of the program (Clifton et al., 2007).

As the final program assignment of the semester, students would have been expected to be able to use any of the data structures or techniques learned throughout the semester. Students would likely utilize various combinations of tests, loops, computations, and program structures to provide correct solutions. Although not a requirement, students could use the HiC graphics package to provide visual results rather than standard textual results. Doing so could earn the student up to 10 point of extra credit (Clifton et al., 2007). Using the graphics package however meant both that (at least some of the) feedback from the automated grader could or should be ignored and that the HintC hints would be unavailable.
CHAPTER 5. ANALYSIS AND EVALUATION

The data set contains a total of 252 submissions provided by 30 students (submitters). Twenty of the students provided solutions with matching output as determined by the automated grader. No student submitted only once with the average being just over 8 submissions per student. Seventeen (57%) of the students provided between 2 and 8 submissions whereas the remaining students, 13 (43%), provided between 9 and 19 submissions.

Zero-Difference Submissions

In all there were 48 zero-difference submissions with an average of just under 2 zero-difference submissions per student. Of those 252 submissions, 28 (11%) were submissions that followed the first submission with matching output from each student. Ten (33%) of the students did not provide a solution with matching output as determined by the automated grader.

Lacking matching-output solutions usually would mean that that student did not provide a working solution but that was not the case with this assignment. Of the ten students without matching-output solutions, nine of them had final submissions with (at least some of the) remaining differences due to either output from the graphics package or discrepancies of floating point calculations. One of those nine had differences due to at least both of these issues. The student without either differences because of floating point calculations or because of output from the graphics package provided a final submission with just one bug, which resulted in two differences.

Differences because of the graphics package Four of those ten students had final submissions with remaining differences due to the automated grading system not accounting for output generated by the graphics package. Students must have been instructed to ignore the feedback from the automated grader because the grader detected output differences for all lines concerning themselves with graphics. It is not clear from the submission log data to determine how many of the students creating graphical solutions provided correct, working solutions.

Differences because of floating point calculations Another issue with the automated grading of this assignment was with the use of floating point numbers. Of the ten students who did not provide matching-output solutions, six of them provided solutions where the produced output was affected by the floating point arithmetic. An example of the HiC-style list of suggestions is shown in Table 5.7. Eventually the students were asked to use setprecision to set the amount of displayed decimal place numbers to three but that too did not always work as expected in that some of the calculations determining when a target was hit still varied.

Analysis of the Submissions

The analysis consists of 204 comparisons between consecutive submission logs. The analysis investigated both how many lines were changed between submissions and which types of things were being changed.
The number of lines changed between submissions when lines were changed is shown in Figure 5.13. As can be seen in the chart, most (58%) of the time when students submitted a solution with changes the students changed just one line. Three-fourths of the submissions with changes were with either changes to one line or to two lines.

Sometimes students submitted solutions without having changed any of the code. Figure 5.14 shows the number of lines changed between submissions across all submissions.

As shown in Figure 5.15 of the types of changes between submissions, changes to the non-computed output contributed to the largest single type of change between submissions. Resubmits, or submissions after changing nothing, accounted for 17% of the submissions. This particular assignment called for the use of setprecision. There were enough submissions following changes to this item to warrant its own category. However, changes to
CHAPTER 5. ANALYSIS AND EVALUATION

Figure 5.14 – Number of Lines changed using HintC (all submissions)

Figure 5.15 – Types of Changes between Submissions using HintC
CHAPTER 5. ANALYSIS AND EVALUATION

setprecision (7%), changes to the calculation of a variable (5%), and changes to the main algorithm (17%) could have been grouped together. Doing so would have brought the total for that group, changes to the main algorithm, to 29%. There were several other types of changes which could not easily be categorized into one of the other categories or could have been categorized into several of the other categories. These appear together under the category labeled miscellaneous.

5.2.2 Correlating First Submissions Differences with Number of Submissions

In previous sections, the correlation between the number of differences in first submissions and the number of submissions was calculated for programs and labs submitted using HiC. This section looks at the same correlation as it may exist with the data from the program assignment submitted using HintC.

A plot of the data for programs submitted through HintC is shown in Figure 5.16. Given that there are few data points, the data points can appear to be scattered rather randomly throughout the plot in Figure 5.16. There are areas, though, without any data points (such as in the lower right-hand area of the plot, which corresponds to no students whose first submission had around 75 or more differences and who provided no more than 5 submissions). The line of best fit should help visualize the patterns in the data. The equation for this plot’s line of best fit is as follows:

\[ NS = 0.05778 \times NFSD + 4.66961 \]  

The Pearson’s product-moment correlation test \((t = 3.5962, df = 28, p-value = 0.001226)\) results in a correlation of 0.5620919. The test results indicate a positive correlation between the two variables in which a higher number of differences in first submissions correlates with a higher number of submissions.

5.2.3 Other Submission Logs generated by HintC

The submission log data discussed in the previous section was the only made available for inclusion in this thesis. This does make the analysis more difficult and the results harder to take as representative. Although there are submissions generated by HintC from 30 submitters, they all come from just one data set.
5.3 The Accuracy of the HintC’s Feedback

5.3.1 Accuracy due to the Type of Comparisons

During the design phase of HintC, it was decided that HintC should be implemented using a character-by-character comparison rather than a segment-by-segment comparison. With a segment-by-segment comparison, both the expected output and the actual output would have been broken into meaningful segments. Each segment from the actual output would have then been compared to the matching segment in the expected output. When an entire segment does not match, the output would be marked as being different and an exception would be raised. The assumption with this approach would be that the student would (or should) print nothing less than a complete segment per stanza, or per an argument of a print, cout, call.

This strategy was not implemented in HintC because the assumption that needs to be made may be false, would result in providing a false positive, and may prove more frustrating than helpful in terms of providing helpful, pedagogical feedback at runtime. For example, perhaps an application is to compute the amount of change from some purchase. Continuing the example, one student elected to provide the solution by storing cents in one variable and dollars in another. The issue comes when the expected output segment at some point is the number 45.32 as one entire segment but the student’s code produces instead three segments: 45, the decimal point ., and 32. With a strict segment-by-segment comparison, if the segments compared do not match in their entirety, then they are marked as differing from one another and the appropriate exception is raised. In this case, the system would provide a false positive.

In the end, HintC was implemented with a hybrid of the two: making character-by-character comparisons but referring to the entire segments when giving hints.
analyzed though used the aforementioned split segment approach and thus should have been handled well enough with a segment-by-segment comparison.

5.3.2 Less than optimal Hints

As for the actual hints themselves, in looking through the submissions provided while using HintC, it seemed that the algorithm for detecting output differences was sufficiently accurate. In all but three of the 252 submissions, the HintC seemed accurate and relevant. These three exceptions are detailed here.

In the one case, the student’s output contained an extra space at the beginning of a line of non-computed output. The feedback from HintC indicated that an earlier line of the output did not match. It also indicated that the expected output at that point was a newline but the student’s program generated nothing at that point (failed to provide a newline when needed). This particular issue occurred because a string constant contained a trailing space at the end of a line, which HintC noticed but HiC would have normally ignored.

In the second case, HintC correctly indicated which piece of the output was not expected, but seemed to point to the wrong two pieces of “expected” and “actual” output in explaining the context. This case could be seen as somewhat misleading because it seemed the hint was providing conflicting information. It seems that HintC could be adjusted to better handle this case. Furthermore, it seemed that the student was not distracted by the apparently conflicting hint but rather addressed another issue and provided a next submission within two minutes.

The third case is similar to the first case. Here the student’s non-computed output contained a space at the end of a line, which HintC accounted for, whereas HiC would have stripped out. The “suggestions” from HiC do not indicate a problem with the line although HintC does, pointing to the segment of non-computed output which was not expected (contained the extra, trailing space), and which line of the output and which line of the source correspond to the issue. It should be possible to make HintC and HiC agree on how to handle these trailing spaces: always ignoring them, always considering them, or defining the rules for when to disregard them and when to take them into account.

5.4 Questionnaire: Student Input on HintC

A small, informal questionnaire was used to assess students’ perceptions HintC in general and its hints specifically. The results of the questionnaire were made available to the master candidate by academic staff in the University of Wisconsin-Platteville’s Computer Science department. Students were not required to complete the questionnaire but could volunteer to do so. Of the approximately 50 students in the two sections who would have HintC (and previously HiC), results from 36 students were collected. It should be noted that there were two sections for the class and around only half of the students from one section completed
the questionnaire. With such a sampling bias, one need be careful before using the data for anything other than anecdotal purpose.

Furthermore, the questionnaire was given to students in a course that finds itself somewhere between CS0 and CS1. The course was meant for CS majors but does not go very deep into the programming. Therefore, this group may represent the best target group for HintC in that this group may be most likely to benefit from HintC.

Three questions comprised the questionnaire. The first two questions asked how helpful the hints were and how helpful they would be for future students. Both were followed by a choice of five responses. The third question was open-ended, asking students to provide any suggestions for changing or improving the program.

Results of the questionnaire provide anecdotal evidence that students thought HintC had helped them and could help future students. Of the two questions regarding the hints, one asked students to describe their experiences with the hints and the other asked if student thought future students would benefit from the hints. To this first question, just under 80% of the responses indicated that the hints were at least mildly helpful in identifying errors or had helped students to fix errors. None of the responses indicated that students thought the hints were misleading.

When asked if students thought the hints would be helpful for future CS1 students, just over 80% of the responses were affirmative, indicating that students thought HintC would be helpful to future students. None of the responses indicated that students thought that the hints would not be helpful to future students. In both instances, around 13% of the responses indicated no exposure with the hints. This can be understood when considering that HintC provided no hints to errors in submissions in which the student’s solution used the HiC graphics package. Furthermore, some students’ submissions contained neither runtime errors nor output differences for which HintC could have provided hints. It is also possible that some students did not know what was meant by hints.

In response to the open-ended question, most of the responses addressed issues with HiC itself rather than the hints specifically. Those replies which did address HintC or its hints were mostly complimentary with a few exceptions. The exceptions suggested that output differences should be highlighted even more clearly, that somehow with the hints the submission log looks cluttered, and that the hints should be more specific to the cause of the problem.

In summary, the results were positive, indicating a general acceptance of HintC and its hints. However, despite roughly 81% of the responses indicating that HintC was helpful and approximately 93.5% of those familiar with the hints finding them helpful, the informal nature of the questionnaire at best suggests the results should be taken lightly. Nonetheless, none of the responses indicated that the hints were misleading, even mildly misleading. Students seemed generally happy with the hints and a clear majority indicated that the hints would be helpful to students in the future when HintC could be available for the full semester.
5.5 Comparing Data from HiC to Data from HintC

This section compares data from the submission logs where HiC had been used with those where HintC had been used. Most of the data was discussed in Section 5.1 (dealing with HiC) and Section 5.2 (dealing with HintC).

5.5.1 Development Environment and Number of First Submissions Differences

In a fashion similar to that used in Section 5.1.5, this section analyzes the assignments using the number of differences on first submissions. The aforementioned section establishes that there are many more first submission differences for programs than for labs and that the number of differences on first submissions varies among assignments within an assignment type.

Considering that the feedback from HintC on non-computed output and output differences does not present itself before that first submission, having used HintC or HiC to provide a first submission in itself should not have affected the submission. A ‘box-and-whisker’ plot of the number of first submission differences grouped just on which development environment generated the submission is shown in Figure 5.17.

As can be seen in Figure 5.17, the mean when using HintC is just a bit higher than those when using HiC but the HiC group is much more varied than its HintC counterpart. Part of the variance present in the HiC group can be explained because this graph, just like the one in Figure 5.20, does not account for the type of assignment and thus includes data from both labs and programs for HiC but only from programs for HintC.

Since the number of differences on first submissions differs based on assignment type, and since data from lab assignments which were submitted using HintC was not available for examination for this thesis, a more appropriate comparison excludes data from the lab assignments. Figure 5.18 contains a plot of the number of differences on first submissions for the program assignments, grouped by development environment.

Here again the HiC group is much more varied than its HintC counterpart. This too can be partially explained because the HiC group contains data from several data sets (which vary from one another) whereas the HintC group contains just one data set. As such, Figure 5.19

---

2The ‘non-computed output’ / ‘output differs’ feedback component of HintC is only available when submitting. However, the rest of the hint system is available for all of the other runtime errors without requiring a submission. The submission log data made available for this thesis contains submissions and no data regarding what the students might have been doing within the IDE before submitting. Although it could have been the case that students were benefiting from HintC’s feedback by running the program without submitting, it seems unlikely, given the in lab observations and the actual contents of the submissions that students were thoroughly using the compilation system for debugging (thus benefiting from HintC) before or instead of providing submissions. It seems more likely that in lieu of compiling without submitting, students were providing submissions.
Figure 5.17 – Number of Differences on First Submissions (all data sets, grouped)

Figure 5.18 – Number of Differences on First Submissions (programs, grouped)
Figure 5.19 – Number of Differences on First Submissions (programs)

shows a graph similar to the one found in Figure 5.9 though this time includes submissions generated by HintC from program assignments.

The finding from Section 5.1.5 that the number of first submission differences varies from one program assignment to the next seems to still hold true with the data set of the program assignment. Given the data and the findings thus far, a more direct comparison of the number of submissions provided when using HiC versus when using HintC might be best obtained by comparing the following two data sets: f05prog03 (HiC) \((M = 89.27536, SD = 81.4618)\) and s07prog03 (HintC) \((M = 64.56667, SD = 48.17904)\). Of the data sets available these two seem most similar based on the mean and standard deviation of the number of differences on first submissions.

### 5.5.2 Development Environment and Number of Submissions

A total of two development environments were used to generate the various data sets. The question this section addresses is if the development environment affects the number of submissions. The assertion is that using HintC (in place of HiC) will result in a decrease in the number of submissions. Therefore, the null hypothesis \((H_0)\) represents the case in which the number of submissions does not differ significantly between the two development environments or that there are fewer submissions when using HiC than when using HintC. The alternate \((H_A)\) represents the case in which the aforementioned assertion holds true: there are fewer submissions when using HintC than when using HiC.

The same data, outlined in Section 3, is used again. The data consists of five labs and
six programs. Ten of the data sets were generated using HiC and one was generated using HintC. The first round of analysis compares the number of submissions from all data sets generated by HiC with all of the data sets generated by HintC. A graph of the means of the two is shown in Figure 5.20.

According to the graph (Figure 5.20), there does not appear to be a big difference between the two. The graph does show that there is greater variance within the HiC data set than within the HintC data set. It also shows that the average number of submissions is actually lower when using HiC than when using HintC. At first glance this result seems unexpected.

However the graph in Figure 5.20 does not account for at least one established source of variance: the type of assignment. According to findings in Section 5.1.1, the type of assignment affects the number of submissions and the number of submissions for labs is much smaller than for programs. Three hundred submitters providing lab submissions, all with relatively few submissions, would bring the average down considerably.

Because the available submission log data generated by HintC is of a program assignment type, a more appropriate comparison would exclude labs. Figure 5.21 contains just such a graph where one sees that the average number of submissions with HiC is higher than the average number of submissions using HintC.

The groups do not share an equal variance and so again the Welch’s t-test is the t-test of choice. The results of the t-test are as follows: $t(150.218) = 6.3524$, $p-value < 1.195e-09$. Given such a small $p-value$, it is unlikely that the observed difference between the two groups is merely a coincidence. As such the null hypothesis is rejected and the alternate hypothesis accepted. Based on this analysis, students provide fewer submissions for programs when using HintC ($M = 8.4, SD = 4.952185$) than when using HiC ($M = 17.65, SD = 17.94482$).

Merely obtaining a small $p-value$, which corresponds to a true difference between the
means, does not necessarily indicate that the findings are significant or interesting. In this case, however, students provide twice as many submissions when using HiC ($M = 17.65$) than when using HintC ($M = 8.4$).

It could very well be the case that factors other than the development environment affected the results. Whereas this statistic does not exclude that possibility, it does indicate that despite the differences within each group, the difference observed between the two groups could be explained with the independent factor (the development environment). Perhaps, for example, the program completed using HintC was simply easier than those completed using HiC. Perhaps the group of students using HintC generally submitted fewer times than the group who had used HiC. The available data does not allow further investigations into the latter but the former can be addressed by using the number of differences on first submissions as a comparison metric.

Section 5.5.1 suggests comparing $f05prog03$ (HiC) and $s07prog03$ (HintC) because they seem most similar based on the number of differences in first submissions. Despite both being named Program 3, these data sets refer to the 3rd program in the given semesters and are different assignments$^3$.

The null ($H_0$) and alternate ($H_A$) hypotheses remain as they were at the beginning of this section. In comparing the number of submissions from the one data set with the other, the results of the Welch’s t-test are as follows: $t(94.073) = 2.3317, p – value < 0.01092$. At the $\alpha = 0.05$ level, the t value is large enough and thus $H_0$ could be rejected and $H_A$ could be accepted. The results back up the claim that the data set where HintC was used had fewer submissions.

$^3$A small description of $f05prog03$ can be found in Section 3.4.2 and a description of $s07prog03$ in Section 5.2.1
However, considering there are many other potential sources of variance, many variables which were not excluded with the design, it seems premature to conclude that using HintC results in fewer submissions. Among other things, the two assignments are different both in their content and in the number of differences on first submissions. Furthermore, the data sets are from different student groups and it is not known if in general the one group merely submitted more than the other group.

5.5.3 Development Environment and Number of Zero-Difference Submissions

This section analyzes the difference, if any, between the number of matching-output submissions provided by the two development environments. The expectation is that students submitting through HintC will provide fewer zero-difference submissions than those submitting through HiC, perhaps because they are more aware of what they are doing submission to submission and need not submit just for the current feedback.

A graph similar to the one found in Figure 5.6 of the zero-difference submission for programs is shown in Figure 5.22. The program data sets generated through HintC have been included. Since it is known that the number of zero-difference submissions varies between assignment types (shown in Section 5.1.3) and there were no lab assignments generated with HintC made available for analysis in this thesis, the test should only include the program data sets. Running an ANOVA across all of the program data sets available provides the results shown in Table 5.8, which are statistically significant at the 0.001 level.
Although it was shown in Section 5.1.4 that the number of zero-difference submission varied among lab assignments, the same did not seem true with the program assignments generated by HiC. Nonetheless, the test can account for variations from particular data sets.

Results from running the ANOVA again but accounting for the differences within each assignment are shown in Table 5.9. The F value of 8.6580 for IDE as an independent variable is significant at the 0.001 level while the F value of 1.8997 for the Data set as an independent variable was not statistically significant. As confirmed by the ANOVA, there is a difference across the IDE variable. Considering there are only two entries, further analysis is not needed to isolate which of the set differs from the others. Instead, the next step is to run a t-test.

The null hypothesis ($H_0$) represents the case in which the number of matching-output submissions does not differ significantly between the two development environments or the case in which there are fewer such submissions when using HiC compared to when using HintC. The alternate ($H_A$) represents the expected case in which there are fewer zero-difference submissions provided through HintC than through HiC.

The Welch’s t-test results are as follows: $t(96.336) = 5.8437, p-value < 3.469e-08$. The t value is large enough to indicate statistically significant results at the 0.05 level. The null hypothesis is then rejected and the alternate accepted. As was the case in with the results in Section 5.5.2, these results could be used to back up the claim that students provide fewer zero-difference submissions when using HintC as than when using HiC.

Although the statistics indicate a likely-not-coincidence difference between the two groups of data, this alone does provide the data needed to assert that HintC was or its hints were responsible for students having provided fewer matching-output submissions.

### 5.5.4 A Comparison of Two Correlations

The correlations from four sets of data were calculated. In all cases the correlations were positive but varied in strength. That they were all positive is not so interesting because the
correlations could not have been negative: the least number of submissions one could submit is one and the least number of differences a submission could contain is zero. The different equations for the least squares regression lines and the differing strengths are interesting and worth further inspection.

Of the four sets, the correlation between the number of differences in first submissions and the number of submissions was the weakest with the lab assignments available in the HiC data sets and strongest with the program assignment in the HintC data set.

The weak correlation with the labs could be indicative of several possibilities. It does imply a poor direct relationship between the two factors in the correlation. Although it seemed that both the number of submissions and the number of differences in first submissions were indicators of an assignment’s difficulty, this is less true, if true at all, for lab assignments. Another possible explanation is that students are usually taught how to use the IDE and the submission process through some of the lab assignments. This too could play into the variation in the number of submissions seen within the labs.

The linear dependency between the two variables for the programs in the HiC data sets is about midway between being a perfect one to one correlation and no correlation at all. Equation 5.2, representing the line of best fit for the data set, claims the number of submissions can be determined by adding 6 to the product of 0.094 and the number of differences in the first submission.

This formula is the equation describing the mean number of differences in first submissions as they relate to the mean number of submissions for the given data set. The data set had an average of 120.7 differences in first submissions and a mean of 17.65 submissions per student. Carrying out the equation provides the expected results:

$$\text{Number of Submissions} = 0.0944 \times \text{Number of First Submission Differences} + 6.2559$$

$$= 0.0944 \times 120.7 + 6.2559$$

$$= 11.39408 + 6.2559$$

$$= 17.64998$$

The dependency between these two variables for the submissions provided through HintC was also calculated. Of the four sets, the correlation between number of differences in first submissions and the number of submissions was the strongest with the program assignment submissions available in the HintC data set. One possible explanation for this is that the group included submissions from only one data set.

Equation 5.4 describes the best fit line for the program data set provided through HintC. With an average of 64.56 differences on first submissions and an average of 8.4 submissions per student, the formula accurately describes the relationship between the two variables via a linear equation:

$$\text{Number of Submissions} = 0.05778 \times \text{Number of First Submission Differences} + 4.66961$$

$$= 0.05778 \times 64.56 + 4.66961$$

$$= 3.730277 + 4.66961$$
Actually this equation, Equation 5.4, seems most similar to the equation corresponding to the lab submissions, Equation 5.1. This might indicate that this particular assignment was closer in difficulty to the labs than to the other programs.

The question is, though, of the use of these pieces of data. The notion is that these variables and the correlation between them can be used to quantifiably describe the assignments. The assignments could then be assessed, not with the intent to reduce the number of submissions or differences in first submissions, but rather just to obtain some comparative data regarding the different assignments.

Other extraneous factors should be ruled out to better assess the true nature of the assignment. Many such potential sources of variation could not be factored out or accounted for with the data available for this thesis.

Further tests would need to be carried out to confirm the validity of the assertions that these variables and the correlations indicate assignment difficulty. Nonetheless, assuming such is the case, then the variables can be used to compare program submission data in the HiC group with that in the HintC group. Even though the number of submissions and number of differences in first submissions were lower in the submissions from the program submitted through HintC, the line of best equations allow any arbitrary number to be used in order to determine the other.

If, for example, the linear dependency shown in Equation 5.4 holds true and the average first submission contained 120.7 differences (the average number of differences from the programs in the HiC data set), one would expect just under 12 submissions per student, on average.

\[
\text{Number of Submissions} = 0.05778 \times \text{Number of First Submission Differences} + 4.66961
\]

\[
= 0.05778 \times 120.7 + 4.66961
\]

\[
= 6.974046 + 4.66961
\]

\[
= 11.64366
\]

With an average of 120.7 differences in first submissions, 17.65 submissions are expected per student in with the programs submitted through HiC while only 11.64 submissions are expected per student with the program submitted through HintC. With the program in the HintC data set, around 33% fewer submissions would be expected than with the programs in the HiC data set. Again, with the data available, this provides a means to compare the data sets but does not provide a means for isolating the effect of one development environment over another.

### 5.5.5 A Comparison of the Number of Lines changed

Beyond just providing submissions, students usually were making some changes to the solution before providing another submission. Data on the number of lines being changed
Table 5.10 – ANOVA : Number of changed Lines by IDE

<table>
<thead>
<tr>
<th></th>
<th>Df</th>
<th>Sum Sq</th>
<th>Mean Sq</th>
<th>F value</th>
<th>Pr(&gt;F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE</td>
<td>1</td>
<td>664</td>
<td>664</td>
<td>5.7344</td>
<td>0.01676</td>
</tr>
<tr>
<td>Residuals</td>
<td>1501</td>
<td>173725</td>
<td>116</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

between submissions has been presented in three sections: in Section 3.3.2 on a lab assignment submitted through HiC; in Section 3.4.4 on a program assignment submitted through HiC; and in Section 5.2.1 on a program assignment submitted through HintC.

The number of changes made between submissions may be an indicator of qualities such as students’ learning experiences, of the effectiveness of an assignment, or of the transition students make from novices to the next level. This unit as an indicator of any of these has not be established yet however. It is not yet understood if a student should be making more changes between submissions or fewer changes and which of these two is more likely a sign of a struggling student. It might also be the case that some students provide submissions when program is mostly finished and just need to adjust a few lines, using the submission process to narrow down which lines. In this case the number of changes per submission might be small though one could imagine students with a better understanding of the program as a whole might make more changes, fixing all known issues, before providing the next submission. In this case then there should be fewer submissions but more changes between submissions.

This section compares data on the number of lines changed between submissions between the two sets of nominal data. It looks at the data as is in an attempt to understand what the data may signify. Of the analyses containing detailed ‘number of lines changed’ data, two are from program assignments and one is from a lab assignment. Given what is known about the differences the assignment type makes, data from the two program assignments is compared.

The data set from Section 3.4.4 has an average of 4.444965 changed lines per submission with a standard deviation of 11.41683. The other data set, the one from Section 5.2.1, has an average of 2.572072 changed lines between submissions with a standard deviation of 5.581298. An analysis of variance, the results (significant at the 0.05 level) of which are shown in Table 5.10, indicates that the means from the two groups are unequal.

Students from the Spring of 2007 did not change the same amount of lines between submissions on their assignment Program 3 as the students from the Fall of 2005 did on their assignment Program 5. At the 0.05 level, the Welch’s t-test indicate that the true difference between these means is greater than 0 ($t(602.98) = 3.8066, p-value = 7.763e-05$) and greater than 1 ($t(602.98) = 1.7741, p-value = 0.03827$). Results of the test for a difference greater than 2 or even 1.1 were not significant.

Thus the difference in the number of lines changed between submissions for these two data sets is rather small. From a professor’s point of view, it might not even be important that the students are changing 5 lines between submissions instead of just 4 lines. Furthermore, the effect of any one particular variable, such as the IDE in use, cannot be determined from this
type of data. Although this data correlates HintC with few changes between submissions, mere correlations do not imply causality and HintC itself is just one of the many factors that can be correlated with the lower number of lines changed.

### 5.5.6 A Comparison of Types of Changes

Along with lines being changed (Section 5.5.5), the types of changes being made were also noted for several data sets. This thesis presents data on the types of changes made between submissions in three of the previous sections: in Section 3.3.2 on a lab assignment submitted through HiC; in Section 3.4.4 on a program assignment submitted through HiC; and in Section 5.2.1 on a program assignment submitted through HintC.

Figure 5.23 shows the types of changes from the three data sets. In two of the data sets shown, one lab and one program, most changes were to the non-computed output. In the other data set, a program, most changes were to the main algorithm followed by non-computed output changes.

Excluding data from the lab assignment and grouping together the several smaller categories results in a graph such as the one shown in Figure 5.24. This graph seems to just make it more obvious the extent to which different things were being changed between submissions between the two assignments which is probably an indication of the differences between the two assignments themselves or of the students providing the submissions.
5.5.7 A Comparison of the same Assignment in two Settings

Two of the data sets contain submissions from the same assignment. Section 3.4.1 and Section 5.2.1 both reference the “Water Everywhere” program assignment. Full submission logs are only available from one of the two data sets. Therefore, statistics on the ‘types of changes between submissions’ and ‘number of lines changed between submissions’ cannot be compared. The data available is sufficient, however, for comparisons of the basic submission statistics: ‘number of differences on first submissions’, ‘number of matching output submissions’, and ‘number of submissions’.

Whereas both data sets use the same, or very similar, assignment, the IDE (HiC or HintC) used and the group of students providing submissions are not the only differences between the two groups. Some of the other differences include the following: the semester in which the course took place (fall or spring), the instructors for the course, the calendar year in which the submissions were provided (2004 or 2007), and the format of the course itself (the former being a special, shorter term).

This need not mean that the data cannot be compared. Without knowing otherwise, the groups are assumed equal. The null hypothesis ($H_0$) in all cases refers to the situation in which the groups do not different significantly on the given measure. The alternate ($H_A$) is that there is a difference between the two groups for the given measure.
First Submission Differences with Water Everywhere

A graph of the number of differences in first submissions is shown in Figure 5.25. A Welch’s t-test comparing the two on first submission differences results in the following: \( t(39.116) = -4.2311, p-value < 0.0001358 \). The results are significant at the 0.05 level (two-sided) where a t value of 1.95996 is required. In this case the null hypothesis (\( H_0 \)) is rejected and the alternate (\( H_A \)) accepted, allowing one to conclude that the observed differences are more than likely not coincidental.

It seems fairly obvious from the graph (Figure 5.25) that the two sets are different. The variance is unequal and average differences between the two groups appears to differ by around 40 differences. Even though the IDE changed between the two, its effects, if any, should only appear after the first submissions. Therefore, it seems there are explanations for the differences between these two groups other than the IDE in use.

Number of Submissions with Matching Output with Water Everywhere

Figure 5.26 contains the graph showing the number of submissions with matching output from the two data sets. The graph shows how fairly similar these two groups seem to be in terms of the zero-difference submissions. The averages from both sets appear to be the same and the group variances appear similar. This is the case with the mean and standard deviation from the \( f04prog03 \) data set at \( M = 1.5833, SD = 1.505042 \) and the mean and standard deviation from the \( s07prog03 \) data set at \( M = 1.6, SD = 1.792706 \).
A Welch’s t-test comparing the number of matching output submissions from both sets provides the following results: $t(24.086) = -0.0306, \ p-value < 0.9758$. The results are not significant at the 0.05 level (two-sided) where a t value of 2.06390 is required. In this case the null hypothesis ($H_0$) is not rejected and the alternate ($H_A$) is not accepted. It could be said that the two data sets do not vary considerably in this regard.

Number of Submissions with Water Everywhere

The numbers on the amount of submissions from each data set can be seen in Figure 5.27. Data from the graph seems to indicate that more submissions were provided, on average, in the $s07prog03$ data set than with the $f04prog03$ data set. Considering that the graph just illustrates the data provided to it, this is the case with these two data sets. Given the data ($M = 6.75, SD = 6.995128$ submissions in the $f04prog03$ data set and $M = 8.4, SD = 4.952185$ submissions in the $s07prog03$ data set), it seems that there is certainly a difference between these two groups. It is possible, though, that the observed values, though different from one another, are not different enough to support claims of the groups certainly being different.

A Welch’s t-test is used to compare the two groups with regard to the number of submissions. Results from the t-test are as follows: $t(15.615) = -0.7458, \ p-value < 0.4669$. The results of the t-test are not significant at the 0.05 level (two-sided) where a t value of 2.06390 is required. The null hypothesis ($H_0$) is not rejected and the alternate ($H_A$) is not accepted. If a true difference between the group means exists, it could be the small sample size that contributes to the low t-test value. However, even assuming all other things equal but with
a sample size large enough to allow for more than 30 degrees of freedom, the results here still fall short of the required t value of almost two. This means that although the groups are not identical, the difference between them is not large enough to compensate for the differences within each.
Chapter 6

Conclusions

Following the analysis of available submission logs, this thesis provides information describing some of the characteristics of the available data. This thesis introduces the concept of non-computed output and pointed to students’ attention to it as well as the concept of what is called thrashing and its attributes.

One may be very tempted, given the data shown in Chapter 3 and Chapter 5, to conclude that students from the Spring of 2007 are inferior to those in the Fall of 2005, that the generally better students take the course in the Fall, that the labs are too easy and the programs are too difficult (Figure 5.7), that students spend most of their time fixing non-computed output (Figure 3.3), that students change less than four lines of code between submissions (Figure 3.8), that students struggle with some of the rudimentary programming concepts (Figure 3.4), that HintC and its hint system actually increase the number of submissions, they decrease the number of changes to the main algorithm and increase the number of changes to the non-computed output (Figure 5.24).

Neither the data nor the analysis presented throughout this thesis supports any of the claims above. The analysis is a meta-analysis of existing data and correlative in nature, not experimental. Thereby it allows for descriptions of the observations and correlations. Correlations alone do not constitute causality.

Furthermore, many factors likely contributed to the properties observed. For example, the data to track a particular submitter was eliminated in the process to provide the non-student-identifiable submission log data for analysis in this thesis. The identifier used to refer to a particular submitter was consistent throughout one group of submission logs (one data set) but not necessarily so throughout the duration of the course (thus spanning several data sets). That is, the extent to which the number of submissions, types of changes, differences on first submissions, and so on varied from one student to the next was lost. Eliminating this data was probably not a design goal of the removal of the identifiable portions from the submission logs but was likely not foreseen as a requirement either.

Understandably, the student’s username (which usually contains the last name) should be removed from the file containing the submission but its replacement should be consistent
for all submissions provided by that same student. This should help achieve the goals of providing an anonymous form of the submission logs and of maintaining data about one possible source of explanation of variance.

Analysis can then use this data to account properly for differences among students within data sets. It can also follow patterns from the same sources among data sets and between assignment types. It might be, for example, that some students generally provide many more submissions than some others or that some others, as a personal rule, submit after each and every change. Without being able to account for such cases, a data set with more one type of student than the other may seem to indicate struggling students or poor assignments.

This, students as sources of variations in the submissions, is one of the more obvious factors but there are plenty of others. The submission log data did not explicitly contain data about which professors instructed the course sections during which the submissions were provided. It seems feasible that this too might make for a good correlation. Without claiming that some professors are worse than others, some probably have their own instructing styles and/or expectations of how students should be using the development environment and the submission process. Even in the theoretical case that the students and assignments alone are the main factors causing the data to be as it is, being able to have accounted for the role of the instructor, if any, would still be beneficial.

Other factors such as the times of classes (morning, midday, afternoon, evening) and days of the week for class were also not available in the submission log data. Some data about the semester in which the course took place during which the data set was created (submissions provided) was implicitly available. To this end, the sample size of data sets was rather small, including just five program samples from the fall semesters and one from the spring semesters. It may very well be the case, for example, that students taking the course in the fall semesters perform ‘worse’ than those taking it in the spring semesters. This might be, though, because those taking the introductory course in the spring probably contributed to the lower performance in the fall when they took it the first time and performed better when taking it a second time in the spring.

Also unknown is the prior programming knowledge/experience students have before taking the course and before providing submissions. It might be meaningful to combine the submission data statistics to performance on exams, achievement in the course, and ultimately success thereafter. Data on students’ exam scores and course grades was not available in the submission logs.

Another factor which seems like it would be very insightful, especially from the pedagogical perspective, is that of the goals of the assignments. That is, different assignments were used to teach different concepts. It would be nice to have the data detailing with which assignment a concept (loops, structures, pass-by-reference parameters, ...) was introduced and which concepts were required with which assignments. Such data, when analyzed well, could provide evidence for the department’s claims that a particular assignment teaches about a particular topic or that one assignment does a better job of it than another. It could also help understand what was labeled as thrashing and seen as struggling. Are the students thrashing? Are they struggling with the newly introduced concept or one that had
been introduced several assignments ago? Are they struggling with a particular assignment but not the particular concept? Or are they struggling with the wording and formatting of some non-computed output appearing in a particular assignment?\footnote{To this point, it would be interesting to know where professors thought students would have problems. These predictions could be correlated with actual struggles to see just how well professors understand the issues facing their students.}

All is not lost. The analysis shows that the wealth of data known as the submission logs can be molded into information and that with a little more work made even more informative and insightful. This alone warrants further analysis and investigation.

With the data available and analyzed, the most insightful glimpses into the submissions come from two sources: one is the several assignments from the the Fall of 2005 and the other is of the lone program assignment that was used in two settings.

With the former, the series of data sets all from the Fall of 2005, the collection of students remained the same and thus the group as a whole was the same. Although variations could not be traced to individual submitters, keeping the group the same is at least one less factor when comparing the submission logs from the assignments. The number of submissions varied from program to program (not so with the labs) and appeared to increase as the course progressed save the last one in the data set. In contrast, the number of matching-output submissions varied from lab to lab (not so with the programs). The number of differences in first submissions appeared to decrease with each consecutive lab assignment whereas the first submission differences in programs seemed to increase with each successive program assignment except for the last one. Actually, the data may have come from different sections and thus different students with different professors. If this is so, the variables for term (Fall semester) and year (2005) remain set while the rest go unexplained.

The latter, the “Water Everywhere” program assignment, is interesting because it was used with two groups of students. The core of the assignment remained the same while the students and group of students probably were not the same. However, the indicator of how similar the submission log data is, that of the number of differences in first submissions, indicates that the two sets are different. This and the plethora of other differences do not provide the basis needed for attributing any particular differences between the two to one particular item (such as the term, the IDE, the instructor).

Although the analysis might not be detailed or encompassing enough to describe a clear picture of how the students complete their assignments, it probably describes the picture well enough to indicate that it is not yet clear just how the students complete their assignments and how the several factors interact with each other. The analysis does seem to indicate that students use the submission process not merely as the method of turning in completed assignments. It seems to indicate that they use the process during development perhaps as a tool to assist with debugging their code. Furthermore, the analysis seems to indicate that things not intended to receive a lot of attention may receive a lot of attention. This can partially be seen in Figure 5.23, the graph showing the types of changes from three data sets.
Future action should harness the data in the logs, extend it where needed to make it more useful and complete, and scientifically examine the correlations and effects of the several aforementioned variables. Also, expectations from the submission process should be well defined and well explained to the students so as to have more consistent use of the tool.

Was HintC a step in the right direction? With the data available and type of analysis performed, it cannot be claimed that submissions or student learning improved (or worsened) through the use of HintC. The hints of HintC could use tweaking. The data presented in this thesis did indicate three instances in which the hints seemed less than optimal. Responses to the questionnaire do indicate that HintC’s more specific, context-sensitive feedback was well received.


Appendix A

Code to which Hints refer

This section shows code samples to which hints mentioned throughout the thesis refer. All of the sample code in this section comes from the data set in which HintC had been used as the IDE. This data set was Program 3 from the CS1 class in the Spring 2007. The assignment is discussed in Section 5.2.1.

Each of the five code snippets comes from a different student. The submissions from which the samples shown in Table 4.4, Table 4.3, and Table 4.5 came contained no runtime errors but were only flagged as having mismatching output.

The other two, Tables A.5 and A.1, contained runtime errors. The submission containing the code in Table A.5 was flagged as probably containing an infinite loop. The submission containing the code in Table A.1 was flagged as utilizing a variable before initializing it or assigning it a value.

The samples are shown with the line numbers appended to the solutions by HintC during the submission process. A complete submission generated through HiC can be found in Appendix B.

```c
38:     void dist (float Distance, float number)
39:     {
40:         cout << "Strike " << number << " goes " << Distance << " feet."
41:     }
```

Table A.1 – Source Code Segment Referred to by Figure 4.7
Table A.2 – Source Code Segment Referred to by Figure 4.4

```cpp
67: while (count < 5)
68: {
69:     count = count + 1;
70:     if (comparison < shortdistance)
71:         {
72:             cout << "Strike " << count << " goes " << comparison << " feet. " << endl;
73:             cout << "You missed: you were " << distance - comparison
74:                 << " feet too short." << endl;
75:         }
76:     else if (comparison > longdistance)
77:         {
78:             cout << "Strike " << count << " goes " << comparison << " feet. " << endl;
79:             cout << "You missed: you were " << comparison - distance
80:                 << " feet too long." << endl;
81:         }
82:     else if (comparison > shortdistance && comparison < longdistance)
83:         {
84:             cout << "**** YOU HIT THE TARGET! ****";
85:             return count;
86:         }
87:     angle = A(count);
88:     velc = V();
89:     comparison = (( velc * velc ) * (sin(2 * Radians(angle))))/(32.2);
90: }
```

Table A.3 – Source Code Segment Referred to by Figure 4.3

```cpp
64: float input(float target)
65: {
66:     float degrees, velocity, dis, rad, num, off, err;
67:     num = 1;
68:     while (num <= 5)
69:         {
70:             cout << "Enter angle (in degrees) for guess " << num << ": ";
71:             cin >> degrees;
72:             cout << "Enter velocity (feet/sec): ";
73:             cin >> velocity;
74:             rad = Radians(degrees);
75:             dis = distance(velocity, rad);
76:             cout << "Strike " << num << " goes " << dis << " feet." << endl;
77:             off = error(dis, target);
78:             if (off == 0)
79:                 return 1;
80:             num = num + 1;
81:             if(num > 5)
82:                 cout << "You need more practice; you missed after 5 tries.";
83:         }
84:     return 0;
85: }
```
APPENDIX A. CODE TO WHICH HINTS REFER

Table A.4 – Source Code Segment Referred to by Figure 4.5

Table A.5 – Source Code Segment Referred to by Figure 4.6
Appendix B

Sample HiC Submission Log

[Student’s username] LAB[#] Differences: [##]
[FirstName] [LastName] Section [#] Date of Submission: [YYYY MM DD]

Your submission as of the above date using HiC, v3.0.16. If this is a programming assignment, please review the assignment writeup and programming ground rules to make sure you didn't miss anything.

01: #include <iostream.h>
02: #include <iomanip.h>
03: const int PRECISION = 3;
04: float average(float total, int count);
05: void updateAverageStats(int& count, float& total, float next_score);
06: void updatePinWinners(float next_score, float& silver_score,
07: float& gold_score, int& silver_count, int& gold_count);
08:
09: int main() {
10: float score, gold_threshold, silver_threshold, sum = 0.0;
11: int count = 0, silver_winners = 0, gold_winners = 0;
12:
13: cout << setiosflags(ios::fixed) << setiosflags(ios::showpoint);
14: cout << setprecision(PRECISION);
15:
16: cout << "Score needed for Gold Pin: ";
17: cin >> gold_threshold;
18: cout << "Score needed for Silver Pin: ";
19: cin >> silver_threshold;
20:
21: cin >> score;
22: while ( !cin.eof() ) {
23: updateAverageStats (count, sum, score);
24: updatePinWinners (score, gold_threshold, silver_threshold,
25: silver_winners, gold_winners);
26: cin >> score;
27: }
28:
```cpp
29:    cout << endl;
30:    cout << "Average: " << average(sum, count) << endl;
31:    cout << "Gold Level " << gold_threshold << "": "
32:        << gold_winners << endl;
33:    cout << "Silver Level " << silver_threshold << "": "
34:        << silver_winners << endl;
35:    return 0;
36: }
37:
38: float average(float total, int count) {
39:    float average;
40:    if (count > 0 ) {
41:        average = total / count;
42:        return average;
43:    }
44:    else
45:        return 0;
46: }
47:
48: void updateAverageStats (int& count, float& total, float next_score) {
49:    total += next_score;
50:    ++count;
51: }
52:
53: void updatePinWinners(float next_score, float& silver_score, float& gold_score, int& silver_count, int& gold_count) {
54:    if (next_score >= 18.0 && next_score < 20.0) {
55:        silver_score++;
56:        silver_count++;
57:    }
58:    else if ( next_score >=20.0) {
59:        gold_score++;
60:        gold_count++;
61:    }
62: }
63: }

******************************************************************
Output generated by your program from test test1 (line numbers added):

1: Score needed for Gold Pin: 20.0
2: Score needed for Silver Pin: 18.0
3: 19.5 20.7 12.2 24.0
4:
5: Average: 19.100
6: Gold Level 21.000: 2
7: Silver Level 20.000: 1

The CORRECT output (line numbers added for reference):

1: Score needed for Gold Pin: 20.0
2: Score needed for Silver Pin: 18.0
3: 19.5 20.7 12.2 24.0
4:
5: Average: 19.100
6: Gold Level 20.000: 2
7: Silver Level 18.000: 1

******************************************************************
The following describes how to turn YOUR output into the CORRECT output by inserting and deleting lines. Use it to help find the errors:

Insert after Line 5: Gold Level 20.000: 2
Insert after Line 5: Silver Level 18.000: 1
Delete Line 6: Gold Level 21.000: 2
Delete Line 7: Silver Level 20.000: 1

******************************************************************

Output generated by your program from test test2 (line numbers added):

1: Score needed for Gold Pin: 15.0
2: Score needed for Silver Pin: 12.0
3: 16.4 11.2 9.6 14.0 13.4 11.8 16.0 15.9 14.3 11.3
4: 
5: Average: 13.390
6: Gold Level 15.000: 0
7: Silver Level 12.000: 0

The CORRECT output (line numbers added for reference):

1: Score needed for Gold Pin: 15.0
2: Score needed for Silver Pin: 12.0
3: 16.4 11.2 9.6 14.0 13.4 11.8 16.0 15.9 14.3 11.3
4: 
5: Average: 13.390
6: Gold Level 15.000: 3
7: Silver Level 12.000: 3

******************************************************************
The following describes how to turn YOUR output into the CORRECT output by inserting and deleting lines. Use it to help find the errors:

Insert after Line 5: Gold Level 15.000: 3
Insert after Line 5: Silver Level 12.000: 3
Delete Line 6: Gold Level 15.000: 0
Delete Line 7: Silver Level 12.000: 0

******************************************************************

Results from test test3:

>>> YOUR OUTPUT MATCHES THE STANDARD OUTPUT <<<
(To see the full output, select Include All Test Output from the Submission menu)