2014

An Intelligent Debugging Tutor For Novice Computer Science Students

Elizabeth Emily Carter
Lehigh University

An Intelligent Debugging Tutor For Novice Computer Science Students

By

Elizabeth Emily Carter

Presented to the Graduate and Research Committee

Of Lehigh University

In Candidacy for the Degree of

Doctor of Philosophy

In

Computer Science

Lehigh University

May 2014

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy

_____
Date

_____
Dissertation Advisor
(Dr. Gang Tan)

_____
Accepted Date

Committee Members:

_____
Dr. Gang Tan

_____
Dr. Jeff Heflin

_____
Dr. H. Lynn Columba

_____
Dr. Glenn Blank

Acknowledgements:

First and foremost I would like to thank my de facto advisor Dr. Glenn Blank. He was the first faculty member I spoke to when I applied to Lehigh's PhD program and he has guided me through the entire process. His guidance and editing assistance have been absolutely invaluable. I would like to thank Dr. Blank, Dr. Henry Odi, Dr. H. Lynn Columba, Dr. Susan Szczepanski, Dr. Jennifer Swann, and Dr. Terry Hart for giving me the opportunity to work on the LVSTEM project, which funded most of my time at Lehigh. I would also like to thank my official advisor Dr. Gang Tan and my committee members (Dr. Blank, Dr. Tan, Dr. Jeff Heflin, and Dr. H. Lynn Columba) for their advice and for serving on my committee.

For their advice on my experimental design and the formal evaluation process I would like to thank Dr. Edwin Kay and Dr. Diane Bates from TCNJ. Professor Sharon Kalafut, Dr. Brian D. Davison, and Dr. Brian Y. Chen for their advice, feedback, and for allowing me into their classrooms to interact with their students. I would like to thank the following people from Phillipsburg High School, Cranford High School, and Warren Hills High School for assisting me in the IRB approval process for their schools and for welcoming me into their classrooms so that I could present my system to their students. From Phillipsburg High School: Mr. George Chando, Dr. Marianne Trapani, and Mrs. Laura Wojick; from Cranford High School: Dr. Paul Ward and Mr. Richard Bell; and from Warren Hills High School: Mr. Earl Clymer, Mr. Daryl Detrick, and Mr. John Hajdu. I'd also like to thank Mr. Detrick for allotting time for me to present my system to high school faculty at two of the CSTA meetings. I would also like to thank all of the students who participated in the evaluation – this dissertation would not have been possible without you!

I would also like to thank Heidi Wegrzyn, Jeanne Steinberg, and Brianne Lisk for all of their assistance with the required paperwork and other administrative issues I encountered.

This dissertation is dedicated to my Mother and Father, Nancy and Randy Carter, who have stood by me through everything and always encouraged me to pursue all of my dreams - even when those dreams ended up being more difficult to pursue than I had ever imagined. I am so grateful to have you as parents. Thank you so much for being my best friends.

Table of Contents

List Of Tables:

List Of Figures:

Abstract:

Debugging is a necessary aspect of computer science that can be difficult for novices and experienced programmers alike. This skill is mainly self-taught and is generally gained through trial and error, perhaps with some assistance from a professor or other expert figure. Novices encountering their first software defects may have few avenues open to them depending on the environment in which they are learning to program. The evident problem here is that the potential for a student to become stuck, frustrated, and/or losing confidence in their ability to pursue computer science is great. For a student to be successful when working professionally or progressing through academia they need to be able to function independently; trusting their own knowledge on par or above that of others so that their productivity does not rely on the knowledge of someone else. In order to solve this problem an Intelligent Tutoring System for teaching debugging skills to the novice utilizing Case Based Reasoning, Static Program Slicing, and the student's preferred learning style was proposed. Case acquisition and automatic Exercise Generation were also explored. The system built for this research program was evaluated using novice students at the College and High School levels. Results of this evaluation produced statistically significant results at the $p<.05$ and $p<.01$ levels, with generated exercises exhibiting significance at the $p<.01$ level. These results prove that the methodology chosen is a valid approach for the problem described, that the system does in fact teach students how to debug programs, and that the system is capable of successfully generating exercises on the fly.

1 Chapter 1: Why a Tutoring System for Debugging?

1.1 Introduction

Debugging is an intrinsic and difficult part of Computer Science for the novice, for the expert, and for software designed to assist in the process. The expert must apply their experiences with past defect encounters and their knowledge of a particular programming language. When encountering a new difficulty an expert may fall back on general problem solving strategies or look for Internet, human, and written resources in order to determine the cause for a given defect and how to resolve it. This differentiates itself from programming in that it requires the programmer to have a deeper comprehension of the machine and the language being used than the act of programming alone.

Computer systems for analyzing and correcting defective software perform static and/or dynamic analysis, use rules (ITS4) and patterns (FindBugs) but are limited by the Halting problem and their own static knowledge bases. The NP Complete nature of the Halting Problem (being able to determine whether or not an application will complete) makes it, in turn, an NP complete problem to determine if a computer program is correct. And a static knowledge base is unlikely to completely cover every defect a programmer may include in their code given that there are an infinite number of ways to write the same program.

The novice's problem is evident – they lack the analytical skills and domain information that experts and software analysis systems have. If students had a resource that could assist them in acquiring the skills and domain knowledge they require to

debug their own programs, they would stand a better chance of succeeding in their current and future course work. Given that there is little room in existing curricula for new material it makes sense that this information should be provided to the student by an external resource that is available to them on demand and in situ.

Students are more likely to succeed if they are not left to spin their wheels indefinitely, yet human tutors cannot be present for every baffling bug any single student may encounter. A system that can intelligently intervene and appropriately assist the student could help to bolster their knowledgebase and confidence, thereby making it less likely the student will drop a course or even switch majors because they find the material too difficult. Such a system could also help the instructor understand how they might teach debugging skills to their students, show the instructor what material their students might be struggling with, enable the instructor to proactively determine what knowledge gaps their students might have or what assistance their students might need, and otherwise assist the instructor in helping their students to learn this difficult skill.

1.2 Research Questions

An ITS for debugging raises many interesting research questions within the fields of Intelligent Tutoring Systems and Computer Science Education. Indeed, in the proposal for this study, a list of 15 questions were identified. The following discussion traces which questions have been actively pursued, which ones were dropped and why, and what interesting questions have been added.

With respect to Computer Science Education, the questions identified were:

- *How should debugging be taught?*
- *How could a system teach the domain?*

With respect to Intelligent Tutoring Systems, the questions identified were:

- *How should the domain be represented and reasoned about?*
- *How can the expert knowledge base be kept tractable?*
- *How can the system acquire domain knowledge?*
- *Could the system generate exercises dynamically?*
- *Should the system support other languages? How would supporting other languages change the system?*
- *How should the system model student knowledge?*
- *How could the system reason about student solutions without incurring the program verification problem?*
- *Does peer assistance factor in to modeling the student? How?*
- *How should debugging issues be remediated?*
- *How could the system determine if a given remediation is successful?*
- *How to communicate remediations?*
- *How to discuss the domain with the student?*
- *Should personae be used?*
- *Should the system facilitate peer communication?*

After pursuing this original list of questions the following questions were dropped, the rationale for dropping the questions follows:

Should the system support other languages? How would supporting other languages change the system?

Supporting other languages has been deemed out of scope, though it would certainly be possible to support other languages with the infrastructure that has been built. Some further discussion regarding this question is taken up in the future research section.

Does peer assistance factor in to modeling the student? How? And should the system facilitate peer communication?

Supporting peer assistance has been deemed tangential to the core research problem and hence out of scope for this dissertation. Additionally, there are concerns that facilitating peer assistance may open new avenues for cheating, an additional complication that would need to be addressed separately.

Should personae be used?

At this point the researcher believes the feedback mechanism of the system without animated personae is sufficient for the target audience. Students and instructors are satisfied with a user interface broadly in the style of an integrated programming (and debugging) environment, in which animated personae might just be a distraction to the students. Therefore this question became out of scope.

Two questions were added after the proposal and include:

- *How could the system generate exercises on demand?*
- *Could the system learn new solutions to known defects from the student interactions with the system?*

These questions were added in order to help address the static nature of the case base and exercise systems, in addition to increasing the Computer Science value of this research on the whole.

1.3 Contributions

This work contributes to the domains of Intelligent Tutoring Systems and Computer Science Education. Although several systems and tools have been built to try to assist the novice with debugging their own code, most of these systems focus on a single class of error (syntax, runtime, logical). ITS-Debug works to provide assistance to the novice over all three classes of defects. Even fewer systems are or aimed to become Intelligent Tutoring Systems None of the systems encountered in the existing research assist the student in the comprehensive manner ITS-Debug supports the novice, and none of the tutoring systems encountered in this domain concentrate on the Java programming language, which is the common introductory language for Computer Science majors today.

ITS-Debug addresses this domain in a novel manner, utilizing Case Based Reasoning (CBR) to represent and reason about the domain. Most existing ITSs utilize either rule-, model- or constraint-based reasoning (discussed in detail in Chapter 2). Few existing ITS utilize CBR and those that do, do not aim to assist students with the debugging domain. ITS-Debug utilizes CBR in three of its core modules. CBR is used

in the Domain Module as the methodology for diagnosing student defects and determining an appropriate fix for the issue. The Pedagogical module uses the case data in three ways. Both of these result in providing feedback to the student in order to assist them in solving their current defect. The first way that the Pedagogical module utilizes the CBR system is in order to determine the appropriate feedback to return to the student. Specifically, the module uses the case identified by the Domain module as the most similar in order to generate the feedback that is shown by the Communication module. Second, the Pedagogical module uses the case data to assist the student in the process of analogical reasoning. If the student has encountered the current case in the past and then encounters the same case in the future, the system will show the student a summary of the previous encounter (described in further detail in Chapter 4). Third, this module uses case data to provide the student with ad hoc, on demand exercises, thereby avoiding the predictability of a static exercise base. Finally, the Student module also uses the case data from the system in order to maintain a history of the cases the student has encountered and the results of these encounters in order to represent the student's current level of understanding of the domain. No tutoring system encountered in the research has employed case based reasoning this comprehensively or robustly throughout all core modules of the system or all the major aspects of CBR, including case acquisition.

Another way ITS-Debug contributes to the field of Intelligent Tutoring Systems (and alluded to above) is by exploring exercise generation. Many ITSs are limited by utilizing a static database of exercises. Once the student has completed all available

exercises in the system, the system is of little use unless the student wishes to continue practicing with the same exercises. ITS-Debug overcomes this limitation by producing exercises on the fly. Other work in generating exercises for debugging practice have utilized templates (Problets) and aspect oriented programming (Enbug). No system encountered in the literature created exercises by utilizing a code base of correct code that exemplified different programming concepts and constructs, then systematically breaking that source code to provide an exercise to the student. How ITS-Debug accomplishes this is discussed in more detail in Chapter 4.

This work also contributes significantly to the field of Computer Science Education. Debugging is often not a skill that is directly taught to the student. Although there are several works pursuing debugging instruction and offering guidance for best practices in this domain, few courses have implemented these recommendations in classrooms. This work explores how debugging could be taught and whether or not a system could be built to teach novices this skill without requiring instructors to adopt a new curriculum or fit new curricular material in already overloaded introductory courses.

In summary, the contributions of this work include:

- An ITS capable of teaching students to debug and providing students with targeted remediations for debugging issues across syntax, runtime, and logical defects;
- A CBR system addressing the unique problem of tutoring debugging skills;
- Insight into the applicability of CBR to program analysis;

- An exercise generation system capable of dynamically generating exercises by utilizing the afore mentioned CBR system;
- Insight into methods for teaching debugging and proof that the chosen methodology is successful in accomplishing this end.

The most important research questions can be organized under these contributions in the following manner:

- Intelligent Tutoring Systems
    - *How could the system reason about student solutions without incurring the program verification problem?*
    - *How should debugging issues be remediated?*
    - *How could the system determine if a given remediation is successful?*
    - *Could the system generate exercises dynamically?*
- Intelligent Tutoring Systems and Case Based Reasoning
    - *How should the domain be represented and reasoned about?*
    - *How can the system acquire domain knowledge?*
- Computer Science Education
    - *How could debugging be taught? Can debugging be taught to novices?*
    - *How could a system teach the domain?*

1.4 What is an Intelligent Tutoring System?

1.4.1 Architecture

It is a well-known fact that a student learns best when they are able to receive one-on-one instruction in what they are studying. Unfortunately it is impossible for every student to receive one-on-one tutelage all the time because there are not enough teachers or tutors to provide such instruction and because students often study on their own at odd hours. Intelligent tutoring systems aim to provide one-on-one instruction that can be available to all students at all times. Artificial Intelligence is combined with knowledge of pedagogical methodologies to provide a system that teaches the student a given subject matter. An ITS (Intelligent Tutoring System) typically represents up to four different kinds of knowledge: domain, student, pedagogical, and communication [18].

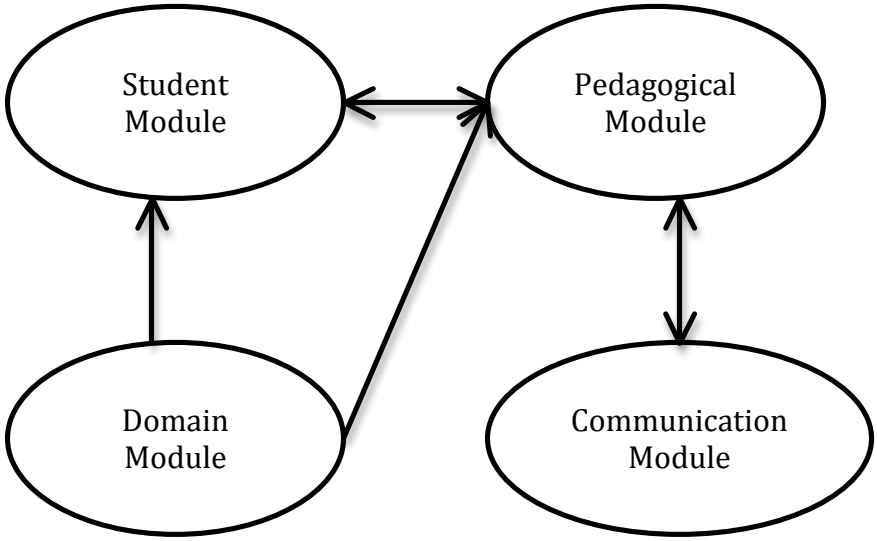


Figure 1: Intelligent Tutoring System Modules and Interaction

Domain knowledge is represented with an expert module representing how to solve problems in this field of expertise [8]. This part of the tutoring system provides the basis for interpreting student actions within the system. Classically the expert module is implemented as an expert system capable of generating solutions to the same

problems the student is solving. Because the expert module is meant to help teach and diagnose student actions within the tutor, it must include an ability to reason about the problem in aa way similar to how a human expert would [8]. When the expert module is not designed in this way, the tutor may recommend optimal actions in the context of solving each problem, yet it remains the student's responsibility to comprehend the context and rationale for an appropriate action. Usage of classical expert systems as the Domain Module is basically the educational equivalent of leaving the student alone with a list of questions and a list of answers. Simply knowing if they god the answer right or wrong does not tell the student how to not make the same mistakes over and over. If the student misunderstands the process by which to reach an answer in the given domain, providing the answers without explanation is not going to be educationally beneficial.

The student model is responsible for recording the student's knowledge state within the system and typically consists of two components: an overlay of domain expert knowledge and a bug catalog [8]. An overlay is a copy of the domain knowledge model; each knowledge unit within it receives a tag containing an estimate of how well a student has learned it. The bug catalog consists of a set of predicted misconceptions, each carrying a tag indicating if the student has exhibited the misconception or not [8]. Unfortunately there are several limitations to the bug catalog model. First, they are really only applicable to domains that are both fairly simple and procedural in nature. Additionally significant effort is required to compile all likely bugs within the domain as students will typically exhibit a wide range of issues within the domain in question,

to be effective the catalog needs to be comprehensive [18]. Finally, compiling a bug catalog by hand is not productive, as it may not be known if the bug will be exhibited by students using the system and the system may not be able to provide appropriate remediation for certain bugs [18]. An alternative to this methodology is to observe the behavior of student users and dynamically construct the catalog using machine learning [18].

The Student Module may also represent student actions, answers, the results of their actions, intermediate results, and verbal protocols [18]. A core assumption of the student model is that student behavior is an indication of what knowledge they have acquired and what misconceptions they may hold. Some representations that have been used to model both domain and student knowledge include semantic networks, rules, constraints, plan recognition, and machine learning.

The Pedagogical Module, embodying teaching knowledge appropriate for the domain, provides the ITS with the ability to determine when the system should intervene within the student's work. The decision to intervene may be based on the system's evaluation of student knowledge, learning style, and emotions [18]. Of these three, tutoring systems primarily utilize what the system has come to believe the student knows when making intervention decisions. The components of pedagogical intervention consist, at a high level, of objects, actions and navigation. Objects refers to the kind of intervention and can include such methods as providing an explanation or example, hints, quizzing the student, providing the student with an analogy, and others. Actions are the actual actions the module can take and include items such as test,

summarize, interrupt, demonstrate, teach the procedure, and others. Navigation refers to moving the student through the material and can include teaching the material step by step, ask questions therein taking a more Socratic approach, moving on to the next topic, going back to a previous topic, or staying on the current topic.

There are three categories of teaching approaches an ITS can take. Please note that these categories can and do overlap. These approaches include basing instruction on human teaching (collaborative learning, apprenticeship training, tutorial dialogue, problem solving, error handling), using methodologies that are informed by learning theory (Socratic learning, cognitive learning theory, situated learning, social interaction, constructivist theory), and facilitating learning with technology (such as using animated pedagogical agents and virtual reality).

Communication knowledge refers to the ability of the system to manage interaction between the system and the student [18]. Human teachers employ various strategies and methods when communicating with students [18]. For example, a teacher who notices that a student is disinterested may try to modify how they are teaching to garner interest. Methods utilized by intelligent tutoring systems include the use of synthetic humans (AI agents engineered to be realistic), virtual reality immersive environments, facial animation, and social intelligence (forming a social connection with the student).

1.4.2 Methods

The two classical methodologies that successful intelligent tutoring systems have used are model tracing and constraint based. Model Tracing tutors try to trace

student input in order to understand how the student arrived at a given answer, comparing and matching student steps to expert steps in order to determine when the student's logic does not match with the recorded solution. The concept of a bug library comes into play after a student has erred, it is used as a resource for identifying student errors and the corresponding reasons behind them in addition to the trace data [10]. If the trace the tutor captures of the student's solution contains one or more instances of rules in the bug library then the library dictates the remediation efforts that the tutor is to apply. Because of the reliance on tracing student input and an overlay or bug library, the capabilities of a model tracing tutor depend on how well the trace is captured (matching student steps with their corresponding expert steps) and how comprehensive the bug library is. In the ideal situation the tutor would be able to trace all student inputs and have a completely comprehensive library of common misconceptions. This ideal is not practical for non-trivial domains. Also there could be several alternate strategies to solve a problem. For each strategy supported the tutor would require expert rules and buggy rules for each strategy and the model tracing process would need to map the student's input to a particular strategy. If the student takes an unexpected path during their solution and reaches the wrong answer the tutor may not be able to remediate the issue.

Constraint-based tutors can avoid some of the pitfalls of their model-tracing counterparts. These tutors operate under the idea that it is enough to know that the student has made a mistake. Model-tracing tutors labor under the idea that diagnostic information is hidden within the student's actions, whereas constraint-based tutors

assume that the diagnostic information required to provide remediation is already in the problem state at which the student has arrived. These tutors have a library of constraints that are relevant to the domain in question. Formally a constraint is an ordered pair $<C_r, C_s>$. C_r represents a relevance condition, which is responsible for specifying when the constraint is relevant. C_s is a satisfaction condition, which should hold for any correct solution satisfying the relevance condition. If the problem state satisfies the relevance condition then the satisfaction condition should be satisfied by the problem state as well. If this is not the case the student has made an error. Constraint violation occurs if and only if the relevance condition for the constraint is true and the satisfaction condition is false [10]. Although knowing that a constraint has been violated or that an error has been made is pivotal to the tutoring process, knowing only that a given constraint has been violated can be a limiting factor on what assistance the system is able to provide back to the student.

Another approach is case based reasoning. Case-based reasoning is an AI paradigm wherein a system utilizes information about what has happened previously (cases) in order to solve a new problem [13]. This methodology is similar to the constraint based approach in that each instance of constraint violation could be seen as a case.

1.5 Nature of the Problem and Defect Representation

There are three general types of software defects: syntactic, runtime and logical. Program compilers detect syntactic defects and generate error messages. This mechanism works reasonably well for experts but may cause confusion for novices. The

error messages may be obscure for novices, especially when they assume knowledge of concepts that novices have yet learned. Also, most syntax issues are reported on the line in which they occur, yet often they are triggered by errors on previous lines (the line of reported as an error may actually be fine syntactically). In this situation, the real error occurs some number of lines above the reported defect and the programmer has to ferret out which line really causes the problem (e.g., unmatched parenthesis or brackets). While seemingly simple to experienced programmers, for novices this kind of defect becomes a larger issue when they do not have an adequate understanding of the syntax of the language they are using. More novice-friendly defect reporting should afford a novice better understanding of what they have done wrong in terms of concepts they should already understand, and how to avoid the mistake in the future.

Runtime errors are somewhat harder for a programmer to solve. Defects in this category do not emerge until the application is executed and the program terminates in an unexpected manner. An example of an exception the novice might encounter includes the InputMismatchException, which occurs when the student is using the Scanner class to read a specific type of data from console input and data of the wrong type is provided. At this point in their education the student may have just started learning about types in programming languages and is unfamiliar with the concept of exception handling. Thus the error message provided is not novice-friendly – the student receives a message like this:

Exception in thread “main” java.util.InputMismatchException

At StudentsClass.main(StudentsClass)

The student additionally receives the line number at which the program failed and a stack trace. For a novice, this type of error message is obscure and may even be a bit intimidating. They know that their program stopped, that there was some sort of problem that uses terms about which they are clueless, and a line number where the issue might be located, buried in a stack trace. Even if they identify the line number in question, the run-time error may well be triggered by conditions that occurred elsewhere in the flow of the program. If a programmer does not have an idea of where the cause of the error could be, an expert might employ a debugger and/or output statements to determine what part of their code causes the fault. However, novices have to learn these techniques and how to effectively apply them to the current problem. Most commercial and open source debuggers are not designed to be novice friendly; they are designed for expert programmers who already understand the concept of a breakpoint, stack traces, watches, and a host of other concepts. Also, finding the fault alone may not be enough to instruct the novice how to fix their code. If the issue is not something obvious to the student now they need to start checking the Web, asking their instructor, or consult with peers to determine what they did wrong.

Logical errors are usually the hardest to diagnose. This class represents programs that compile and run, but produce incorrect results. An example could be a function meant to produce factorials but instead produces products, or a stack implementation where the pop operation does nothing. Often novices don't even check for logical errors, let alone know how to diagnose them. However, the class of problems related to

determining if a program is correct is NP-complete, so there is no generalized solution possible for determining if input code will produce correct results. Likewise the problem of determining the cause of a logical error can be time consuming and frustrating for the novice. Experts know they should at least confirm that the output and/or behavior of the program matches what they planned. Once a discrepancy is found, heuristic knowledge about debugging can guide an expert to correcting the problem. It could be a simple mistake involving the incorrect use of an API function or operator. Or the error could really be due to the student having a fundamental misconception about the problem they are attempting to solve with their program, the programming language they are using, or the algorithm they have been instructed to implement.

Defects from any of these three classes may well pose a greater difficulty for a novice than an expert. There are many possible reasons for bugs inadvertently included in a student's code. Despite these challenges, they all point to the same overarching problem: there is knowledge the student does not yet have that is holding them back and, one way or another, they need to gain that knowledge to fix their programming defect. The student also needs to generalize this knowledge as a skill in order to gain confidence for debugging similar problems in the future.

2 Chapter 2: Teaching Debugging

Debugging and programming go hand in hand as it is unlikely that any programmer will code a perfect program the first time they compile and run it. Issues that a novice is encountering for the first time are issues they are likely to see again and again, in different environments and even with different programming languages. It is crucial that the student programmer learn how to investigate program defects for success as students and potential computing professionals. How should these skills be passed on to the student? How do experts solve their debugging problems? How do novices, with little or no previous programming knowledge, approach the debugging problem? Background research into these questions is pursued in the following sections.

2.1 Explicitly Teaching Debugging Skills

Some of the earliest work found related to teaching debugging was completed by [25]. Mathis implemented a course on debugging techniques with 15 separate modules, including 3 modules related to case studies, projects, and review. Some of these modules included software engineering concepts, different types of bugs, and aids for debugging. The course specifically used Fortran as the programming language and included techniques for designing algorithms in order to make them easier to debug. Though some of these ideas have found their way into computing courses on software engineering, courses that are explicitly focused on debugging or quality assurance are few and far between [25]. Moreover, there is a chicken and egg problem with respect to novices: how do instructors teach them what they need about debugging while also teaching them about programming and other computer science topics?

With regard to analyzing the novice debugger, Almazadeh et al. analyzed students' compile time errors over the course of a semester long class encompassing 15 exercises and 2 exams. During their analysis they identified 226 distinct semantic errors. Their results were distilled into 6 classes of errors over 7 computer science topics. The topics included conditionals, loops, methods, arrays, classes, files, and strings. The error classes included field not found, use of non-static variable inside static method, type mismatch, using non-initialized variable, method call with wrong arguments, and method name not found [4].

A second phase of this study compared good debuggers to weak debuggers. Participants were asked to correct a faulty program containing compile time and logic errors within 2 hours. The errors included in the code corresponded to the errors students participating in phase one of this research most often committed. Their results showed statistically that a good programmer is not necessarily a good debugger but that a good debugger is usually a good programmer. The authors explain these results as follows: a good programmer who is not necessarily a good debugger is able to write relatively simple programs that compile and run correctly more often;; therefore it appears that they do not need to debug as often and avoid developing a skill they do not quite have. On the other hand, good debuggers also tend to be good programmers, because the knowledge of the system and the language that make them good debuggers enhances their expertise as programmers. The authors also believed that programming assignments are not being evaluated with the proper criteria, thereby allowing students with less topic comprehension to achieve higher marks [4].

Similar to Almazadeh et al.'s work, a tool called Retina [30] helps identify which students were having debugging difficulties through the use of passive observation. The idea behind providing such a tool is the fact that many students either do not recognize that they need help, or do recognize that they need help but are reluctant to ask for assistance [30]. This tool allows instructors to see how long students take to complete a given assignment, how often they have to recompile, and what types of errors were reported by the compiler. In addition, students are allowed to see a limited view of their classmate's activities on assignments in order to gain perspective on where they fit within the rest of the class.

The idea behind providing the latter kind of functionality is that a student who is discouraged by the amount of time they are expending on a given assignment may feel more confident in their own abilities when they realize that they are not the only ones experiencing a given issue on an assignment. A downside to this functionality that is not mentioned in the paper is that if the student is truly the only one struggling with a given topic then this sort of information on where their peers fall within a given assignment could be detrimental to the student's self confidence and therefore performance. The instructor could proactively remedy this however if said instructor is monitoring the class via the teacher console, determines that student X needs assistance, and intervenes [30].

The results from Retina were not so much related to improving scores as they were to determining correlations within the data. For instance, the authors of this paper noticed that when they viewed data for an entire semester, students who required less

time for assignments generally received higher scores than those students who required more time [30]. They also found that the number of compilation errors could be an indicator of whether or not a student was struggling. The last important correlation they found correlated the number of compilation errors to time of day that the project was being worked on. As anyone familiar with the habits of college students might suspect, the majority of errors recorded by the system occurred between 8 PM and 5 AM [30].

Retina helps support the idea of providing a tutoring system to help teach debugging by monitoring student activities, which a tutor would also be expected to do, only not strictly in a passive fashion but more to determine when to intervene and provide instructive feedback. Additionally, this work is able to help prove the validity of such data and exemplifies a framework for analyzing it [30].

Wang and Souders discuss an undergraduate research project that aims to help tutor debugging skills via a web-delivered quiz system. Their research discusses an undergraduate research program they ran centered on assisting students' debugging with their quiz system. Some of their more interesting results included characteristics required for debugging and certain defects undergraduates tend to include in their code. Specifically, knowledge of the domain, program, compiler messages and also meta-cognitive skills are very important for successful debugging. In terms of actual defects, the researchers found that it is more difficult for novices to find their bugs than it is for them to fix them. They also found that the following are the most common defects coded by freshmen and sophomores: omitting operations or components, adding unnecessary operations or components, and incorrectly implementing operations or

components. The authors also mention that they were unable to identify any dominant debugging strategies [41].

Chmiel and Loui explored how to teach debugging skills and how to measure the debugging competency of a student. Their approach consisted of four aspects: debugging exercises, debugging logs, development logs and reflective memos, and collaborative assignments [9]. Exercises required students to either identify the defect by performing code reviews without the aid of a computer, or fix the defect with or without (depending on student preference) the aid of a debugging tool called Turbo Debugger.

The first type of exercise bolstered student ability to understand what a given section of code would do while the second line gave the students extra experience with the problem solving required to debug their own code. The next aspect of their methodology was to introduce debugging logs. Log entries consisted of the name of subroutine the defect was found in, how long it took to remedy the defect, the incorrect behavior exhibited by the program, the incorrect code, and how the defect was fixed. This was modeled after another logging scheme cited in their paper with the addition of the solution to the defect [9]. As a side note it should be mentioned that the inclusion of the solution to the defect reinforces the idea of representing defects within a debugging tutor via case-based reasoning. The students in this study are basically creating their own personal case base for later review if such a defect should arise again.

Chmiel and Loui's approach also involves the use of development logs and reflective memos. The development log was used to document what occurred during

the completion of a project including design decisions, debugging experiences, their development plans, and the time expended on each of these aspects of the assignment. The reflective memo was based off of the development log and forced the student to reflect on the contents of the development log including what type of defects were encountered, how they discovered the defects logged, and when the defects were discovered. Finally, students were asked to answer the question of what would they do differently for subsequent assignments [9].

Students in this study were to participate in a collaborative final project in teams of four [9]. The idea of incorporating collaborative work is pedagogically sound as it is well documented that cooperation has the benefit of helping students attain grater productivity and higher achievement in addition to the other social and psychological benefits associated with working in a collaborative manner [40].

The results of this study showed that actively teaching debugging in this manner presented a statistically significant decrease in time spent debugging [9], which indicates that students who received the debugging instruction in the end spent less time struggling with debugging activities than their control group counterparts. The class this study was performed in was a third programming course for Electrical and Computer Engineering majors, the language presented to the students was Assembly—one of the most difficult languages to debug effectively [9].

Perhaps the most significant result of this work however is not the statistical proof that teaching debugging is relevant to producing more competent programmers, but instead the model that the authors developed in order to categorize a student's

debugging skills. They created a model based on the Dreyfus model of skill development [9], which consists of five stages: novice, advanced beginner, competent, proficient, and expert. Novice debuggers in this model are absolute beginners—they repeat the same types of defects frequently, debug haphazardly, expend considerable time on the debugging process, and may give up easily or depend on others for help. Advanced beginners differ from novices in that defect repetition is reduced and they are capable of recognizing defect symptoms more readily. Competent debuggers approach debugging systematically, alternate techniques, are capable for working mostly independently, and know several different techniques. Proficient debuggers build on these skills further, working on their skills in other areas in order to help improve their debugging ability and perhaps providing assistance to peers. For experts, debugging is second nature and their ability to identify defects far surpasses counterparts in previous categories.

Studying a novice compared to other novices is informative and helps to ascertain where students are going wrong, where students are going right, and what characterizes both groups. But what differentiates expert from novice when focusing on the debugging process? Gugerty and Olson performed experiments to determine these expert-novice differences in debugging by running two experiments that compared how the two groups approach the domain. Novices in this study were students who had recently completed a first or second Pascal course; experts were graduate students in the Computer Science department. For the first experiment each participant received training on the LOGO language and then were provided with 3 buggy programs on

floppy disk accompanied by hard copies of the programs' output. Each participant was given a 30-minute time frame for each program. Sessions were videotaped and subjects were asked to think aloud for later analysis. Three different types of bugs were given in the defective programs provided, including: incorrect graphic procedure parameter, missing graphics interface statement, and an error dealing with variable scope. In this experiment experts found the defect slightly more often than novices but tested fewer hypothesis—that is, although there was not a great difference in the ability to locate the defect in this study the experts were more efficient at identifying the correct cause for a given program defect. Novices were only able to immediately identify the correct cause of a given defect 21% of the time [15].

In the second experiment, similar subjects were selected. This time the programs in question were written in Pascal and the subjects were given one program to debug. Additional materials included a printed description of what the program was meant to do, what the correct output should look like, scratch paper, and a calculator. The results found from analyzing how the different groups attacked the problems were somewhat surprising. The authors expected that the novice programmer might jump right in to solving the problem, editing the program without trying to understand it first; however, this was not the case. The main differences between novice and expert groups in this study were that novices required more time to identify the correct cause for a defect and in general were less successful than the expert group in resolving the defects they were presented with. Their study also revealed that novices tended to add defects while trying to fix the original problem; experts participating in the study rarely did [15].

3 Chapter 3: Systems for Teaching Debugging, Defect Detection and Supporting the Novice Debugger

Analyzing and categorizing the student debugger helps to identify and establish what issues students may encounter while debugging, when they might have them, and suggests how to assist them when they have come to a debugging impasse. The next step requires providing the student with the tools they need to succeed. Intelligent Tutoring Systems and specialized IDEs for teaching programming necessarily include representation and specialized detection of defects in order to help remediate program flaws. Also, tools for detecting software defects at the source code level also need to represent defects within their systems. The rest of this section will discuss relevant defect detection systems, specialized IDEs, specialized debuggers, and intelligent tutoring systems.

3.1 Defect Detection Systems

Several software bug detection systems have been developed to remediate security issues in code at the source code level and at the compiled source level, ITS4 and FindBugs are two such tools. Each deals with a different language and therefore has different defect concentrations.

ITS4 is a static vulnerability scanner for C and C++ that checks for 2 main types of issues that are recognized via a defect database. The database, at the time of [35], contained 131 issues representing calls to unsafe C functions with the largest class of issues being file accesses [35]. For each of these unsafe calls their database contains a description of the problem, a description of how to fix it, an assessment of the severity

of the issue belonging to the set {NO_RISK, LOW_RISK, MODERATE_RISK, RISKY, VERY_RISKY, MOST_RISKY}, instructions on what type of analysis to perform when the function occurs in the token stream, and if the function involved can or cannot retrieve input from some external source like a socket or file [35].

As for classifying the issues represented in the database, the first type is related to the usage of unsafe C libraries and involves the sanity checking of string constants used as parameters to these unsafe libraries in order to prevent the different types of overflow attacks. The second type of defect targeted is race conditions [35]. The authors of the paper (and subsequently, the tool) refer to these issues as "Time-Of-Check, Time-Of-Use" problems, or TOCTOU. Race conditions identified in this tool are strictly related to file operations; other concurrency issues are not looked for. The developers of ITS4 classify these issues within a program through the usage of handlers for functions that are 'uses' and 'checks'. Each time a function that performs file operations is encountered the variable name in that function call that represents the filename is stored. Then a mapping is created that matches the variables in the stored list to their list of TOCTOU functions utilizing that same variable. Because of their methodology certain file related race conditions may go undetected. Also aliasing is unsupported. Only string constants are recognized by this tool as valid arguments for files;; therefore if someone were to use perhaps argv[x] as the filename related race conditions would not be caught. This particular example also poses the problem of unverified input. After the mapping is created and the scanning process is complete the list of file calls using each filename variable and if that variable has been noted to be used in at least one

check and one use this is combined into a single issue to be reported to the user with a higher severity level [35].

FindBugs is another static analysis tool that processes Java code and utilizes bug patterns [7]. It works at the byte code level after Java compilation; the software itself is written in Java. FindBugs has a catalog of about 300 different bug patterns. Each pattern belongs to one of the following categories: correctness, bad practice, internationalization, malicious code vulnerability, multithreaded correctness, performance, dodgy code, and security. Priorities in the range of high, medium, and low are assigned to each instance of a bug pattern and an associated priority is determined via heuristics unique to each instance. The tool is available in many different formats including through the command line, as a plugin for Eclipse or NetBeans, or through Ant or Maven. One interesting issue discussed in [7] was that of defects that were true defects but presented a low impact to the execution of the project. These included deliberate errors, masked errors, infeasible statement branches, or situations where the program is already doomed [7].

3.2 Whyline

The Whyline tool [18] tries to aid experienced developers in the debugging progress by enabling the programmer to perform a visual dialog with their program. The authors claim that developers have an internal dialogue about why a given program behaves the way it does. The Java centric version of Whyline will be discussed here, though it should be noted that CMU first created a version of Whyline for use within their educational Alice framework, demonstrating that the Whyline tool has applications

for both novice and experienced programmers.

Whyline enables developers to choose questions regarding why or why didn't a program perform in a certain manner and then receive an answer generated programmatically. Answers are generated via several forms of program analysis.

The architecture of the Whyline application consists of the following five parts, all implemented in Java: instrumentation framework, trace data structure, user interface, question extractor, and question answerer. The first of these modules, the instrumentation framework, contains an API for reading, representing, and writing java classfiles that is similar to other existing bytecode APIs. Support for reading, instrumenting, and analyzing classfiles is included in order to capture an execution trace of the application along with the ability to modify a given program so that a trace will be produced on subsequent executions.

Next, the trace data structure is responsible for encapsulating all aspects of the program for Whyline's use including the source code, class files and execution history [18]. This structure enables static access to both static and dynamic facts about a trace including potential callers of a method and all executions of a given method. Performance is taken into account here through the serialization and caching of the results of these queries so that future utilization of a given trace can reuse a particular analysis.

Following the trace data structure in the architecture is the user interface, which itself consists of five components, and enables the user to see different views of the trace data structure. The views are enabled by the different UI components including

output, search, file, call stack, and answer. Select output within the output UI causes the question extractor to come into play, which populates the answer UI. The question extractor queries the trace data structures and turns this data into views for the user. When the user selects one of the questions generated by the question generator the user interface generates the answer UI. This component is responsible for querying the question answerer in order to provide visualizations and answer related text.

The most crucial aspect of Whyline is probably its method of recording program execution. Without this component the system would be unable to provide the question and answer interface that is crucial to its functionality. The types of information recorded during an execution trace in Whyline include executed class files and associated source files [18]. When launched, Whyline first scans a user-defined directory for source code and makes a copy. Then the tool begins to perform bytecode instrumentation via the java.lang.instrument package available after Java version 1.5, intercepting byte arrays when each Java class is loaded. Instrumentation in Whyline involves both an analysis step and an instrumentation step. The analysis step is utilized for identification of data and control instructions that Whyline should instrument, including method invocations, catches, branches, and exceptions, all identified via simple parsing. Whyline specifically focuses on data instructions that affect either local variable or heap space within the application as opposed to also including all instructions that could affect the operand stack of the JVM. For example, consider the assignment x = a + b + c. Whyline will only instrument the value of the assignment and final addition, disregarding other aspects of the computation. The reasoning behind this

is performance, the values of a and b should not have changed after their prior assignments and these would have previously been instrumented by the tool. Another important aspect of this step is the determination of stack dependencies for each instruction within a method. The system explores all execution paths possible for a method, pairing push and pop instructions affecting the operand stack. During this exploration a simulated operand stack is maintained.

Method instrumentation is the next step of Whyline's analysis. Whyline visits each instruction and inserts calls to a global instrumentation method around the instrumented instruction. Each of these records information as an event, of which there are six types: assignments, invocations and returns, exception throws/catches, object instantiation, thread synchronization, and specialized native I/O events. Events are encoded as binary data with the following format. First, each has a header containing a 1-bit switch flag to determine if this event is occurring for the first time after switching threads. Following this is a 32-bit serial event ID that is set only if the switch flag is set. Next is a 1-bit io_callstack flag which is set to true if the code in question is representative of input or output or is necessary for call stack maintenance. This indicates to the trace loader which events it will need to process immediately. The last 32 bits are utilized for an instruction ID which is split into 2 sections. The first section is 14 bits long and represents a class ID. The last 18 bits are used to represent the instruction's index within the class file [18]. Objects are recorded differently. Each new object is assigned a unique 64-bit ID which is stored in a hash table and receives an entry into a file containing this 64-bit ID and the type represented as a class ID, thread

IDs are maintained similarly during runtime.

On each class load in the JVM Whyline intercepts the class load in order to perform several preprocessing steps before commencing instrumentation. These steps include creating a copy of the uninstrumented code, noting which classes don't have to be instrumented because the user specified to skip them, determining which classes are referenced by each loaded class (in aid of answering 'why not' questions), and caching the instrumented code for later use and subsequent performance gains [18].

Whyline's trace loading process is somewhat involved. Code is loaded first, both source and class files, and static information is extracted. Then three levels of associative activities are performed, associating invocations with methods and references with declarations. Lastly output instructions and thrown and caught exceptions are considered. All of this information is utilized by Whyline to produce a precise call graph. After generating this call graph thread traces are considered. Finally, the last responsibility of the loader is to generate an I/O history. This last step is extremely important because Whyline's question / answer framework depends on the ability to correlate output to program logic. An important aspect of Whyline to note here is that it has the ability to handle graphical output in this framework as well [18].

Question extraction is the next important aspect of Whyline to discuss. Broadly, the system supports questions of the form "Why did…" or "Why didn't…". Available questions are determined by a user selected time step and encompass either the data or caller affecting output. After the user selects an output primitive the system proceeds to generate possible questions related to each entity and objects that can indirectly

influence a selected output. "Why did…" questions can be of the form "why did x's attribute = y." "Why didn't…" questions can refer to why certain fields did not receive values, why weren't certain methods executed, and why certain objects were missing after the time the user selected [18].

Finally, after defining questions, comes a mechanism for answering them. Whyline's answering logic varies depending on whether the question is of the "Why did…" variety as opposed to "Why didn't…" Each "Why did…" question has a direct mapping to the program's execution history. Answers are therefore generated via dynamic slicing and presented to the user as a tree of relevant events sequenced chronologically. A program slice consists of "…the set of all statements that might affect the value of a variable occurrence [1]." Dynamic slicing refers to finding the set of statements that actually had this impact [1]. The difference between a static and dynamic slice is that a static slice is related to a given program location while a dynamic slice is defined according to the end of the execution history [1].

"Why didn't…" questions are more difficult because they represent why something did not happen as opposed to concrete references to specific code. Ko and Myers' methodology for these questions involve the analysis of one or multiple possibly unexecuted statements following the user selected I/O event. Instructions are analyzed to determine both why it did not execute and why the wrong value was used, involving both temporal scope and identity scope. Temporal scope refers to the timeframe following the user selected time point and the end of the program. Identity scope refers to objects that the user has indicated they are interested in exploring through their

question selection. For answering questions regarding why certain code did not execute Whyline first verifies that the instruction indeed did not execute and then, an algorithm the authors named *whynotreached* is run. This algorithm aggregates all the statements that have not executed and then runs a separate algorithm called *explain* to determine reasons for those statements remaining unexecuted. The answer boils down to either a list of other instructions that needed to be executed first or an execution trace providing the explanation as to why the instruction went unexecuted [18].

All of these mechanisms together combine allowing the programmer to turn an internal dialog on possible failures replete with code changes, hand coded instrumentation, and multiple compile and retest phases, into a single question regarding why something did or did not happen within their program. Evaluation of Whyline supports the benefits of such a tool. Developers were split into two groups and asked to determine why a graphical application failed to use the correct user specified color to draw a line. The control group that was not provided the Whyline tool took in the range of 3 to 38 minutes to determine why the wrong color appeared on the screen. The Whyline enabled group took from 1 to 12 minutes to determine the issue. If such a tool can produce this level of success with experienced Java developers, incorporating this methodology into a debugging tutor has the potential to assist in the creation of a new generation of computer science professionals who do not waste as much time spinning their wheels speculating on program failure. Providing a framework wherein the developer can actually query the program for why it failed could at the least help teach the programmer which issues manifest in which ways, making them better

problem solvers in general and more efficient software producers.

3.3 Toolkits

Within the realm of toolkits, Dereferee contributes to educating novices to better debug pointer issues. Specifically, this system provides an educational toolkit for producing better feedback on pointer problems within C++ code. Usage of the tool requires the inclusion of a header file, linking to the Dereferee library, and modification of pointer declarations to be of the form checked(T*)[3]. The tool was intended for student use in situations where pointer errors in their code would be masked by the execution environment or cause an unexplained runtime exception. Pointer usage is tracked via a memory allocation table that stores reference counts for each allocated section of memory. Other details stored by this system are the types allocated objects, if each monitored block refers to a single object or an array, a stack trace detailing the context and location of each allocation, and the length of dynamically allocated arrays. The system views each pointer as a state machine with transitions between live, null, dead and out-of-scope. These states are reached through the operations performed on a given pointer during program execution. Unfortunately the paper does not discuss testing involving student usage. Instead, the authors of the paper displayed Dereferee's performance in identifying pointer defects by instrumenting a set of code containing both student code and the author's code, then tabulating the results of running the programs with Dereferee versus running the programs without using Dereferee. Additionally their testing utilized an external tool called CxxTest (similar to JUnit) in order to uniformly collect data on their test results and provide crash recovery. Their

results showed that, had the students been provided the toolkit, 95.6% of the students whose code was involved would have benefitted from using the toolkit [3].

3.4 Specialized IDEs and Libraries

Thetis is a specialized IDE (Integrated Development Environment[1]) to provide the novice with better feedback on C program defects. The system was designed by students at Stanford to address novice issues encountered when using C as the programming language for a CS1 / CS2 progression [13]. The main concern of the faculty at Stanford when making this decision was that students would end up spending more time on debugging their assignments and less time on actually learning computer science, which was in the end the actual result. The main cause of this time difference was determined to be a function of the programming environment instead of the language itself. Thetis was developed to remedy this problem and dispel the shortcomings of commercial C compilers in the educational setting. Some properties that commercial C compilers lack included uninformative and misleading error messages regarding certain syntactical errors and little to no runtime checking. Also interactive debuggers for C, such as the GNU Project Debugger (GDB), usually require students to have an understanding of advanced concepts before they are prepared to learn the relevant material needed to use the debugger.

Thetis is able to provide better feedback to novices because it is a C interpreter instead of a C compiler. A compiler generates machine code, thus separating the

[1] Software that assists developers in writing and testing code

compilation and execution phases. An interpreter on the other hand combines both of these into a single phase, simulating program execution [13]. The drawback of interpreting C as opposed to compiling it is the introduction of interpreter related latency, but this is not of the utmost importance in introductory CS classes.

The system provides more comprehensive error messages, runtime error detection, debugging and visualization tools, and what the authors call a listener [13]. The debugging / visualization tools and listener are very similar to what is currently included in software suites such as Microsoft's Visual Studio, allowing the programmer to easily view what is happening at runtime in their program by allowing for graphically placed break points, the ability to evaluate expressions while the program runs (via the listener), and the ability to view the call stack, function variables, arrays, and structures. As for runtime error detection, Thetis was configured to check for certain defects including division by 0, use of uninitialized variables, dereferencing invalid pointers, out of bounds array accesses, assignment of an out of range value to an enumerated type, exiting a non-void function without returning a value, accessing a bad function pointer, and passing invalid arguments to functions.

Thetis was evaluated at Stanford University within an introductory level computer science course. 30 students from this course were involved, the results of their usage of the system allowed for further refinement. Results from this first study were very encouraging, causing the authors to expand and test the system with all the students enrolled in the course (about 300) the next time it ran. Despite reporting positive testing results, they do not list explicit details of these results. The authors do however give

minimal results from a smaller set of 39 students who used the system during a summer iteration of the same course. These students were asked to complete a 4 question survey, asking the students how easy they found the system to use, if they were ever confused by the system, if they felt that the system helped them understand programming, and if the error messages reported by the system helped them find mistakes in their programs. The numbers for these results were very encouraging and bolster the previous statements made about positive research results. 82% of the students felt the error messages Thetis reported helped them find their mistakes more easily and 71.8% of the students reported that the system helped them understand programming [13].

Two now well known novice-oriented IDEs for the novice Java programmer are BlueJ and DrJava. While the user experience is vastly different between the two systems, certain design concerns are taken into account by both systems. These concerns include simplicity of use, ease of understanding, and facilitating education in Java programming.

BlueJ, the first chronologically of these two systems, began as thesis work by K**ö**lling in the late 1990's as the Blue environment ([21]) and since evolved into BlueJ. This system favors an Objects-first pedagogical approach, where novices are introduced to the concepts of object-orientation immediately in their first CS courses. The student is presented first with an interface for creating a UML diagram that allows for "uses" and "inherits" edges. Before any code can be written the student must articulate their design in this interface. The student is able to code their class by "opening" it. When

the student creates a new class in the main interface the Blue system creates a code skeleton for it that is viewable by double clicking the box representing the class in the main window. This brings up an editor pre-populated with certain details as defined by the design the student created. Changes in the editor window that affect the design of the application being developed are echoed in the design view, keeping the application consistent automatically across views. The student can compile their application via a compile button visible in the top menu. Errors reported back to the student from Blue are presented in more comprehensible language than is standard for a compiler [20,21]. BlueJ has been successfully used in many introductory courses and it is still being expanded today.

DrJava, developed near the same time as BlueJ, strives for interface simplicity and is the second "Dr" system developed at Rice (the first was DrScheme). An IDE such as Eclipse, CodeWarrior,or Visual Studio bogs the user down in many menus overflowing with fancy options that, to the novice, are intimidating and overwhelming. DrJava, on the other hand, presents a much simpler interface, and enables the student to achieve the write-compile-test loop of program development from a simple run-time command-line, and by providing a limited set of menu options that focus the student more clearly on those tasks.

Instead of the many panels and tabs, the student sees just three panels—an editor, a file listing, and a bottom pane. This bottom pane has three tabs: interactions, console, and compiler output. The console pane allows students to provide console input and view console output. The compiler output pane shows student the results of any

compilation attempts. The interactions pane enables students to run individual Java statements interactively in a "Read-Eval-Print" loop as one might see in a scripting environment typically found in support of LISP, Perl, Python or Mathematica. This methodology enables simpler debugging via immediate evaluation of programming statements. Additionally, DrJava makes it easier for the student to locate and understand their error messages. When a syntax error exists in the student's code after a compilation attempt, DrJava displays the error message with its associated line number in the interactions pane at the bottom of the screen. When the student clicks on the error message, the line number affected by the error message is highlighted in bright yellow while the error message gets highlighted in a light green color. This draws the student's eye to the error at hand and helps them to focus their attention on a given issue [2].

3.5 InSTEP

InSTEP (Independent Student Tutoring by Example Programs) was developed in 2001 to tutor C programming on the Web [31]. This system presents feedback in context of a current problem and limits the domain by using a fill in the blank approach, where students are presented with partial programs and then must complete the example by filling in the blank. The workflow of this system progresses in the following manner. First, the student attempts to provide the correct snippet of code. After providing their answer, they submit the form and InSTEP compiles the submission. If any compiler errors are present the student receives a report containing those errors and the current source code.

If the submission compiles, InSTEP runs the code through a series of predefined test cases. If the submission passes these test cases, the student receives feedback letting them know they have successfully completed the exercise. Otherwise, InSTEP analyzes the code and the output it produces for a known common set of errors. If InSTEP is able to recognize the error then it provides appropriate remediations to the student. If not, it attempts to at least locate where the issue lies. If InSTEP is able to localize the error then it provides an appropriate hint back to the student along with the output the faulty code was able to produce and a description of InSTEP's analysis results [31].

Evaluation of this system was conducted at the University of Utah within an entry level programming course for engineering majors. Of 120 students in the class, 66 opted to participate. All students at the time of participation had received some experience within this class in Maple and in C. Participants were divided into 3 groups: group 1 was given a standard environment for C programming on the Linux command line including gcc and emacs; group 2 was provided InSTEP but without the analysis results previously described; and group 3 received the full InSTEP system. All students were presented with the same templates in appropriate formats for their test group and group 1 received a tool that was able to verify their solution provided the correct output [31].

The exercises presented to the students consisted of 6 standard programming problems involving 2 problems dealing with loops and iteration, 3 problems dealing with summation, and 1 problem involving computing the maximum sum of a series. Students were asked to take both a pretest before and a posttest after the study

concluded and lectures were held as normal for the 1 week duration of the study, which occurred at the end of the semester. The pre and posttest instruments were evaluated for the level of understanding displayed as opposed to calculating the number of errors committed [31].

Other measures taken included the amount of time students spent on each exercise, the amount of time TAs and other course staff spent during the evaluation period assisting each student, homework grades, midterm grades, and exam grades. Of all these measures, only one showed statistical significance—the amount of time course staff spent helping students. This result indicates that InSTEP was quantitatively helpful to the students who received the full system, as a significant portion of the students were confident enough in their answers that they did not need to seek outside assistance [31].

3.6 FLINT

Ziegler and Crews from Western Kentucky University recognized that the IDEs in existence for programming C++ were novice un-friendly. Because most development environments are built as tools for the expert developer, the feedback from these systems in the presence of faulty code is a poor match for the novice.

Ziegler and Crews built a specialized IDE focused on helping students to debug programs called FLINT (FLowchart INTerpreter). This somewhat rigid system focuses the student on designing their procedural code before performing any implementation. First, the student is required to articulate their design in the form of a hierarchy of actions the code must perform. Then the student is required to compose a flowchart.

The creation of this flowchart and any subsequent algorithm implementation is restricted by their initial design; changes in the system the student is trying to build must first occur in the design itself [44].

Algorithms in the system are represented using flowcharts;; the authors explain that this decision is in part due to existing research on expert programmers indicating greater speed and comprehension when reading flowcharts as opposed to pseudo-code. When constructing the flowchart only complete programming constructs may be added, removed, or moved. Flowchart items are manipulated from a "point and click" interface, making this a system that supports the drag and drop programming methodology [44].

When the student needs to perform debugging, FLINT facilitates the activity by allowing stepwise execution and breakpoints. While performing stepwise execution the system highlights the current lines and affected variables in order to provide the student with a visual cues, highlighting how code runs sequentially and helping the student understand what actually occurs in the underlying machine [44].

3.7 Backstop

Some researchers have recognized a need to tailor the debugging process so that the novice is able to debug their programs on their own and understand their results. One system built explicitly to help the novice Java programmer debug is Backstop, built by Murphy et al. [29]. They take a much narrower approach than other systems and methods. Instead of concentrating on debugging or programming in the general sense, they elect to concentrate solely on runtime errors and helping students resolve them.

Specifically, the authors aim to provide detailed and helpful error messages for uncaught runtime exceptions as opposed to the standard messages returned by the Java runtime system. Their tool also produces step-by-step explanations of running code in order to help the student understand what the code did at specific points during execution. This tool, called Backstop, is essentially a novice-centered debugger. Previous research cited by the authors highlights the fact that there have been several efforts to remediate novice issues with compiler messages (including BlueJ, ProfessorJ, DrJava, Gauntlet, Expresso, and HiC). Additionally the authors mention two systems for specifically helping novices remediate logical defects in their code, InSTEP (discussed earlier) and DEBUG. Another tool similar to Backstop and also listed in this paper is also a novice-centric debugger but targets C++ programs and is called CMeRun [29].

Backstop is able to support the student through finding and fixing their runtime exceptions by catching uncaught exceptions and then helping them walk through their code as with a normal debugger but with language and details more suited to the novice. Unfortunately, their experimental design makes it unclear as to whether the tool actually helped the students debug runtime issues. Their control group was specifically selected from their top students and all students in this group were able to solve the runtime error without assistance. Additionally, most of the students in the test group (76%) were able to solve the runtime error with Backstop's assistance within a 15-minute time frame.

The main benefit of this tool is the fact that it turns the jargonized output of what is usually an expert-oriented process into a friendlier, more naturally worded

representation that highlights the sequential nature of the underlying machine. In addition to helping the student with traditional debugger output, the tool is also capable of assisting the student in understanding runtime errors. Some of the features of Backstop include: displaying runtime exception details for uncaught runtime exceptions in an easier to understand wording, displaying which line of code is currently executing (which eventually could become a program trace), and displaying the values for variables affected by the line currently being displayed [29].

Although most of this functionality is available in standard debugging environments in one form or another, there are two interesting details that set Backstop apart in addition to its more user-friendly messages. While most debuggers can help the programmer delineate a trace of some form, this system will actually print out line-by-line what is running, if so directed by the user. This functionality makes it much easier to follow trace data than traditional environments. Additionally, displaying the current values for variables at a given time step (what many debugging environments call a "watch" for a variable) is handled by default and with much greater ease than environments like expert-oriented environments [29].

Before the student can use Backstop on their code, a preprocessor must ensure that their code follows certain conventions required by the system. Some of these conventions include that each line may only contain one Java statement, braces must appear alone on their own line, multiple assignments should not be contained within the same line of code, and case statements must also be presented on their own line. The preprocessor has the ability to force the student's code into most of these conventions if

the student did not originally follow them. Two conventions the tool was incapable of enforcing at the time of writing however were the need for the student's code not to have one statement spanning multiple lines and that each block of code be enclosed between curly braces. Once the student's code is in the appropriate format, Backstop can start running on the student's code [29].

Backstop was evaluated in two scenarios, one in which the student was instructed to use the tool to help them fix a runtime error and one in which they were instructed to use the system to help them fix a logical error. Access to features in Backstop were isolated in order to test different features separately. A group of 17 students of evenly distributed gender were selected for the study, all 17 had recently completed Columbia's CS1 course. Each student evaluated the system individually and was interviewed at the end of the experience by the author [29].

The evaluation of Backstop's effectiveness on assisting the student with runtime errors went as follows. The student was presented with a program that contained an unchecked runtime exception, a description of the algorithm needed for the problem and a demonstration of what situations forced the error to appear. Students were asked to interpret the error message, find the defect and fix it [29].

Results from the study show that the participating students viewed Backstop somewhat favorably. In scenario 1, debugging a runtime error, 76% of the students using Backstop were able to find and fix the error within 8 minutes. All control group students (4 of the top students in the class) were able to solve the problem within 5 minutes. In scenario 2, where students were asked to debug a logic issues, results were

similar except the time required to fix the error was generally much shorter. Although their choice of control group makes it impossible to evaluate if the system truly helped, the students' perception and reception of the tool is encouraging [29].

3.8 Intelligent Tutoring Systems for Programming

Several Intelligent Tutoring Systems have been built in order to teach programming. Explicit discussion of how the tutors represent defects was not encountered in the relevant ITS papers explored during this study but each of the tutoring systems researched had to deal with incorrect student input in one way or another. Tutoring systems of specific relevance to this study are ELM-PE, ELM-ART, CBRMETAL, the Problets tutor and its precursor, and the Object Oriented Design Tutor developed by Moritz, Parvez, and Wang at Lehigh University. The following consists of an overview of Intelligent Tutoring Systems in the general sense and summaries of Anderson et al.'s LISP tutor, ELM-PE, ELM-ART, and CBRMETAL.

3.8.1 Anderson Lisp Tutor

Anderson et al.'s LISP tutor was developed in the early 1980s and follows the ACT* theory of skill acquisition. This theory holds that "…knowledge becomes proceduralized with initial usage…" and that subsequent use of that knowledge further strengthens it [5]. ACT* later became the ACT-R theory of human cognition which assumes there are two types of fundamentally separate knowledge. These types are declarative and procedural. Declarative knowledge takes the form of a fact or an experience. An example, as given in [6], would be the following: *when you divide both sides of an equation by the same value, both sides of that equation are equal*. People

originally gain skills initially by learning declarative facts. Through the application of those facts, people gain procedural knowledge. This theory assumes that these skills are very orderly and able to be represented by production rules associating problem states and goals with actions and state changes in the environment. Adherence to these theories enabled Anderson et al. to construct a model tracing tutor for teaching LISP programming called the APT LISP Tutor, or simply the Lisp Tutor in other publications. This tutoring system was evaluated and successfully used at Carnegie Mellon University to assist students taking a LISP course [5]. However, like other programming language tutors, this system appears to contribute little to helping students develop debugging skills.

3.8.2 ELM-PE

ELM-PE (Episodic Learner Model Programming Environment) uses Case-Based Reasoning (CBR) [36] and is a precursor to both ELM-ART and CBRMETAL. The tutor has three modes—listener, editor and exercise; the different modes exhibit different levels of assistance. ELM-PE starts in listener mode, which provides the least amount of support. Basically, listener mode is similar to utilizing a regular text editor with subtle hints appearing at the bottom of the screen. Editor mode provides more support features including examples, visualizations of expression evaluations, displaying where in the code errors are detected, and a more intelligent structure editor. The last mode, exercise mode, provides the highest level of support. An interesting feature existing in this mode is automatic cognitive diagnosis, which is where case-based reasoning is utilized to explain how a student arrived at their current exercise

answer attempt. This system builds a derivation tree representing all concepts and rules that the system identified as explanation for the student's current solution [36].

3.8.3 ELM-ART

Also teaching LISP, ELM-ART is served over the Internet and is described as an "on-line intelligent textbook with an integrated problem solving environment [8]." Every problem in this book is capable of being supported in the same way ELM-PE originally did. In order to help keep students from getting lost in the course material, ELM-ART adapts link annotation and sorting. The authors use a traffic light based annotation, placing green circles next to items that it believes the student is ready to learn and the system recommends, red circles mean the student has not yet read the material, and yellow means that the student may be ready but the system does not recommend the student pursue the material yet. When the student is solving exercises ELM-ART is capable of predicting how the student will solve different problems and will use this information to select the next problem to be served. ELM-ART is a successor of ELM-PE and therefore is also capable of the same diagnosis of code that was described in ELM-PE [8].

3.8.4 CBRMETAL

ILMDA (Intelligent Learning Material Delivery Agent) with CBRMETAL (Case Based Reasoning and Meta Learning) incorporates this approach, utilizing machine learning to accomplish meta-learning in order to improve the Pedagogical Module of that system over time [32]. Specifically, metal-learning is "…a learning mechanism of a system that learns about the system itself and how to improve the

systems performance over time." [34] The different pedagogical strategies utilized by the system are themselves cases delineating which tutoring action to perform in a given situation. All of this effort goes into producing software applicable to only, at best, a single subject matter. There are not many intelligent tutoring systems utilizing machine learning and fewer that utilize it to improve the Pedagogical Module, however there is one other tool that the authors mention called CBIP.

CBIP (Case-Based Instructional Planner) is a tool that has been integrated in existing ITSs for enabling the Pedagogical Module to learn and thus become a self-improving system that can learn from what it experiences. The planning aspect of this system is utilized to map sequences of instructional goals and actions in order to provide coherence, continuity, and consistency. Cases in this system differ from cases in CBRMETAL in that here a case is used to define part of a previously used plan. Encapsulated by the case are the context, plan or sub plan if plans are layered, and related results [34].

The authors of CBRMETAL also cite CAPIT, an ITS utilizing Bayesian networks and decision theory for modeling the student and selecting subsequent tutorial actions. The authors of CBRMETAL claim that Mayo and Mitrovic categorize, generally, student models into three categories for all ITS systems: expert-centric, data-centric, and efficiency-centric [34]. On examination of Mayo and Mitrovic, it has become apparent that the authors were referring solely to those ITSs that utilize Bayesian networks in their student models [27]. Therefore, within the purview of ITSs utilizing Bayesian Networks in this manner, the three categories consist of the following.

Expert-centric Bayesian Network student models are those in which an expert is responsible for directly or indirectly creating the structure and conditional probabilities of that model. Efficiency-centric Bayesian Network models are partially specified or restricted and filled in with domain knowledge forced to fit the model. The mentioned restrictions are chosen in order to help maximize efficiency, thus efficiency-centric. Data-centric student Bayesian Network student models were introduced in [27]. In this type of model the system does not try to model unobserved student states, instead attempting to predict student performance by observing certain variables and trying to model relationships between those variables. Because ILMDA does not utilize a Bayesian Network for modeling students, the subsequent claim that ILMDA follows the data-centric model is not completely valid.

Modules within the CBRMETAL framework include a case learning module, similarity heuristics, and adaptation heuristics [34]. The case learning module is employed for learning new cases and is responsible for deciding newness of a potential case when compared to the rest of the existing case base.

The similarity heuristics adjuster module is utilized for performing reinforcement learning and similarity computations. These calculations are utilized to determine relative weights for the attributes within a case. Reinforcement learning within the system is based on the assumption that for the most similar case retrieved from the casebase, $C_{best,}$ there exists a solution description pc_{best} and solution sc_{best}. If sc_{best} was previously observed as successful and is adapted to a new situation p_{new} similar to pc_{best}, then sc_{best} should also be successful. If the described situation has not observed then it

is possible that p_{new} is not actually similar to previously postulated pc_{best}. Whether or not the contributing heuristics lead to successful identification of a similarity or not leads to related penalties and rewards for those heuristics involved. If p_{new} was erroneously determined to be similar to pc_{best} then heuristics contributing positively to the proposed correlation are penalized and heuristics contributing negatively to the proposed correlation are rewarded [34].

Finally, the adaptation heuristics adjuster module is responsible for helping the system learn to adapt to new situations. This is accomplished though applying reinforcement learning to the adaptation heuristics utilized by the system. Different heuristics contribute to creating the best case solution, this module records those heuristics that contribute changes to the best case solution sc_{best}. If sc_{best} is observed to be successful when applied then the related adaptation heuristics are rewarded, if it is unsuccessful the converse is true. The reward or penalty is prorated based on the impact of each individual adaptation heuristic involved [34].

The authors discuss several advantages and disadvantages related to their approach when compared to traditional CBR approaches. All three modules of this system are adjusted at once which helps avoid biased learning. However, this also presents a disadvantage. By having so many variables changing the system may never actually converge. One aspect's improving could negate improvements to other parts of the system. In order to minimize this instability the authors have defined six principles, which are explored in turn below [34].

Their first principle relates to learning new cases that are based on diversity. The

objective of such a case is to expand situation coverage of a given case but not to affect the solution coverage of the case base. Therefore the adaptation heuristic should have the responsibility of generating new solutions making it less likely for learning a new case to impact solution coverage significantly [34].

Second, their system contains the ability to learn new cases with failed solutions, allowing the system to learn similarly to a human. For instance, a person tends to learn more from their failures than their successes. Given this, an intelligent system could also utilize failed solutions that have been adapted incrementally to eventually obtain a successful solution. Their system uses what they call failure-driven adaptation heuristics, which adjust solution parameters opposite to the way that different-drive adaptation heuristics might [34].

Their third principle involves tagging cases with a utility vector recording the successfulness of the case, how many new cases have resulted from this case's retrieval, and the rate of retrieval. Cases with a larger history of success are given more weight when calculating similarity heuristics. Cases that have caused the creation of more new cases also are given more bearing in the determination of adaptation heuristics [34].

Fourth, the authors utilize parameters for dictating aggressiveness for each learning module – t for the similarity heuristics learning module and h adaptation heuristics learning module. The CBR aspect of the system can then be modified according to the confidence of the developer. If the heuristics for calculating similarity are considered by the developer to be correct then they can set $t<h$. Their fifth principle is that their system will learn conservatively by modifying only those heuristics that are

the most influential in each learning episode, rewarding or penalizing only the highest contributor to a given success or failure. Finally, their last principle involves staggering learning activities based on the frequency of learning for cases or heuristics in order to help prevent the learning oscillation discussed earlier. Within the system actually implemented in this paper, principles 1 and 2 were incorporated into their case learning module, principle 6 is utilized over the entire system with regard to learning management, and principles 3 through 5 were incorporated into the two heuristic learning modules [34].

After discussing these principles, the authors discuss the actual tutor in this system, ILMDA. This system serves a topic to the student as an instructional content set including a tutorial, related examples, and exercise problems for assessment of student understanding. ILMDA uses the assessment and profile of a student to select appropriate examples and exercises to serve to the student and makes this decision through the use of case based reasoning. The case structure is specified by ILMDA. An individual case is comprised of a situation, a solution, an outcome, and a performance metric. Each of these in turn has sub properties. The situation aspect of a case refers to the student's static and dynamic profiles and instructional content characteristics. The solution aspect comprises characteristics of the next appropriate example or exercise. Outcome refers to the usage history of the case in question and Performance exhibits the difference between the behavior that was expected and the behavior that was observed [34].

Within this system and structure, the static and dynamic profiles provide the

foundation for student modeling. The dynamic profile is utilized to capture observed real-time student behaviors and is populated with how many times a student attempted to answer the same exercise, how many different modules have been provided to the student so far, an average of mouse clicks observed throughout the student's use of the tutorial, the average mouse clicks when the student is viewing examples, number of times a student quits after a tutorial, number of successes, and the average time the student spent on the tutorial [34].

The solution parameters are utilized to specify certain characteristics of example and exercise problems including length, Bloom's taxonomy, interest, scaffolding (level of learning support) amount, times viewed, average time spent on the exercise per use, level of difficulty, and average clicks per use. ILMDA's scaffolding consists of cues, references, elaborations and hints. Outcome parameters are calculated over the usage history of a given case, including: how many times the case has been employed successfully, if the student quit the problem, and if the student provided a correct answer. This data is accumulated from observations each time a given case is used. The performance parameter aspect of these cases within ILMDA is utilized to document differences between expectations and observations with regard to behavior and is accumulated from real time observations [34].

CBRMETAL utilizes meta-learning in order to adjust both similarity and adaptation heuristics and to help fill in the case base. The CBR module performs this type of learning after a student has completed a session in ILMDA. At the end of a session ILMDA sends the CBR part of the system relevant results. The similarity and

adaptation heuristics adjustor modules are run offline after the completion of multiple sessions, allowing for more accurate statistics to be presented on the cases. The case learning module runs online with the rest of the system. ILMDA's decisions about whether or not a case will be stored are made directly after a session finishes, allowing for immediate use of new cases [34].

The final architecture of this system consisted of 8 modules: GUI frontend, CBRMETAL Reasoning, Casebase, Student Data, Session Data, Content Set, Online Report Review and Analysis, and Online Authoring Upload. This system was tested at the University of Nebraska in CS1 courses with 5 content sets: File I/O, Event-Driven Programming, Exceptions, Inheritance and Polymorphism, and Recursion. Each problem in the content set received a Bloom's taxonomy level in the range of 1 to 6 representing the sequence {knowledge, comprehension, application, analysis, evaluation, synthesis} and a difficulty level between 1 and 10 inclusive. Bloom's taxonomy defines a "…framework for categorizing learning goals" [http://cft.vanderbilt.edu/teaching-guides/pedagogical/blooms-taxonomy/] that was first created in 1956 and has since been greatly utilized in education and revised. The case base was pre-populated with cases containing estimated values for each parameter. Also, the adaptation heuristics and similarity heuristics were initialized. All similarity heuristics initially received a weight of 1.0, in essence forcing all situation parameters to have the same importance when determining similarity calculations. Adaptation heuristic initialization was a little more complex because not all of these heuristics were continuous (i.e. scaffolding, difficulty level) [34].

The system was utilized over 2 semesters within the University of Nebraska's Department of Computer Science and Engineering in CSCE155, which is their version of CS1, and is comprised of about 150 students each year from CS, different Engineering disciplines, and Math. In the Fall 2004 semester, two versions of ILMDA, learning and non-learning, were employed. In the learning version ILMDA utilized the full architecture. The non-learning version was ILMDA with learning turned off—the heuristics and case base were utilized statically over the initialized values given to the system by the developers. ILMDA alternated automatically per student, sometimes presenting the learning version and sometimes presenting the non-learning version. Results were discussed in regard to teaching Recursion were promising. On average, the situations posed to the learning version required less examples and exercises than the non-learning version and students scored about 6% better within the learning system than within the non-learning system. On the other hand, student utilizing the learning version spent more time than their non-learning user counterparts. The authors suggest this was due to better problem selection that in turn caused students to be more invested and engaged [34]. When the authors expanded to consider all content sets they found that learning ILMDA was consistent in utilizing fewer problems to bring students to the same level of comprehension as the non-learning system. The Recursion aspect itself was found to consistently require the use of more questions, however, and the authors suspected that this requirement arose from ILMDA serving more difficult than easy questions within this topic. For other topics learning ILMDA served easy questions more frequently [34].

The second round of testing occurred in the Spring of 2005 within the same class. This time a third version of ILMDA was introduced. This static version differed from its learning and non-learning counterparts in that the static system did not utilize CBR for the selection of a tutoring strategy. Instead, the static system consistently used the same teaching strategy case, learned nothing, and would select the next easiest problem to serve to the student when a wrong answer is input. Over this trial the static version of ILMDA outperformed non-learning ILMDA, suggesting that the initial case base was less effective than a single heuristic and that the non-learning ILMDA adapted poorly to different situations. However, learning ILMDA was able to outperform both counterparts through adaptation heuristic adjustments and learning new cases [34].

Thus the results from CBRMEMTAL are encouraging. It supports the idea that an ITS can successfully learn how to better teach a given subject area through observation and analysis of interactions with a student user. It is also encouraging for the utilization of case-based reasoning within a debugging tutor. CBRMETAL is more interesting than other CBR based ITS systems however because of the meta-learning component that enables it to, in essence, reason about its own teaching strategies.

3.9 **Intelligent Tutoring Systems for Debugging**

In addition to the systems discussed above, there have been three documented systems that aimed to become Intelligent Tutoring Systems for teaching debugging and a paper that discusses some of the issues and requirements an Intelligent Tutoring System for Debugging would face in an Object Oriented environment. The following discusses the 3 most relevant of these systems: PROUST, DebugIt, and the Problets

tutor.

3.9.1 PROUST

PROUST was built in the early 1980's by Johnson and Soloway [17]. It utilized intention based analysis in order to understand the programmer's intended outcome for Pascal programs. The system aims to understand the novice's buggy programs through the analysis of the program's source code and a non-algorithmic description of the intended outcome of the program. The authors note that in traditional analysis methods there is a lack of correlation between program text and programmer intention. To solve this gap the author built the PROUST system's domain and Pedagogical Modules, intending to utilize this system as part of an Intelligent Tutoring System.

This analysis is performed through the use of programming plans, where a plan is defined as "…a procedure or strategy for realizing intentions in code where the key elements have been abstracted and represented explicitly [17].. This work builds on results in earlier work analyzing how expert programmers think when reading and writing programs. Although the expert would create valid plans that once executed would result in a largely correct program, the novice will utilize plans that the expert would not consider and may be faulty. This difference in behavior is due to the novice lacking the background to determine proper courses of action consistently. In order to accommodate the novice, PROUST includes not only plans an expert would formulate while working on a programming problem but also the somewhat questionable plans a novice might create. The system also considers the order in which identified plans are implemented.

PROUST produces a goal decomposition that is then utilized to determine plans the programmer may have applied. Goal decomposition in PROUST includes: hierarchy or subtasks identified, relationships and interactions for subtasks,, and a mapping from goals identified to plans used to implement them. The system develops transformations and interpretations; interpretations constitute a search space the author refers to as the interpretation space. In the event of multiple valid interpretations existing for the same program, PROUST uses a series of heuristics that the author does not go into further save to mention that when bugs are present they are ranked in severity and then utilized later to predict which plan is the most likely of the set of matching plans [17].

The system was analyzed over a series of 206 programs. Within this sample PROUST was able to generate a complete analysis for 161 of them. When PROUST completes an analysis it also calculates a confidence level. In some situations, PROUST is only able to complete a partial analysis. During evaluation this occurred 17% of the time during trials. The paper does not go in to detail about which situations make only a partial analysis possible but it is assumed that this occurs in situations where PROUST does not have a plan to match the goals determined during the intention based analysis. In situations where PROUST is able to complete a partial analysis, feedback given to the student deals only with those parts of the program that PROUST was able to analyze. There are certain situations where PROUST is able to complete its analysis but generates false positive bug reports back to the student. In testing this only occurred about 5% of the time. The creators of PROUST originally intended the system

to become a full ITS; however the system remained a sophisticated analysis tool for assisting in the debugging process without further development into a full featured ITS [17].

3.9.2 DebugIt

A more recent system working towards becoming an ITS for debugging is called DebugIT. Greg Lee and Jackie Wu developed this system at the Taiwan Normal University in Taipei in 1999. The authors approach this system from an exercise-oriented standpoint. Students are presented with an interface that allows them to debug short buggy Pascal programs. The student attempts to solve the problem using a preset selection of actions and then submits it, receiving feedback as they proceed. The system then compares their solution to solutions held in its database; comparison is done in terms of correction steps taken by the student. A total of 3 attempts are allowed before the system presents the student with the answer, along with an explanation. Assistance is available as hints on an on demand basis via a button in the user interface [24].

The system that was implemented and evaluated consisted of an exercise-based system focusing on loops with a library of 20 exercises. The system was presented to two groups of introductory computer science students: a group of college freshman and a group of High School sophomores. All participants were enrolled in Pascal courses. The experiment consisted of both test and control groups. Both groups were presented the same set of 20 buggy Pascal programs and instructed to try to fix them within 150 minutes. Control group students were given a standard Pascal programming

environment, test group students were given access to the DebugIT system alone. After the 150 minutes were up students were given 50 minutes to complete a posttest for comparison. Results showed that college students using the DebugIT system achieved a statistically significant higher score on the posttest than their control group counterparts. Tests at the High School sophomore level did not yield the same results. The authors attribute this to the students having had much less previous experience with Pascal. Additionally, students were asked to complete a qualitative questionnaire about their experience with the DebugIT system. Questions included "Program debugging is easy", "DebugIt is a good tool for debugging practice", and others related to the student's satisfaction with different aspects of the experiment and their views on debugging in general. Results from this questionnaire were generally positive, but the High School students in general gave less positive feedback. This is somewhat expected as the High School students had less previous programming experience than the college students which might have caused them to have less success than the college students and therefore view the testing session less positively [24].

3.9.3 Problets and its Precursor

The aim of Problets is to build an ITS that could help students better analyze programs, which is a crucial ability for successful debugging episodes. Specifically, Problets focus on pointer issues in C++ with a concentration on step-wise evaluation, output prediction, and finally debugging. Specific defects that this system aims to help remediate include semantic and runtime difficulties.

An additional interesting aspect of this work and other publications by Kumar is

the idea of automatic problem creation. Other researchers in the field of Intelligent Tutoring Systems recognize that finite exercise sets are a limitation to the effectiveness of the system—students may run out of exercises, or end up viewing an exercise multiple times [22,]. Kumar's approach uses templates in a Backus-Naur Form like formalism where non-terminals represent data types, identifiers and literal constants. Exercises are created dynamically by randomly filling the non-terminals in with appropriate values. This approach has the added benefit of being syntactically independent—if two languages share a similar semantic structure the same template may translate to another language. Domain modeling in this system uses the model based approach, the model consisting of state diagrams to model different aspects of the domain [23].

A module called the Problem Sequencer selects which problem to present to the student. This module analyzes the data stored in the Student model and determines from that data which exercise is appropriate to provide next. The student model contains information about how the student has performed on previous exercises; this data comes from the grader module of the system [23].

Feedback to the student is provided on an on-demand basis consisting of an explanation of how the current problem's code runs. The explanation is created via a two-stage algorithm involving simulation and reflection, and the explanation itself has three forms: simulative, diagnostic, and customized. Simulative feedback consists of a complete explanation of the program in question, this type of feedback is presented during the beginning of a students use of the system or if the student has been identified

as a novice. Diagnostic feedback is an abbreviation of Simulative feedback and contains 2 two components: an abbreviated explanation of where errors in the program are described and process explanation dealing with the inputs and outputs for the defective program. Diagnostic feedback is provided to more advanced students and/or as the student uses the system more. Customized feedback is described as feedback that includes only explanations related to the processes and objects that the system has determined the student is deficient in [23].

Evaluation of the system was performed on a version that focuses on semantic and runtime errors. This version supported 4 types of feedback delivery, including none, demand, error-flag, and immediate. "Demand" refers to feedback that is provided when the student requests it. Error-flag refers to a methodology where the system informs the student of the correctness of their answer by changing its color – green for correct, red for incorrect. Immediate feedback in this system is a mode where as soon as the system determines that an incorrect answer has been entered it guides the student. This type of feedback utilized 3 levels of scaffolded hints: abstract, concrete, and bottom-out. Abstract hints are simple reminders of facts in the domain, such as the definition of a dangling pointer. Concrete hints are hints that draw the student to some specific incorrect behavior in the program and is vaguely Socratic. For example, if the student references a variable before assigning it, the system would ask them if the variable is being referenced before it is assigned. Bottom-out is more to the point and simply tells the student a fact about their answer in English [23].

The interface provided to the students consisted of a panel for the program and a

panel containing of answering options and feedback provided from the tutoring system to the user. The system was evaluated three ways: in isolation, against a workbook, and against the different feedback modes it provides [23].

The first evaluation was tested with the pretest-practice-posttest methodology without a control group. The results of the evaluation compared favorably to human tutoring, achieving slightly higher learning effect size than a human tutor. In the second evaluation, the author utilized the pretest-practice-posttest methodology where control group students received a workbook and test group students received the tutoring system. The end difference between the two groups was not statistically significant but the tutor group did attain slightly higher scores. The third evaluation compared minimal feedback to simulative. The minimal feedback mechanism corrects the students' answer without explanation; simulative provides both answer and explanation. The same methodology was used and incorrect answers were penalized. Effect sizes calculated when comparing pre and posttest results slightly favored the minimal feedback system, with students commenting that the simulative feedback was too verbose [23].

4 Chapter 4: Implementation

The research discussing how to teach students to debug their programs and that analyzes the novice's defect and debugging strategies is crucial – how can we help the student learn if we do not understand, fully, the problem they are facing. The systems that have been developed to assist the novice and teach them this domain are also crucial but support the novice in narrow ways, support languages the novice is less likely to be using, or are aimed at teaching programming on the whole as opposed to focusing on debugging.

What is needed is a system that enables the student to practice debugging broken code and assist the student in debugging their own code when they encounter a problem. This section discusses ITS-Debug, built to assist the novice programmer who is taking an introductory level course taught using the Java programming language. This system is capable of three different modes of operation and contains all four standard modules of an Intelligent Tutoring System (Domain, Student, Pedagogical, Communication). The first mode provides handwritten exercises to the students that consist of broken code; the student is tasked with fixing the program presented. This first mode is limited by the static nature of its exercise base; a second mode alleviates this limitation by determining what the student needs to learn and presenting a suitable exercise generated on the fly by breaking a working Java program that exemplifies a certain Java programming topic. The third mode breaks from this exercise-based scaffold and provides the student with a workspace where they can work on an assigned program they are developing from scratch, receiving assistance on the defects they create within

their own program. This section discusses the architecture and implementation of the ITS-Debug system.

4.1 **High Level System Overview**:

ITS-Debug has been split into two main systems consisting of an analyzer module and the ITS itself. For workflow clarity, the Front End user interface is depicted separately from the ITS despite being part of the Communication module. The basic workflow of the tutoring system consists of the following steps. First, the student goes to the system's URL in a modern web browser (it is known to work in Firefox, Chrome, or IE 10+). Then they log in to the system. The system then determines which phase the student is mapped to – choices include phases 1 through 3. Phases are discussed below; all phases either present a broken problem to the student and ask them to fix it or allow the student to write a program themselves and receive assistance on any defects they create in their own code. The student is then able to edit the code, compile and run the code, and receive assistance on a host of syntax, runtime, and logical defects that might be present in the exercise or that they may inadvertently create themselves.

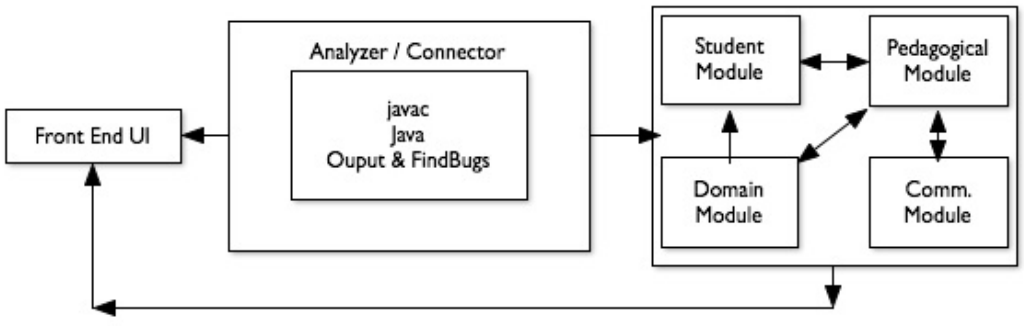


Figure 2: System Design - High Level

Phase 1 is an exercise-based system where the exercise database consists of handwritten exercises meant to exemplify some coding defect that the student is asked to fix. Phase 2 is also an exercise-based system but exercises are generated dynamically by breaking a working piece of Java code that already exists within the system and presenting the newly broken code to the student as an exercise. Phase 3 breaks from the exercise-based model and allows the student a more open environment to work in. In this phase students are able to work on a predefined lab exercise within the system where they receive a minimal code template. The template defines a class with instructions given in a comments section and an empty main method to start the student out. From there the student can start working on the lab exercise and receive assistance on any bugs they code into their own program.

For experimental purposes, when the student logs in for the first time, they are asked to complete a series of surveys. The first document they receive is the informed consent form. After this, they receive a pre-survey and a pre-test. Once they have completed these the system starts in earnest, asking the student to identify their modality. The system has been built to be multi-modal, meaning that it is meant to support different learning styles. Three modalities are currently supported: kinesthetic, verbal, and visual. Kinesthetic learners "learn by doing," which the system handles trivially, as it is all about providing the student an environment in which they can work on real debugging problems with some support. Verbal learners learn better through textual examples, while visual learners learn best through visual or animated examples. When the student first signs on to the system they receive two choices for learning style.

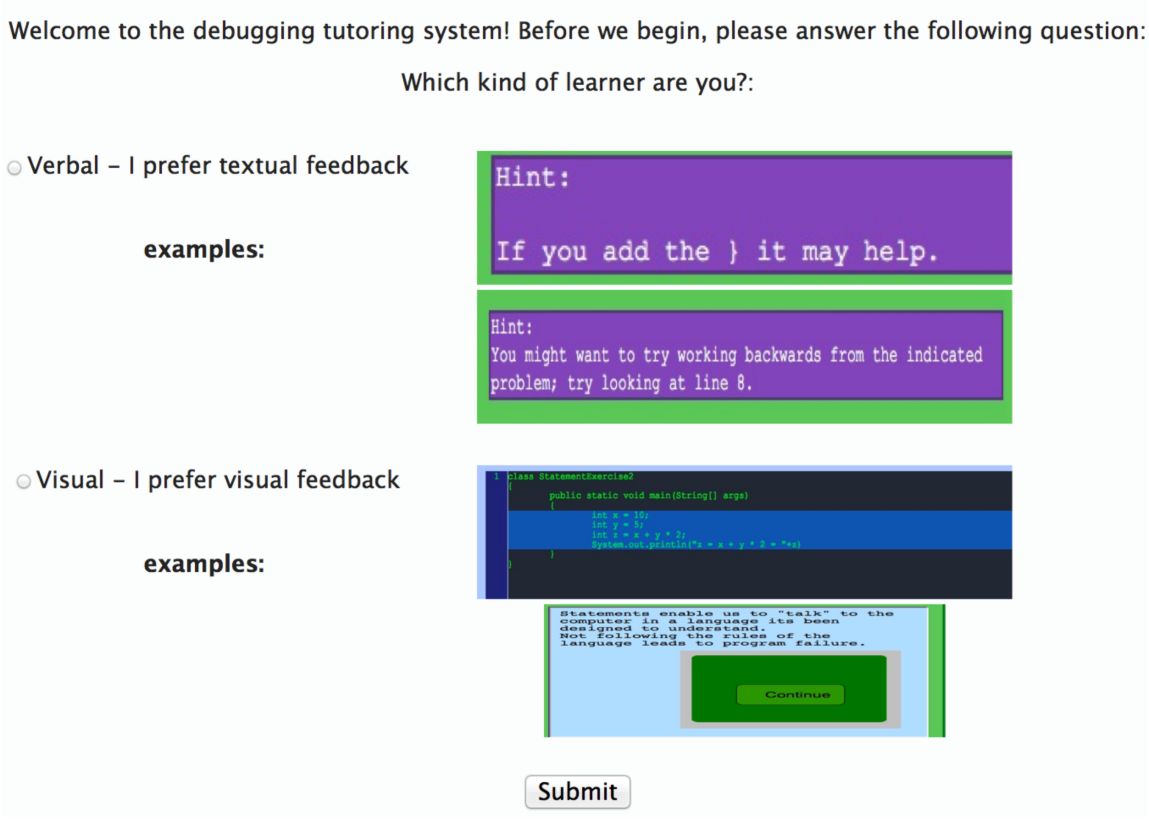


Figure 3: Form – Modality Selection

The student is able to select their learning style by selecting the appropriate radio button and clicking the "submit" button. Students are only asked to complete this step once, in order avoid the student switching modality during evaluation and introducing more variability into the data collected by the system. The selection of learning style helps drive the Pedagogical Module's selection of remediations. The impact of this choice on the system's feedback is described in more depth in the section regarding the Pedagogical Module specifically.

Once this initial set up is complete, the student's interaction with the tutoring system is driven in part by the Analyzer/Connector module described in the next section.

4.1.1 Analyzer/Connector module

The analyzer module is responsible for preprocessing and output analysis. This part of the system was built as two components. The first component is a Java program that ties in to the javac compiler and outputs details about the Abstract Syntax Tree generated during compilation. Connection to the javac compiler and extraction of compilation data is facilitated through the javax.Tools.CompilationTask package and the Java Diagnostics Collector. Information from this component is used in phase one to determine more accurate line numbers. In the event of an unsuccessful compilation, this module iterates through a Diagnostic Collection generated by javac to extract the javac key and generated error message. These details are then passed on to the Domain Module. If a compilation attempt is successful, this module runs the student's code through the Java Runtime Environment in order to determine if the code produces an exception or if the code produces any output. If an exception is thrown, the analyzer module passes the type of exception thrown to the Domain Module for further analysis. Otherwise, the module compares the student's output to the expected output for the exercise at hand. If the output matches the output expected after removing any heading and trailing whitespace characters the student's solution is deemed correct. Otherwise, the student is informed that their output does not match with the expected output for the exercise and their code is run through the static analysis tool FindBugs, described earlier in this document. If FindBugs finds a logical error pattern it recognizes, these details are passed to the Domain Module for analysis and later consumption by the Pedagogical Module to determine appropriate further courses of action.

In phase 1, abstract syntax tree (AST) data is used to determine more accurate line numbers for remediation. In phases 2 and 3 AST information is utilized further to help with learning from student solutions and for effecting causal analysis. The Analyzer module is contained within the Communication Module as part of the main workflow and is responsible for calling the first component. On successful compilation, the second component then tries to run the program. If the output matches what the system has been told is the correct output for a given exercise, then the analysis stops there. Otherwise, the analyzer module runs the Findbugs static analyzer to search the code for logical defect patterns. The Analyzer module routes this data from javac, the Java Runtime, and FindBugs to the tutoring system. The three phases of the system are discussed in more detail in the next three sections of this chapter.

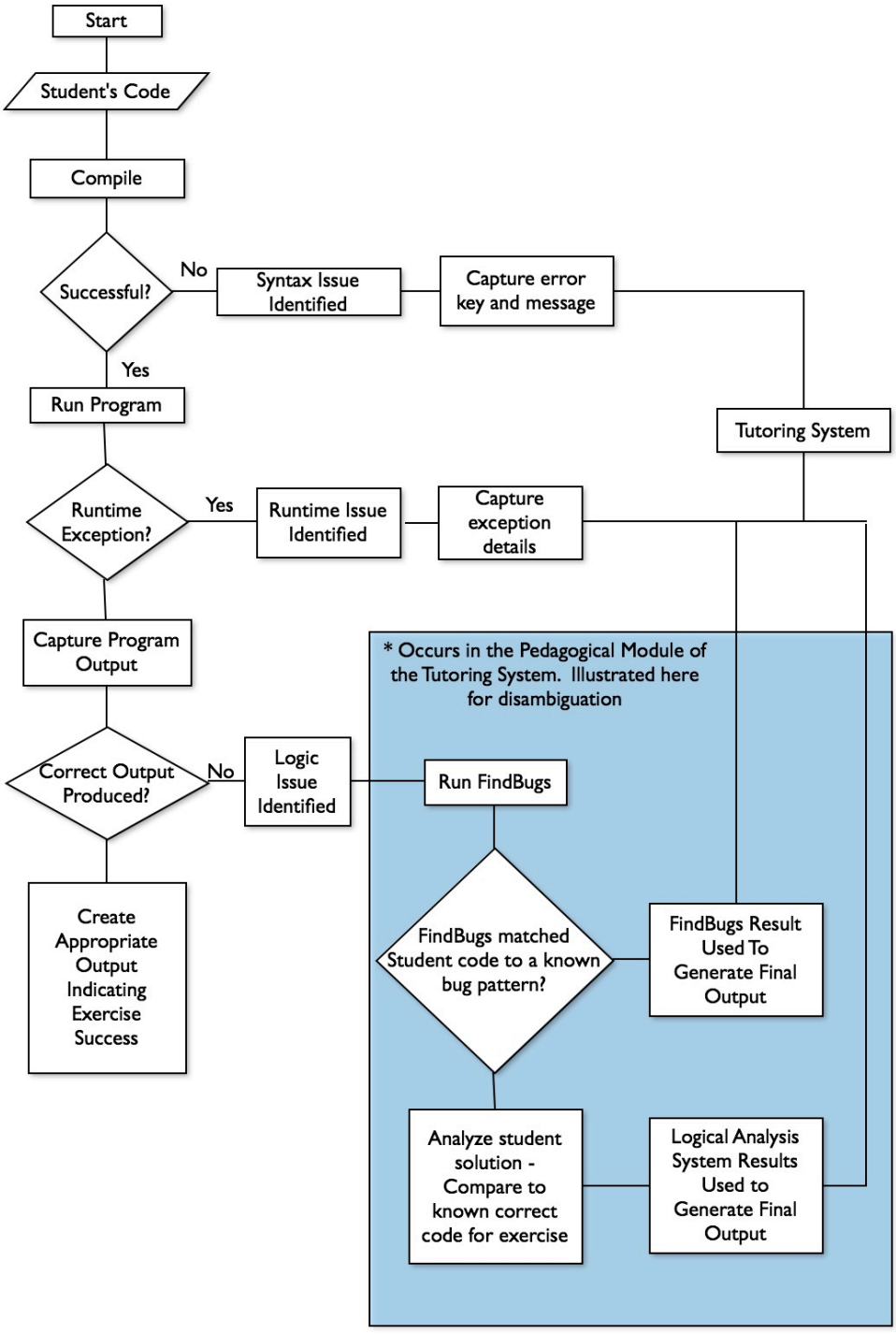


Figure 4: Connector / Analyzer Workflow

4.2 Phase 1: Handwritten Exercises

Phase 1 consists of the core of the tutoring system and includes an exercise system that allows the student to practice with a database of buggy programs, each containing one defect that the student is tasked with fixing. Though the database is static, exercise selection and feedback are dynamic, with a Domain Module for reasoning about the student's programming defects, a Student Module for representing what the system currently believes the student knows, a Pedagogical Module for teaching the student debugging concepts, and a Communication Module for student-system interaction. The Pedagogical Module is also responsible for exercise selection, which is based on the student's history within the system.

4.2.1 Domain Module

The Domain Module is utilized to diagnose programming defects and to suggest a solution to be implemented by the user. This research proposes that the debugging domain is predominantly case based; therefore this module is implemented using case based reasoning. A formal definition for cases in this system follows.

Case c = <uid, mt, st, dl, eid, uc, ss, es>

uid = unique numeric identifier [0..infinity]

mt = main type, represented by integer, member of {SYNTAX, RUNTIME, LOGIC}

st = sub type, represented by an integer [0..infinity], predefined in a subtopic table

dl = difficulty level

eid = error identifier for main type

uc = usage count

ss = solution steps, provided in a meta language construct

es = error symptoms, presented as a key

The main type field differentiates the source of the error message for a given case. If a case's main type is syntax, the error generated by the java compiler. Runtime errors in this system are synonymous with uncaught Runtime Exceptions and are therefore generated by the Java Runtime Environment. Logical errors, the hardest type of error for human and computer alike to recognize, are handled with the open source static analysis tool, FindBugs.

The subtype field maps a case to the subtypes of a topic. Topics in this system correspond with overarching topics of the first progression computer science for majors course and include: elementary programming, selections, loops, methods, single dimensional arrays, multidimensional arrays, objects and classes, strings, and I/O. Subtopics represent specializations of these topics. For example, the selections topic has subtopics if, if/else, nested if, and switch/case; covering the major constructs used in

branching. A table of all subtopics is provided as an appendix to this document (appendix *SUBTOPICS*).

Each of the main types requires specialized representation due to the fact that different details are provided by the different systems utilized by this system as subsystems. Because of this, there are three sub tables to the main case table. For compiler messages there are 3 details of importance. The compiler generates 3 types of messages: errors, warnings, and miscellaneous. Errors and warnings flag syntax level errors or possible pitfalls. Miscellaneous is a catchall category for certain remaining errors and javac messages.

The error identifier field is a foreign key into a specific table built to represent extra details for the main type of the case. Syntax cases, runtime cases and logic cases require some additional data that is not common between the three types. This extra data is described in further detail later.

Usage count represents the number of times this case has been calculated to be the most similar to the current defect in the front end of the tutoring system.

The solution steps field consists of a suggested solution for the case defined using a simple language of actions and constructs developed for this system. Sentences in this language are of the following form :

<action>:theAction; <item>:someJavaConstruct;

The language, as implemented in the system, may be defined as follows:

S:
<action>:Action; <item>:Item;
Action:
add | remove | delete
Item:
LanguageConstruct | SyntaxToken
SyntaxToken:
{ | } | (|) | ‘ | “ | [|] | semicolon | */
LanguageConstruct:
array access | array type | assignment | binary | block | break | case | catch | class | compound | do while | enhanced for | erroneous | expression statement | for | identifier | if | import | instance of | literal | member select | method | method invocation | new array | new class | parameterized type | parenthesized | primitive type | return statement | switch | throw | try | typecast | type parameter | unary | variable | while

This formalism allows for the automatic generation of remediations for the student and in phase 2 will facilitate the automatic creation of exercises and assist in the process of learning new solutions; this will be defined further in the subsequent section discussing the design of Phase 2.

The error symptoms field currently consists of the error message generated when the error is present. In the case of a logical defect, the error symptom field holds the FindBugs short key, later disambiguated with the appropriate FindBugs long key.

The case base holds two types of compiler messages: the rules from the properties file and specific variants of those rules. The rules in the properties file define unique string keys that are supplied for different classes of syntax errors. In the case of compiler.err.expected, the original rule and message from the properties file is included

along with specialized rules for ';' expected, '(' expected, '{' expected and so forth. This is because cases in the case base are further separated by their suggested solution

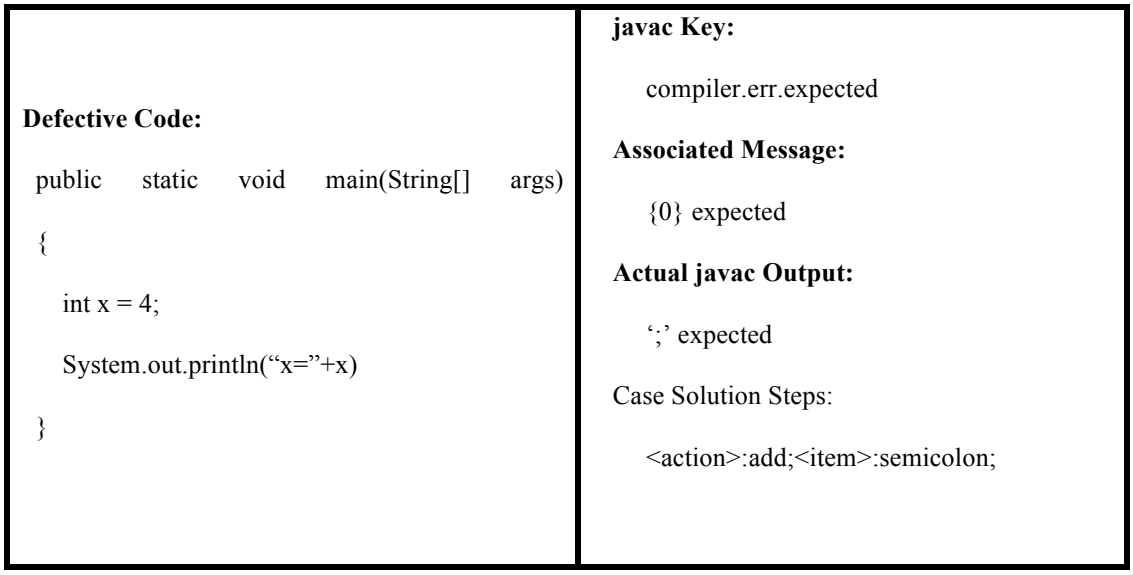


Figure 5: How the javac properties file is used to handle a missing semicolon

For example, error messages for missing syntax tokens are created via the compiler.err.expected rule in the properties file. The error message for this rule in the properties file is "{0} expected". The compiler fills {0} in with the missing token at runtime.

Runtime exceptions are represented by a text key that corresponds to the class names of unchecked Java runtime exceptions. An extensive list of runtime exceptions was generated for the case base, totaling 755 entries at present.

Defective Code:

```
public static void main(String[] args)
{
    int[] numbers = {1, 2, 3};
    System.out.println(numbers[3]);
}
```

Runtime Exception:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:3
        at forceArrayAccessEx.main(forceArrayAccessEx.java:6)
```

Redacted Runtime Exception:

```
Exception in thread {0} java.{1}.ArrayIndexOutOfBoundsException:
```

Case Solution Steps:

<action>:edit;<item>:array access;

Figure 6: Example Runtime Exception

Logical defects are represented using fields defined by the designers of the FindBugs software. This includes a short key, a long key, a category, a short description, and a long description. The logic map table also has a "source" column to allow for future use of other static analysis tools. Of this data, the main fields utilized by the tutoring system are the long key and the short description. The long key is favored due to the fact that it is a unique identifier and the short key is not. The short description is favored over the long description due to the excessive length of some of the long descriptions.

Defective Code:

```
public static void main(String[] args)
{
    int sum;
    int a = 2;
    int b = 3;
    sum = a +b;
    System.out.println(b);
}
```

Findbugs Output:

M D DLS_DEAD_LOCAL_STORE DLS: Dead store to $L1 in ForceDls.main(String[]) At ForceDls.java:[line 8]

Long Key:

DLS_DEAD_LOCAL_STORE

Case Solution Steps:

<action>:edit;<item>:variable;

Figure 7: Example Logical Defect

As mentioned earlier, FindBugs is a static analysis tool that utilizes defect patterns. The tool reads the compiled byte code for a program and then scans it to see if any known defect patterns have been implemented. Each long key corresponds to a defect pattern. Defects within FindBugs belongs to 1 of 9 categories, including dodgy code, experimental, bad practice, correctness, internationalization, malicious code vulnerability, multithreaded correctness, performance, and security. ITS-Debug launches FindBugs as a command line tool; the resultant output is parsed by the analyzer and later consumed by the tutoring system to match the long key to a case in the case base.

4.2.2 Student Module

The Student Module is responsible for modeling the student's knowledge and knowledge gaps. Within this particular system, the Student Module mainly keeps a record of the student's exercise and case progress. With regard to exercise progress, ITS-Debug logs which exercises the student has attempted, how many attempts the student has made on a given exercise, and whether or not the student has submitted a correct answer for that exercise. Exercise progress is used by the Pedagogical Module for exercise selection logic and to determine the proper remediation level to return to the Communication module. With regard to case progress, the Student Module logs each case the student has encountered while using the system. The information recorded includes how many times the student has encountered the case and how many times the student has correctly resolved the case. Correct case resolution is equivalent to solving an exercise by solving the case. In the future, system case resolution may be redefined as the absence of the associated symptom in a subsequent compile operation. In other words if the error existed when the student compiled the program at time X and then no longer exists when the student re-compiles the program in the future at time Y, it may be possible in some situations to assume that the code at time Y successfully removes the previously reported error at time X.

4.2.3 Pedagogical Module

The Pedagogical Module uses information from the Student and Domain Modules to determine what the student's current problem is, what reason there might be for the current problem's occurrence, and how to help the student learn from that

problem. The Domain Module supplies the problem description to the Pedagogical Module, this information is used by the Pedagogical Module to select an appropriate remediation from a library of remediation options.

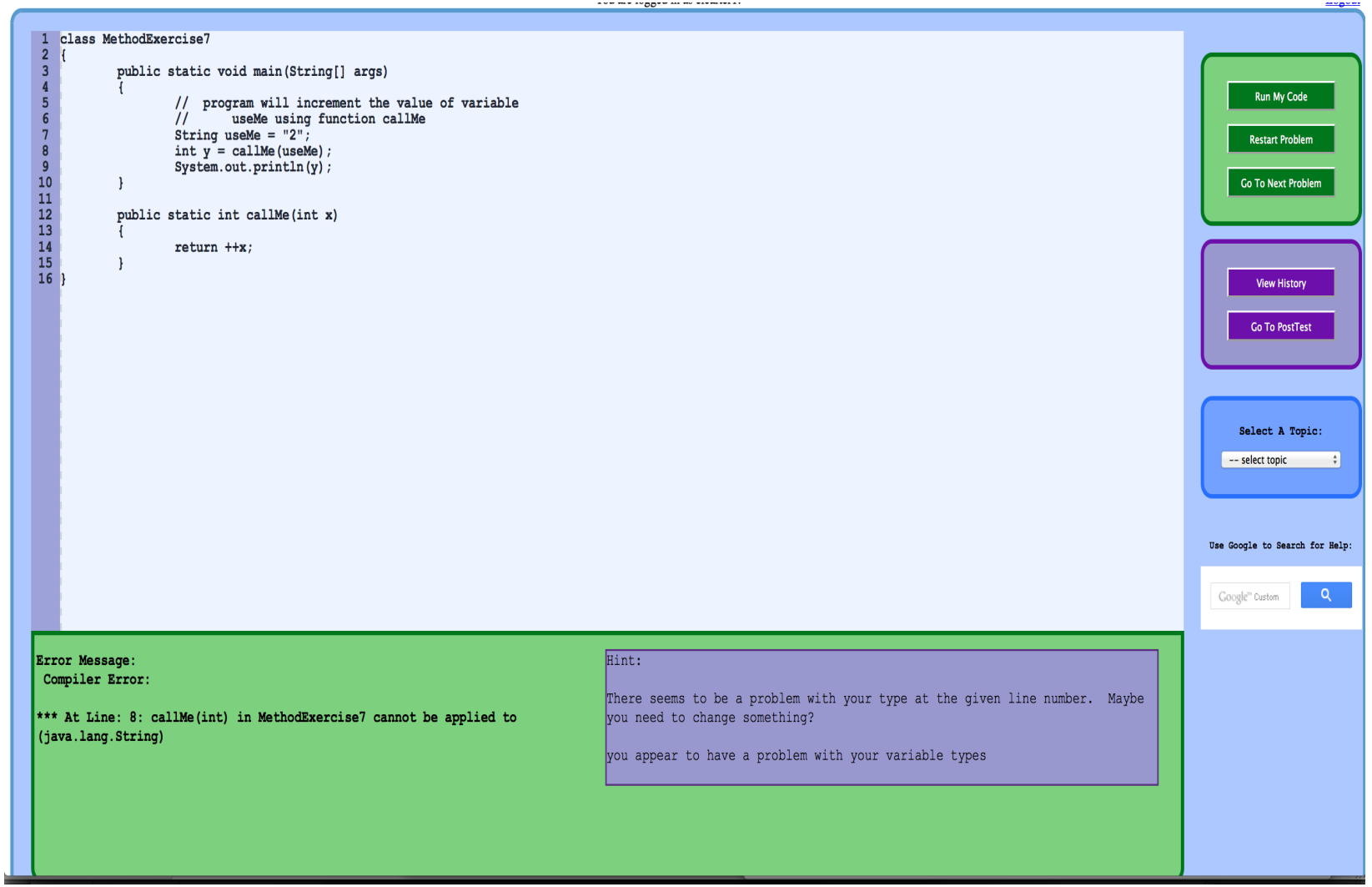


Figure 8: System providing Verbal Remediation

Figure 9: Example of visual remediation supplied to a student

The Pedagogical Module is also responsible for exercise selection. Exercise history is retrieved from the Student Module and then analyzed in order to determine the proper topic, subtopic, and difficulty level of exercise for the student. Factors used in this calculation include what exercises the student has been presented with, what exercises the student has completed, and how many attempts the student has made to complete the exercise. Based on this history, an appropriate exercise is selected and then returned to the Communication module.

The Pedagogical Module also is responsible for determining how to teach relevant material to the student. If the student submits an incorrect answer, the Pedagogical Module tries to assist the student and teach them how to properly debug the error at hand. A range of different remediations are supplied to the student, depending on the student's chosen learning style and number of attempts on a given exercise during a given episode with that exercise. A student who selects or is determined to be verbal will receive only text-based remediations. Conversely, a student who selects or is determined to be visual will receive only visual cues and animations with examples.

Depending on how many attempts have been made on a given exercise at a given time, different remediations a provided. There are currently 4 levels of remediation for each learner style. All remediations are scaffolded, with each subsequent remediation assuming the student knows less sabot the defect and topic at hand.

4.2.3.1 ***Verbal Remediatons:***

Verbal remediations are scaffolded in the following manner. The first time a

student requests assistance with a given exercise they receive a hint directing them to pay more attention to the line number given in the error message and advice on what to look for when they examine that line. The second level remediation suggests a tactic. Currently, the system supports two debugging tactics: forward trace and backward trace. The tactic is chosen based on either a tactic associated with the case or, if no tactic is defined, a tactic is chosen based on the main type of the case corresponding to the current error. If a case has no suggested tactic and is either a syntax or runtime error, then a backward trace is suggested. Otherwise, a forward trace is suggested. There was no existing research encountered as to dominant decomposition methods for certain classes of program errors but an automatic solution needed to be chosen for situations where no tactic has been suggested. Reviewing syntax, runtime, and logic errors revealed a possible tendency towards backward tracing for syntax and runtime errors and forward tracing for logic errors. However, this decision is somewhat arbitrary and needs to be examined in further detail in the future.

The third level of verbal remediation is based on the main type of the case found in the case base. Students receiving this level of remediation will receive a hint pointing them to pay attention to their syntax or their logic near the line the compiler, runtime system, or FindBugs indicated the problem was located. For the fourth level of remediation, students receive a tutorial based on the topic they are currently dealing with. For example, if the student is working on an exercise that deals with selection statements, they will receive a textbook like tutorial on selection statements and an example meant to exemplify how to properly write selection statements. After all

remediations have been exhausted for a given exercise the student is shown the answer to the current exercise, with the differences between the two exercises highlighted.

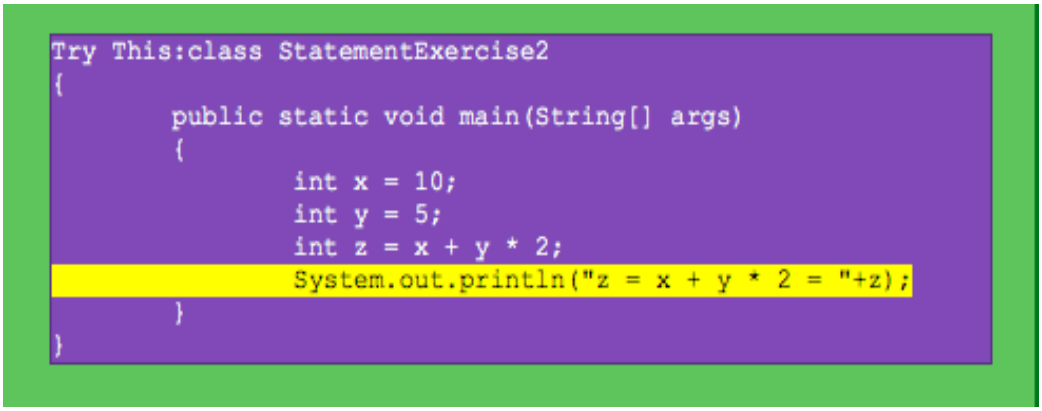


Figure 10: Answer to student after all remediations for an exercise are exhausted

4.2.3.2 ***Visual Remediations:***

The visual remediations in this system were designed to be as equivalent as possible to their verbal counterparts. The first level remediation for a verbal student highlights the line number that the student should be looking at to solve or start solving the current defect. Second level visual remediation consists of a visual representation of the suggested decomposition. For instance, if the current defect recommends that the student work backwards, the visual student will see multiple lines highlight starting from the line the student was originally directed to backwards to the beginning of the affected block of code. For cases suggesting a forward trace, the animation works from the suggested line forwards to the end of the current block of code. Third level visual remediations, as with verbal remediations, provide assistance based on the main type of the case. To satisfy this type of assistance from a visual perspective three animations

were create; one for syntax issues, one for runtime issues, and one for logical issues. Each of these animations contains a discussion of the type (syntax, runtime, logic) as well as how to approach resolving the problem. These videos are capable of switching between their original content, which discusses the main type of the error in a broad context, and a more case-specific version that replaces certain aspects of the broad details with specific details from the case's solution steps.

The Communication module in this system ties all of the other modules together. (The tutoring system is served as an ASP.NET web site built on the Mono platform. Mono is an open source implementation of the .NET framework that can run on Linux, Unix, and Macintosh operating systems. The Communication Module also utilizes HTML5, javascript, and jQuery to accomplish its goals.) .

The user interface for this system was built using the design patterns already employed by existing IDEs, with a special preference for the DrJava and Eclipse layouts since students at this level of their Computer Science education often utilize those environments.

4.2.3.3 Remediations Not Tied to a Modality

Two other remediation levels are present in ITS-Debug that are not tied directly to a modality. Both of these are new levels of remediation added during the Fall 2013 semester. The first of these levels is an analogical reasoning component. This component becomes available to the student once they have successfully solved a case in the system. Successfully solving the case is considered to be the successful removal of the error from the code. When the student successfully solves a case, their solution is

saved to the database along with the case that represents the original error message. If the student encounters the same case again, the system reminds the student at the second attempt to solve the exercise that they have seen this problem before. It also asks the student if they would like to see what they did last time. If the student clicks the button provided, they receive a popup that shows the case, the original buggy code, and the solution that they coded to the defect. This level of remediation takes advantage of the case based nature of the domain and the implementation, providing the student with an opportunity to revisit exactly what they did the last time they encountered the issue.

4.2.3.4 Logical Analysis Module –Feedback When FindBugs Finds Nothing

The other new form of remediation not tied to a modality specifically applies to logical defects. This new level of assistance aims to provide more relevant details for logical defects to the student than the reesarcher was able to obtain using the FindBugs system. Specifically, two new levels of remediation are introduced in the presence of a logical defect: indicate to the student roughly how far from the answer they are and let the student see an overview of where their solution differs from the known solution. This module is invoked when FindBugs does not match a bug pattern to the student's code, though the code still does not produce correct output.[2] In this scenario, ITS-Debug uses one of the many possible correct solutions to any given exercise it can generate. This solution is passed through the Connector module to gather the parse tree

[2] Note that this module is **only** utilized in the generation of feedback for logical defects – no other part of the system utilizes this module.

information generated by the javac compiler. ITS-Debug reads the parse tree data in the form of a depth-first pre-order traversal generated by javac.[3] Once the tutoring system has reconstituted the traversal into a list of nodes it performs the same actions with the student's current code. Additionally, separate lists are kept for each Java construct (as defined in the meta language earlier). Once both programs' parse data have been read into memory, the individual lists for each construct are compared between the two programs. Each node in the known solution is compared to each node in the student solution. The comparison generates a confidence level, represented by a percentage, indicating how closely the nodes between the two lists match (a higher confidence level implies a closer match).

The exact comparison details differ between different constructs. Each construct, and consequently each type of node, has a different set of meta data defining the construct. For example, a For Loop node contains the following sub constructs: initializer statement, conditional statement, update statement, and a code block. When the comparison runs, the subconstructs between the two nodes are each compared. If the sub-constructs are not equivalent, the difference is logged and saved to assist in providing feedback.

After the comparison is complete, the nodes in the known solution are separated into three lists: items that appear to exist between both programs but have different configurations, items that are in the known solution but missing in the student's code, and items that appear to exist in both programs but are placed in a different location in

[3] For more details on parse tree utilization in this system see Appendix F

the student's code. The confidence levels for each node in the solution code are used to classify which list the node belongs to. If the confidence level of the node is between 50 and 90 percent, the system assumes that the node exists between both programs and adds it to the "different configuration" list. If the confidence is less than 50, the node is considered to be missing from the student's solution and is added to the "missing" list. Additionally, if the confidence level is in the range (50, 100) and has different parent or children nodes, it gets added to the "different location" list[4]. These lists are then used to build scaffolded feedback messages for the student, resulting in three different levels of feedback. The first of these levels behaves as before – a message is chosen at random from a bank of messages and provides a hint to the student that a logical defect exists in their code. The second level utilizes the above analysis to provide a message indicating how close or far the student is from the known solution and consists of a hint from the following list:

- You seem to be really close to the answer!
- You're pretty close to the answer.
- Your answer seems kind of far from the answer I know. If you thought you were closer, you might want to review the problem to make sure you're approaching it the right way

[4] These specific thresholds were a design decision made during pre-evaluation testing of the system.

The selection of which hint to show is based on the percentage that the student's solution differs from the known solution. This is calculated via the previously discussed confidence level mechanism.

The second level of logical error remediation shows the student a list of what nodes differ and a brief overview of the difference. These differences are computed at the same time as the confidence levels and differ in granularity. The exact form of the feedback message in this instance takes the form of a bulleted list of items identified as being configured differently, missing, or transposed in the student's code (as determined by the assignment to the similarly named lists described above). For example, if the answer and the student both have a method called sum but the student's parameter list is different than the answer's parameter list, the feedback will include a message that the student's parameter list is not as expected. Alternatively, if the solution contains a method called sum that appears to be missing from the student's code, the system will inform the student that one of the items that appears to be missing from their solution is a method named sum.

4.3 **Phase 2: Dynamic Exercise Generation**

Phase 2 changes the system in two ways. First of all, and the main goal of Phase 2 since the original design of this project, the system gains the ability to generate novel exercises on the fly. This is facilitated by the meta-language utilized as the solution step for cases in the case base. The full definition of the terminals of the meta-language, described briefly earlier, follows:

```
SolutionAction A ∈{Add,Remove,Edit_towardsCorrect}
ProblemAction P ∈ {Remove, Add, Edit_awayFromCorrect} == ~A
Construct C ∈ {ArrayAccess, ArrayType, BinaryOperation, Block, Break,
              Case, Catch, Class, CompoundAssignment, DoWhile,
              EnhancedFor, ErroneousStatement, Expression, ForLoop,
              Identifier, If, Import, InstanceOf, Literal,
              MemberSelect, MethodInvocation, Method, NewArray,
              NewClass, ParameterizedType, Parenthesized, Primitive,
              Return, Switch, Throws, Try, TypeCast, TypeParameter,
              Unary, Variable, While}
```

The second main alteration involved in Phase 2 is the ability to acquire cases from student's solutions. The next two sections will discuss the implementation of these two systems in detail.

4.3.1 Exercise Generator

The exercise generator tries to remedy a basic issue with phase 1 – the student will, eventually, run out of questions. If the student runs out of questions, the exercise-based functionality of the system becomes useless to that student. If the system could create targeted exercises on the fly there is the potential for an almost infinite number of exercises within the system, making the system a more long term assistant during the student's educational process.

The basic workflow of the exercise generator is represented in the pseudo-code below, followed by a detailed description.

```
GenerateExercise(underlyingCase,studentData,selectedTopic)
  if(underlyingCase.SolutionSteps != null)
    intermediateRep[]←parseSolution(underlyingCase)
    action ← intermediateRep[0]
    construct ← intermediateRep[1]
    action ← reverse(action)
    selectedCode ← constructDet(construct)
    startLocation ← selectedCode.StartLine
    endLocation ← selectedCode.EndLine
    original ← selectedCode
    if (action == "edit")
      newCode←selectedCode.manipulate(case,studentData)
      selectedCode ← replace(selectedCode, newCode)
    else if (action == "add")
      selectedCode ← insert(selectedCode, construct)
    else
      selectedCode ← remove(selectedCode, construct)
```

Figure 11: Pseudocode – Exercise Generation

First, the Pedagogical Module identifies one case from the case base as the most relevant case for exercise generation given the student's selected topic and the student's history within the system. This case can be any case in the system – syntax, runtime, or logic. The Pedagogical Module also supplies this module with the student's topic selection. After receiving this information the Exercise Generator selects an example class from a bank of examples containing correct code. Once selected, the Exercise Generator runs this code through a process similar to the Communication module's Connector module in order to generate syntax tree data about the example program. This data, as when the student is sending in their solution for compilation, is written to disk by the Connector module and then picked up by the Exercise Generator and read into memory as a hash table by using an instance of the Student Module reserved for the

use of the tutoring system. The instance of the Student Module parses the selected code in to memory and then proceeds to parse the solution steps attached to the earlier selected case. The solution is then reversed – add actions become remove actions, editing actions become editing actions to insert a defect, and remove actions become add actions. After the solution step is reversed, the Exercise Generator locates an appropriate construct as defined in the solution steps field of the case. For instance if the meta-language in the solution steps field indicated that a for loop should be modified, then the generator would look for a for loop in the example code that was previously compiled and parsed. The generator also takes into account whether the case represents a syntax, runtime, or logical defect in the code and concentrates on generating an error of the correct type. The actual error generation occurs in the Java Representation classes utilized in the system to model the different language constructs and their sub-constructs. Each construct representation class contains an overloaded method *manipulate* which receives case details and uses them to modify the construct towards the defect the case describes. This implementation was chosen because each construct would need to be modified in a different manner to create the different kinds of defects described in the case base. For example, the class representing if statements is capable of generating syntax or logical defects in any given if block in the sample code. If the manipulate method is instructed to generate a syntax defect, a random piece of key if statement syntax is removed. If the manipulate method is instructed to generate a logical defect, the system will either change the conditional statement for the if statement block to use a different comparison operator than it originally did or, if

present, remove the associated else block. The manipulate method is responsible for implementing the edit action for each construct towards the desired type of defect. Currently, only the remove and edit actions are supported in the generator. Remove implies that the case would advise the student to add a construct to the code. As the edit action is difficult to define within the current framework, random number generators are utilized in order to determine what sub construct to break (if applicable) or what syntax element to make malformed. Implementation of the add action was deferred as few cases utilize the 'remove' action from the solution steps meta-language that is the inverse of the 'add' action utilized by the generator. After the Exercise Generator has created the exercise, it is written back to file and saved to the database as a new exercise. Then the exercise is presented to the student and the original Tutoring System workflow proceeds from there.

```
public class StatementCode
{
     public static void main(String[] args)
     {
          System.out.println)"This is a small program ");
          System.out.println("meant for practice with ");
          System.out.println("simple Java statements.");
          String ABCs = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
          System.out.println("ABCs: " + ABCs);
          int number = 1;
          int number2 = 10;
          double number3 = number / (double)number2;
          System.out.println("Simple Math: number = "
               + number + " number2 = " + number2 + "
               number/number2 = " + number3);
          String numbers = "123456789";
          String lettersAndNumbers = ABCs + numbers;
          System.out.println("letters and numbers: " +
               lettersAndNumbers);
     }
}
```

Figure 12: Generated Syntax Error Exercise

Figure 12 above depicts a syntax error generated by the exercise generator module. In this exercise, the defect occurs in the first System.out.println statement – a closing parenthesis has been substituted for the opening parenthesis.

```
public class SingleDimensionArrayCode
{
     public static void main(int[] args)
     {
          int[] fibonacciNumbers = {1,1,2,3,5,8,13,21,34,55};
          int sum = 0;
          for (int i = 0; i<fibonacciNumbers.length; i++)
          {
               sum += fibonacciNumbers[i];
          }
          System.out.println("Sum = " + sum);
          double[] fibonacciNumbers_pt2 = new double[10];
          double runningSum = 0.0;
          for (int i = 0; i<fibonacciNumbers_pt2.length;i++)
          {
               fibonacciNumbers_pt2[i] = fibonacciNumbers[i]*.5;
               runningSum += fibonacciNumbers_pt2[i];
          }
          System.out.println("Second sum = " + runningSum);
     }
}

// exception generated = Exception in thread "main"
//java.lang.NoSuchMethodError: main
```

Figure 13: Generated Runtime Error Exercise

Figure 13 depicts a runtime error generated by the system, resulting from the manipulation of the SIngleDimensionArrayCode example class. Specifically, the error generated deals with manipulating statements using array types. The specific issue generated here involves the type of the array parameter args in function main, changing it from the expected String to an int. The attempt to invoke main with these parameters causes the Java Runtime system to throw a NoSuchMethodError exception because there is no definition for main that accepts an int[] parameter.

```
public class IfCode {
        public static void main(String[] args){
                int decision = 0;
                String result1 = "";
                String result2 = "";


                switch(decision){
                        case 0:
                                result2 = "King";
                                break;
                        case 1:
                                result2 = "Queen";
                                break;
                        case 2:
                                result2 = "Jack";
                                break;
                        case 3:
                                result2 = "Ace";
                                break;
                        default:
                                result2 = "Doh!";
                                break;
                }

                System.out.println(result1 + " " + result2);
        }

}
```

Figure 14: Generated Logical Defect

Figure 14 exhibits a logical defect introduced by the system in to the IfCode example from the sample code (correct code appears in the appendix). In this sample, the exercise generator has removed the if / else if / else block that was originally part of the sample. Due to this removal, the code no longer produce the appropriate output and a logical defect has been created. The comments in the sample (omitted here) instruct the student as to what output the original code was meant to produce so that the goal of the exercise program is clear.

In order to implement this functionality, the classes representing individual Java

constructs all contain a manipulate method and a specialized toString method. The manipulate method is responsible for editing any construct of the given type according to a set of rules specific to that construct and its sub-constructs. For instance, an if statement would require a different set of manipulation logic than a variable declaration. With an if statement we could modify the test condition, the block of code to be executed when the condition is true, or the block of code contained in the else condition. Or we could alter the syntax to make it incorrect. With a variable declaration, we are limited to manipulating the variable's name, the type, or removing a space or a semicolon.

After the manipulate method mangles the working construct, the construct is rebuilt by the specialized toString method and then sent back to the Exercise Generator for re-integration with the rest of the example code by overwriting the original version of the construct with the new version. The specialized toString is responsible for taking the mangled construct and formatting it properly for its re-integration into the example code. After exercise generation has completed the exercise is sent back to the Communication module and the system workflow proceeds as it did in phase 1.

4.3.2 Case Acquisition

The other major new functionality introduced in Phase 2 consists of case acquisition. This step completes the CBR cycle and turns the Domain Module into a full Case Based Reasoning system. The CBR consists of 5 stages: Retrieve, Reuse, Revise, Review, Retain. Phase 1 of the system implemented the Retrieve, Reuse and Revise steps. Review and Retain were postponed until Phase 2 in order to complete the

implementation of the main and most important aspects of the system contained in phase 1. Several different methodologies were reviewed to determine the best way for this research with its current goals to implement the last 2 stages of the CBR cycle. A hard coded, static case base is a proof of concept but is limited in its usefulness for tutoring – there are a potentially infinite number of defects, each with an infinite number of solutions. In order to keep the system relevant it should have the ability to acquire new cases. New case acquisition potentially could focus on two aspects of cases as they exist within the system. The first would be errors the system has never encountered before and does not yet know how to remediate. The second would be encountering a new solution to a known defect. This system favors the second method since the preexisting case base is comprehensive over the domain of errors a novice might encounter, making it unlikely that the novice will code a defect that is completely new to the system. Specifically, the case base contains cases for every syntax issue javac is capable of reporting and every exception built into the Java Core Library, plus the ability to diagnose and remediate many logical errors. It is thus considered more likely that the students in question will come up with a different solution to an existing case than that they will create an entirely new case.

The case acquisition workflow follows an offline model, saving student solutions to be analyzed at a time when the system is not being actively used. This model was favored in order to avoid introducing another potential cause of latency at runtime. Every time the student solves an exercise in the exercise-based areas of the system, a pointer to the solution is saved in a separate file and stored to disk. The case acquisition

component is then launched later manually at an off peak time. Once the module is launched, it reads the file of pointers and prepares to process the solutions. The following pseudo-code provides an overview of the algorithm used to analyze each solution file.

```
GetSolutions()
     For each student solution
          Q ← enqueue(Q, solution)

ProcessSolutions(Q)
  Exercises[] ← getSystemExercises(), symptom ← ""
  diffResult ← null, proposedSolution ← null
  studentAST ← null, studentCode ← ""
  relatedExercise ← null, mostSimilarCase ← null

  while (Q.count > 0)
     proposedSolution ← Q.dequeue()
     studentCOde ← getStudentCode(proposedSolution)
     relatedExercise ← Exercises[proposedSolution.whichEx]
     originalCode ← relatedExercise.Answer
     if (studentCode != originalCode)
       diffResult ← handleDiff (studentCode,originalCode)
       questionCode ← relatedExercise.Question
       if (diffResult != null)
          symptom ← diagnose(current)
          studentAST←retrieveAST(proposedSolution)
          ml ← generateLanguage(questionCode, studentCode)
          if (!anyWithSameSolution(cases, ml, symptom)
            if(mostSimilar == null||mostSimilar.symptoms == null)
               newCase(symptom, studentSolution)
            else if (mostSimilar.Solution == null)
               alterExistingCase(mostSimilar, studentSolution)
```

Figure 15: Pseudocode – Case Acquisition

First, the system determines which student files to load for analysis based on the previously mentioned file containing pointers to student solutions. For each program referenced, the system first determines if the student's answer is textually equivalent to the original answer for the exercise. If not, diff is run to compare the student's answer

to the original answer and isolate the differences. Then the original exercise question code is run through the Connector/Analyzer module to determine what defect it originally contained. After this the javamap file for the student's answer, as generated by the Connector/Analyzer module, is loaded in to the system and the most similar case to the defect in the original code is selected. The case acquisition module then takes the student's solution and determines what steps the student took to reach their solution, ending in the generation of the solution step meta-language utilized within the system to both provide assistance and create phase 2 exercises. Once the meta-language has been generated, the system checks if no case relevant to the original defect contains the new solution or if the original solution was empty. If the solution was empty then the solution steps for that case are filled in. Otherwise, the case is cloned and the solution steps of the clone are replaced with the new solution from the student's code.

4.4 **Phase 3: Debug-Time Support for Novice Programmers**

The last phase of the system takes it away from being an exercise drill system and brings it closer to a novice-centric IDE that focuses on providing debug time support. This ability was the original goal of the system – to provide assistance within the context of a program the student is writing at the time that they need it. In order to accomplish this the system includes a small set of phase three exercises consisting of a minimal code template, whether or not the exercise will require standard in, any input options in the event that the program requires console input to run, the expected output for all input options, and one possible answer for the problem. An example of one of these exercises follows.

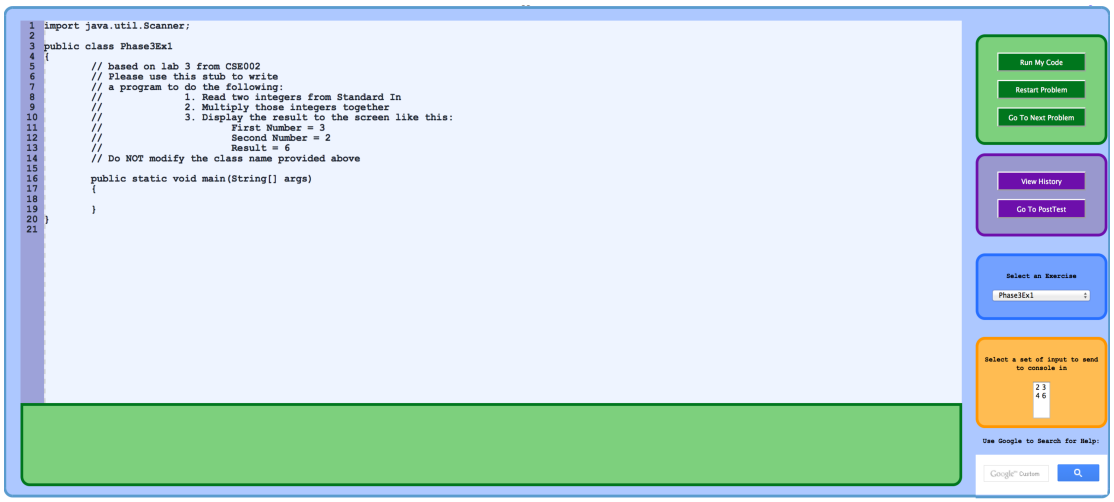


Figure 16: Interface for Phase 3: Debug Time Support

```
public class Car
{
        // This is a class for representing a Car
        // Please add a getter method for currentSpeed
        // and a setter method for mileage.
        // You should also create a default constructor
        // to set the member variables to default values.
        // The default value of currentSpeed should be 45.
        // The default value of mileage should be 0
        // make should default to Pontiac
        // model should default to TransAm

        private String make;
        private String model;
        private int currentSpeed;
        private double mileage;


        public static void main(String[] args)
        {
                Car myCar = new Car();
                myCar.setMileage(500.6);
                System.out.println(myCar);
                System.out.println("current speed = " + myCar.getCurrentSpeed());
        }

        public String toString()
        {
                return make + " " + model + " mileage = " + mileage ;
        }

}
```

Figure 17: Phase 3 Exercise Example

As displayed in the example above these exercises include a comments section that provides details pertaining to the program the students are to develop for a selected exercise. Students using the system can work on the exercises provided and receive

assistance from the system on any defects they code into the program themselves, as opposed to previous phases that served programs written by the author that were broken either manually or dynamically. Assistance for student created defects works in the same manner as assistance for defects in phases 1 and 2. In fact, phases 1 and 2 were both capable of assisting the student with defects they might have introduced into the exercises that were not already present.

The model for these exercises was chosen so that the system would retain the ability to tell the student they had or had not successfully completed the exercise. Additionally, this model allows the system to perform further logical analysis in the event the student codes a logical defect that is not recognized by FindBugs. This extended logical analysis was deemed necessary when, during early evaluation episodes, the coverage of the FindBugs system over novice logical defects was found to be insufficient. The new analysis augments the feedback available to the Pedagogical Module and provides the system with the ability to compare the student's solution to the known answer. This ability allows the module to indicate to the student how close or far they may be from the solution and provide a list of items that are different between the student's solution and the known answer.

The approach of forcing the student to examine one error at a time is maintained by having the system focus on the most relevant error. The following assumptions have been made in order to determine the most relevant error in the presence of multiple errors. For syntax errors, the expert programmer generally starts with the first reported defect. Runtime errors, meaning a runtime error/exception that causes the program to

stop running, usually occurs only once per run, as this error will cause the runtime environment to crash. For the last class of errors, logical defects, FindBugs is run against the student's code and the first reported error is retrieved

4.5 Research Questions Revisited

4.5.1 How should the domain be represented and reasoned about?

There are several ways that the domain of debugging has been viewed and different ideas exist as to how to represent and reason about this domain. Andreas Zeller wrote a comprehensive book on the subject of debugging including many methodologies and an account of research into automatic debugging techniques. Different from the other literature reviewed as part of this research, Zeller recommends adopting the idea that debugging is a scientific process. In other words, human debuggers should employ the scientific method of defining a hypothesis, testing that hypothesis, and then either accepting or rejecting that hypothesis based on what occurs after attempting the code change suggested by the hypothesis. According to Zeller, applying the scientific method will help programmers eliminate their defects more efficiently [43].

Weiser, on the other hand, analyzed expert programmers scientifically [38]. He provided a group of expert programmers a faulty program and gave them a timeframe to finish fixing the program. While the programmers worked on their problems, Weiser videotaped and logged their actions and instructed the participants to "think out loud." A major outcome from Weiser's work is the idea of program slicing. That is, a programmer will actually mentally slice their code into pieces that do and pieces that do

not affect the current outcome when the program is run. This approach has been adopted by the static analysis community and used to define several different kinds of formal slices and slice operations including backward slice, forward slice, chop, dice, and backbone [37, 43].

Forward slice and backward slice are static with respect to the code. A forward slice follows dependencies from a slicing criterion A, such that all statements affected by A are included in the slice. This type of slice is formally denoted as $S^F(A)$, and creates the set of elements defined as: {B | A influences B}. Statements not included in a forward slice are those that cannot be influenced by slicing criterion A. A backward slice is a backward trace that determines all the statements that could affect B. It is formally denoted as $S^B(B)$, such that the set of elements defined by the following is created {A | A influences B}. For backward slices it is not uncommon for all preceding statements to be included in the slice [37, 43].

Three types of slice operations are defined. Each of these requires two different slices. Backbones are defined by an intersection of two slices and are useful in determining if there exist multiple values affected by a given defect. A dice is the difference between two slices and is useful for determining how the backward slice of some variable affects the backward slice of some other variable. This operation is most useful if the programmer knows that the program in question is mostly correct. A chop is the intersection between a forward slice and a backward slice. This operation is most beneficial for determining how some statement A, utilized to create an originating forward slice, influences another statement B, utilized to create an originating backward

slice. This operation is useful for determining paths of influence within a program [43].

Developing from static analysis is the idea of dynamic analysis. Dynamic analysis takes into account how the machine's state changes as the program in question runs. Taking a subset of time steps corresponds to taking a dynamic slice of the program; the slice is called dynamic because it deals with the code in its running state as opposed to executable sub-programs pulled from the original program in order to identify a defect. [42]

Static checkers like FindBugs and ITS4, discussed earlier, take a different approach. These tools include a library of defect patterns (FindBugs), unsafe function calls (ITS4), and other similarly well-defined pattern-based data that is later used in analyzing input programs. If the input program implements a pattern known to the static checker, the checker outputs information to the programmer that they have included a logical defect of one form or another.

My system utilizes both the analytical ideas of Weiser and adopts the analogical approach implicit in FindBugs, ITS4 and the "Have I or some other expert seen this before" behavior of many expert debuggers. Weiser's ideas about slicing are used to determine what debugging tactics are and what should be taught to the student. FindBugs and ITS4 utilize a set series of patterns that could be viewed as cases, thereby reinforcing the theory that the domain is amenable to the case-based reasoning methodology. Additionally the system takes further advantage of the case based nature of the problem by using the cases and the student's history with the cases to remediate programming defects. Specifically, the system is able to apply analogical reasoning by

determining a relationship between a current programming defect and a previously encountered programming defect. The system is also able to help the student apply analogical reasoning by storing the student's past successful encounters with each case and then allowing the student to review what they did during the last encounter to solve the issue.

4.5.2 How can the expert knowledge base be kept tractable? How can the system acquire domain knowledge?

The system in phase 1 contains a static set of cases that do not change over time. These static cases were drawn according to the error definitions defined around this work. Specifically, syntax errors are synonymous with compiler errors, runtime errors are considered synonymous with runtime exceptions, and the FindBugs static analysis tool defines the logical errors the system will attempt to handle. Because these are all definitions that have a finite number of associated errors, all compiler error rules, unchecked Java runtime exceptions that are part of the JRE, and all FindBugs patterns were added to the case base semi-automatically by reading in a set of property and configuration files for javac, FindBugs, and a file listing JRE exceptions.

However, this methodology breaks part of the CBR cycle. The CBR cycle has 5 stages: Retrieve, Reuse, Revise, Review, Retain. Phase 1 of the system (described in more detail later) implements Retrieve, Reuse, and Revise. Review and Retain were pushed to Phase 2 in order to get the main and most important aspects of the system up and running. Several different ideas were reviewed to determine the best way for this research with its current goals to implement the last 2 stages of the CBR cycle. A hard

coded, static case base is a good start but it is not all-inclusive – there are a potentially infinite number of defects, each with an infinite number of solutions. Therefore, the system employs a static case base for Phase 1 and then works to expand its own case base by analyzing student solutions in Phase 2.

In Phase 2, as described in the discussion of the design of that phase earlier, the system gains the ability to review student's solutions to an exercise, compare the students' solutions against the existing solution, and determine what new solutions to add to the case base.

4.5.3 Could the system (ITS-Debug) generate exercises dynamically?

One of the weaknesses of the phase 1 design of this system is that the exercise system is static. The concept of dynamically creating exercises to meet student educational needs has been explored, those works most closely related to this research include work done by Kumar [23] and Williams-King et al. [39].

Within the C++ debugging tutor discussed earlier, the researchers utilized templates to generate exercises dynamically. Specifically, the authors draw on the model the system is utilizing to perform its domain reasoning to generate exercises using a BNF-like grammar. Templates are created using this grammar by the researchers and then filled in at runtime with the appropriate details by the tutoring system [23].

Conversely, Williams-King et al. utilize aspect oriented programming [39] to create exercises on the fly. Their system was called Enbug and it was developed to help Computer Science professors at the University of Calgary create problems for a course

on computer viruses, malicious software, and creating robust code. Enbug is capable of causing controlled failures and introducing defects into defect free code through the utilization of concepts from aspect-oriented programming, Programming aspects are created with 'pointcuts' that specify to Enbug where in a given program to pause that program's execution. Then, advice modules dictate what to do when the pointcut has been reached. An example from the paper involves forcing a malloc to fail in a C program. The aspect consists of the following:

```
aspect mallocfail {
    pointcut main(): execution(* main(..));
    pointcut malloc(): execution(*malloc(..));

    advice before(): main() {
        set $n = 0
        set $N = 2
    }

    advice before(): malloc {
        if $n > $N
            @proceed(0)
        else
            set $n = $n + 1
        end
    }
}
```

The idea in this snippet is to cause malloc to fail after the nth call to the function. The variable $n keeps track of the number of calls into malloc. In this particular snippet malloc will be forced to fail on the third call to it via the @proceed(0) command [39].

This disseration's research takes an approach similar to that used by Kumar. Suggested solutions for cases are reversed, those instructions are then used to automatically damage a codebase of example code in a controlled manner. The damaged code is then presented to the student as an exercise.

4.5.4 Should the system support other languages? How would supporting other languages change the system?

When this research was proposed, the idea of supporting multiple languages was explored. Java was initially selected as the language to use first for several reasons: it is a well supported language of choice for introductory progression courses for majors; it is the language used at Lehigh University for their introductory progression; the structure and functionality of Java well support the learning objectives of the first progression computer science courses; and it is possible to integrate directly with the standard java compiler, javac, via the openjdk project and the Java runtime environment via reflection libraries built in to the JDK.

C and C++, although more difficult than Java, were favorites for novice courses before Java existed. Now there are other languages that did not exist when Java was released that are gaining popularity for teaching novices including Python, Ruby, Visual Basic (VB.NET) and C#.

Java was adopted as the language of choice for the scope of this dissertation for a few reasons, most notably because it is currently the language utilized for the AP Computer Science exam and the first language Lehigh University and many local High Schools teach their Computer Science students. However, the design of the system allows for future addition of other languages and environments with modest modifications. To support another language, a new Connector and Analyzer module would need to be written and a new set of cases needs generated, in ways analogous to how they were generated for Java. Additionally, the case base and supporting database

tables would likely require an additional column to indicate what language the case comes from. A more fundamental change would be to abstract the cases further than they are at present so that the case instead represents a universal defect instead of an environmental specific defect but these concerns are currently beyond the scope of this dissertation.

4.5.5 How should the system model student knowledge?

According to Weiser, programmers use slices when they are debugging [38]. The person solving the problem creates a mental slice of their application according to the slicing definitions described earlier. He determined this by studying groups of novices and experts; he additionally found that the main difference between the novices and experts he studied was not the approach taken to solving a defect but the knowledge base and ability to apply programming knowledge to the debugging problem that was different.

While Weiser studied both experts and novices and all of Weiser's participants had previous programming knowledge of the Lisp programming language [38], Chmiel and Loui focused on students taking an Electrical Engineering course where they were learning Assembly for the first time. Discussed in further detail earlier, a major contribution of this work was a qualitative rubric for differentiating different levels of debugging skill. Metrics in this rubric include: repetition of defects; planning for programming and debugging; time spent on debugging; demonstrated knowledge of debugging techniques; reliance in others for assistance; and the ability to utilize previous knowledge to solve a new problem [9].

McCartney et al. utilized a different and more qualitative tactic [26]. His work recorded interview answers from 14 students in total from the United Kingdom, Sweden, and the United States. The result of this study was a list of 35 different strategies that the authors distilled into 11 broader categories of strategies. Those categories include: learning from other people; learning from tools or written materials; obtaining and following step-by-step instructions on how to solve a given problem; gaining experience; visualization; learning from examples; tracing code; dividing the problem into manageable sub problems; relating the problem to something in the real world; looking at the problem from a higher level; and transferring existing knowledge to the problem at hand [26].

In order to model the student according to Chmiel and Loui or McCartney et al.'s approaches, sophisticated reasoning and computations would be required. Also some of these metrics would likely require some degree of pattern recognition. The Student model designed for this dissertation's research employs a limited implementation of Chmiel and Loui's approach and utilizes data from Weiser's work as part of the Pedagogical Module in determining what to teach to the student. The student model for this work consists of which topics, subtopics, exercises and cases the student has encountered and what the results of those encounters were. Additionally, the student model includes what the student's learning style is in order to provide individualized and meaningful feedback.

4.5.6 How should the system reason about student solutions without incurring the full program verification problem?

Many programming tutors take a model-based or a constraint-based approach, including the work done by Anderson et al. [5,6] and Mayo and Mitrovic [27]. Model-based approaches generally utilize a model of the perfect solution, containing branches that represent possible faulty paths to incorrect solutions. This approach allows for encoding a large amount of details and reasoning into the system but requires a great deal of time to produce. Additionally it is nearly impossible to foresee every possible wrong turn a student may take or even every possible right turn. Also such systems utilize a great deal of computation time to determine the precise branch the student has taken.

A production level system that employs a model-based approach appears to be Arnow's CodeLab system (turingscraft.com) that comes packaged with several introductory textbooks. If the demonstration exercises given on their site are indicative of how the system works, their approach has what appears to be a single correct answer for each exercise. Users receive assistance pointing out exactly where their code differs from the known solution, which remains hidden until the student answers correctly according to the feedback given. The feedback this system provided when used as a demonstration copy would seem to point to a model-based approach where there is a limited set of known solutions saved in the backend of the system. Deviations from this known solution set are considered incorrect and remediated according to the model within the system.

A different approach is constraint-based reasoning. Systems utilizing this approach have a list of constraints for a given situation that are encoded such that it is easy to reason about them. When all constraints for a given situation are met, the answer is correct. Otherwise, the answer is incorrect. Quicker than model-based reasoning, this approach is favored by Mayo and Mitrovic [27] and has been used successfully in tutors for SQL programming, CAPIT, and other tutoring systems.

The approach adopted for this work could be seen as favoring constraint-based systems where exercises have one constraint—they produce the correct output. Because there are an almost unlimited number of solutions to any given programming problem except for some of the very simplest cases, it was felt that this approach allowed for student creativity and would be able to handle situations that might not be anticipated by the researcher before performing formal testing. This approach also allows for the system to learn from the student. Just because the student is a novice does not mean that they are unable to come up with valid and novel solutions to programming problems. Therefore, as described in more detail in an earlier section, the system collects students' solutions in Phase 2 and analyzes them for novelty. If a solution is novel, the system will save the new solution as a new case with new solution steps.

4.5.7 How should debugging issues be remediated?

This work believes that students would benefit from receiving timely, focused, and meaningful feedback on errors when they occur. This belief stems from field research that was performed during the 2011-2012 academic year. To gain a better understanding of novice difficulties and how to solve them, the researcher tutored

undergraduates and High School students taking their first programming course or their first Java course (in the case of the High School students who were taking AP CS). Additionally, the researcher helped answer student questions in their laboratory periods and graded assignments for the Lehigh University CSE 15 course.

This fieldwork shed light into issues it is easy to forget novices have when one is no longer a novice oneself. Students had many questions about syntax, error messages, and how to produce the output requested from their instructor. Additionally, questions arose about how to understand runtime exceptions. Another interesting fact is that the novice does not always understand that it is not enough to just get their code to compile and run, that they must also verify their code produces the correct output.

Many students requested assistance from the researcher during this fieldwork, with a subset of the students coming back often for further assistance. It became clear from these students that were seen multiple times that the same approach does not work with every student. Certain students learned best just being talked through the problem; some students just needed to be pointed towards the right text or example; and other students did best with physical examples. This difference in learning styles highlighted the need for a tutoring system not only to provide assistance but to try to provide that assistance in the format that would best help the students using it. The system has therefore been built to handle multiple modalities, further details of which can be found in the discussion of the design and implementation of this work.

The need for incorporating multiple learning styles into the classroom and in Intelligent Tutoring Systems has been highly debated. Some work cites no significant

difference when multiple modalities were utilized among groups of students. Other work finds the opposite to be true. Significant work in Intelligent Tutoring Systems to provide multi-modal support was done by S. Parvez and G. Blank at Lehigh University. Their system was developed to teach introductory students according to the design first mentality of computer science instruction, where students are taught proper design tactics and UML before any significant coding. Parvez did her dissertation work on how to create a Pedagogical Module for an Intelligent Tutoring System that could support different learning styles, eventually deciding to employ the Felder-Silverman model after researching several other models of student learning [32].

The Felder-Silverman model, developed by Richard Felder and Linda Silverman at North Carolina State University, aimed at improving engineering education. The model implements the belief that optimal learning can take place when information delivery is aligned with the manner in which the student best processes information [32].

The Pedagogical Module of the DesignFirst-ITS was designed to address multiple aspects of the Felder-Silverman model. The module was capable of creating 7 different types of feedback according to the Felder-Silverman model. These included: definitions, examples, questions, scaffold (or, instructing the user to use a certain tutorial), pictures, relationships for concepts, and applications of a concept [32].

The system was evaluated with one level of feedback in two introductory level courses at Lehigh. Students were asked to fill out the Index of Learning Style questionnaire to determine which dimension of the Felder-Silverman model represented them. Then students were given instructions on how to use the system and proceeded to

use the ITS to create a design. Students were all asked to create a specific design in this system for a movie ticket vending machine. All students participating were able to complete the exercise. Most of the students in this class were identified as visual learners and received visual feedback. The remaining students received text-based feedback [28].

The effectiveness of tailoring feedback to learning style was later evaluated with High School students from High Schools participating in the LVSTEM and Launch-it programs. Participants were divided into 3 groups. Group 1 received no feedback, group 2 received strictly textual feedback, and group 3 were given the appropriate feedback for their identified learning style. The evaluation also utilized a pretest and a posttest [32].

The deviation of group 1's pretest and posttest scores was statistically insignificant when using the paired t-test; which was as expected. Group 2's results were also statistically insignificant; the variance between pre and posttest scores was not large enough to indicate learning took place. Analysis of group 3's results showed a statistically significant deviation between pre and posttest scores and therefore implies that learning did take place within this group of students due to their use of the tutoring system. These findings support work in tailoring tutoring system feedback to the learner according to learning style [32].

4.5.8 How could the system determine if a given remediation is successful?

Previously, this question was only relevant in terms of dynamically determining which learning style best suits a student. Dynamically determining learning style was

dropped as discussed with the members of the dissertation committee earlier this year due to the potential that this part of the system could introduce more variability and confound the statistical results of evaluations.

4.5.9 How to communicate remediations?

How to communicate remediations is non-trivial and much work in ITSs has sought to answer this question for different domains and situations. An ITS may provide different forms of feedback for different types of learners [32], utilize personae to discuss the domain with the student in a more human-like manner [40], interrupt the student strategically for guidance, or wait for a student to request assistance.

While tutoring students taking a college level CS1 course, the researcher reached an important conclusion regarding this question. Interrupting the student at the wrong point led either to confusion or the student not truly learning from their interaction with defective code. It is difficult to determine an appropriate time to interrupt the thought processes of the novice and there is no clear indicator as to what path the student's thoughts are taking without asking them to explicitly describe what they are thinking. Interrupting the student too soon can interrupt the student from reaching important conclusions on his or her own. Interrupting the student too late could cause the feedback to be misunderstood; if the student has moved on from issue x to issue y and a remediation is provided for issue x the assistance will be confusing and irrelevant.

The methodology therefore chosen offers remediations as part of the normal edit-compile-test loop, when students are already looking for error messages that are intended to guide programmers to solving problems. Of course, novices often don't

understand compiler error messages, let alone Java Exceptions. So it is at this point that novices are likely to appreciate assistance. The web-based interface provides one button for both compiling and running and provides remediation within the scope of this loop. When the student clicks this button, their code is sent back to the server for evaluation. In the event that there is an error in the student's code, the system analyzes the error and provides feedback that is relevant to the current situation. The remediation itself is inserted into the interface in the same panel that output appears in, in order to keep results in the area of the screen the student should be looking at after trying to run their code. It is believed that this methodology ties remediations in nicely with error messages and erroneous output and keeps assistance relevant and in context.

4.5.10 How to discuss the domain with the student?

The domains of programming and debugging are difficult for novices to comprehend. Standard environments, built for expert programmers, do not take this into account in either their layout or the language used to describe errors. Several systems including BlueJ, Backstop, and Intelligent Tutors for teaching programming strive to demystify one or both domains for the novice.

ITS-Debug is meant to be a tool for the student who does not yet have a good grasp of debugging. Because of this the system must speak plainly and clearly in its remediations and avoid jargon. Explanations are in plain English, and the detail of remediations increases as the system's confidence in the student's ability to resolve the error decreases. Additionally, the system attempts to discuss the domain in the manner that makes most sense to the effected learner.

5 Chapter 5: Evaluation

Testing of this system has been performed from Fall 2012 through the end of the Fall 2013 semester. Students self selected into the study from the populations of Lehigh University's CSE 002, Phillipsburg High School's AP and intermediate Computer Science Courses, Cranford High School's AP computer Science course, and Warren Hills' AP and Introductory Computer Science courses. Students in these courses were generally novice students in their first or second Java course, depending on the individual student's background.

Evaluation of the system occurred both actively and passively. Active evaluation was performed using two survey instruments (included as appendices) and the pretest-practice-posttest methodology, where students were asked to complete a pretest, use the system for practice, and then complete a posttest. Passive evaluation occurred automatically. Certain metrics that the system must monitor in order to provide appropriate exercises and assistance to the student were logged over time. These include: exercises encountered, attempts per exercise, topics learned, and time required to complete an exercise.

During the course of the Spring 2013 semester the system was presented to the students at Lehigh University three times. For the first two of these encounters the system was presented in a closed, supervised session where the system was presented to the students as a way to practice for an impending exam. The last session allowed students to use the system independently on their own time.

5.1 Spring 2013 – Experimental Setup

The system was evaluated over the Spring and Fall semesters of 2013. Students from Lehigh University were presented with Phase 1 of the system during the Spring 2013 semester. Two separate supervised sessions were held as study sessions for upcoming exams. These sessions each had the students practice with two separate programming topics, as the students' upcoming exams dealt with two separate programming topics. For the first session, students were asked to use the system to practice debugging programs that incorporated Looping structures and Methods. For the second session, students were asked to use the system to practice debugging programs that incorporated Single and Multidimensional Arrays.

All students in these sessions were asked to use an up-to-date version of the Firefox web browser to access the tutoring system at its URL. Once there, students were instructed to log in with usernames and passwords that were distributed on receipt of the informed consent form. The system then presented the students with a multiple-choice pretest consisting of four small broken programs and three more qualitative questions about debugging. After submitting the pretest, students were asked to select their learning style from two choices: visual and verbal (described in more detail in Chapter 4). Then, after submitting their learning style, the students received the tutoring system. Students were asked at this point to select the first topic for the session from a drop down menu on the user interface and practice with that topic for 20 minutes. After 20 minutes had passed, students were asked to switch to the second topic for the session. After the second 20 minute period, students were asked to click on the 'Go To

PostTest' button on the User Interface in order to proceed to the posttest and exit survey. The posttest followed the same format as the pretest, the exit survey asked the students questions about their experience with the system and is included in the appendix section of this document.

The first two supervised sessions combined consisted of 16 students. Because of the somewhat low number of students who were able to attend the supervised sessions, the system was disseminated at the end of the semester to any student who wished to try the system on their own time. These students used the system to practice exercises from all topics available in the system up to Multidimensional arrays. Students who used the system independently all took the pretest but many failed to go on to the posttest and exit survey.

5.2 Fall 2013 – Experimental Set up

The second formal evaluation trial of this system occurred during the Fall 2013 semester at two separate High Schools. All participating students at the first High School were taking Advanced Placement Computer Science. Almost all participating students at the second High School were taking an introductory level computer science class where they were learning Java. The same evaluation methodology as before was performed with the addition of a pre-survey before the pretest in order to determine if any students participating in the study were not true novices.

5.3 Results and Discussion

The following discusses the results of the experiments performed over the

course of the evaluation period. In discussion of the attempts required to complete an exercise, each data point on the scatter charts represents the mean number of attempts one student required to complete the exercises they attempted (unless otherwise stated). For instance, if student x completed 7 exercises with an average of 10 attempts, they are represented in the scatter chart with a circle at position coordinate (7,10). In discussion of the time required to complete an exercise, each point on the scatter charts represents the mean time for an individual student to complete an exercise.

5.3.1 Pretest vs. Posttest

The pretest and posttest results for high school and college students were analyzed separately. College students were evaluated as three groups: two monitored formal sessions and a final session where students were able to use the system on their own. The first set of college students using the system exhibited an increase in the means between the pretest and the posttest but the increase was not statistically significant. The second set of college student evaluation suffered from receiving a total of four students and some unexpected evaluation time issues with the system. Additionally, the third set of college student data for this dimension of all the data collected suffered inconsistencies due to the fact that the sessions were unsupervised. Because of the unsupervised nature of this aspect of the data, students participating did not follow instructions – namely, completion of the posttest. For this subgroup, in terms of the pretest and posttest, the data is flawed in the following ways: too short a practice time, selection of disparate topics unrelated to the pre and posttest, and/or

allowing too much time to pass between the completion of the pre and post test (i.e. more than 6 hours).

During the second semester of the formal evaluation the pretest and posttest each gained a question and focused on a single topic instead of the previous design which included two separate topics. Students' answers were tabulated over the pretest and posttest and the number of correct answers for each test were analyzed using the Paired Samples T-Test in SPSS. The results of this analysis, included below, yielded statistical significance for at the $p<.01$ level.

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	pretest	1.74	103	1.590	.157
	posttest	2.57	103	1.600	.158

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	pretest & posttest	103	.664	.000

Table 1: Paired Samples T-Test Supporting Statistics

Paired Samples Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1	pretest - posttest	-.835	1.307	.129	-1.090	-.580	-6	102	.000000003231316473

Table 2: Paired Samples T-Test Results, High School only

5.3.2 Attempts Data – all students

The attempts required for completed exercises for each participating student

for each exercise are represented by the following scatterplot and table. Specifically, attempts are defined in this work as the number of times the student has submitted their code back to the tutoring system for analysis. This includes the final correct submission for exercises the student has successfully completed. Attempts data was analyzed using the Pearson correlation in SPSS. The hypothesis for this aspect of the evaluation was that there would be a negative correlation between the attempts students required to complete an exercise given more practice with the system. When using a two tailed Pearson correlation, the combined results of all participating students' attempts data was found to be statistically significant at the $p<.01$ level.

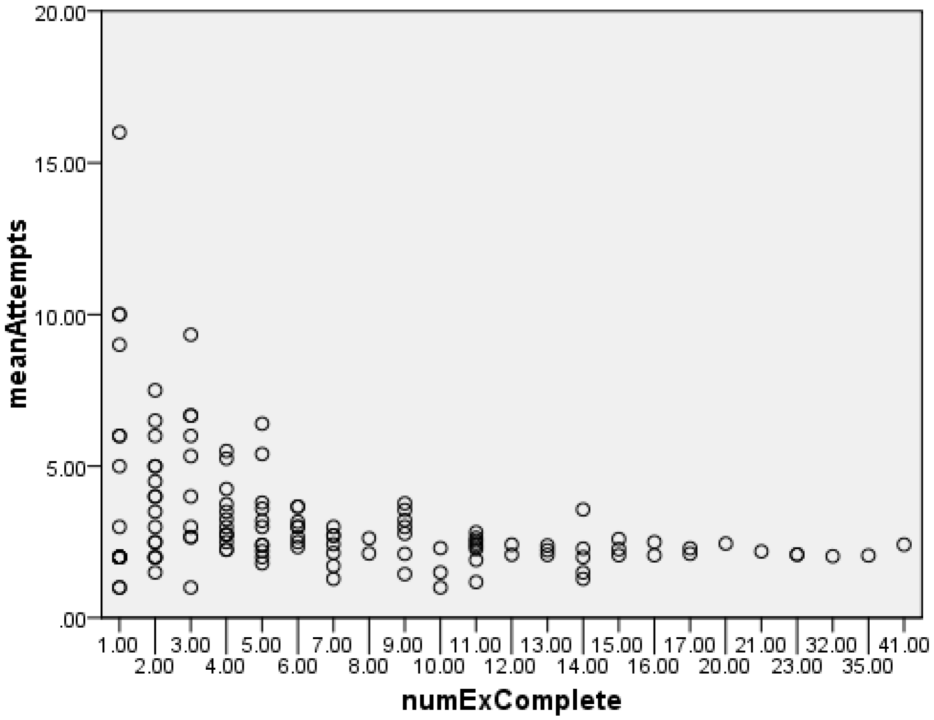


Figure 18: Scatter Plot of all students' attempts data for completed exercises

Correlations

		numExComplete	meanAttempts
numExComplete	Pearson Correlation	1	-.325**
	Sig. (2-tailed)		.000
	N	127	127
meanAttempts	Pearson Correlation	-.325**	1
	Sig. (2-tailed)	.0001941100	
	N	127	127

**. Correlation is significant at the 0.01 level (2-tailed).

Table 3: Pearson correlation for all students' attempts data

The above scatterplot and statistical data show that the students required significantly fewer attempts to complete an exercise with more practice with ITS-Debug..

Another analysis of attempts data was performed over all attempts data, regardless of whether or not the student completed the exercise. The following table and graph show the results of another two-tailed Pearson Correlation over the extended data.

Correlations

		numExAttempted	meanAttempts
numExAttempted	Pearson Correlation	1	-.412**
	Sig. (2-tailed)		.00000011225
	N	154	154
meanAttempts	Pearson Correlation	-.412**	1
	Sig. (2-tailed)	.000000112252	
	N	154	154

**. Correlation is significant at the 0.01 level (2-tailed).

Table 4 Pearson Correlation – All Attempts Data, regardless complete

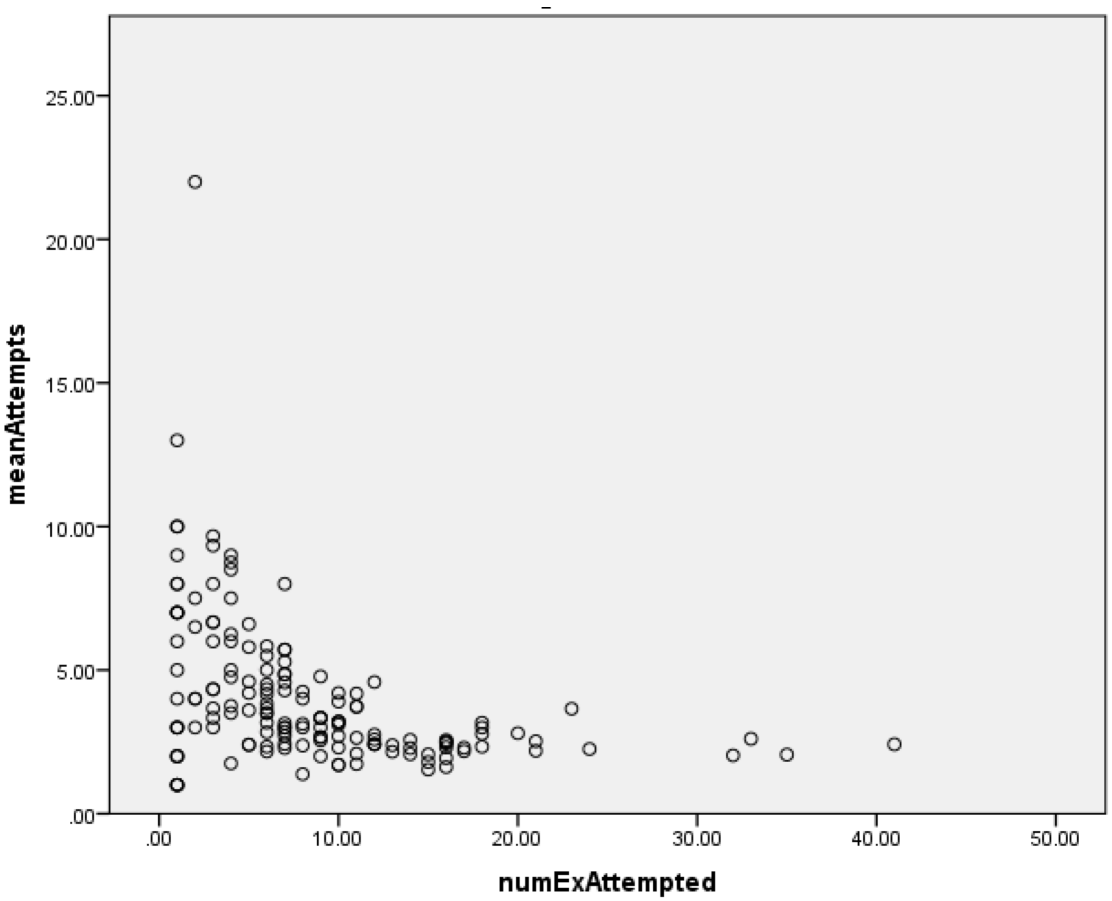


Figure 19: Scatter Plot– All Attempts Data

5.3.3 Attempts data – High School students vs college students

After analyzing the data for all students combined, the data was re-analyzed for each level of student who used the system (i.e. high school vs. college). This data was re-analyzed using the Pearson Correlation for each group; relevant tables and graphs appear below. When the college students' attempts data was analyzed alone, statistical significance was not found in this group. However when the high school students were analyzed separately, statistical significance was again found. It is believed that this difference in results can be explained by the limited number of college students that participated, the fact that the experiment changed between the two groups

to concentrate on practice with a single topic instead of two topics, and the fact that the high school students were closer to the true novice level than the college students. Additionally, most of the high school students who used the system were closer to the "true novice" level the system aims to support and therefore, stood to learn the most from using the system.

Correlations

		numExComplete	meanAttempts
numExComplete	Pearson Correlation	1	-.339**
	Sig. (2-tailed)		.000
	N	103	103
meanAttempts	Pearson Correlation	-.339**	1
	Sig. (2-tailed)	.000462144	
	N	103	103

**. Correlation is significant at the 0.01 level (2-tailed).

Table 5: Pearson Correlation over high school data, completed exercises only

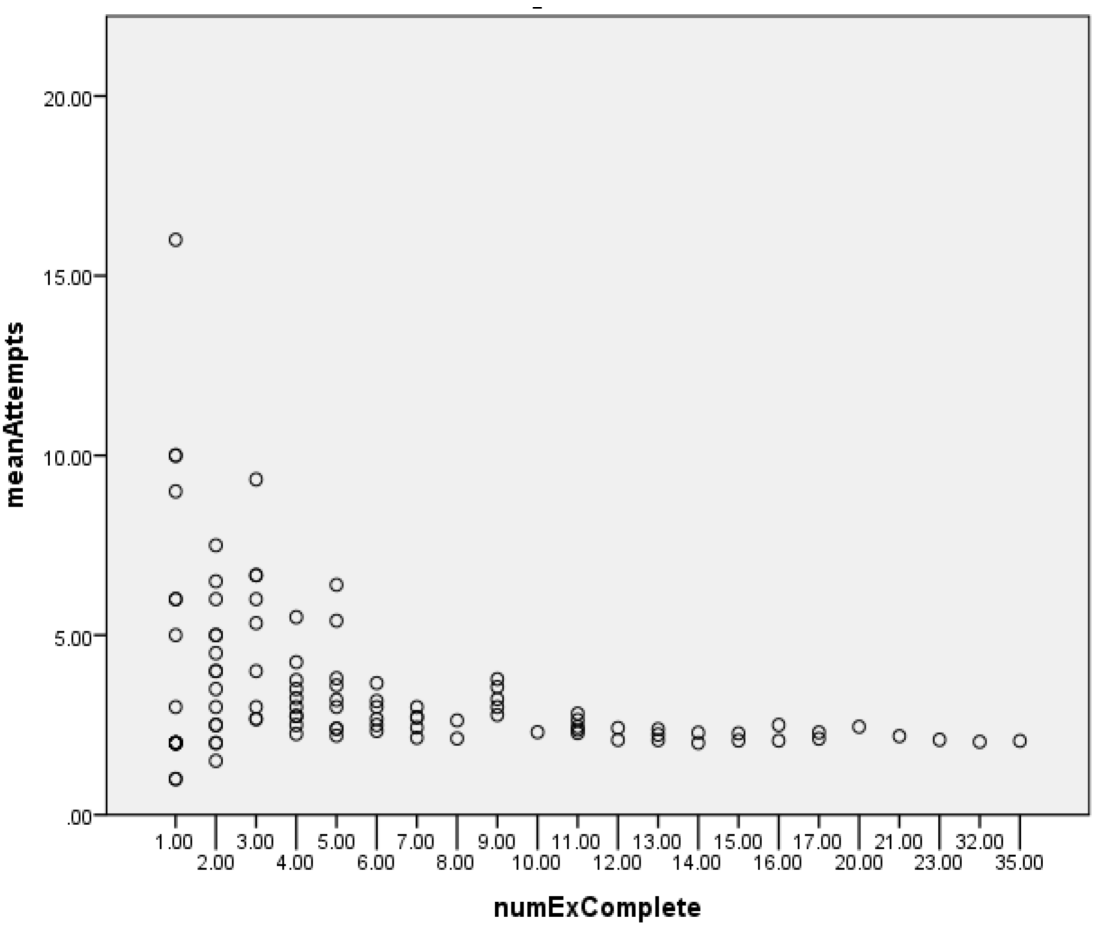


Figure 20 Scatter Plot over all High School data, completed exercises only

Correlations

		numExComplete	meanAttempts
numExComplete	Pearson Correlation	1	-.041
	Sig. (2-tailed)		.851
	N	24	24
meanAttempts	Pearson Correlation	-.041	1
	Sig. (2-tailed)	.851	
	N	24	24

Table 6: Pearson Correlation over college data, completed exercises only

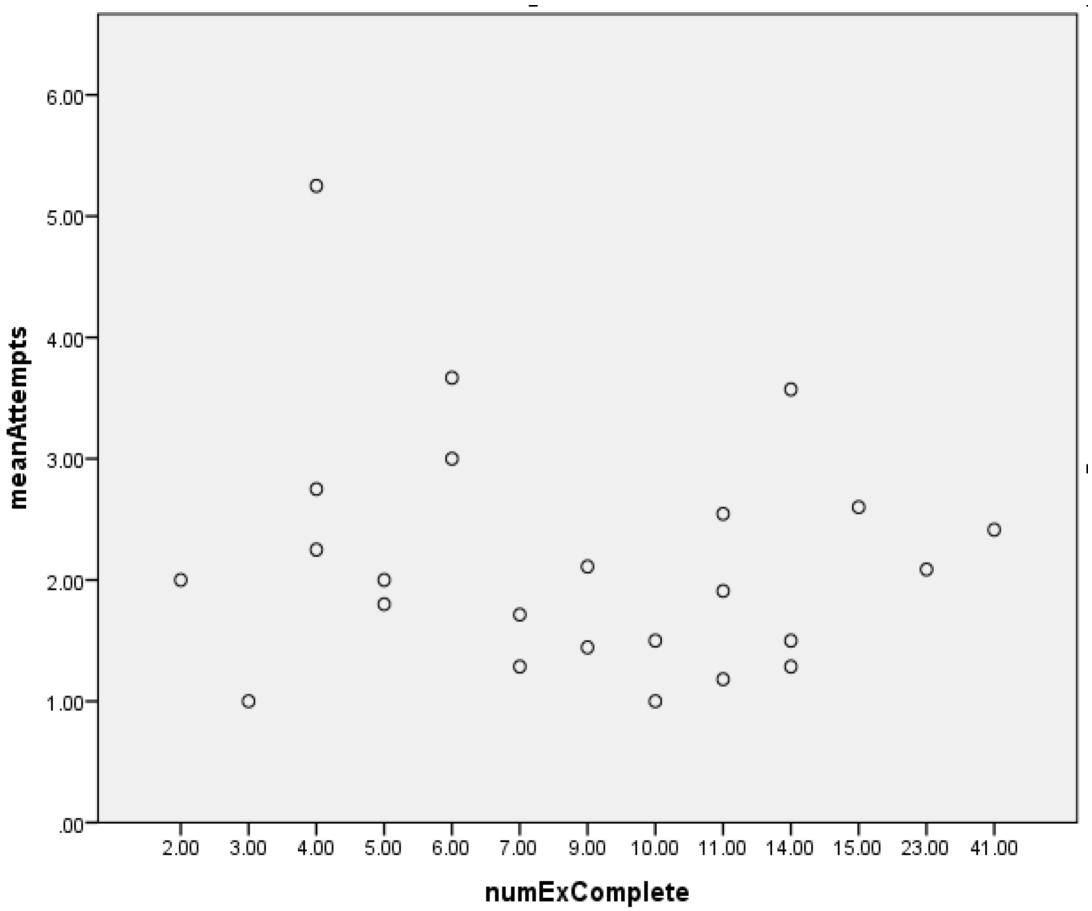


Figure 21: Scatter Plot of college attempts data, completed exercises only

5.3.4 Attempts Data By Modality

The attempts data was furthered analyzed to compare the effectiveness of the two modalities offered. As before, the data was analyzed using the Pearson Correlation and then visualized with a Scatter Plot. When all students were combined, significance at the $p<.01$ level for verbal students. Significance at the $p<.05$ level was observed over all combined students for those who chose to receive visual assistance. High school students' attempts data alone exhibited for both modalities exhibited the same level of significance, at the $p<.05$ level, indicating no significant difference between the two

modalities. When college student data was isolated, no significance was found within either groups' data.

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	-.415**
	Sig. (2-tailed)		.003
	N	49	49
meanAttempts	Pearson Correlation	-.415**	1
	Sig. (2-tailed)	.003	
	N	49	49

**. Correlation is significant at the 0.01 level (2-tailed).

Table 7: Pearson Correlation: Attempts – Everyone –Verbal

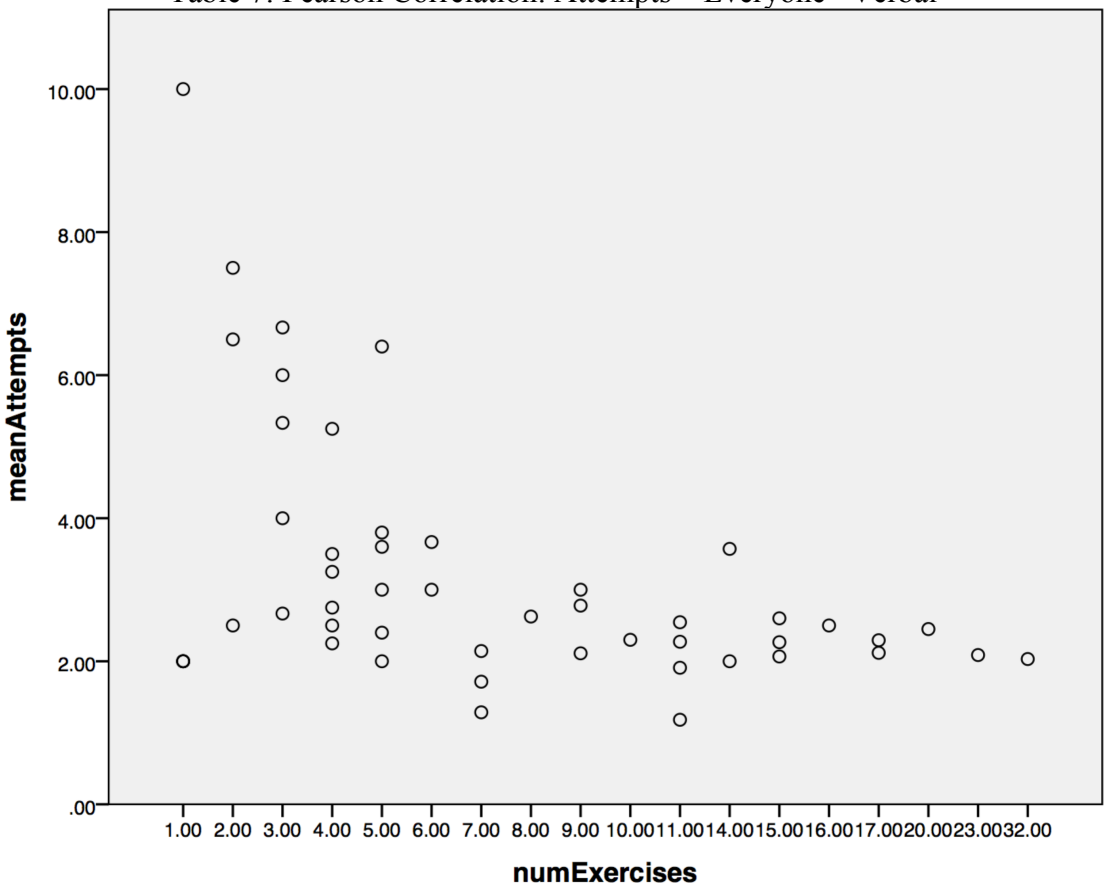


Figure 22: Scatter Plot: Attempts – Everyone – Verbal

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	-.284*
	Sig. (2-tailed)		.016
	N	72	72
meanAttempts	Pearson Correlation	-.284*	1
	Sig. (2-tailed)	.016	
	N	72	72

*. Correlation is significant at the 0.05 level (2-tailed).

Table 8: Pearson Correlation: Attempts – Everyone – Visual

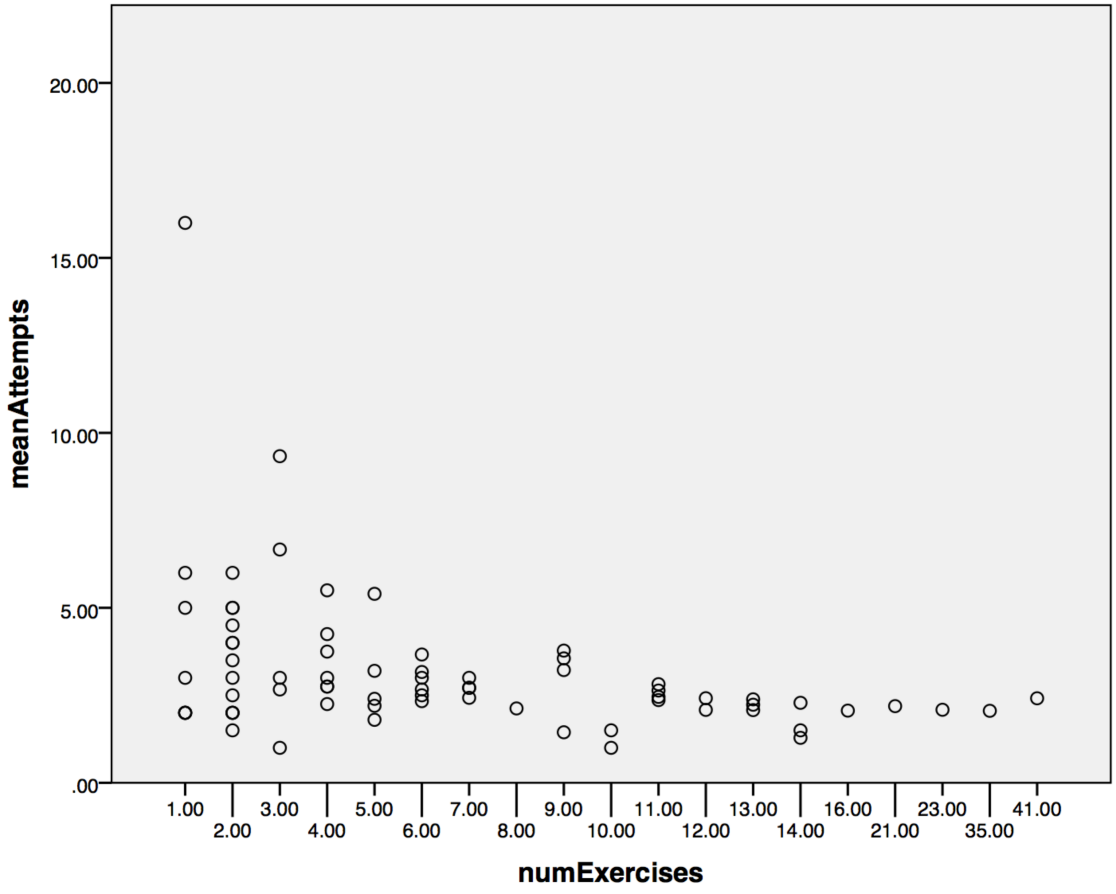


Figure 23: Scatter Plot: Attempts – Everyone - Visual

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	-.416*
	Sig. (2-tailed)		.010
	N	37	37
meanAttempts	Pearson Correlation	-.416*	1
	Sig. (2-tailed)	.010	
	N	37	37

*. Correlation is significant at the 0.05 level (2-tailed).

Table 9: Attempts - High School Verbal

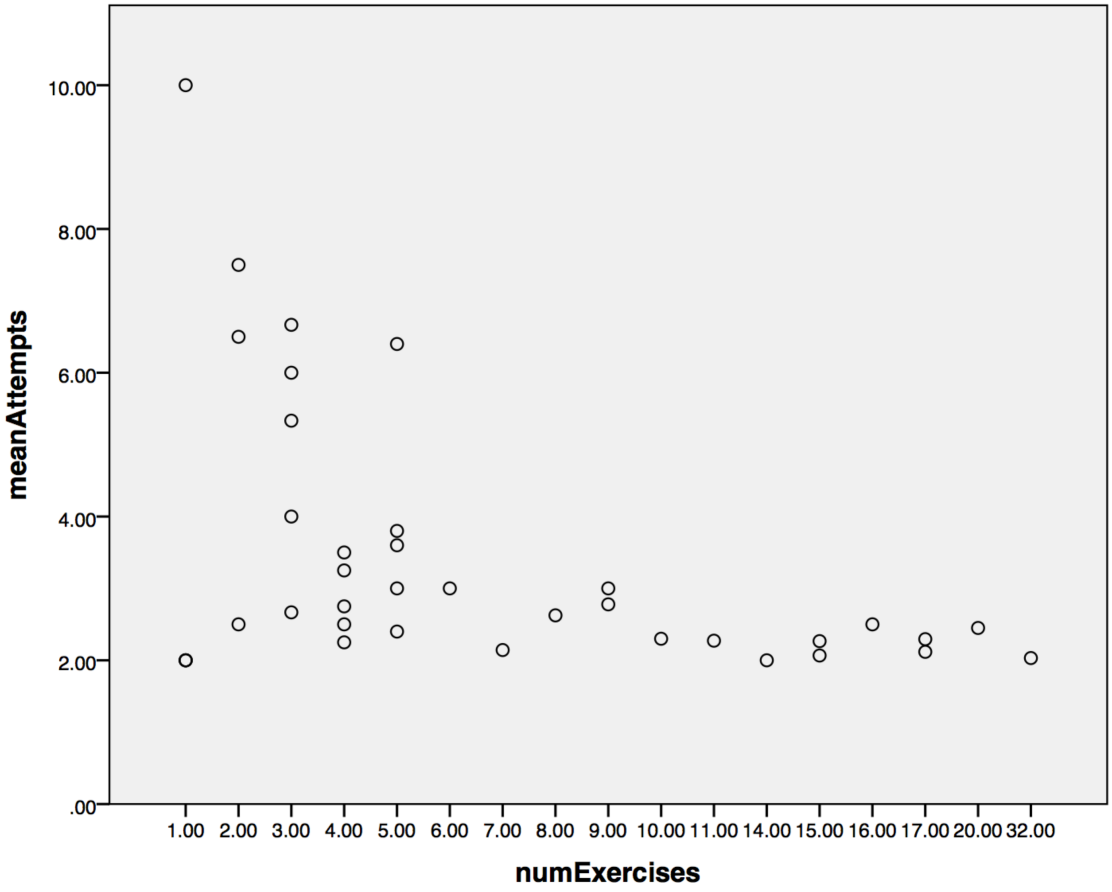


Figure 24: Scatter Plot– High School – Verbal

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	-.314*
	Sig. (2-tailed)		.015
	N	60	60
meanAttempts	Pearson Correlation	-.314*	1
	Sig. (2-tailed)	.015	
	N	60	60

*. Correlation is significant at the 0.05 level (2-tailed).

Table 10: Attempts – High School – Visual

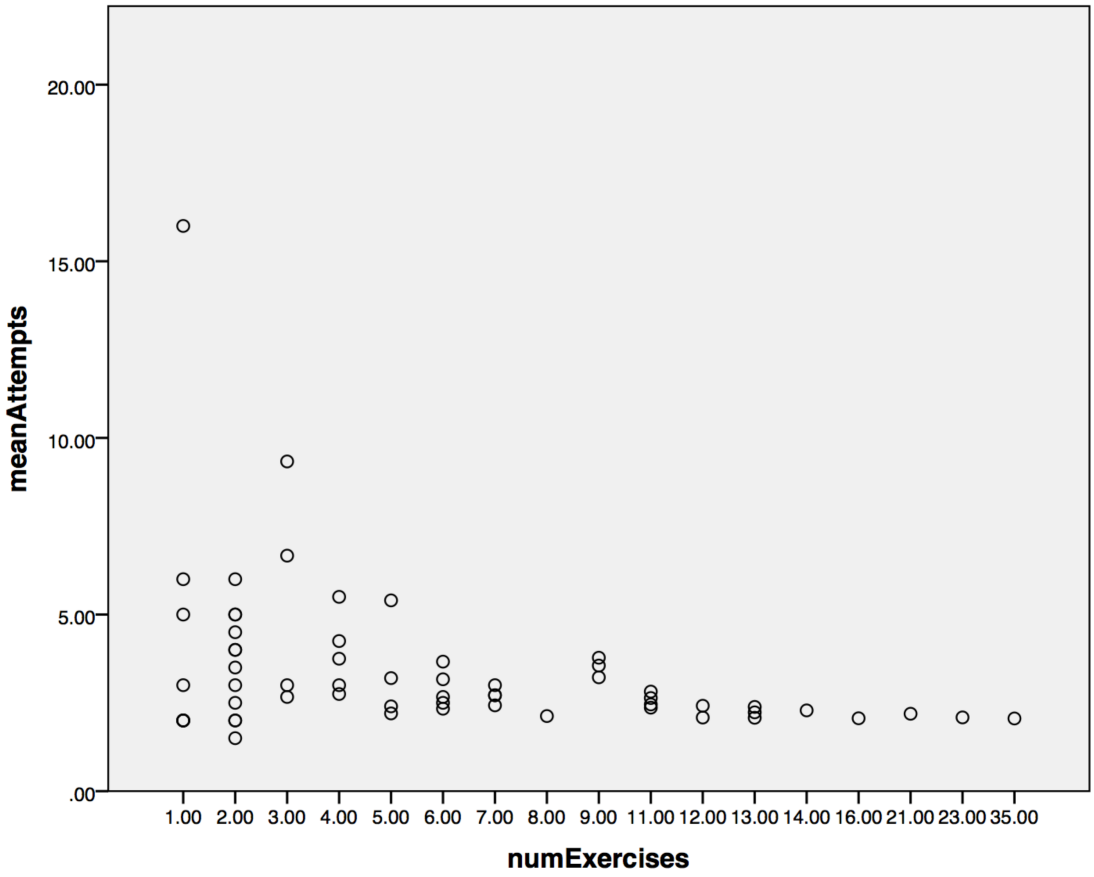


Figure 25: Scatter Plot– High School – Visual

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	-.217
	Sig. (2-tailed)		.497
	N	12	12
meanAttempts	Pearson Correlation	-.217	1
	Sig. (2-tailed)	.497	
	N	12	12

Table 11: Pearson Correlation – Attempts – College – Verbal

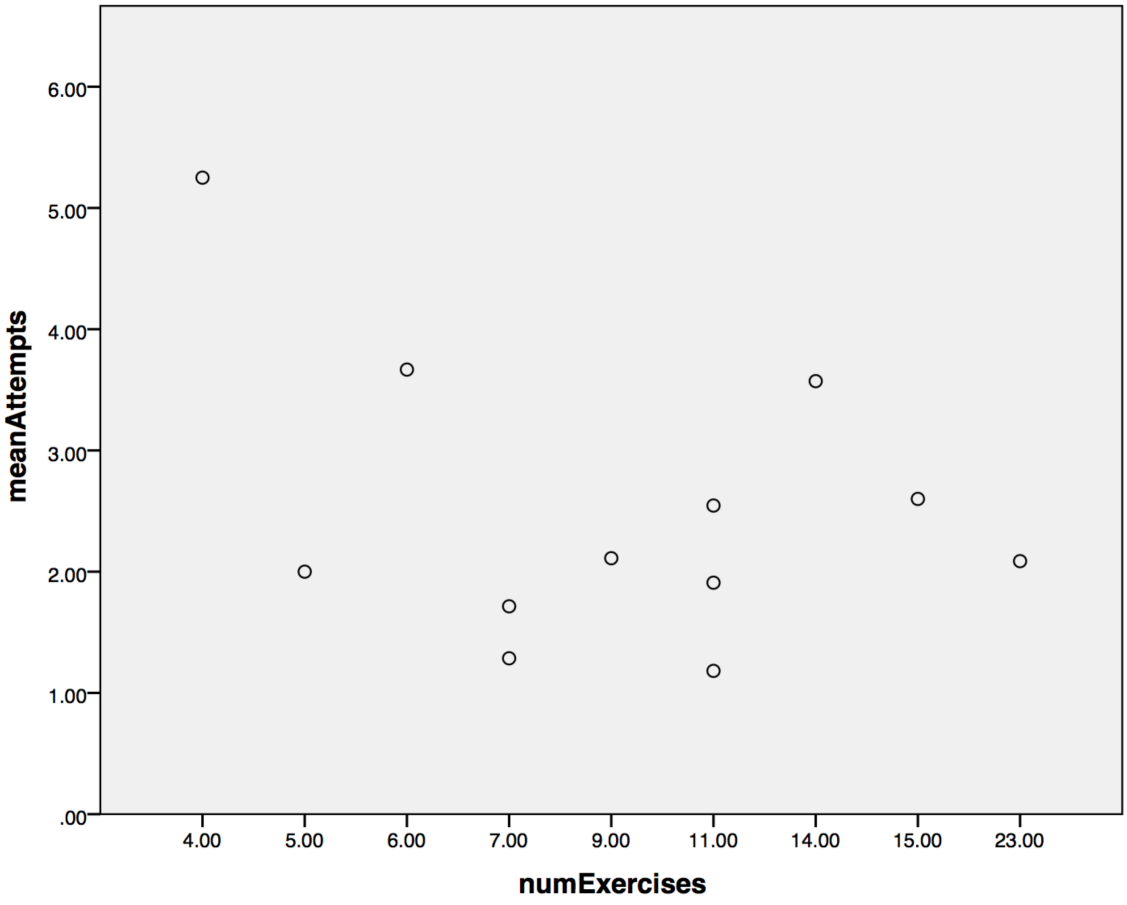


Figure 26: Scatter Plot– Attempts – College – Verbal

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	.090
	Sig. (2-tailed)		.781
	N	12	12
meanAttempts	Pearson Correlation	.090	1
	Sig. (2-tailed)	.781	
	N	12	12

Table 12: Pearson Correlation – Attempts – College – Visual

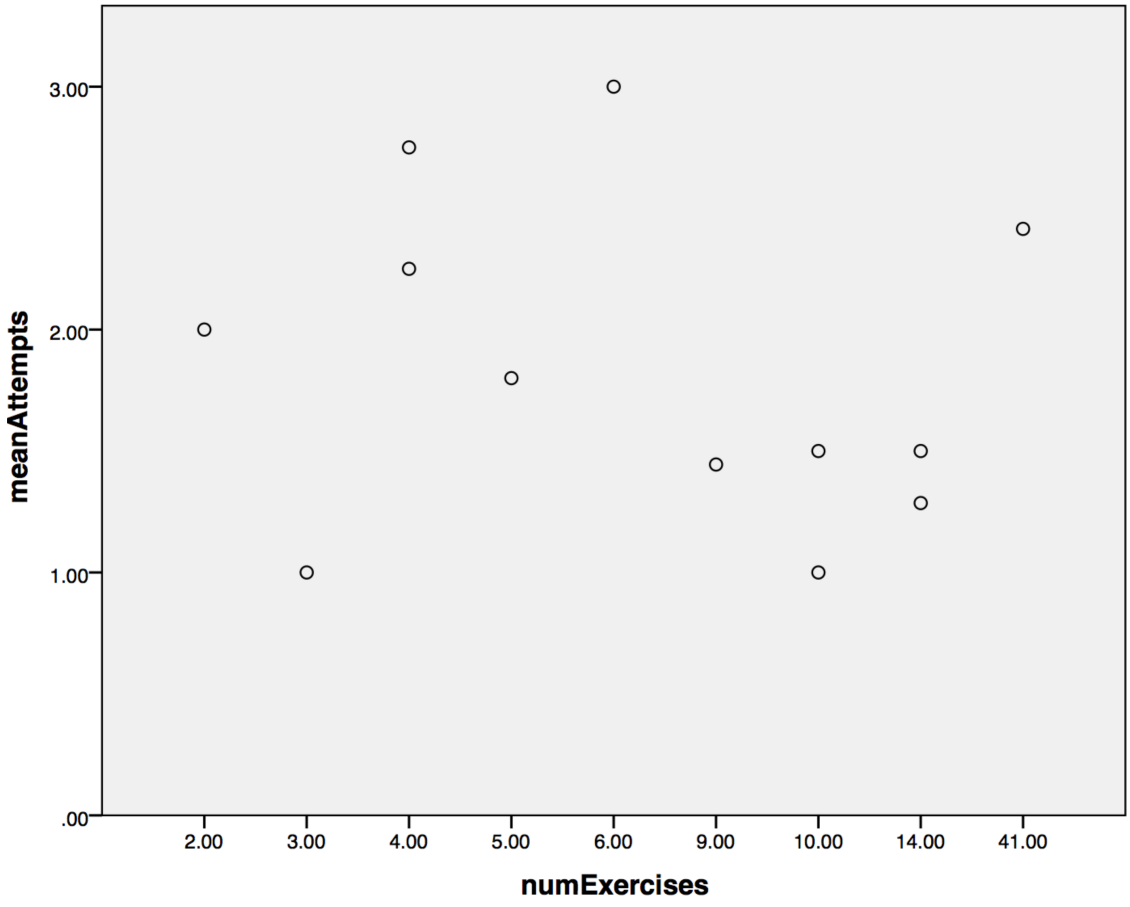


Figure 27: Scatter Plot– Attempts – College – Visual

5.3.5 Time to Complete Exercises

The time required for each student to complete an exercise was analyzed using the Pearson correlation to determine if students would require less time to complete an exercise given more practice with the system. The results of this analysis are displayed in the following scatterplot(s) and table(s). A statistically significant negative correlation between the number of exercises completed and the mean amount of time required to complete an exercise for each student was found at the $p<.01$ level for all students combined and for high school students in isolation. For college students in isolation statistical significance at the $p<.05$ level was exhibited. The original data exhibited some students with an average time to complete of less than a second, further review of the data collected shows that this was in fact in line with the collected data. Whether or not the collected data is skewed due to an error with the logging mechanism is unclear at this time. The data is included, excluding all data points exhibiting an average time to complete less than 2 seconds, regardless because even if the data is skewed it appears to be skewed in a regular fashion and may still be of some value to this work. No conclusions in this work are drawn solely from the timing data.

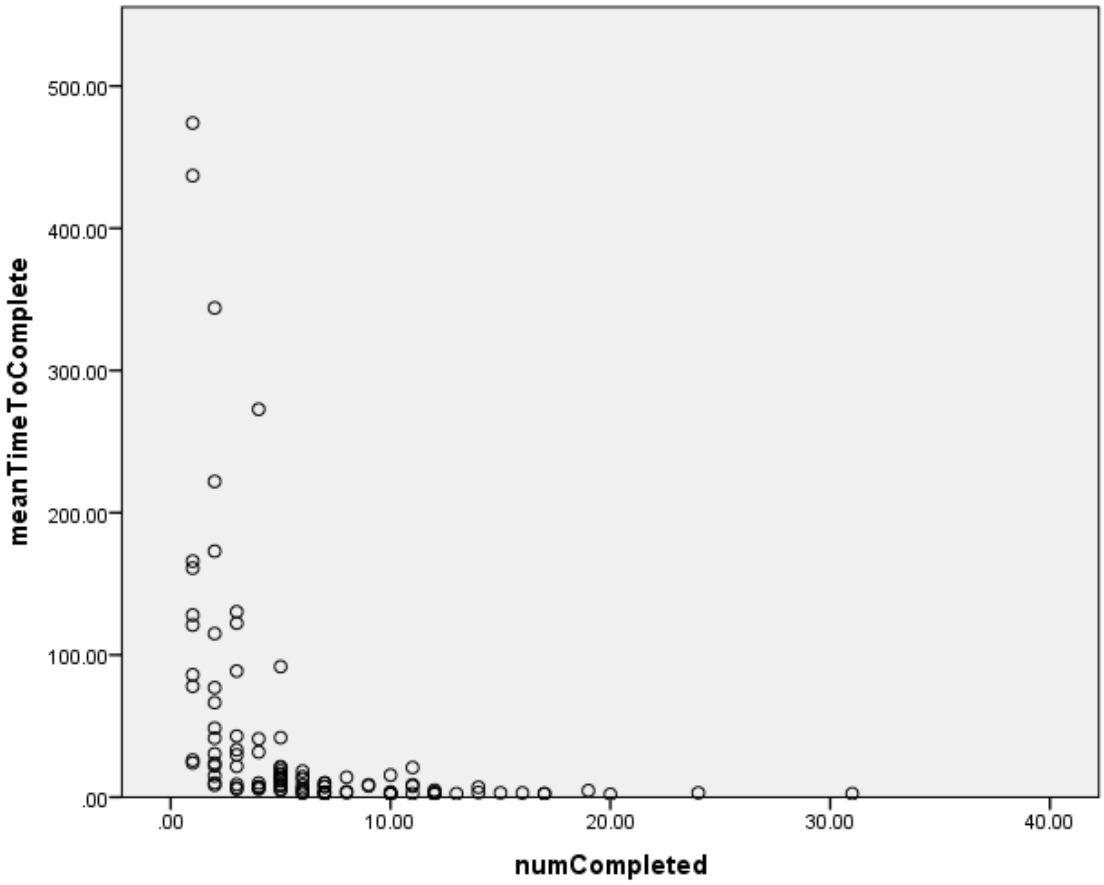


Figure 28: Scatter Plot of timing data, all students

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.386**
	Sig. (2-tailed)		.000
	N	103	103
meanTimeToComplete	Pearson Correlation	-.386**	1
	Sig. (2-tailed)	.000	
	N	103	103

**. Correlation is significant at the 0.01 level (2-tailed).

Table 13: Pearson Correlation of timing data, all students

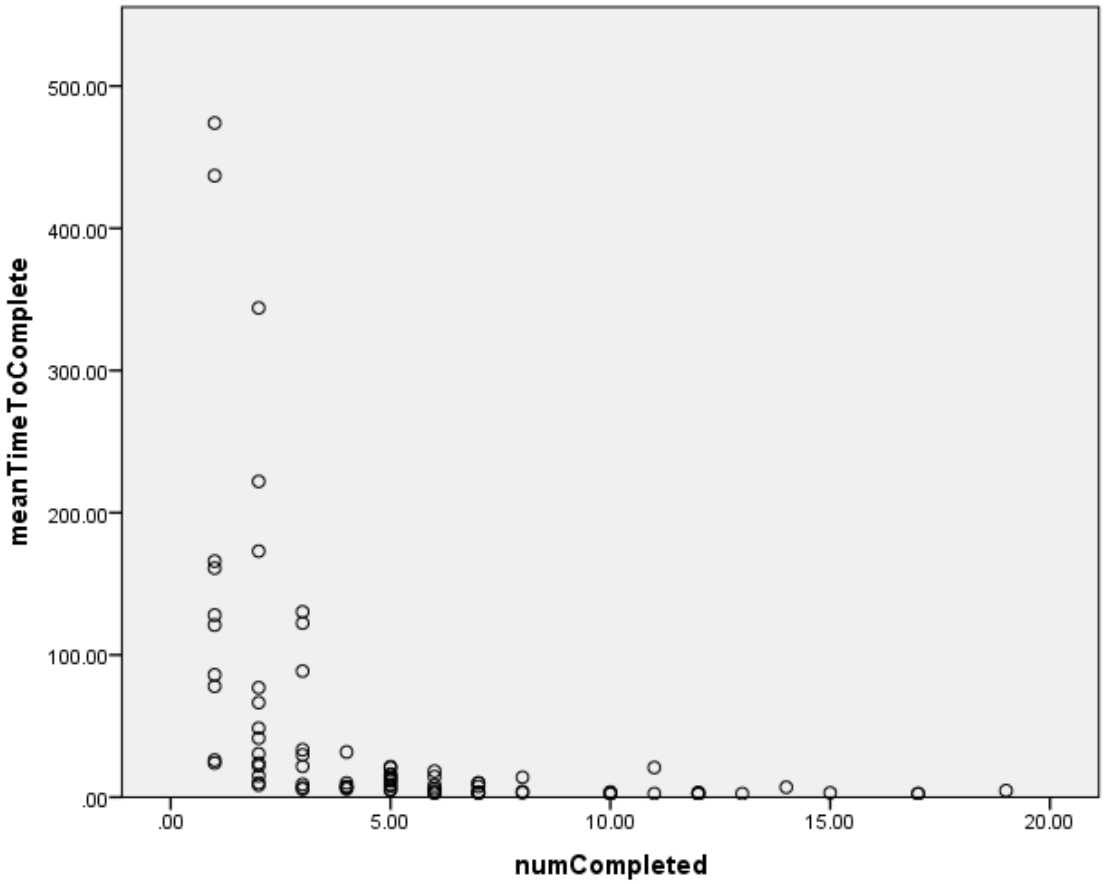


Figure 29: Scatter Plot for timing data, High School only

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.417**
	Sig. (2-tailed)		.000
	N	80	80
meanTimeToComplete	Pearson Correlation	-.417**	1
	Sig. (2-tailed)	.000	
	N	80	80

**. Correlation is significant at the 0.01 level (2-tailed).

Table 14: Pearson Correlation for timing data, High School only

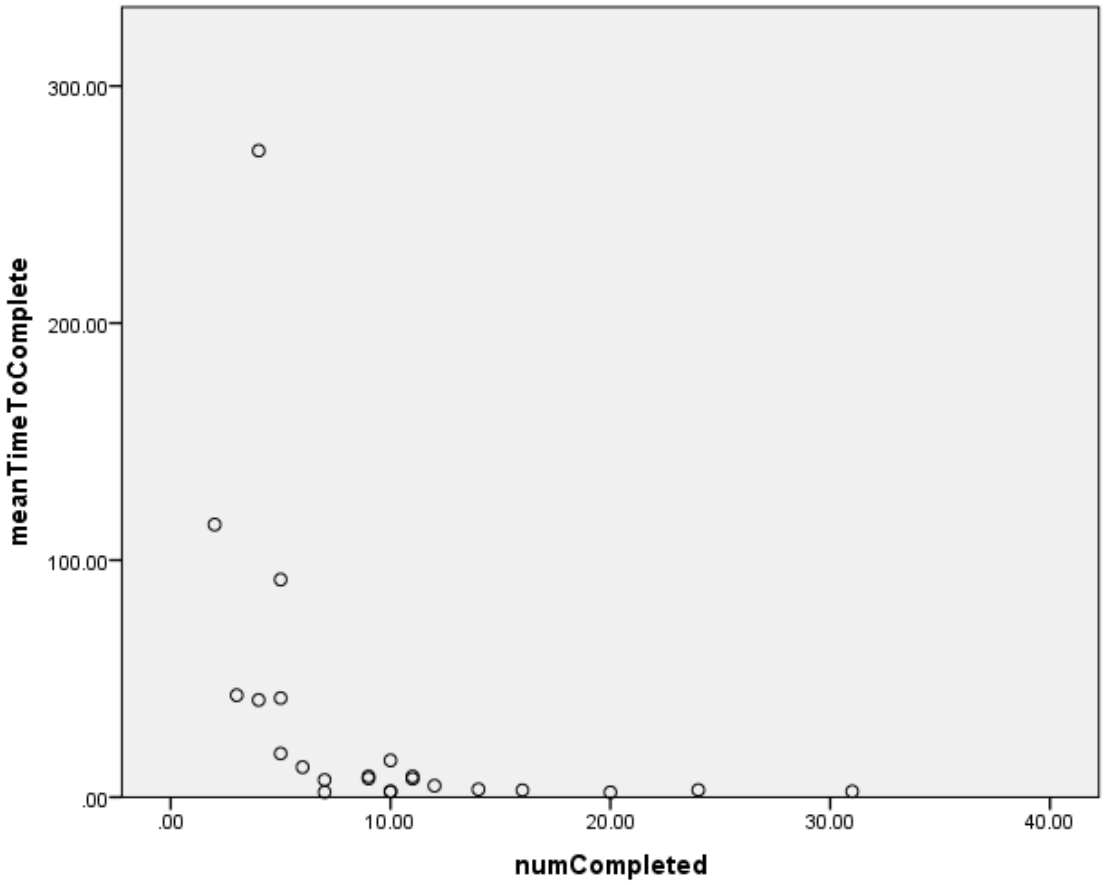


Figure 30: Scatter Plot for timing data, college students only

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.422*
	Sig. (2-tailed)		.045
	N	23	23
meanTimeToComplete	Pearson Correlation	-.422*	1
	Sig. (2-tailed)	.045	
	N	23	23

*. Correlation is significant at the 0.05 level (2-tailed).

Table 15: Pearson Correlation for timing data, college students only

5.3.6 Timing Data By Modality

The students' timing data was also analyzed according to the students' chosen modality. Results were once again correlated using the Pearson Correlation in SPSS and then visualized with a scatterplot. With all students combined the data exhibits significance at the $p<.05$ level for students who selected verbal as their modality and significance at the $p<.01$ level for students who selected visual. College students alone did not exhibit statistical significance for either modality. High school students exhibited significance at the $p<.05$ level for those who selected Verbal remediations and $p<.01$ level for those who selected Visual remediations. As in other sections regarding timing data, the data has been stripped of data points exhibiting a mean time to complete less than 2 seconds and is not used in drawing conclusions of this work.

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.339*
	Sig. (2-tailed)		.030
	N	41	41
meanTimeToComplete	Pearson Correlation	-.339*	1
	Sig. (2-tailed)	.030	
	N	41	41

*. Correlation is significant at the 0.05 level (2-tailed).

Table 16: Pearson Correlation – All Students – Verbal

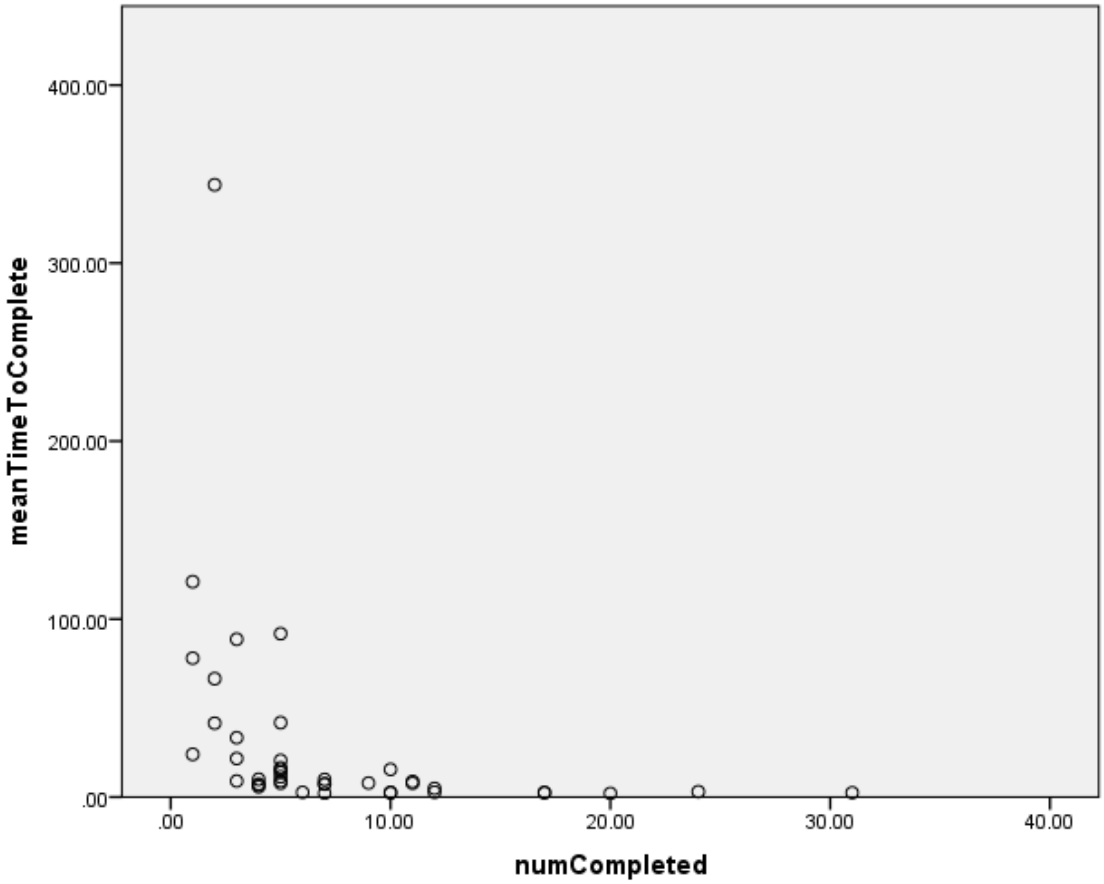


Figure 31: Scatter Plot– All Students – Verbal

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.450**
	Sig. (2-tailed)		.000
	N	62	62
meanTimeToComplete	Pearson Correlation	-.450**	1
	Sig. (2-tailed)	.000	
	N	62	62

**. Correlation is significant at the 0.01 level (2-tailed).

Table 17: Pearson Correlation – All Students – Visual

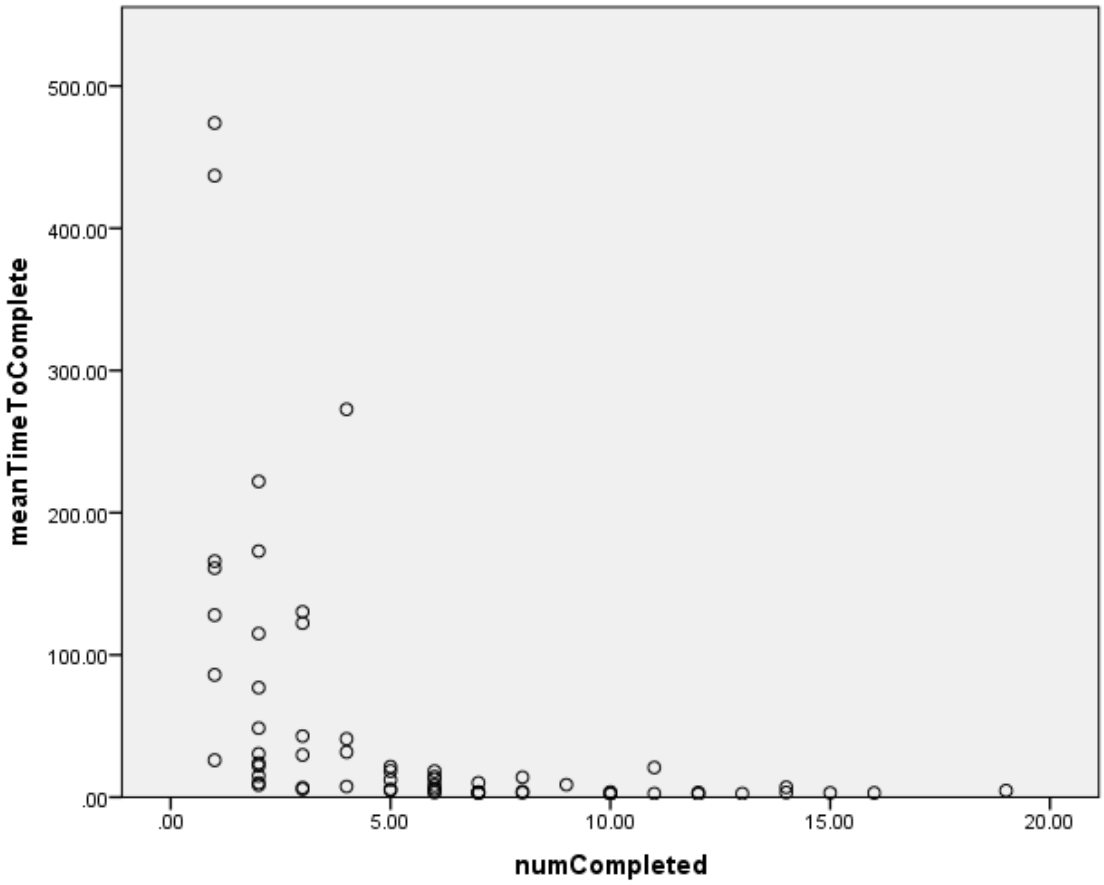


Figure 32: Scatter Plot– All Students – Visual

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.463
	Sig. (2-tailed)		.129
	N	12	12
meanTimeToComplete	Pearson Correlation	-.463	1
	Sig. (2-tailed)	.129	
	N	12	12

Table 18: Pearson Correlation – College – Verbal

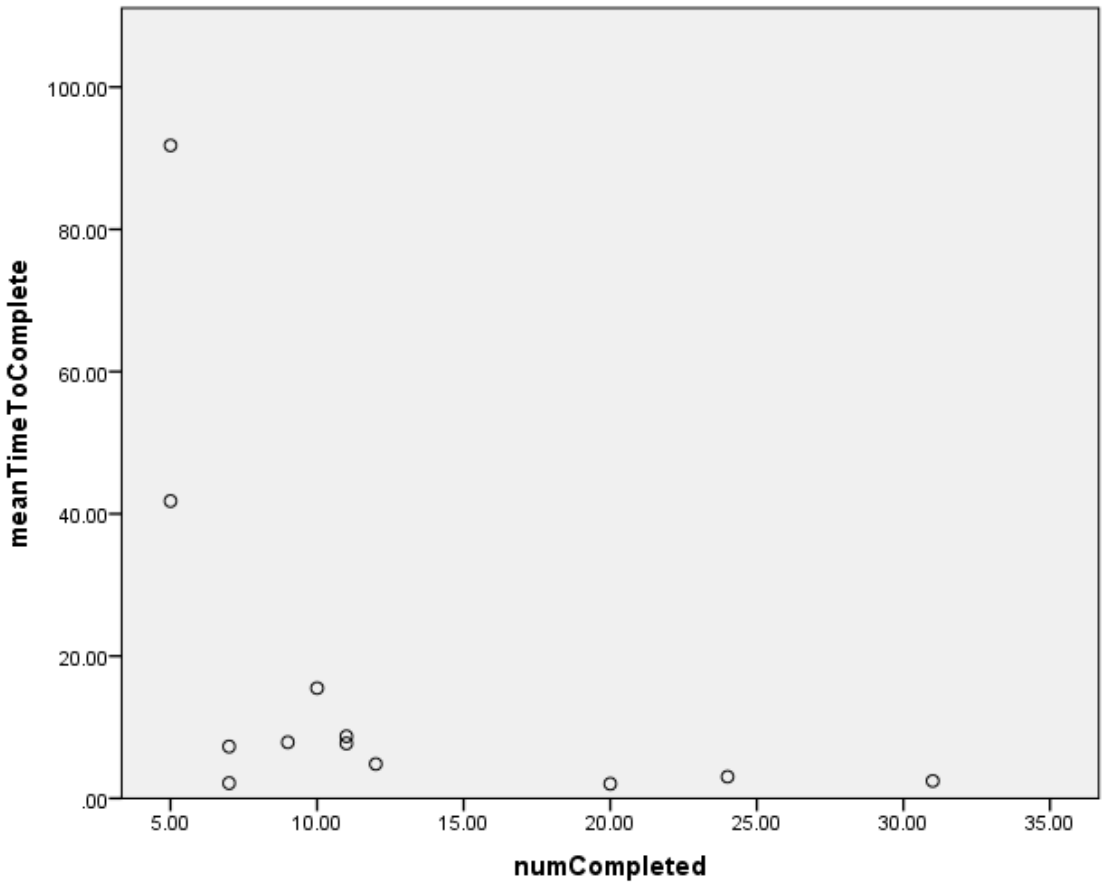


Figure 33: Scatter Plot– College – Verbal

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.515
	Sig. (2-tailed)		.105
	N	11	11
meanTimeToComplete	Pearson Correlation	-.515	1
	Sig. (2-tailed)	.105	
	N	11	11

Table 19: Pearson Correlation – College – Visual

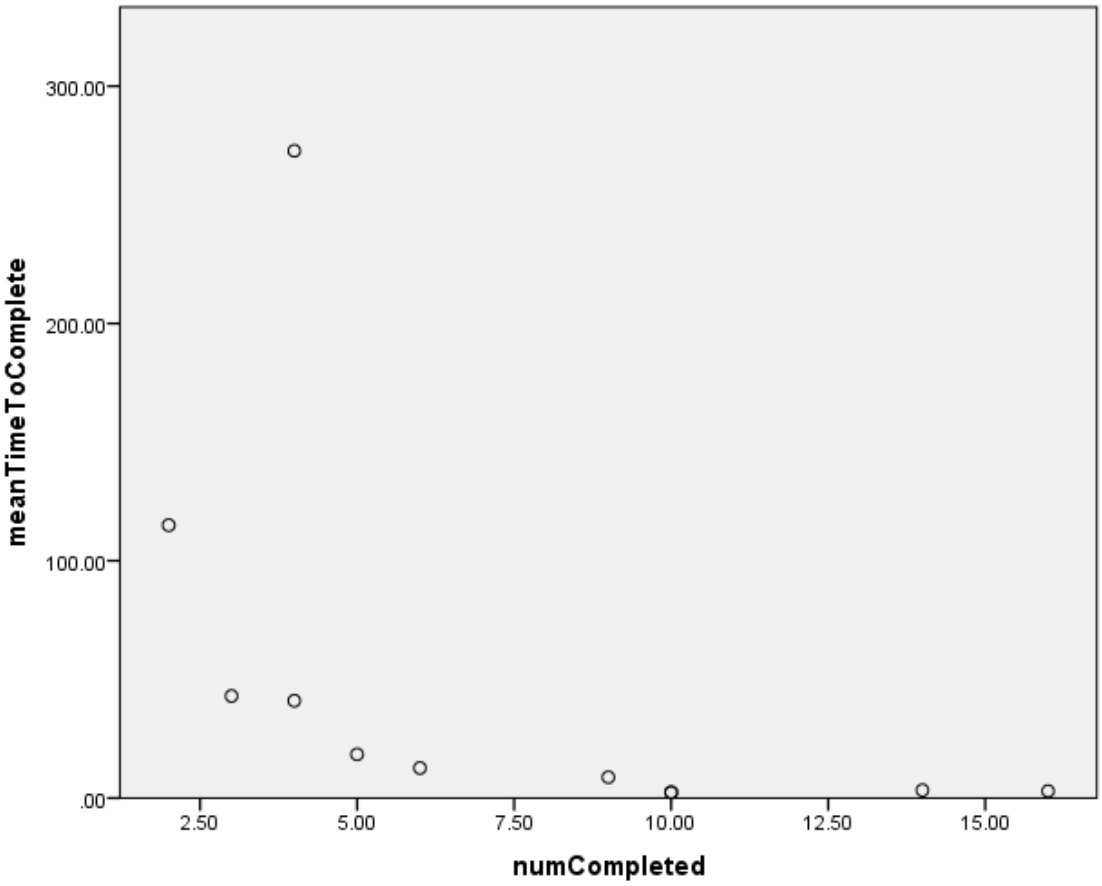


Figure 34: Scatter Plot– College – Visual

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.372*
	Sig. (2-tailed)		.047
	N	29	29
meanTimeToComplete	Pearson Correlation	-.372*	1
	Sig. (2-tailed)	.047	
	N	29	29

*. Correlation is significant at the 0.05 level (2-tailed).

Table 20: Pearson Correlation – High School – Verbal

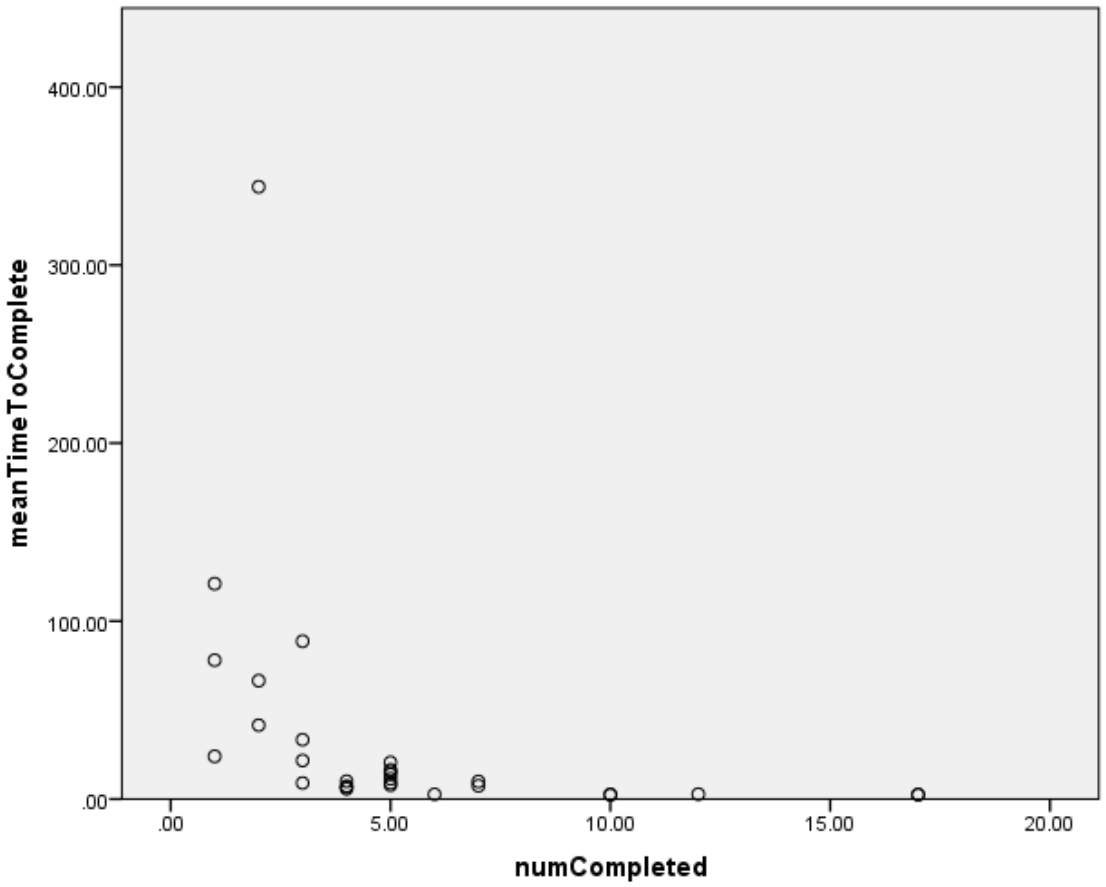


Figure 35: Scatter Plot– High School – Verbal

Correlations

		numCompleted	meanTimeToComplete
numCompleted	Pearson Correlation	1	-.446**
	Sig. (2-tailed)		.001
	N	51	51
meanTimeToComplete	Pearson Correlation	-.446**	1
	Sig. (2-tailed)	.001	
	N	51	51

**. Correlation is significant at the 0.01 level (2-tailed).

Table 21: Pearson Correlation – High School – Visual

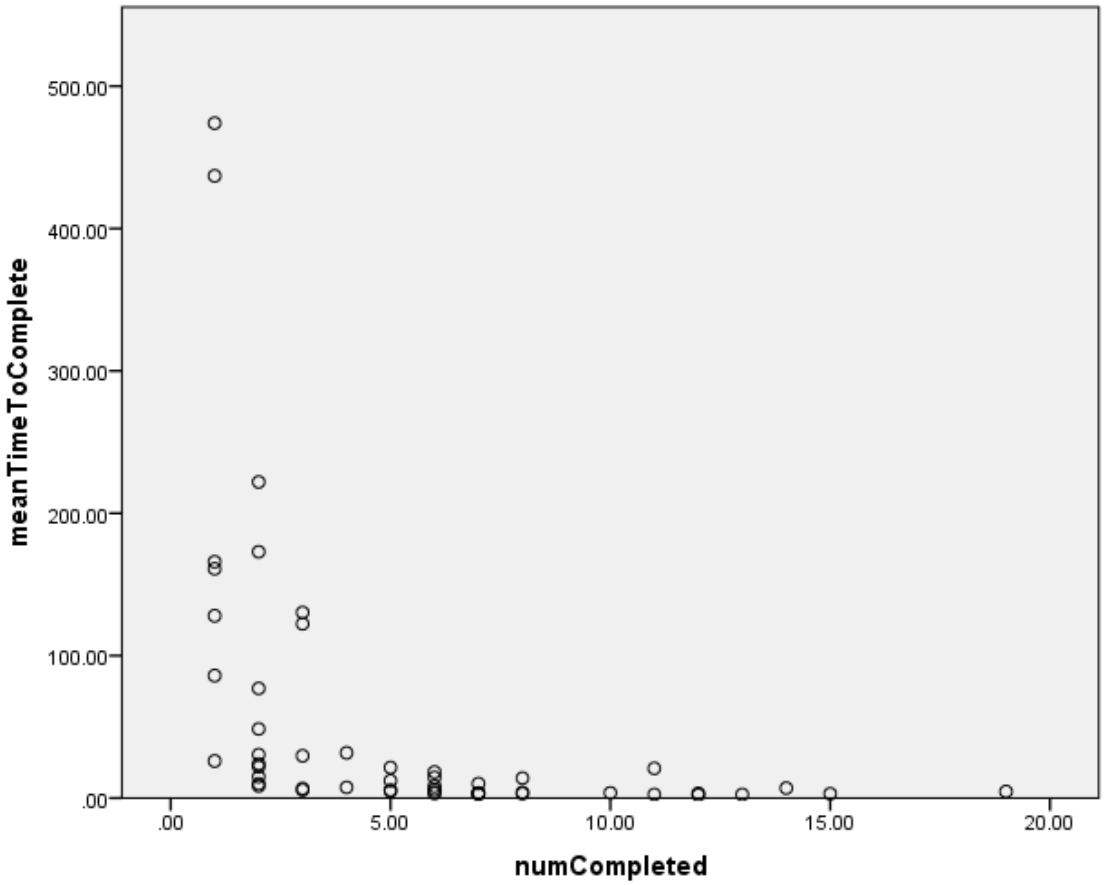


Figure 36: Scatter Plot– High School - Visual

5.3.7 Phase 1 vs Phase 2 – Pretest vs. Posttest

The students' pretest and posttest scores were compared on a by-phase basis. The results of this comparison appear below, as before this result compares only High School participants due to previously discussed issues with the College data in this regard. The first set of 4 tables represents Phase 1 students, the results of running the Paired Samples t-test in SPSS indicate statistical significance at the $p<.01$ level for this group of participants. The second set of 4 tables represents Phase 2 students, results for this group of participants with the same statistical measure also exhibit statistical significance at the $p<.01$ level.

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	pretest	1.88	51	1.519	.213
	posttest	2.55	51	1.487	.208

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	pretest & posttest	51	.622	.0000010806

Paired Samples Test

		Paired Differences			
					95% Confidence Interval of the Difference
		Mean	Std. Deviation	Std. Error Mean	Lower
Pair 1	pretest - posttest	-.667	1.306	.183	-1.034

Paired Samples Test

		Paired Differences			
		95% Confidence Interval of the Difference			
		Upper	t	df	Sig. (2-tailed)
Pair 1	pretest - posttest	-.299	-3.644	50	.001

Table 22: Phase 1 Pretest vs. Posttest t-test

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	pretest	1.63	51	1.661	.233
	posttest	2.53	51	1.678	.235

Paired Samples Correlations

		N	Correlation	Sig.
Pair 1	pretest & posttest	51	.747	.0000000003078

Paired Samples Test

		Paired Differences			
					95% Confidence Interval of the Difference
		Mean	Std. Deviation	Std. Error Mean	Lower
Pair 1	pretest - posttest	-.902	1.188	.166	-1.236

Paired Samples Test

		Paired Differences			
		95% Confidence Interval of the Difference			
		Upper	t	df	Sig. (2-tailed)
Pair 1	pretest - posttest	-.568	-5.424	50	.0000016923

Table 23: Phase 2 Pretest vs. Posttest t-test

5.3.8 Phase 1 vs Phase 2 – Attempts

The attempts data recorded by the system was further evaluated to determine if one exercise phase provided a better or worse experience than the other. Due to the fact that only high school students received both phases of the system, this subset of the data represents only High School students. This data was again analyzed using the Pearson Correlation. For phase 1 attempts data in isolation, a statistically significant negative correlation was found at the $p<.05$ level. For phase 2 attempts data in isolation, a statistically significant negative correlation was found at the $p<.01$ level for all High School students.

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	-.321*
	Sig. (2-tailed)		.032
	N	45	45
meanAttempts	Pearson Correlation	-.321*	1
	Sig. (2-tailed)	.032	
	N	45	45

*. Correlation is significant at the 0.05 level (2-tailed).

Table 24: Pearson Correlation for attempts data for phase 1 students only

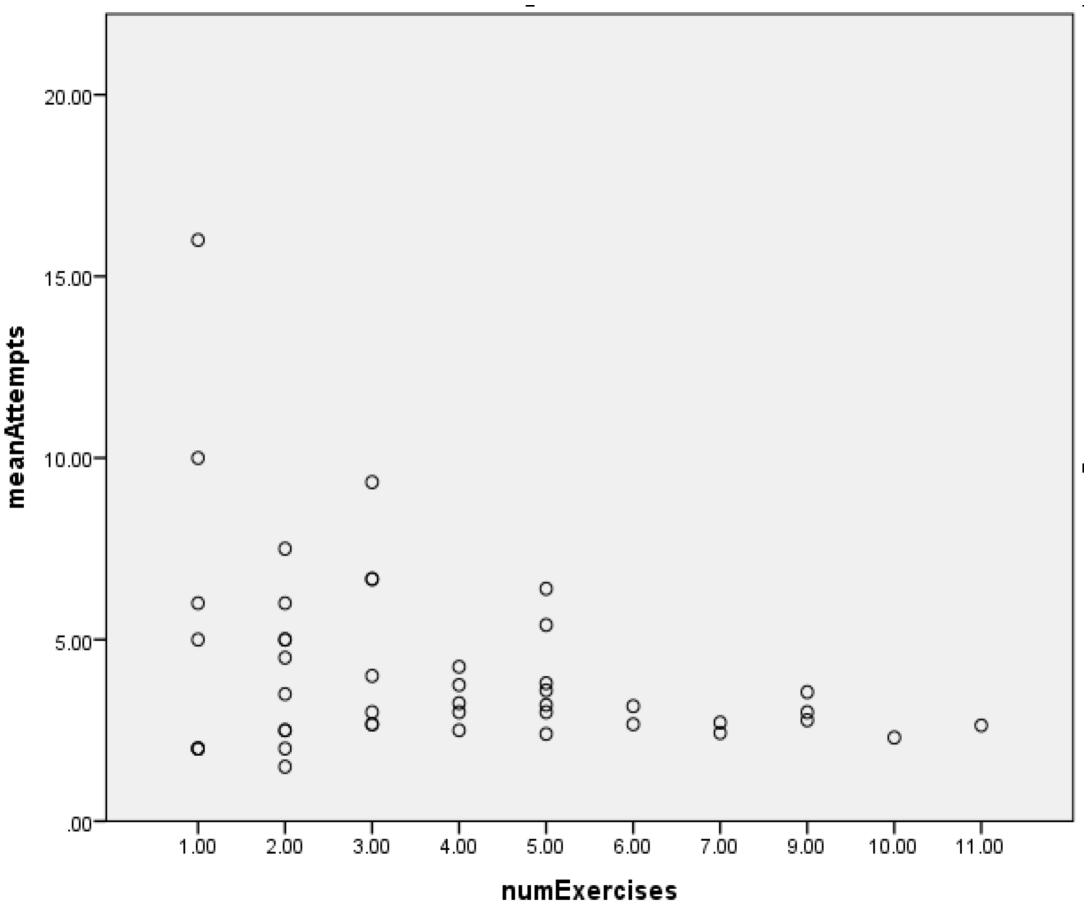


Figure 37: Scatter Plot for attempts data for phase 1 students only

Correlations

		numExercises	meanAttempts
numExercises	Pearson Correlation	1	-.392**
	Sig. (2-tailed)		.004
	N	52	52
meanAttempts	Pearson Correlation	-.392**	1
	Sig. (2-tailed)	.004	
	N	52	52

**. Correlation is significant at the 0.01 level (2-tailed).

Table 25: Pearson Correlation for phase 2 attempts data

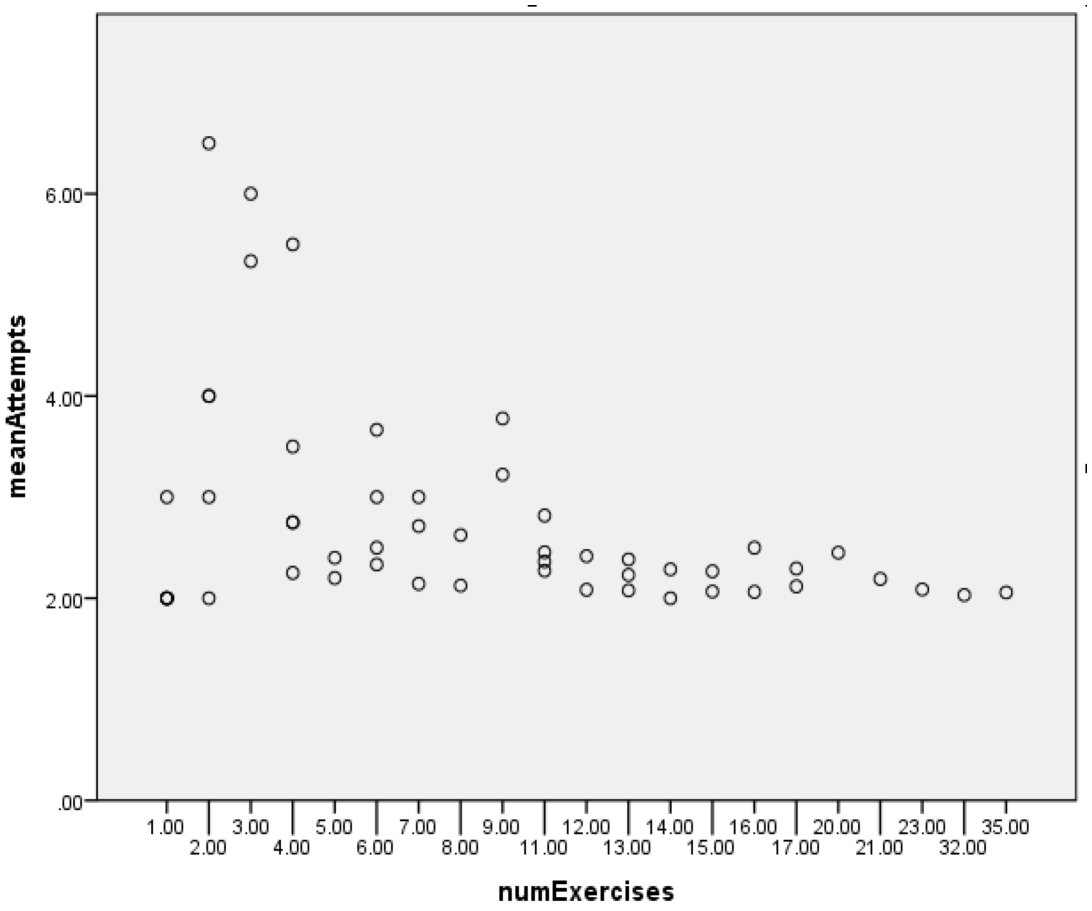


Figure 38: Scatter Plot for phase 2 attempts data

5.3.9 Phase 1 vs Phase 2 – Timing

The time required to complete an exercise was further evaluated to separate Phase 1 from Phase 2 to determine if one group was more or less benefitted from a given phase of the system. This data was analyzed in the same manner as before and includes only high school students as in the previous section. Timing data from both phases presented significance at the $p<.01$ level. Again, this data has been stripped of data points exhibiting an average time to complete less than 2 seconds and is not used as supporting data for the conclusions of this work.

Correlations

		numExercisesComplete	meanTimeInSeconds
numExercisesComplete	Pearson Correlation	1	-.417**
	Sig. (2-tailed)		.004
	N	45	45
meanTimeInSeconds	Pearson Correlation	-.417**	1
	Sig. (2-tailed)	.004	
	N	45	45

**. Correlation is significant at the 0.01 level (2-tailed).

Table 26: Pearson Correlation: Phase 1, timing data, high school only

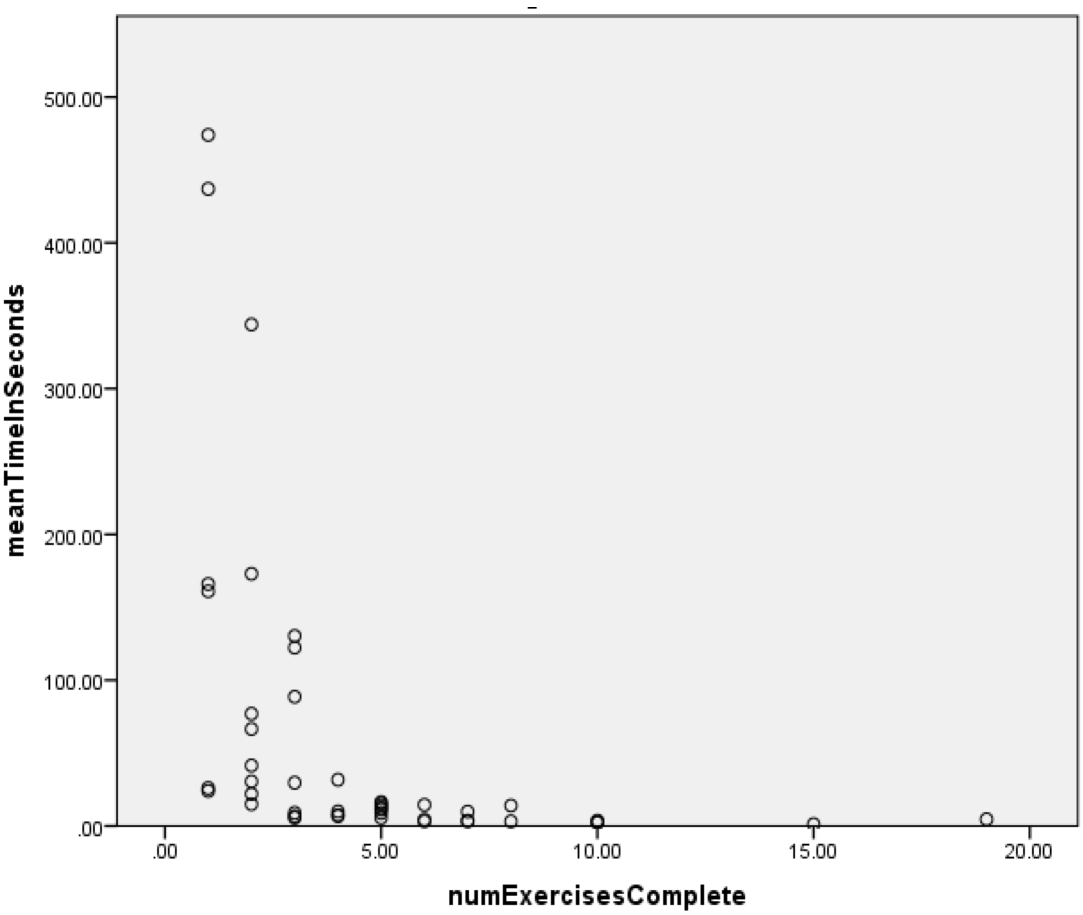


Figure 39: Scatter Plot: Phase 1, timing data, high school only

Correlations

		numExercisesComplete	meanTimeInSeconds
numExercisesComplete	Pearson Correlation	1	-.449**
	Sig. (2-tailed)		.001
	N	52	52
meanTimeInSeconds	Pearson Correlation	-.449**	1
	Sig. (2-tailed)	.001	
	N	52	52

**. Correlation is significant at the 0.01 level (2-tailed).

Table 27: Pearson Correlation: Phase 2, timing data, high school only

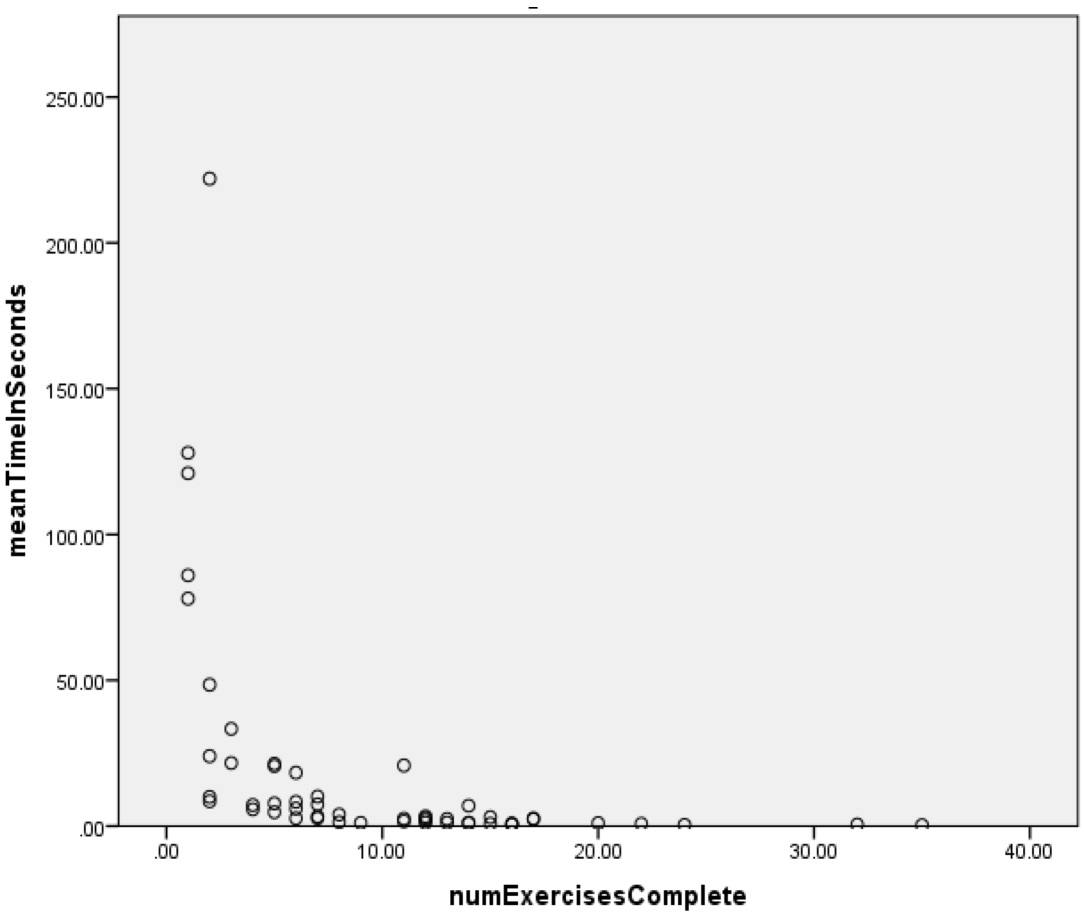


Figure 40: Scatter plot: Phase 2, timing data, High School only

5.3.10 Phase 3

Phase 3 evaluation occurred using two AP high school classes. The first time Phase 3 was introduced during the evaluation period was at the same time Phases 1 and 2 were being evaluated by another class. The current implementation of the system was unable to accommodate this many users at once and therefore very few results were obtained from the Phase 3 group. The second attempt at an evaluation of Phase 3 was performed using a class of 10 AP high school students at another school. The class was partitioned into control (four students) and test (5 students) groups. All students in this session were asked to complete a small lab assignment using the system. The lab

assignment consisted of a loop exercise where students had to print a certain series of characters to the screen multiple times (please see appendix for the exact exercise).

The difference between the start and end times for the participating students was calculated and then evaluated using the Independent Samples t-test in SPSS. End times for two students were missing from the data collected by the system. It is known that one of the control group students gave up due to issues encountered with using the system, this student's end time was set to the maximum observed end time in order to perform the statistical test. It is unknown why the other student did not have an end time, this student's end time was defaulted to the average of all observed completion times. The results of the statistical evaluation below show a decrease in means between the test and control groups (with the observed mean of the test group evaluating to 16.47332). Despite the difference in the means, the results of the evaluation did not produce significant results. Further discussion of these results appears in the next chapter.

Group Statistics

	isControl	N	Mean	Std. Deviation	Std. Error Mean
phase3TimeToComplete	0	5	16.47332	9.266682	4.144186
	1	4	17.99738	8.104843	4.052422

Table 28: Observed Means

Independent Samples Test

		Levene's Test for Equality of Variances		t-test for Equality of Means						
									95% Confidence Interval of the Difference	
		F	Sig.	t	df	Sig. (2-tailed)	Mean Difference	Std. Error Difference	Lower	Upper
phase3TimeToComplete	Equal variances assumed	.279	.613	-.259	7	.803	-1.5241	5.8949	-15.463	12.415
	Equal variances not assumed			-.263	6.898	.800	-1.5241	5.7962	-15.271	12.223

Table 29: Independent Samples t-Test of Phase 3 Timing Data

5.3.11 Pre and Post Test – Metacognitive Questions

Students participating in the study were also asked to complete a set of 3 multiple-choice metacognitive questions about debugging. These questions asked the student to think about what debugging is and how to successfully complete the debugging task. These questions are included in the appendix with the rest of the pre and post test questions. The results of the students' responses appear below. Results for the first question remained relatively the same between the pre and posttests, with the "Iterative edit / recompile / re-run" answer choice gaining 8 students and the less specific iterative analysis and modification choice losing 7 students. For the second question, the largest change in student responses consisted of a decrease in students selecting Forward Slice and more students selecting Backward Slice. The last question exhibited an increase between the pre and post tests of students indicating that the tutor's assistance with error messages does help novice programmers identify errors in the presence of syntax, runtime, and logical defects.

What is debugging?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a - Programming	2	1.7	1.7	1.7
	b – Iterative modification	9	7.8	7.8	9.5
	c – Iterative analysis and modification	72	62.1	62.1	71.6
	e – Iterative edit / recompile / re-run	29	25.0	25.0	96.6
	f – Don't know	4	3.4	3.4	100.0
	Total	116	100.0	100.0	

If you were trying to solve a problem would you…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Trial and Error	33	28.4	28.4	28.4
	b – Search	7	6.0	6.0	34.5
	c – Ask for Help	5	4.3	4.3	38.8
	d – Backward Slice	52	44.8	44.8	83.6
	e – Forward Slice	16	13.8	13.8	97.4
	f – Don't Know	3	2.6	2.6	100.0
	Total	116	100.0	100.0	

Error messages usually help programmers identify errors in…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Syntax	27	23.3	23.3	23.3
	b – Runtime Behavior	3	2.6	2.6	25.9
	c – Program Output	1	.9	.9	26.7
	d – a and b	25	21.6	21.6	48.3
	e – a, b, and c	45	38.8	38.8	87.1
	f – b and c	3	2.6	2.6	89.7
	g – don't know	12	10.3	10.3	100.0
	Total	116	100.0	100.0	

Table 30: Metacognitive – Everyone – Pretest

What is debugging?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a - Programming	2	1.7	1.7	1.7
	B – Iterative modification	8	6.8	6.8	8.5
	c – Iterative analysis and modification	65	55.6	55.6	64.1
	d – Internet Search	1	.9	.9	65.0
	e – Iterative edit / recompile / re-run	37	31.6	31.6	96.6
	f – Don't know	4	3.4	3.4	100.0
	Total	117	100.0	100.0	

If you were trying to solve a problem would you…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Trial and Error	37	31.6	31.6	31.6
	b – Search	4	3.4	3.4	35.0
	c – Ask For Help	7	6.0	6.0	41.0
	d – Backward Slice	65	55.6	55.6	96.6
	f – Don't Know	4	3.4	3.4	100.0
	Total	117	100.0	100.0	

Error messages usually help programmers identify errors in…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Syntax	25	21.4	21.4	21.4
	b – Runtime Behavior	2	1.7	1.7	23.1
	c – Program output	4	3.4	3.4	26.5
	d – a and b	20	17.1	17.1	43.6
	e – a, b, and c	57	48.7	48.7	92.3
	f – b and c	3	2.6	2.6	94.9
	g – Don't Know	6	5.1	5.1	100.0
	Total	117	100.0	100.0	

Table 31: Metacognitive – Everyone - Posttest

What is debugging?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Programming	1	8.3	8.3	8.3
	c – Iterative analysis and modification	10	83.3	83.3	91.7
	e – Iterative edit / recompile / re-run	1	8.3	8.3	100.0
	Total	12	100.0	100.0	

If you were trying to solve a problem would you…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Trial and Error	3	25.0	25.0	25.0
	b – Search	2	16.7	16.7	41.7
	c – Ask For Help	1	8.3	8.3	50.0
	d – Backward Slice	5	41.7	41.7	91.7
	e – Forward Slice	1	8.3	8.3	100.0
	Total	12	100.0	100.0	

Error messages usually help programmers identify errors in…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Syntax	5	41.7	41.7	41.7
	d – a and b	6	50.0	50.0	91.7
	e – a, b, and c	1	8.3	8.3	100.0
	Total	12	100.0	100.0	

Table 32: Metacognitive – College – Pretest

What is debugging?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Programming	1	8.3	8.3	8.3
	b – Iterative modification	1	8.3	8.3	16.7
	c – Iterative analysis and modification	9	75.0	75.0	91.7
	e – Iterative edit / recompile / re-run	1	8.3	8.3	100.0
	Total	12	100.0	100.0	

If you were trying to solve a problem would you…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Trial and Error	5	41.7	41.7	41.7
	b – Search	1	8.3	8.3	50.0
	c – Ask for Help	1	8.3	8.3	58.3
	d – Backward Slice	5	41.7	41.7	100.0
	Total	12	100.0	100.0	

Error messages usually help programmers identify errors in…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Syntax	4	33.3	33.3	33.3
	d – a and b	4	33.3	33.3	66.7
	e – a, b, and c	4	33.3	33.3	100.0
	Total	12	100.0	100.0	

Table 33: Metacognitive – College - Posttest

What is debugging?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Programming	1	1.0	1.0	1.0
	b – Iterative modification	9	8.7	8.7	9.6
	c – Iterative analysis and modification	62	59.6	59.6	69.2
	e – Iterative edit / recompile / re-run	28	26.9	26.9	96.2
	f – Don't Know	4	3.8	3.8	100.0
	Total	104	100.0	100.0	

If you were trying to solve a problem would you…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Trial and Error	30	28.8	28.8	28.8
	b – Search	5	4.8	4.8	33.7
	c – Ask for Help	4	3.8	3.8	37.5
	d – Backward Slice	47	45.2	45.2	82.7
	e – Forward Slice	15	14.4	14.4	97.1
	f – Don't Know	3	2.9	2.9	100.0
	Total	104	100.0	100.0	

Error messages usually help programmers identify errors in…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Syntax	22	21.2	21.2	21.2
	b – Runtime Behavior	3	2.9	2.9	24.0
	c – Program output	1	1.0	1.0	25.0
	d – a and b	19	18.3	18.3	43.3
	e – a, b, and c	44	42.3	42.3	85.6
	f – b and c	3	2.9	2.9	88.5
	g – Don't know	12	11.5	11.5	100.0
	Total	104	100.0	100.0	

Table 34: Metacognitive – High School – pretest

What is debugging?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a - Programming	1	1.0	1.0	1.0
	b – Iterative modification	7	6.7	6.7	7.6
	c – Iterative analysis and modification	56	53.3	53.3	61.0
	d – Internet Search	1	1.0	1.0	61.9
	e – Iterative edit / recompile / re-run	36	34.3	34.3	96.2
	f – Don't Know	4	3.8	3.8	100.0
	Total	105	100.0	100.0	

If you were trying to solve a problem would you…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Trial and Error	32	30.5	30.5	30.5
	b – Search	3	2.9	2.9	33.3
	c – Ask for Help	6	5.7	5.7	39.0
	d – Backward Slice	60	57.1	57.1	96.2
	f – Don't Know	4	3.8	3.8	100.0
	Total	105	100.0	100.0	

Error messages usually help programmers identify errors in…

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	a – Syntax	21	20.0	20.0	20.0
	b – Runtime Behavior	2	1.9	1.9	21.9
	c – Program output	4	3.8	3.8	25.7
	d – a and b	16	15.2	15.2	41.0
	e – a, b, and c	53	50.5	50.5	91.4
	f – b and c	3	2.9	2.9	94.3
	g – Don't Know	6	5.7	5.7	100.0
	Total	105	100.0	100.0	

Table 35: Metacognitive – High School - Posttest

5.3.12 Qualitative Data

In addition to the passive and formal evaluation measures discussed above, the students were also asked to complete a short survey in order to gauge the students' opinions of the system. The exact survey instrument used is included as an appendix. Aggregate student responses appear in the following tables. Tables 13 and 14 exhibit an overwhelming majority of students indicating that concepts they learned in the system would be applicable to programs they may write outside of the system and to their coursework in a more general sense.

I will be able to apply concepts learned in the system to programs I write outside of the system

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid		1	.8	.8	.8
	N	19	15.0	15.0	15.7
	Y	107	84.3	84.3	100.0
	Total	127	100.0	100.0	

Table 36: Y/N – Concepts applicable to programs written outside of the system

Concepts I learned in the system are applicable to my coursework

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid		4	3.1	3.1	3.1
	N	11	8.7	8.7	11.8
	Y	112	88.2	88.2	100.0
	Total	127	100.0	100.0	

Table 37: Y/N – Concepts applicable to coursework

When asked to indicate, using a Likert scale, how helpful the system was in teaching them to debug programs 81.1% of all students rated the system at least 3 out of 5.

How helpful did you feel the system was in teaching you how to debug programs

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1.00	9	7.1	7.1	7.1
	2.00	14	11.0	11.1	18.3
	3.00	46	36.2	36.5	54.8
	4.00	48	37.8	38.1	92.9
	5.00	9	7.1	7.1	100.0
	Total	126	99.2	100.0	
Missing	System	1	.8		
Total		127	100.0		

Table 38: Likert – System was helpful in teaching you to debug programs

When asked to indicate with a Likert scale how helpful concepts taught by the system will be when debugging programs outside of the system, 73.2% of the students rated the system at least 3 out of 5.

How helpful do you think concepts taught by the system will be in debugging programs you wrote outside of the system?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1.00	8	6.3	6.4	6.4
	2.00	13	10.2	10.4	16.8
	3.00	49	38.6	39.2	56.0
	4.00	39	30.7	31.2	87.2
	5.00	16	12.6	12.8	100.0
	Total	125	98.4	100.0	
Missing	System	2	1.6		
Total		127	100.0		

Table 39: Likert – How helpful will system concepts be in debugging other programs

On the last question students were asked to rate, again using the Likert scale, the helpfulness of the feedback mechanism. 70.3% of the students rated the system at least a 3 out of 5 on this question.

How helpful was the feedback the system gave for individual debugging problems?

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1.00	9	7.1	7.1	7.1
	2.00	15	11.8	11.9	19.0
	3.00	43	33.9	34.1	53.2
	4.00	38	29.9	30.2	83.3
	5.00	21	16.5	16.7	100.0
	Total	126	99.2	100.0	
Missing	System	1	.8		
Total		127	100.0		

Table 40: Likert - How helpful was feedback produced by the system

5.3.13 Pre-Survey – Student's Previous Programming Experience

In addition to the pretest and end survey discussed earlier, students were also asked to complete a brief survey at their first log in to the system that would indicate what previous programming experience they had, if any. 128 students completed the survey. Three students submitted multiple, conflicting answers, their programming survey data was subsequently omitted from this discussion.

Have you ever taken a programming course before?

	Frequency	Percent	Valid Percent	Percent of Total Population (N = 127)
Y – Yes	48	39	39	37.8
N - No	75	61	61	59
Total	123	100	100	96.8

Table 41: Have you ever taken a programming course before?

Have you tried to learn programming outside of a course / in another course?

	Frequency	Percent	Valid Percent	Percent of Total Population (N = 127)
Y – Yes	35	28.5	28.5	27.6
N - No	88	71.5	71.5	69.3
Total	123	100	100	96.9

Table 42: Have you tried to learn programming outside of a course / in another course?

Of all students who participated in the evaluation about half of the population indicated that they had some prior programming experience, whether formal classroom experience or on their own (48%, or 61 students).

As shown in the two tables above, the majority of students using the system did not have prior programming experience before the course they were taking at the time of the evaluation. The students were also asked to supply what programming languages they had learned, if any. Five of the students who indicated they had previously taken a programming course did not indicate what language they had learned. Additionally,

four of the students who indicated prior programming assistance provided answers that were not actual programming languages. These included Eclipse (an Integrated Development Environment generally used for Java development), HTML (a markup language for website development) Dreamweaver (a web design/development Integrated Development Environment) and GML (an ambiguous reference that could point to a few different markup languages). These selections are excluded from the table below. Additionally, there exists overlap between students and languages – some students indicated multiple valid programming languages. Each language tally represents the number of respondents indicating prior experience with the language. Of all participating students, about 32% had Java programming experience prior to the course where the evaluation took place.

		Frequency	Percent of total evaluation population (N = 127)
Valid	Java	41	32.3
	Python	7	5.5
	Visual Basic .NET	2	1.6
	C# .NET	1	.8
	C++ / C	4	3.1
	Alice	9	7.1
	javascript	1	.8
	Jeroo (pseudo Java)	1	.8

Table 43: Languages students had previous experience with

5.4 Discussion of the System's ability to generate exercises and obtain cases from student solutions

5.4.1 Exercise Generation

During evaluation, the system was required to generate exercises over the Elementary Programming topic. The Generator looks for cases that map to the selected topic from the front end. A case is selected from this topic from the case base and sent to a method for generating the exercise. This method then takes the meta language solution associated with the case and turns it into instructions for the exercise generator to create the bug instead of reasoning about what assistance to offer in the presence of the bug represented by the case.

During the evaluation period, 786 exercises were generated by the system and served to students. Of the generated exercises, 196 of them were distinct. These exercises were all generated over the same single base class that represented practice with simple statements and was 28 lines long.

5.4.2 Case Acquisition

During the course of the evaluation, every successful solution to an exercise that a student coded was tagged, so that at a later time the case acquisition process could analyze each of these solutions. This analysis reviews each student solution for uniqueness as compared to the original solution for a given exercise. Then, if the student's answer is different, the system turns the student's edits to the exercise code into the meta-language used in the system for providing remediations and generating exercises. If the generated solution is not already used in the cases that are similar to

the defect that was present in the exercise code, the system either updates an existing case (if that case had no solution associated with it) or generates a new case. 1195 solutions were collected and analyzed. Of these, 511 were marked as new and prepared for retention by the system. These results are discussed further in Chapter 6.

6 Chapter 6: Conclusions

This chapter discusses further what the statistical results presented in chapter 5 mean for this research and then provides answers to the research questions posed at the beginning of this work. For convenience, the answers to the research questions are presented in brief here:

- Intelligent Tutoring Systems
    - *How could the system reason about student solutions without incurring the program verification problem?*
        - The system has the following abilities to address this problem: it knows at least one possible answer for each question it generates; it knows the expected output for each possible question, and it is able to compile and run each student solution to determine if the output created was correct. The statistically significant pretest-practice-posttest results indicate that students learned while using the system, which in turn show that ITS-Debug successfully provides feedback on student programming defect issues.
    - *How should debugging issues be remediated?*
        - Changed to "How *could* debugging issues be remediated?" There are many possible approaches to remediation. The methodology chosen in this work has been proven valid by statistically significant pretest-practice-posttest results.
    - *How could the system determine if a given remediation is successful?*

        - This functionality was excluded from formal evaluation as it was made redundant by the committee's decision to exclude the automatic selection of student learning style during the evaluation. The methodology of ITS-Debug is discussed further below.
    - *Could the system generate exercises dynamically?*
        - In phase 2 of ITS-Debug, which adds automatically generated exercises, results of attempts data indicate that students required less attempts to complete exercises the more exercises they completed. This result suggests that Phase 2 exercises enhanced the experience of students participating in the evaluation.
- Intelligent Tutoring Systems and Case Based Reasoning
    - *How should the domain be represented and reasoned about?*
        - Case Based Reasoning was the chosen methodology, with cases presenting as individual syntax, runtime, and logical defects. The success of ITS-Debug in evaluation trials indicates that the chosen methodology is valid.
    - *How can the system acquire domain knowledge?*
        - The system was given the ability to acquire new cases from student solutions, discussed in Chapter 4. The system was able to extract 511 cases from 1195 student solution. Tthe unique-ness of all these new cases is yet to be determined. This question is discussed further below and in Chapter 7 as future work.

- Computer Science Education
    - *How could debugging be taught? Can debugging be taught to novices? How could a system teach the domain?*
        - The chosen methodology was proven successful by pretest-practice-posttest t-test results and by analysis of students' attempts data, both at the means and as individual exercise episodes (see Appendix G).

6.1 Further Discussion of Obtained Results

6.1.1 Pretest vs. Posttest Results

The design of this aspect of the experiment improved after the college students used the system. When the evaluation was being conducted with the College students it was required by the class instructor that the system be used to help the students prepare for an exam. These exams consisted of two main concepts each. For example, the students' second exam focused on Loops and Methods. Therefore the students were asked to practice with Loops for twenty minutes and then Methods for twenty minutes. Additionally, the pre- and posttests needed to be shortened from their original form in order to fit this design. This part of the evaluation also suffered from certain system flaws that have since been remedied. Furthermore, the system was later opened to the students for more freeform use. Although this provided valuable data regarding attempts and time to complete exercises, the pre and posttest methodology did not work well in this environment. Some students didn't practice long enough, some students waited a long period of time before returning to complete the posttest, and some

students simply did not complete the posttest. Due to these issues, the first half of the college student data showed an increase in means between the pretest and posttest that was not statistically significant and the second half of the data for the pre and posttest evaluation was rendered useless.

When high school students were presented with the system these issues were removed from the experiment by only holding the experiment in supervised sessions and by having the students practice with only one topic in a session. A dramatic improvement in the results received from this aspect of the experiment was observed. This time, a two tailed t-test result of $p<.01$ was observed. Some external reasons for this result could include that there were much more of these students and that these students were much closer to being true novices. The larger group of high school participants had mostly only previously programmed using Karel the Robot and were all taking an introductory level Computer Science course. Other students using the system were either Advanced Placement students or college level students, several of which had some kind of previous programming experience at a more advanced level than Karel the Robot.

The results from the pretest-practice-posttest methodology used in the evaluation of this system proves that learning of debugging skills did in fact occur between the time the pretest was taken and the time the posttest was taken. The only event that occurred in between the pretest and posttest was practice within the system. Therefore, the system did in fact teach the students debugging skills.

6.1.2 Attempts Results

The results of the analysis of the attempts data are encouraging. When all student data, regardless of phase or scholarly level, is analyzed as one data set a significant correlation at the $p<.01$ level is obtained, indicating that with more practice within ITS-Debug fewer attempts are required to complete a given exercise. This result may imply that the system is helping the student to learn how to debug programs. When the data is split based on scholarly level, high school students exhibit a statistically significant negative correlation at the $p<.01$ level when comparing the number of exercises seen in the system with the number of attempts required to complete an exercise. Conversely, when looking at the attempts data obtained from college students alone, no statistically significant correlation is found. It is believed that this is due to two factors: the much smaller sample size that the college students represent and the fact that the evaluation changed after the college students used the system. Specifically, college students were asked to practice for 20 minutes each with two different topics. After analyzing the college student data this was deemed a weakness of the experimental design. The data would start to trend towards the expected negative correlation then rise sharply up as the students changed topics. After reviewing this data the decision was made to have students practice with a single topic in a single session, with the hypothesis that the originally hypothesized negative correlation would be evident if the students were not required to change topics during the session. This hypothesis appears to have been proven correct with the high school students. Proving this hypothesis over college students is included as an item for future work in the next chapter.

The significant negative correlation found between the mean number of attempts to complete an exercise and the number of exercises completed within the system proves that with more practice in the system, less attempts are required to complete an exercise. Although the phase 2 participants all received variations of the same base program, both sets of results exhibited statistically significant negative correlations. Therefore, this part of the evaluation also shows that participants acquired debugging skills over the course of the evaluation.

6.1.3 Timing Results

Timing data for all students using the system was evaluated using the Pearson Correlation to determine if a statistically significant negative correlation between number of exercises completed and the mean amount of time required to complete an exercise existed. As mentioned earlier in Chapter 5, this data may contain a skew due to an at this time unknown error in the logging mechanism given that some students present with a mean time to complete of less than a second. This data is not used in this work to prove any of the relevant hypotheses. The analysis has been retained however as, if a skew exists, it appears to exist in a regular manner and may therefore still be relevant to this work.

Analysis of this data produced a significant result at the $p<.01$ level. As with attempts data, the timing data was split by the students' educational level (high school vs. college). Similar to the attempts data, a significant negative correlation was found over high school students' time to complete an exercise at the $p<.01$ level. Again, College students alone did not yield a statistically significant result. However this data

set differentiates itself from the College student attempts data when displayed as a scatter plot in that a trend does appear to exist but not at a statistically significant level. It is believed that, had more college students participated, this result would have also yielded a statistically significant negative correlation between exercises seen and the time required to complete an exercise due to the fact that the data is close to statistical significance at the $p<.05$ level.

6.1.4 Pretest and Posttest Metacognitive Questions Results

The results from the metacognitive questions do not appear to show a significant change in the way the novice students thought about the debugging process. This is believed to be due to four factors. First, the students' limited time within the system would have limited this aspect of the study. Second, the data appears to show that most of the students already had some understanding of what the debugging problem entails. Third, most of the answers that did change over the course of the study moved towards the more detailed and specifically correct choices provided. The second result is in line with previously mentioned background research – students already understand, to a degree, how to approach the debugging problem. The difference between the novice and the expert in the debugging process is mainly that the novice debugger is less efficient at debugging programs due to a lack of domain knowledge and that the novice has a greater tendency to introduce new defects while trying to solve the problem.

Finally, the fact that some of the metacognitive questions do not all have a discrete correct answer could have had a negative impact on the results obtained. The design of the questions was intended to hedge against the second factor above – that the

students have some prior understanding of what the debugging problem entails. It was anticipated that this prior knowledge would have led to similarly non-significant results if the questions gauged only whether or not they knew what debugging was and how to go about it. Therefore the chosen set of metacognitive questions aimed to measure whether or not the student's knowledge of debugging matured over the course of the study. Modifying the metacognitive questions to better suit this purpose and performing the evaluation over a longer period of time in order to allow this maturation of debugging knowledge to occur are left for future evaluation efforts.

6.1.5 Phase 1 vs. Phase 2 Results

The results obtained from comparing Phase 1 to Phase 2 were pleasantly surprising. The hypothesis had been that the automatically generated exercises would be at least as effective as the hand generated exercises. Originally, only the attempts and timing data were considered for this aspect of the evaluation. Further statistical analysis was performed, splitting the Pretest and Posttest results by phase and performing separate paired t-test analyses over both data sets. Both sets of students' scores exhibited statistical significance at the $p<.01$ level, with Phase 2 students exhibiting a higher level of significance than Phase 1. Therefore, both aspects of the system were effective in teaching students debugging skills, and both were at least as effective as each other.

The attempts analysis of the experiment show a significantly higher level of effectiveness for students who received automatically generated exercises. This result could in part be due to the consistent nature of the exercises generated, since a single

base program was used in the evaluation to generate exercises (though the system is able to use multiple base classes). While Phase 1 exercises each consisted of semi-unique individual programs, Phase 2 exercises were all manipulations of a single program. Therefore, as students gained experience with the particular program being used by the Exercise Generator they may have found it easier to fix each subsequent bug produced by the system due to their growing familiarity with the base program. Although previously questions were raised as to possible confounding effects from the phase 2 students only receiving one base program for their exercises, the t-test results suggest that perhaps the regularity of the exercises presented to this group was in fact more helpful than having the students view and correct more disparate base programs.

This evaluation design was chosen purposefully in order to avoid the possibility of confounding the results with base programs that were too disparate. However this design decision leaves open questions as to how students would perform on more disparately generated exercises, over multiple base programs, for a single topic. Given the evaluation performed it is unclear that students would definitely have performed in the same manner if multiple base classes had been utilized. It seems likely that, since both phases produced statistically significant results, evaluation with multiple base classes would not yield significantly different results. Future evaluation over multiple base classes and multiple topics may be merited.

6.1.6 Phase 3 Results

Although the results from the evaluation of Phase 3 did not produce statistical significance, a decrease in the means between the control and test groups was present.

When the Phase 3 experiments were run, the system presented certain server and software oriented issues not present in pre-evaluation testing. In addition, students completed the exercise more quickly than anticipated. Therefore the evaluation session for Phase 3 was much shorter than for Phases 1 and 2. It is believed that a future evaluation of Phase 3 of the system, with more students, a longer evaluation exercise, and in the absence of the issues described above, would produce statistically significant results. Further augmentation and evaluation of Phase 3 is discussed in the next chapter.

6.1.7 Qualitative Results

An analysis of the qualitative data retrieved by the system displays an encouragingly positive response from the students using the system. 84.3% of the students who used the system agreed that concepts they learned within the system would be applicable to programs written outside of the system and 88.2% of students agreed that the concepts they learned within the system were applicable to their coursework. Therefore, not only was learning observed from the data collected by the system but the students also perceived that what they learned from the system would transfer to work outside of the system.

Three additional survey questions were presented to the students, The Likert levels are taken to indicate the following intervals: 1 = not at all, 2= not helpful, 3 = somewhat helpful, 4 = helpful, 5 = extremely helpful. The reason for this deviation from the usual interpretation of Likert ranges (where 3 usually indicates neutrality) is because of the overlap between the second survey question ("How Helpful do you think concepts taught by the system will be in debugging programs you wrote outside of the system?")

and the first Y/N question ("I will be able to apply concepts learned in the system to programs I write outside of the system"). Due to the overlap of these questions and the fact that all but 19 students answered 'Y' to the very similar Y/N question, 3 is taken to indicate they found the system 'somewhat helpful' as opposed to taking a neutral position. With this interpretation of the results, an encouraging majority indicated that they found the system to be helpful in debugging programs. The results from all three of the continuous scale questions indicated the majority of students selected 3 or above. With the interpretation described earlier, this leads to the conclusion that the system was well accepted by the students.

The last question of the survey was free form and asked the students to indicate if there was something they would change about how the system works. During the course of the evaluation this feedback was taken very seriously and helped drive changes to the system as it evolved. The following provides a summary of student responses to this question:

- Confusion on modality selection
- Indication that the system was very helpful and that the student would have liked more time to practice
- Request for more specific/targeted feedback
- Dislike/intimidated by the original color scheme chosen for the interface
- More exercises / More difficult exercises
- Feedback on the pre/post tests to see what they got right/wrong
- Descriptions of glitches/issues with the system that were encountered

- Request for exercises to teach more computer science concepts

As the college students were the first formal evaluation group, it was possible to take some of this feedback under consideration and modify the system accordingly. Specifically, the user interface underwent a major overhaul before evaluation with high school students. The system also received more levels of remediation and the ability to generate exercises on demand.

During the testing sessions certain feedback was given directly to the researcher from the students and the instructors involved. Over the course of this evaluation of the system several students from both the college and high school levels asked if they could continue using the system outside of the testing session, a few of these students did actually continue to use the system of their own volition.

Additionally, participating high school teachers provided their own observations informally at the end of evaluation sessions. One high school teacher asked if they could continue to use the system with their students. Another commented that, while helping to oversee the evaluation, they could see that their students were learning while using the system. This teacher also indicated that they might be interested in continuing to use the system with their students.

6.2 Research Questions Recap

The following reviews the contributions and pursued questions of this research, and how the implementation and results serve to answer the questions posed.

6.2.1 Intelligent Tutoring System

This work successfully created an Intelligent Tutoring System capable of teaching novice students the domain, as exhibited by the results of the pretest – posttest aspect of the evaluation. Significantly lower scores were obtained on the posttest after the students used the system, implying that the system did indeed teach the students the material taught by the system. Since ITS-Debug taught students how to debug small programs, it follows that the results support the conclusion that it does in fact help teach students how to debug programs.

The successful evaluation results obtained for phases 1 and 2 additionally imply that the methodology chosen is indeed valid for representing and reasoning about the domain. As discussed earlier, Case Based Reasoning and Static Program Analysis were chosen for representing and reasoning over the domain. Case Based Reasoning was chosen because the analogical nature of CBR provides is close to the way expert programmers typically reason about the domain. I.e., expert programmers frequently ask themselves, have I seen this error before? What did I do last time to resolve this error? What I did differently this time that I should remember next time? In addition, when programmers try to analyze why their program failed to run or failed to produce the desired results they may make mental static slices of their program code. In essence they may try to think like the computer and determine where the process may have gone wrong by traveling forward and backwards through their code and analyzing it in this manner.

The results obtained from students using the system indicated a high level of statistical significance; including the t-test results and the analysis of each students' mean attempts required to complete an exercise. This researcher therefore believes that the choices of CBR and program slicing are conducive to the goals of ITS-Debug, and thus help novice Computer Science Students learn how to debug programs. It is possible that a combination of CBR with richer model-based reasoning could help the system be more relevant to more advanced users.

Another research question asked how the knowledge base for this system could be kept tractable, given that the domain is potentially infinite. This question was posed before the design or implementation of the system was clear. Although the domain is potentially infinite in the number of ways to program a specific problem and the number of ways to break a given program, it was found to be limitable for this research through several factors. These included: the choice of target audience, the interaction with javac, the Java Runtime Environment, and the FindBugs static analyzer. Each of these resources has a static knowledge library.. All possible compiler issues that javac can produce are represented with the knowledge base provided as it comes directly from the configuration file javac uses to generate all of its error messages. Runtime errors are represented by the core Java library's Exception classes. Again, this is completely represented by the system's knowledge base. Additionally, many of these exceptions are out of scope for the target audience. For instance, Socket exceptions are part of this library. Novice students are extremely unlikely to be working with sockets.

Additionally, none of the target students were at a level where they were writing their own exception handlers – most of them had not yet even learned try/catch.

Novice level logic errors were not, however, as well represented with the FindBugs library. FindBugs missed many logic errors that novices tends to code often make, probably because FindBugs targets a more expert programmer. In order to address this problem, two different ideas were considered: augment Findbugs with new patterns representing novice level logical defects, or build a different form of logical analysis into the system. The latter approach was chosen, developing a separate module of logical error assistance. This part of ITS-Debug compared the student's solution to the known solution for a given exercise. In the presence of a logical error the student received assistance indicating roughly how close or far they were from the known solution. If the student still had a logical error after receiving the rough estimate of correctness, they then received a comparison indicating at a high level what they were missing or had included unnecessarily in their solution. These new levels of assistance are described in more detail in Chapter 4.

This approach was specifically chosen in order to supplement the use of CBR. While CBR is a valid approach for the domain, as the results of evaluation show, it started to become clear during initial evaluation efforts that as the system progresses in the future it will need to incorporate other forms of reasoning. Specifically, if the system were to progress to assist students dealing with more complex programming (class design, generics, inheritance, etc.), it may need to incorporate some model-based reasoning in order to help students reason more deeply about these problems. The new

logical analysis included in the Pedagogical Module begins to explore the incorporation of model-based reasoning into the tutoring system. These new levels of analysis were present in the system during the high school evaluations performed this past semester.

Additionally, questions about student modeling were posed. The current student module models the student in terms of what exercises the student has seen, what cases the student has seen, what exercises and cases the student has successfully solved, and what the topic coverage is of the student's experience within the system. The Pedagogical Module uses this data to help determine how to teach the student and what exercise to serve back to the student. Some of the reasoning with this module was not utilized in the most recent evaluation due to the nature of the evaluation process chosen – the students needed to concentrate on one topic at a time for this version of the evaluation. The system does incorporate a mechanism for completely controlling the student's navigation through the topics the system is able to present. Improvement and evaluation of this aspect of the system is considered future work. However, as with the domain module, the methodology chosen in this research is considered valid at this time due to the significance of the obtained t-test results for the pretest and posttest, in addition to the statistically significant correlations found regarding students' mean attempts required to complete an exercise.

Another aspect of the research questions pertaining to Intelligent Tutoring Systems involved reasoning about student solutions without incurring the full program verification problem. This question was revisited a couple of times during the course of this research program. The combination of CBR and having every exercise have at

least one known solution was used to address this concern originally. And for syntax, runtime, and some logical errors this worked very well, as exhibited by the results for phases 1 and 2. Later, it became apparent that FindBugs needed to be augmented to help the novice with Logical defects – during initial trials it was noticed that certain logical defects novices are prone to coding were not being picked up by the FindBugs system. ITS-Debug was then modified to perform a deeper analysis comparing the student solution to the known solution and determining where the student's solution differed. For Phases 1 and 2 this worked very well. The system was able to complete its analysis without causing a timeout and the student was able to receive extra assistance with logical defects. When the students were using Phase 3 however this part of the system started to slow down more dramatically and in some instances fail. Thus, though both of these forms of analysis avoid the NP-complete program verification problem, the end solution may still in certain scenarios be too computationally expensive for a client-server web application. Modifications to alleviate this computational expense are discussed as future work.

Another question pertaining to Intelligent Tutoring Systems was how should debugging issues be remediated? It was determined very early on in this research program that the system could benefit from including multiple learning styles in its ability to teach the domain. Because the lengthy surveys usually used to determine a student's learning style would not have worked well within the evaluation conditions, ITS-Debug takes a broad approach to employing a student's learning style in its presentation of remediations. Specifically, two types of learning styles are supported:

verbal learners and visual learners. ITS-Debug asks the student to identify at the start of their first session what kind of learner they believe they are. ITS-Debug then uses this learning modality for this student, unless the student alters his or her selection.

As with other tutoring systems, the remediations ITS-Debug provides are scaffolded. Students receive more information with subsequent assistance from the system. Visual students received visual cues and animations disambiguating the domain while verbal students received textual feedback.

When all students' data was analyzed as one data set, split by modality, attempts data recorded by the system for students that selected Verbal as their modality required significantly fewer attempts to complete an exercise. Informal feedback from some students who saw both forms of remediation also suggested that they preferred the verbal presentation. However, when students' data regarding time to complete an exercise is analyzed, the opposite result is obtained – students who selected Visual as their modality required significantly less time to complete an exercise ($p<.01$) than those who selected Verbal ($p<.05$). It is believed that this difference (if it is not skewed by previously discussed issues with timing data) may in part be due to the fact that although the Verbal assistance may have been more effective in teaching the domain and therefore helping the student to require less attempts to finish an exercise, the Visual assistance may have more obviously indicated to students what they needed to change. The first two levels of Visual assistance draw the student's eye immediately to the lines of code that could be affected, through the use of animated line highlighting. The student may be able to process the Visual information more quickly and, although

they are using more attempts, they may be reaching the solution in less actual time than their Verbal counterparts. Future analysis may be merited to refine this comparison.

The last relevant question regarding contributions to Intelligent Tutoring Systems involved determining whether or not a given remediation was successful. This question was originally more relevant to selecting the student's learning style on the fly. When this functionality was included, the student would indicate that they would like the system to determine what kind of learner they were from their interactions with ITS-Debug. It would then observe the student and randomly provide different remediations. In this scenario ITS-Debug would start by oscillating randomly between the two available learning styles and assigning a weighting to each style. Initially, both learning styles were assigned a weight of 0. This weight would increase or decrease as the student used the system and responded positively or negatively to a given remediation. If a given remediation resulted in a correct answer to an exercise, this learning style received an increase in weighting. If a given remediation resulted in an incorrect answer to an exercise the weighting for that learning style was reduced. The oscillation would continue until one learning style received a weighting greater than 60%. Once one of the learning styles reached this threshold, the student would remain in the system as this type of learner unless they returned to the modality selection screen and selected a different learning style. Due to a unanimous decision of the dissertation committee, this aspect of the system has not yet been formally evaluated. However, when this aspect of the system is eventually evaluated, it is anticipated that the students would

better receive this methodology than a lengthy survey tied to a given learning style model.

6.2.1.1 Acquisition of Domain Knowledge

ITS-Debug records every solution the student creates for every exercise the system presents. Subsequently (after the student has finished working with ITS-Debug), a separate module analyzes all of the solutions, comparing them to the original known solution for the exercise. Each exercise represents one defect, with few exceptions. Each solution is compared to its related exercise. If the student coded a new solution to the exercise, the original defective code for the exercise is analyzed to determine to which case it corresponds. The student's solution is then distilled into the ITS-Debug meta-language. If the original case did not have a solution, the new solution is added to the case. Otherwise, a new case is created using the student's new solution.

The system's performance for the case acquisition task is encouraging, while also suggesting avenues for further improvement. The current implementation assumes that each solution contains one change. This assumption will not always hold—some defects may involve multiple changes, especially certain logical defects. Additionally, as ITS-Debug matures its representation language may become more complicated. Evolution of this language may present opportunities to glean additional useful information from student solutions. Finally, it is possible that a different approach would yield better/more exact results. The module currently compares the student's solution to the system's original solution and determines which nodes of the student's code least match the known solution. The least match is the item identified as the

affected construct. This approach was chosen when the researcher realized that identification of the affected construct could be viewed as the opposite of a traditional pattern recognition problem—the element farthest away from the known elements is what is desired, not the closest match to an existing model of the solution. Although the current implementation of the algorithm uses this approach, there may be a better algorithm for solving this problem or further refinement of the solution utilized here. Continued improvement of the case acquisition module is an item for future work.

6.2.1.2 Exercise Generation

As discussed earlier in Chapter 4, the original base system was augmented with an Exercise Generator in order to help remedy the fact that Phase 1 had a fixed database of exercises that students would eventually complete. The Phase 2 Exercise Generator was built to generate exercises over many Java language constructs. The system, as discussed in Chapter 5, was evaluated on Elementary Programming exercises alone (as necessitated by the students using the system) and performed very well. The results obtained from the by phase t-test analysis discussed earlier and the analysis of attempts required to complete an exercise prove that the system is successfully capable of generating relevant Elementary Programming exercises on the fly. Further details about exercises the exercise generator is capable of generating appear in Chapter 5.

6.2.2 Computer Science Education – Teaching Debugging Skills to Novices

The question that was posed under this category at the beginning of this work was how should debugging be taught to novices. Two methods were chosen to answer this question. Phases 1 and 2 exemplify the first method – show the students defective

code they did not write and assist them in the debugging process. The second method presents students with a system capable of assisting them with defect in any program they might write in the system (Phase 3).

The results of evaluating Phases 1 and 2 show that the first method is indeed a valid way to teach novices to debug programs. However, to answer the question as it was originally posed – how **should** debugging be taught to novices? – a comparative analysis not completed yet would need to be performed, implementing various teaching methods researched and comparing the results of each of those methods. In this research the first method is supported by statistically significant results obtained from evaluation. The results from Phase 3 are inconclusive and would require further evaluation in future work.

The background research performed for this dissertation described several different approaches to teaching the domain and building systems to teach the domain. Classes instructed students on best practices in software development and design and had students practice with broken programs and take handwritten logs. Some systems decomposed the problem by bug type (syntax, runtime, logical) and provided targeted assistance for the chosen type. Other systems were developed as Intelligent Tutoring System Components (PROUST) or full ITS systems. Each of these systems targets a subset of the domain when what the student really needs is a more all purpose assistance solution. Students are going to code all three types of errors, often multiple errors from each of these types within the same program. They need a single system capable of handling syntax, runtime, and logical errors over a broad range of defects

possible within the programming language they are using. ITS-Debug aims to be that one stop assistance solution for the student, and Case Based Reasoning helps the system to achieve this end. Although future improvements and evaluation are warranted, the current implementation has proven its effectiveness during the evaluation period and during this period all three error types were encountered within the system during live evaluation. Future work could help the system to evolve into its full potential as an overarching resource for the novice computer science student.

7 Future Work

7.1 Intelligent Tutoring Systems

Future directions for the Student Module were identified during the course of the implementation and evaluation. This module evolved as the system was built and many different design choices were reviewed. The other design that was considered involved using Pattern Recognition to map students back to knowledge levels as described in Chmiel and Loui's work [9]. Although a very interesting research question, a simpler model sufficed for this particular research program. In the future it would be interesting to consider how to create a Student Module using Pattern Recognition. The researcher does not at this time know of an ITS that uses Pattern Recognition techniques to help model students.

The Pedagogical Module also contains room for improvement. As discussed in Chapter 6, the new logical analysis introduced while developing Phase 3 is sometimes too computationally expensive for a web application that relies heavily on server-side computation. Two solutions could improve the analysis process: shifting from a client-server web based solution to a desktop application implementation or moving more of the computation and analysis to a client-side code library. The latter approach would still preserve the advantages of a web-based application.

Additionally, a few items were noticed as possible future work for the exercise generation system. ITS-Debug was evaluated solely on its ability to generate exercises under the topic of Elementary Programming. It would be interesting to determine through further evaluation how well the Exercise Generator covers other topics and how

to increase the abilities of the Exercise Generator. For instance, the Generator is capable of generating exercises over a subset of the topic the system is able to represent. It would be interesting to determine how to generate novel exercises involving defects in class development and design. The fact that this domain is more open and difficult to reason about would make the Exercise Generator itself more interesting.

7.2 Future Work Indicated From Evaluation Results

The evaluation of Phases 1 and 2 showed a direct correlation between the time spent practicing with the system and the number of attempts and the length of time required to complete a given exercise for high school students. The college group alone did not show a significant correlation. As discussed earlier this is believed to be in part due to the original design of the experiment. It is believed that if the college students had used the system under circumstances similar to those the high school students experienced that a significant correlation would also have been found with the college student population. It is also believed that if more college students had participated this would also have positively affected the results. This further evaluation is left as future work.

For Phase 2, future work includes evaluating this aspect of the system over other topics. Evaluation of this phase was limited, due to environmental constraints, to Elementary Programming alone. It would be interesting to determine how well the exercise generation system is able to handle the demands of the students over other topics.

And, as discussed earlier, it may be beneficial to perform a future evaluation session with an improved implementation of Phase 3 with more students. One of the eventual goals of this research has been to provide students with a system that could provide them with assistance on programs they are being asked to write. In some ways, the system is in fact ready to provide this form of assistance. However in the presence of certain student issues in the more open environment of Phase 3 the system could benefit from further improvements before performing this proposed future evaluation.

7.3 Postponed Questions

Three of the original questions were postponed as out of scope and three more were discovered while performing the research for this system. It is the researcher's intention to eventually pursue these remaining questions and whatever other questions are discovered as work with the system continues. The following discusses the postponed research questions:

7.3.1 How do male and female students compare when using the system? Is there an increased benefit to using the system for one group as opposed to another?

This question was originally posed during the depth study preceding this dissertation. One of the main goals behind this system is to provide a support mechanism for when students become stuck on a programming exercise. It follows that students without peer support or external resources may become more profoundly stuck than students who have someone to turn to or work with. Given that there is still a

gender disparity in the study and pursuit of Computer Science, it stands to reason that perhaps a system like this, that provides in situ and on demand assistance, would stand to be of greater assistance to students in the minority. This line of research was not pursued due to confidentiality concerns – given that there are still so few female students, discussion of results on gendered differences would have inadvertently been traceable to individual students. Because of these concerns it was deemed more important to vet the system first, then pursue research into the impact the tool might have on different gender/minority groups as future work.

7.3.2 Should the system support other languages? How would supporting other languages change the system?

Currently the answer to this question appears to be a strong "yes." Although the question was postponed it became apparent that the novice students at many schools are starting their computer science education with Python, Ruby, and VB.NET. In order to be able to assist as many students as possible going forward this is considered a very important question for future pursuit and may in fact end up being the next question pursued for this research.

It is believed at this time that other languages could be supported by this system if the analogous compilers/interpreters and runtime systems allow for an external system to gather information about these steps in the programming process and then reason about them in the same manner as the current system. It is believed that there will be correlations between the messages produced by the systems of these other languages and the messages produced by the Java compilation and runtime

environments. If this is true it may be possible to redesign the underlying architecture to take advantage of these analogies between different languages and perhaps make use of this information in assisting the student. When encountering a problem using a new programming language being able to correlate the new issue with an issue that was encountered with a previous programming language can be helpful.

7.3.3 Does peer assistance factor in to modeling the student? How? And should the system facilitate peer communication?

Supporting peer assistance in the classroom is a known educational technique, especially in educational environments that include peer programming. Originally, the researcher intended to include a message board so that students would be able to assist each other anonymously. It would be interesting in the future to revisit this question and determine if there is a better way to facilitate per assistance and how to analyze its effectiveness.

7.3.4 Support for personae?

Personae were originally discussed as a possible mechanism for discussing the domain with students. It is unclear at this time whether or not personae would be an improvement to this system. Student feedback did not indicate that they would like more personalized, human like assistance. They indicated that they would like more direct / clearer assistance. It is unclear at this time whether the pursuit of this question is worthwhile for this research.

7.3.5 Should further levels of assistance be introduced?

During the evaluation period a few students indicated that they would like more direct assistance from the feedback mechanism. There is a fine line here between assisting the student to find the answer themselves and giving the student the answer. It is believed at this time that the system could benefit from further levels of assistance and a more intelligent selection of what assistance to provide to the student in what situation.

7.3.6 Could the Student Module be realized via pattern recognition?

During the design phase of the system built for this research, a different Student Model was planned out that would require the use of pattern recognition to map students back to competence levels as described in Chmiel and Loui's work [9]. There is no system known to the researcher at this time that utilizes pattern recognition in this manner, this may be a worthwhile future research direction but requires further background research in order to even determine if it is in fact a unique research area.

7.4 Conclusion

The driving goal behind this research was to create a system that could successfully teach the novice student about the domain of program debugging and provide a significant contribution to both Intelligent Tutoring Systems research and Computer Science Education research. No Intelligent Tutoring System to date has been built to deal with the domain in this manner, namely using Case Based Reasoning. In addition, very few systems have been built to specifically assist students in learning how to debug programs. All of the background research, design, implementation,

evaluation, and results have come together and produced an ITS like no other at this time – a system that has been proven to help teach debugging to novices in a general sense, that can produce exercises within this generalized domain on the fly, and that utilizes Case Based Reasoning and Static Program Slicing to achieve these ends. Furthermore, there are still many interesting research questions and future system improvements to pursue. It is the intention of this researcher to continue this research into the future, eventually bringing the system to its full and final potential, and providing students with a legitimate source of debugging assistance that they can access when and where they need it without being concerned that they will be judged for not knowing something that they have likely never been taught.

8 Bibliography

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic Program Slicing. In *Proceedings of ACM SIGPLAN '90*. ACM, White Plains, NY. 246 – 256.

[2] Eric Allen, Robert Cartwright, and Brian Stoler. 2002. DrJava: a lightweight pedagogic environment for Java. In *Proceedings of the 33rd SIGCSE technical symposium on Computer science education* (SIGCSE '02). ACM, New York, NY, USA, 137-141.

[3] Anthony Allevato, Steven H. Edwards, Manuel A. Pérez-Quiñones. *Dereferee: Exploring Pointer mismanagement in Student Code*. SIGCSE 2009, March 3 – 7, Chattanooga, TN, pp. 137 – 177.

[4] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (ITiCSE '05). ACM, New York, NY, USA, 84-88.

[5] John R. Anderson, Albert T. Corbett, Kenneth R. Koedinger and Ray Pelletier: Cognitive Tutors: Lessons Learned. The Journal of the Learning Sciences 4(2), 167-207 (1995)

[6]. Albert T. Corbett, Kenneth R. Koedinger, John R. Anderson. Intelligent tutoring Systems. *Handbook of Human Computer Interaction, Second, Completely revised Edition.* M. Helander, T. K. Landauer, P. Prabhu (editors), Elsevier Science B. V., 1997. Chapter 37

[7] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, YuQuian Zhou. *Evaluating Static Analysis Defect Warnings on Production Software.* PASTE 2007. San Diego, CA.

[8] Peter Brusilovsky, Elmar W. Schwarz, Gerhard Weber. 1996. ELM-ART: An Intelligent Tutoring System on World Wide Web. In Proceedings of the Third International Conference on Intelligent Tutoring Systems (ITS '96), Claude Frasson, Gilles Gauthier, and Alan Lesgold (Eds.). Springer-Verlag, London, UK, 261-269.

[9] Ryan Chmiel and Michael C. Loui. 2004. Debugging: from novice to expert. In Proceedings of the 35th SIGCSE technical symposium on Computer science education (SIGCSE '04). ACM, New York, NY, USA, 17-21.

[10] Albert T. Corbett, Kenneth R. Koedinger, and John R. Anderson (1997). Intelligent tutoring systems (Chapter 37). M. G. Helander, T. K. Landauer, & P. Prabhu, (Eds.) *Handbook of Human- Computer Interaction*, 2nd edition.

Amsterdam, The Netherlands: Elsevier Science

[11]Albert T. Corbett, John R. Anderson, and Eric J. Patterson. Problem compilation and tutoring flexibility in the Lisp tutor. In Proceedings ITS'88: Intelligent Tutoring Systems, pages 423--429, 1988.

[12] Adair Dingle and Carol Zander. 2000. Assessing the ripple effect of CS1 language choice. In Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference. Consortium for Computing Sciences in Colleges, USA, 85-93.

[13] Stephen N. Freund, Eric S. Roberts. *Thetis: An ANSI C Programming Environment Designed for Introductory Use*. SIGCSE 1996.

[14] Nicolas Guibert, Patrick Girard, and Laurent Guittet. 2004. Example-based programming: a pertinent visual approach for learning to program. In *Proceedings of the working conference on Advanced visual interfaces* (AVI '04). ACM, New York, NY, USA, 358-361.

[15] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '86), Marilyn Mantei and Peter Orbeton (Eds.). ACM, New York, NY, USA, 171-174.

[16] M. Helander, T.K. Landauer, P. Prabhu (Eds). 1997. *Handbook of Human-Computer Interaction, Second, Completely Revised Edition*. Elsevier Science B. V.

[17] W. Lewis Johnson and Elliot Soloway, PROUST: Knowledge-Based Program Understanding. *Software Engineering, IEEE Transactions on* , vol.SE-11, no.3, pp. 267- 275, March 1985

[18] Andrew J. Ko, Brad A. Myers. 2010. Extracting and answering why and why not questions about Java program output. ACM Trans. Softw. Eng. Methodol. 20, 2, Article 4 (September 2010), 36 pages.

[19] Viswanathan Kodaganallur, Rob R. Weitz and David Rosenthal. 2005. *A Comparison of Model-Tracing and Constraint-Based Intelligent Tutoring Paradigms*. IJAIED, Vol. 15, No. 2 (2005), pp. 117-144.

[20] Michael Kölling and John Rosenberg. 2001. Guidelines for teaching object orientation with Java. In *Proceedings of the 6th annual conference on Innovation and technology in computer science education* (ITiCSE '01). ACM, New York, NY, USA, 33-36.

[21] Michael Kolling. Teaching Object Orientation with the Blue Environment. Journal of Object Oriented Programming, 12(2), May 1999, 14--23.

[22] Amruth N. Kumar. Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors, Technology, Instruction, Cognition and Learning (TICL) Journal, Sept 2006.

[23]Amruth N. Kumar. 2002. Model-Based Reasoning for Domain Modeling in a Web-Based Intelligent Tutoring System to Help Students Learn to Debug C++ Programs. In *Proceedings of the 6th International Conference on Intelligent Tutoring Systems* (ITS '02), Stefano A. Cerri, Guy Gouardères, and Fábio Paraguaçu (Eds.). Springer-Verlag, London, UK, UK, 792-801.

[24] Greg C. Lee , Jackie C. Wu, Debug it: a debugging practicing system, Computers & Education, v.32 n.2, p.165-179, Feb. 1999

[25] Robert F. Mathis. 1974. Teaching debugging. In *Proceedings of the fourth SIGCSE technical symposium on Computer science education* (SIGCSE '74). ACM, New York, NY, USA, 59-63. DOI=10.1145/800183.810443

[26] Robert McCartney, Anna Eckerdal, Jan Erik Mostrom, Kate Sanders, and Carol Zander. 2007. Successful students' strategies for getting unstuck. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education* (ITiCSE '07). ACM, New York, NY, USA, 156-160

[27] Michael Mayo and Antonija Mitrovic (2001). Optimizing ITS behavior with Bayesian networks and decision theory. International Journal of Artificial Intelligence in Education, 12, 124-153.

[28] Sally H. Moritz, Fang Wei, Shahida M. Parvez, and Glenn D. Blank. 2005. From objects-first to design-first with multimedia and intelligent tutoring. *SIGCSE Bull.* 37, 3 (June 2005), 99-103. DOI=10.1145/1151954.1067475

[29] Christian Murphy, Eunhee Kim, Gail Kaiser, and Adam Cannon. 2008. Backstop: a tool for debugging runtime errors. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education* (SIGCSE '08). ACM, New York, NY, USA, 173-177.

[30] Christian Murphy, Gail Kaiser, Kristin Loveland and Sahar Hasan. *Retina: Helping Students and Instructors Based on Observed Programming Activities.* SIGCSE 2009.

[31]Elizabeth Odekirk-Hash and Joseph L. Zachary. 2001. Automated feedback on programs means students need less help from teachers. In *Proceedings of the thirty second SIGCSE technical symposium on Computer Science Education* (SIGCSE

'01). ACM, New York, NY, USA, 55-59.

[32] Shahida M. Parvez. 2005. Title. Ph. D. Dissertation. Lehigh University, Bethlehem, PA.

[33] Stuart Russel and Peter Norvig. 2003. *Artificial Intelligence – A Modern Approach*. Pearson Education, Inc. Upper Saddle River, NJ.

[34] Leen-Kiat Soh and Todd Blank. 2008. Integrating Case-Based Reasoning and Meta-Learning for a Self-Improving Intelligent Tutoring System. Int. J. Artif. Intell. Ed. 18, 1 (January 2008), 27-58.

[35] J. Viega, J.T. Block, T. Kohno, G. McGraw. *ITS4: A Static Vulnerability Scanner for C and C++ Code.* ACSAC 2000, December 11 – 15, New Orleans, LA, pp. 257 – 267.

[36] Gerhard Weber, A. Mollenberg. ELM-PE: A Knowledge-based Programming Environment for Learning LISP. In *Educational Multimedia and Hypermedia (ED-MEDIA 1994).* Vancouver, British Columbia, Canada. June 25-30, 557 – 562.

[37]Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering* (ICSE '81). IEEE Press, Piscataway, NJ, USA, 439-449.

[38] Mark Weiser. 1982. Programmers use slices when debugging. *Commun. ACM* 25, 7 (July 1982), 446-452.

[39] David Williams-King, John Aycock, Daniel Medeiros Nunes de Castro. *Enbug: When Debuggers Go Bad*. ITiCSE 2010, June 26-30, Bilkent, Ankara, Turkey.

[40] Beverly P. Woolf. 2009. *Building Intelligent Interactive Tutors – Student-centered strategies for revolutionizing e-learning.* Morgan Kaufmann Publishers, Boston, MA.

[41]Xiaohong (Sophie) Wang and Joshua Souders. 2012. Improving debugging education through applied learning. *J. Comput. Sci. Coll.* 27, 3 (January 2012), 138-145.

[42]Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (March 2005), 1-36.

[43] Andreas Zeller. *Why Programs Fail*. Elsevier, New York, 2009.

[44] Uta Ziegler and Thad Crews. 1999. An integrated program development tool for teaching and learning how to program. *SIGCSE Bull.* 31, 1 (March 1999), 276-280.

9 Appendix

Appendix A: Pre Study Survey:

1. Have you ever taken a programming course before?
   Yes No

2. Have you tried to learn programming outside of a course or within a course on some other subject?
   Yes No

3. If you answered "Yes" to questions 1 or 2, what language(s) did you learn? (select all that apply)
   Java C++ Python Visual Basic .NET C#.NET Other (please list below)

Appendix B: End of Study Survey:

1. Please select "Yes" or "No": I will be able to apply concepts learned in the system to programs I write outside of the system:
   Yes No

2. Please select "Yes" or "No":Concepts I learned in the system are applicable to my coursework:
   Yes No

3. On a scale of 1 to 5 with 1 meaning "not at all" and 5 meaning "extremely": How helpful did you feel the system was in teaching you how to debug programs?
   1 2 3 4 5

4. On a scale of 1 to 5 with 1 meaning "not at all" and 5 meaning "extremely": How helpful do you think concepts taught by the system will be in debugging programs you wrote outside of the system?
   1 2 3 4 5

5. On a scale of 1 to 5 with 1 meaning "not at all" and 5 meaning "extremely": How helpful was the feedback the system gave for individual debugging problems?
   1 2 3 4 5

6. Is there something you would change about the system or how the system works? If so, please indicate any relevant information below.

Appendix C: Pre/Post Test

Please complete this survey to the best of your ability. This survey DOES NOT count towards your grade, nor are there any "right" or "wrong" answers. This survey IS NOT indicative of what your instructor(s) expect you to know nor will they be informed of how you personally answered this survey. This is COMPLETELY ANONYMOUS.

Pre Survey:

1. Please select "Yes" or "No":Have you ever taken a programming course before?
    a. Yes
    b. No

2. Please select "Yes" or "No": Have you tried to learn programming outside of a course or within a course on some other subject.
    a. Yes
    b. No

3. If you answered "Yes" to questions 1 or 2, what language(s) did you learn? select all that apply):
    a. Java
    b. C++
    c. Python
    d. Visual Basic
    e. C# .NET
    f. Other (Please list below):

Pretest:

1.
```
public class PreTest1
{
  public static void main(String[] args)
  {
    int a = 3, b = 4;
    int sum = 0;
    // add a and b together
    sum = a + b;
    // output the result
    System.out.println sum
  }
}
```

a. change the statement "int a = 3, b = 4" to "int a = 3; int b = 4;"

b. Change the statement System.out.println sum to System.out.println("sum");

c. Change the statement System.out.println sum to System.out.println(sum);

d. Change int sum = 0 to double sum = 5.4

e. I don't know

2.
```
public class PreTest2 {
  public static void main(String[] args)
  {
    double firstNumber = 5.4;
    int secondNumber = 3.5;
    // Multiply firstNumber by secondNumber
    // and output to the screen
    System.out.println(firstNumber *
      secondNumber);
  }
}
```

a. Change double firstNumber to float firstNumber

b. Change System.out.println to System.out.print

c. Change double firstNumber to int firstNumber

d. Change int secondNumber to double secondNumber

e. I don't know

3.
```
public class PreTest3
{
  public static void main(String[] args) {
    System.out.println("This will print ");
    System.out.println["some text to the
      screen"];
  }
}
```

a. Change the ( ) in the first output statement to [ ]

b. Change the [ ] in the second output statement to ( )

c. Change double quotes (") in the output statements to single quotes (')

d. Remove the .out from the output statements

e. I don't know

4.
```
public class PreTest4 {
  public static void main(String[] args)
  {
    int a = 2;
    int b = 5
    int result = a * b;
```

a. Change the ints to doubles

b. Add a semicolon to the end of int b = 5

c. Change result = a * b to

```
    System.out.println("a * b = "
      + result);
  }
}
```

result = a/b

d. Change the semicolons to colons

e. I don't know

5.)

```
public class PreTest5 {
  public static void main(String[] args)
  {
    int a = 6;
    int a *= 5;
    System.out.println("a = " + a);
  }
}
```

a. Change the ints to doubles

b. Change System.out.println ("a = " + a) to System.out.println(a)

c. change int a *= 5 to a *= 5

d. Change System.out.println to System.output

e. I don't know

1. What is debugging?
    a. The act of programming itself, where a person writes a program in order to accomplish some task or calculation
    b. The process of making changes to a program in order to obtain a working application
    c. The process of analyzing and modifying a program's code to isolate and correct an error
    d. The process of using an Internet search engine in order to diagnose and fix a program
    e. The process of editing and re-compiling and re-running an application to remove defects
    f. I don't know

2. If you were trying to solve a problem would you
    a. Trial and error: make a change and see if that fixed the problem. If not, make another change.
    b. Search: Using keywords from the problem, look for a solution on the Web.
    c. Help: ask a peer or instructor what they would do to solve this problem.
    d. Backward slice: starting from where things went wrong, look at *previous* lines of code to find the cause.
    e. Forward slice: starting from where things went wrong, look at *following* lines of code to find the cause.
    f. I don't know.

3. Error messages usually help programmers identify errors in:
    a. The program's structure (syntax).
    b. The program's behavior when it runs.
    c. The program's output (i.e., the output is not correct).
    d. a and b.
    e. a, b and c.
    f. b and c.
    g. I don't know.

Posttest

1.)
```
public class PostTest1 {
 public static void main(String[] args)
  {
    int a = 3, b = 4;
    int sum = 0;
    // add a and b together
    sum = a + b;
    // output the result
    System;out;println(sum)
  }
}
```

a. change the statement "int a = 3, b = 4" to "int a = 3; int b = 4;"

b. Change the statement System.out.println(sum) to System.out.println("sum");

c. Change the statement System;out;println(sum) to System.out.println(sum);

d. Change int sum = 0 to double sum = 5.4

e. I don't know

2.)
```
public class PostTest2 {
 public static void main(String[] args)
 {
    int firstNumber = 5.4;
    double secondNumber = 3.5;
    // Multiply firstNumber by
    // secondNumber
    // and output to the screen
    System.out.println(firstNumber *
      secondNumber);
  }
}
```

a. Change int firstNumber to double firstNumber

b. Change System.out.println to System.out.print

c. Change double firstNumber to int firstNumber

d. Change int secondNumber to double secondNumber

e. I don't know

3.)
```
public class PostTest3 {
 public static void main(String[] args)
 {
    System.out.println["This will print
     "];
    System.out.println("some text to
      the screen");
 }
}
```

a. Change the ( ) in the second output statement to [ ]

b. Change the [ ] in the first output statement to ( )

c. Change double quotes (") in the output statements to single quotes (')

d. Remove the .out from the output statements

e. I don't know

4.)
```
public class PostTest4 {
```

a. Change the ints to doubles

```
 public static void main(String[] args)
  {
    int a = 2;
    int b = 5:
    int result = a * b;
    System.out.println("a * b = " +
      result);
  }
}
```

b. Change the colon at end of int b = 5 to a semicolon

c. Change result = a * b to result = a/b

d. Change the semicolons to colons

e. I don't know

5.)

```
public class PostTest5 {
 public static void main(String[] args)
 {
   int b = 6;
   int b += 5;
   System.out.println("b = " + b);
 }
}
```

a. Change the ints to doubles

b. Change System.out.println("b = " + b) to System.out.println("a = " + a)

c. change int b += 5 to b += 5

d. Change System.out.println to System.output

e. I don't know

1. What is debugging?
    a. The act of programming itself, where a person writes a program in order to accomplish some task or calculation
    b. The process of making changes to a program in order to obtain a working application
    c. The process of analyzing and modifying a program's code to isolate and correct an error
    d. The process of using an Internet search engine in order to diagnose and fix a program
    e. The process of editing and re-compiling and re-running an application to remove defects
    f. I don't know

2. If you were trying to solve a problem would you
    a. Trial and error: make a change and see if that fixed the problem. If not, make another change.
    b. Search: Using keywords from the problem, look for a solution on the Web.
    c. Help: ask a peer or instructor what they would do to solve this problem.
    d. Backward slice: starting from where things went wrong, look at *previous* lines of code to find the cause.
    e. Forward slice: starting from where things went wrong, look at *following* lines of code to find the cause.
    f. I don't know.

3. Error messages usually help programmers identify errors in:
    a. The program's structure (syntax).
    b. The program's behavior when it runs.
    c. The program's output (i.e., the output is not correct).
    d. a and b.
    e. a, b and c.
    f. b and c.
    g. I don't know.

Appendix D: Topic Listing

ID	Topic Name	Associated Subtopics
1	Elementary Programming	1,2,3,4,5,6
2	Selections	7,8,9,10,11
3	Loops	12,13,14,15,16,17
4	Methods	18,19,20,21,22,23,24
5	Single Dimension Arrays	25,26,27,28,29,30
6	Multi-Dimensional Arrays	31,32,33,34
7	Objects and Classes	35,36,37,38
8	String	39,40,41,42
9	I/O	43,44,45

Appendix E: Subtopic Listing

ID	Subtopic Name	Description
1	Math	Issues with mathematical computations
2	Variables	Issues with individual variables
3	Identifiers	Issues with individual identifiers
4	Assignment	Issues involving the assignment operator
5	Primitives	Issues involving primitive values
6	Operators	Issues involving operators
7	Booleans	Issues involving boolean statements
8	If	Issues involving single If statements
9	If / Else	Issues involving If statements with Else statements
10	Nested If	Issues involving nested If statements
11	Switch / Case	Issues involving switch and case statements
12	While Loop	Issues involving the While Loop construct
13	Do While Loop	Issues involving the Do While Loop construct
14	For Loop	Issues involving the For Loop construct
15	For Each Loop	Issues involving the For Each loop construct
16	Loop Sentinels	Issues involving loop sentinels
17	Nested Loops	Issues involving nested loops
18	Defining Methods	Issues involving method definitions
19	Invoking Methods	Issues involving invoking methods
20	Returning Values from Methods	Issues involving returning values from methods
21	Handling Values Returned from Methods	Issues involving dropped or mishandled return values
22	Passing Arguments to Methods	Issues involving passing arguments in to methods
23	Overloading Methods	Issues involving overloading methods
24	Variable Scope	Issues involving a variable's scope
25	Declaring and Creating Single Dimensional Arrays	Issues involving declaring or creating single dimensional arrays
26	Single Dimensional Array Initialization	Issues involving the initialization of a single dimensional array
27	Accessing Single Dimensional Array Elements	Issues involving accessing single dimensional array elements
28	Traversing Single Dimensional Arrays	Issues involving single dimensional array traversal
29	Copying Single Dimensional Arrays	Issues involving copying single dimensional arrays
30	Sending Single Dimensional Arrays as Arguments	Issues involving sending single dimensional arrays as arguments to methods
31	Declaring and Creating Multidimensional Arrays	Issues involving creating and declaring multidimensional arrays
32	Accessing Multidimensional Array Elements	Issues involving accessing multidimensional array elements

33	Traversing Multidimensional Arrays	Issues involving traversing multidimensional arrays
34	Multidimensional Arrays as Arguments	Issues related to sending multidimensional arrays as arguments
35	Constructors	Issues involving constructors
36	Object Variables	Issues involving objects variables
37	The '.' Operator	Issues involving the '.' operator
38	Members	Issues involving class members
39	Using Strings	Issues involving String variables
40	Using String Functions	Issues involving using String functions
41	Using Printf	Issues involving the use of printf
42	Using Format Strings	Issues involving the use of Format Strings
43	Using PrintWriter	Issues involving using PrintWriter to output data
44	Using Scanner	Issues involving using the Scanner class
45	Using Buffered Reader	Issues involving the use of Buffered Reader
46	Try	Issues involving Try blocks
47	Catch	Issues involving Catch blocks
48	Finally	Issues involving Finally blocks
49	Miscellanious Syntax	Catch-all for syntax errors not covered by other subtypes
50	Miscellanious IO	Catch-all for I/O errors not covered by other subtypes
51	Type System	Abuses of the type system
52	Infinite Loop	Infinite Loop

Appendix F: Discussion of Parse Tree Data and Sample Tree

The following shows a sample program and its parse tree data as generated by javac and the connector module. The numbering on the tree indicates the order in which the connector module visited each node in the parse tree generated by javac, with 1 representing the first node visited. The selected code for this example is the answer for one of the phase 3 exercises. The answer code was selected in order to generate a correct parse tree. Please note, the javac compiler automatically creates a default constructor called <init> , this method is included in the sample parse tree for the sake of completeness.

```
public class StatementExercise_C2
{
    // program is meant to perform some math
    // and then output the result to the screen

    public static void main(String[] args)
    {
        int x = 3;
        int y = 4;
        x = 2*y+5;
        System.out.println("2 * 4 + 5 = " + x);
    }
}
```

Figure 41: Sample code, used to generate parse tree

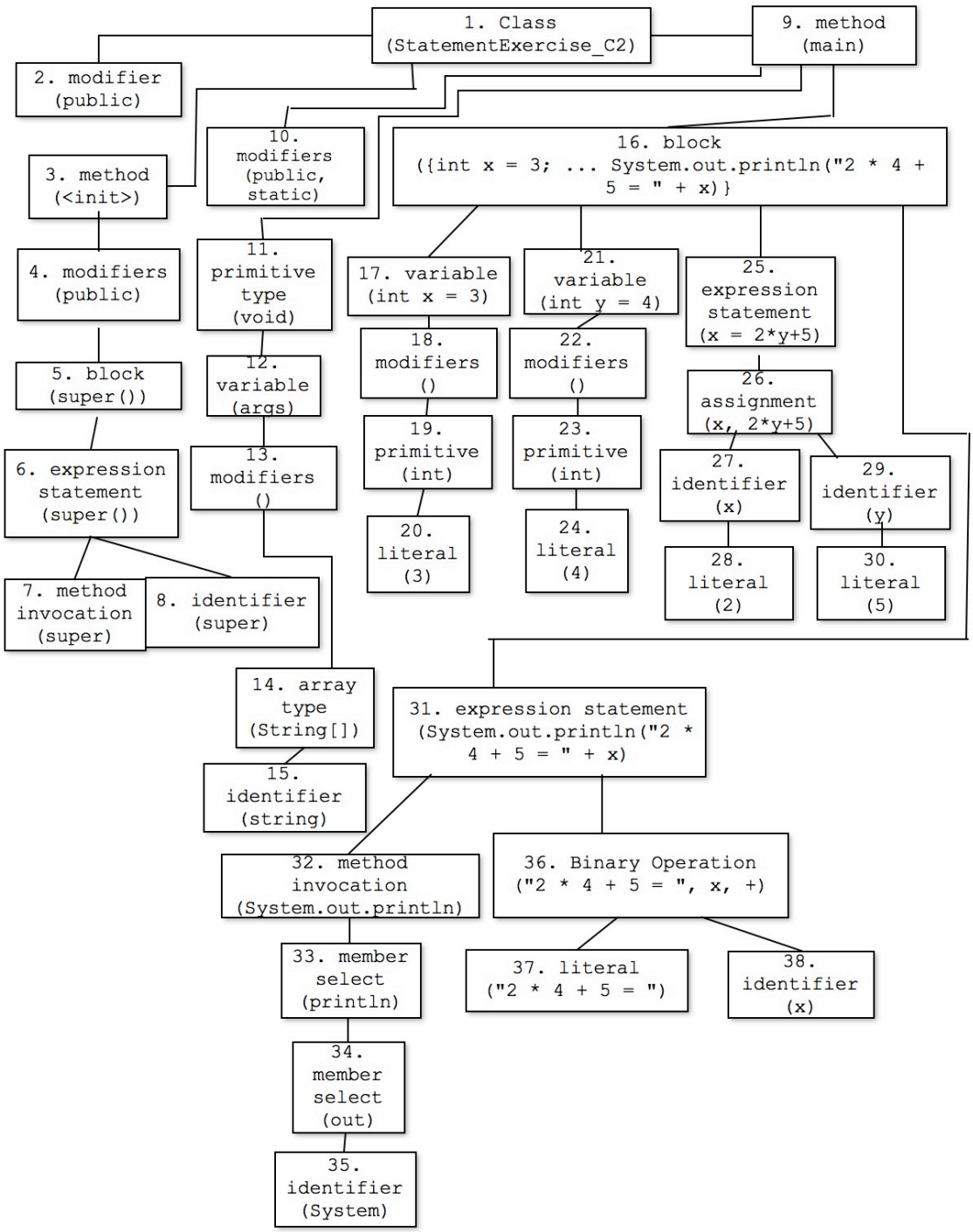
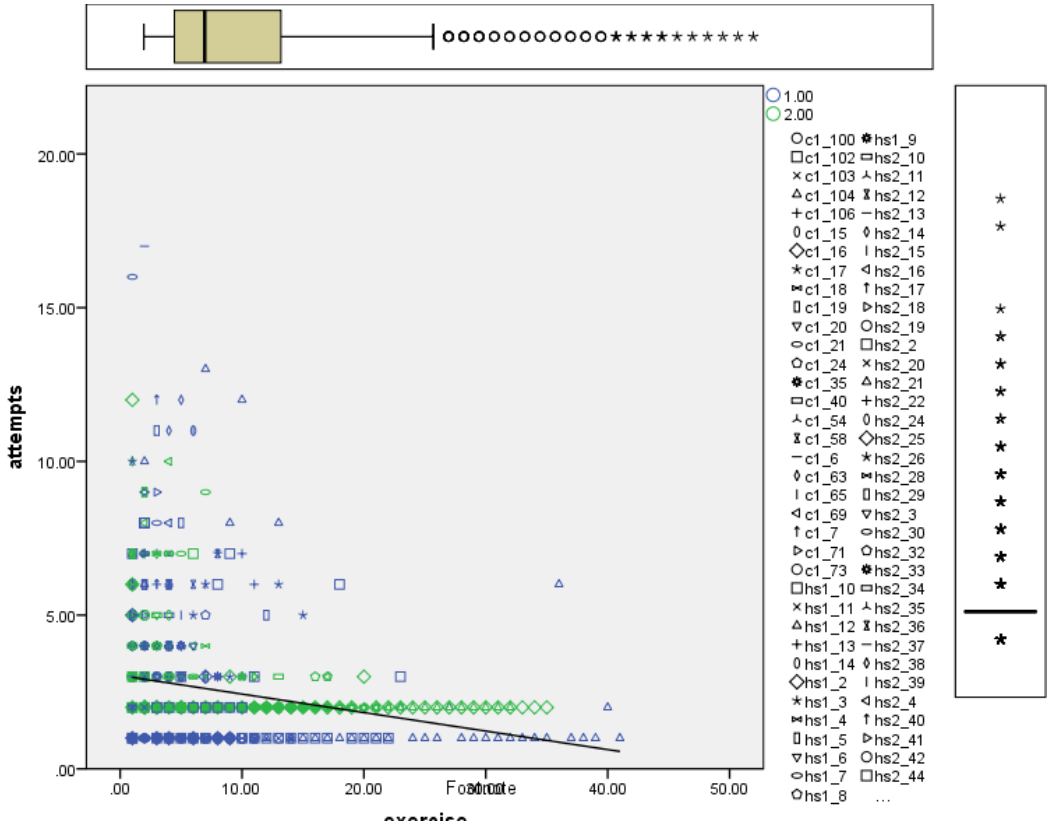


Figure 41: Parse Tree

Appendix G: Analyzing each students' required attempts for each completed exercise

The following presents scatter charts for each subset of the data relevant to this study, comparing each students' attempts for every completed exercise by dividing the data up into different subsets of the collected data. Each chart contains regression lines depicting the general direction of the data, with some subsets having two scatter charts: one that contains a regression line for all participants in the subset and one that contains 2 regression lines for all participants in each phase of the subset. Phases are additionally depicted separately as their own subsets, additionally broken down into subsets by range of exercises completed. Finally, each scatter chart that has one regression line also is accompanied by the results of the Pearson's Correlation over the data set indicated to determine if a significant correlation exists between the number of exercises completed and the number of attempts required to complete an exercise. As was shown in Chapter 5 when the data was evaluated at the means, we have similar results here: the more exercises students complete, the less attempts are required to complete an exercise. In addition to the t-test results also discussed in Chapter 5, these results are taken as further proof that students using the system did indeed learn how to debug the problems being presented to them while using the system.

Subset: Everyone



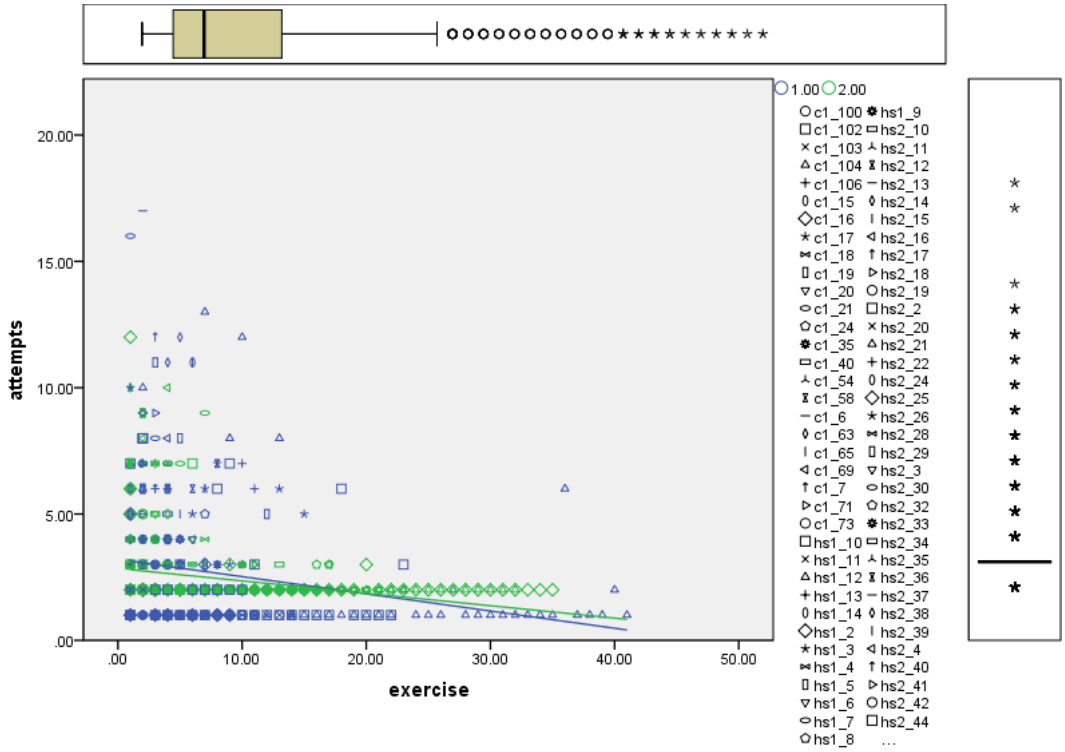
Pearson Correlation:

Correlations

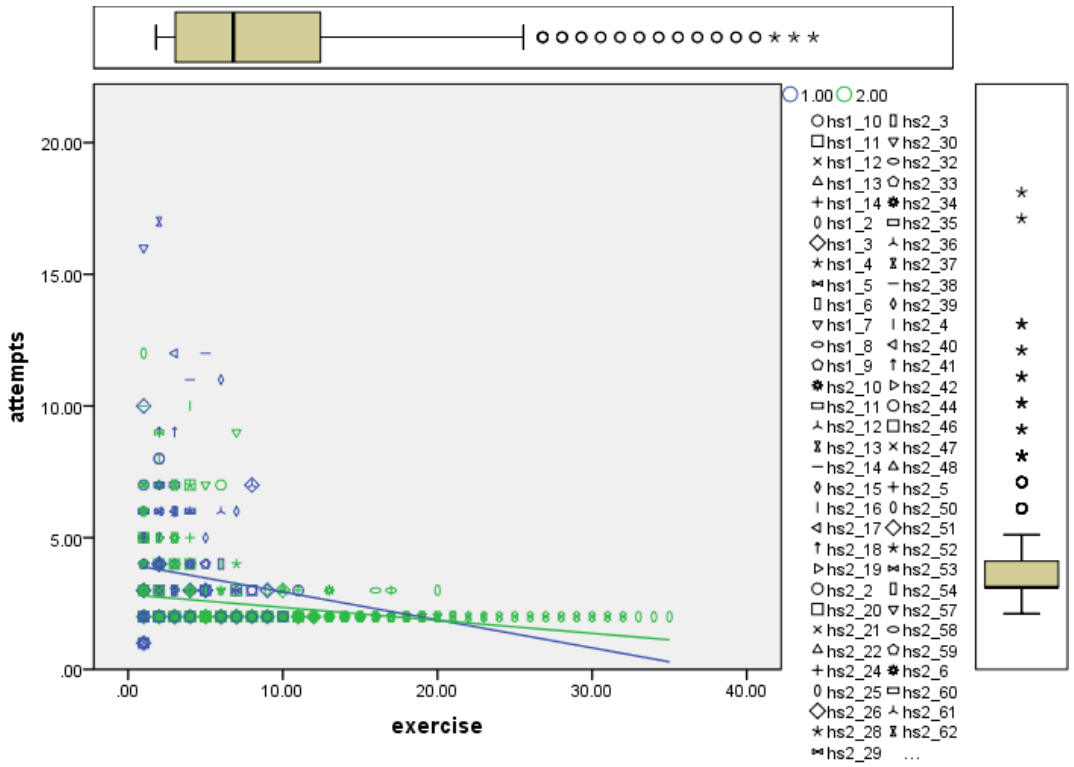
		exercise	attempts
exercise	Pearson Correlation	1	-.217**
	Sig. (2-tailed)		.000
	N	913	913
attempts	Pearson Correlation	-.217**	1
	Sig. (2-tailed)	.000	
	N	913	913

**. Correlation is significant at the 0.01 level (2-tailed).

Subset: Everyone, Regression Lines by Phase



Subset: All High School Students



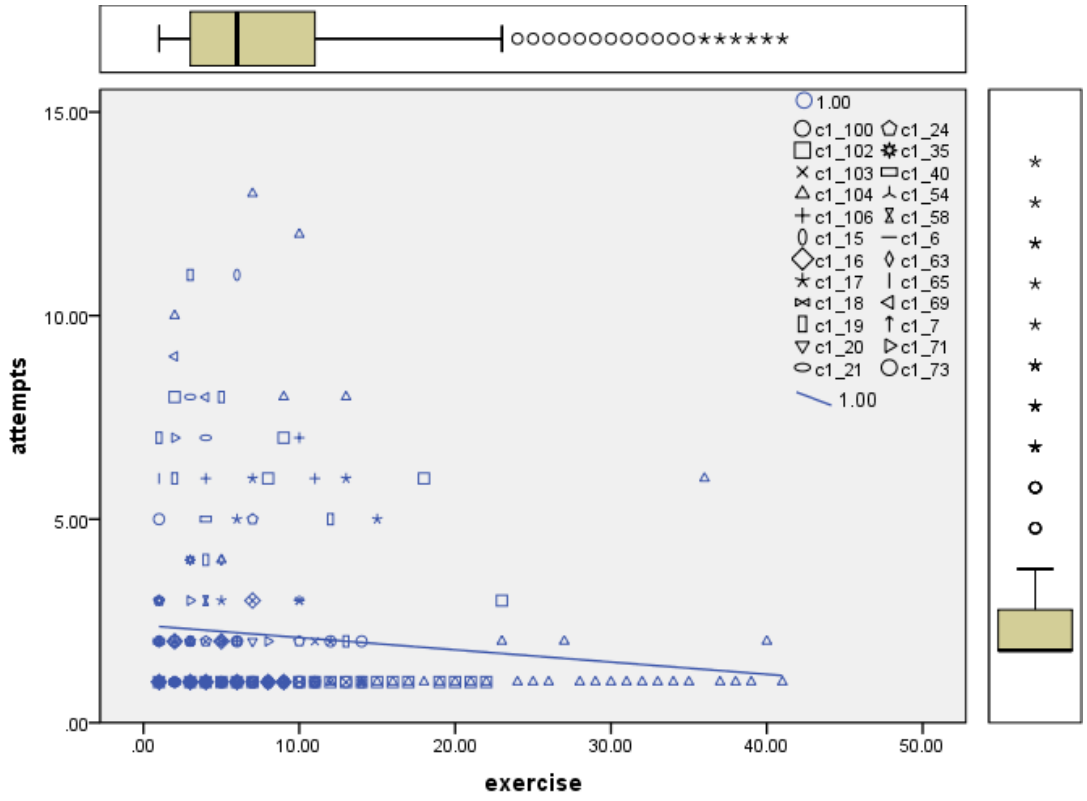
Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.259**
	Sig. (2-tailed)		.000
	N	668	668
attempts	Pearson Correlation	-.259**	1
	Sig. (2-tailed)	.000	
	N	668	668

**. Correlation is significant at the 0.01 level (2-tailed).

Subset: All College Students

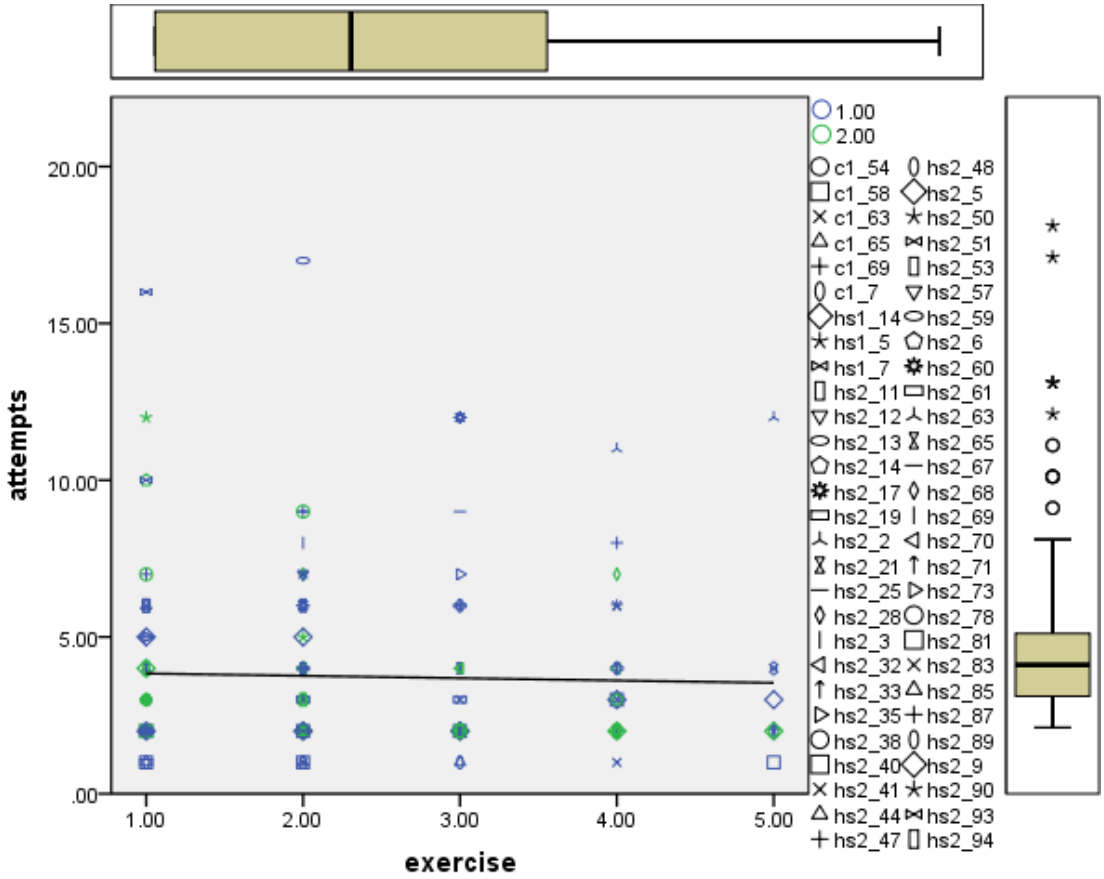


Pearson Correlation

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.114
	Sig. (2-tailed)		.076
	N	245	245
attempts	Pearson Correlation	-.114	1
	Sig. (2-tailed)	.076	
	N	245	245

Subset: Everyone, 1 to 5 exercises completed

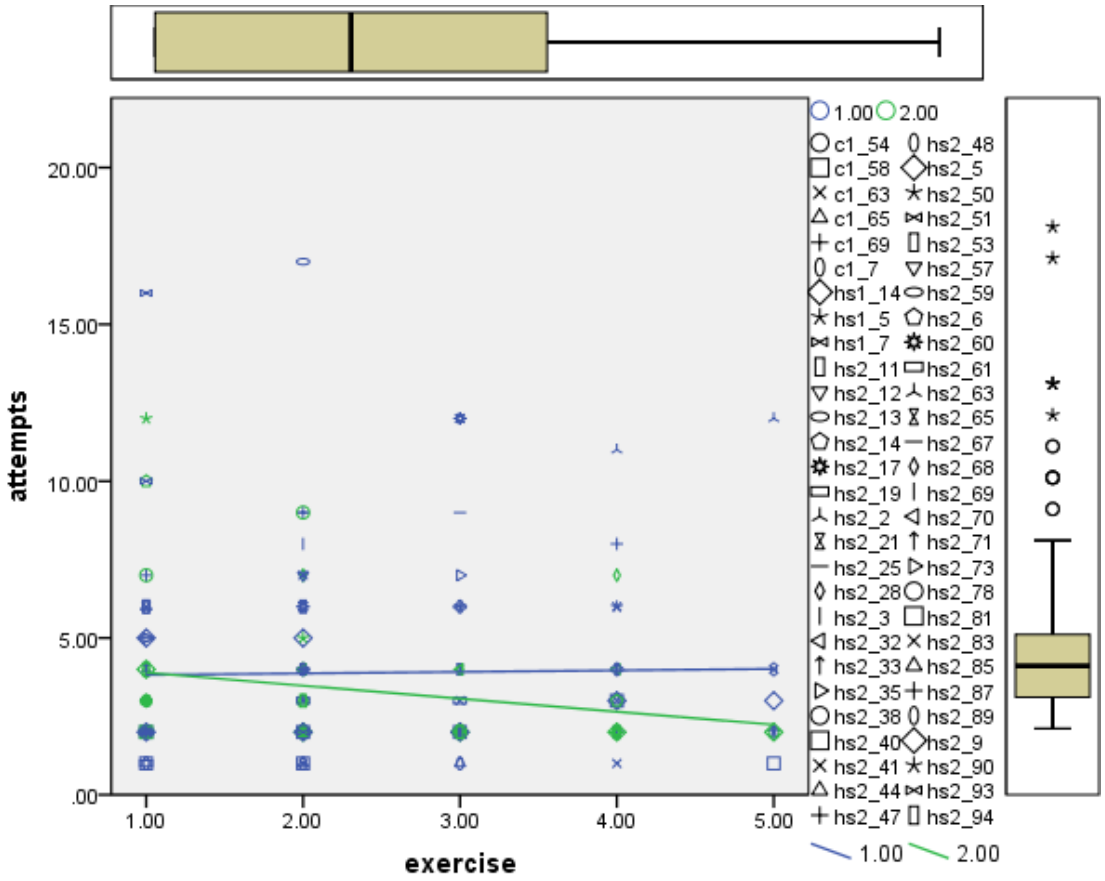


Pearson Correlation:

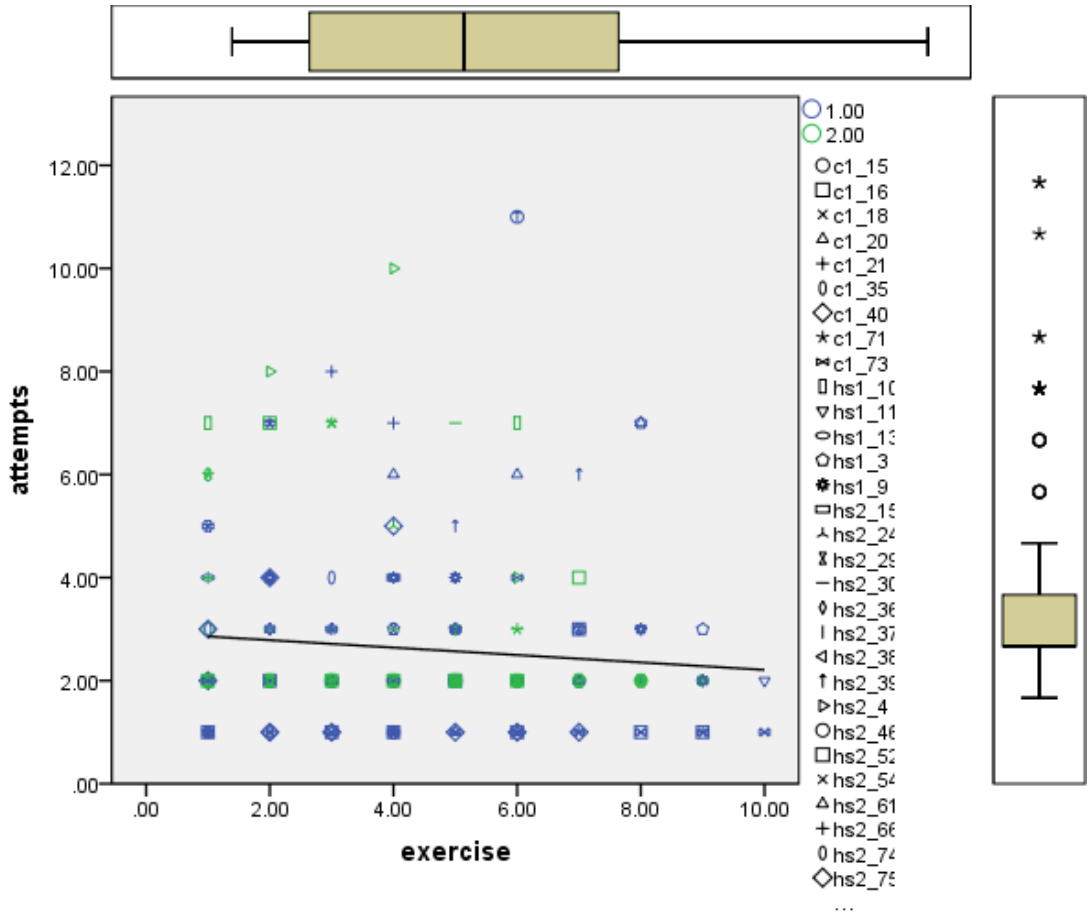
Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.033
	Sig. (2-tailed)		.674
	N	164	164
attempts	Pearson Correlation	-.033	1
	Sig. (2-tailed)	.674	
	N	164	164

Subset: Everyone, 1 to 5 exercises completed, Regression lines by phase



Subset: Everyone, 6 to 10 exercises completed

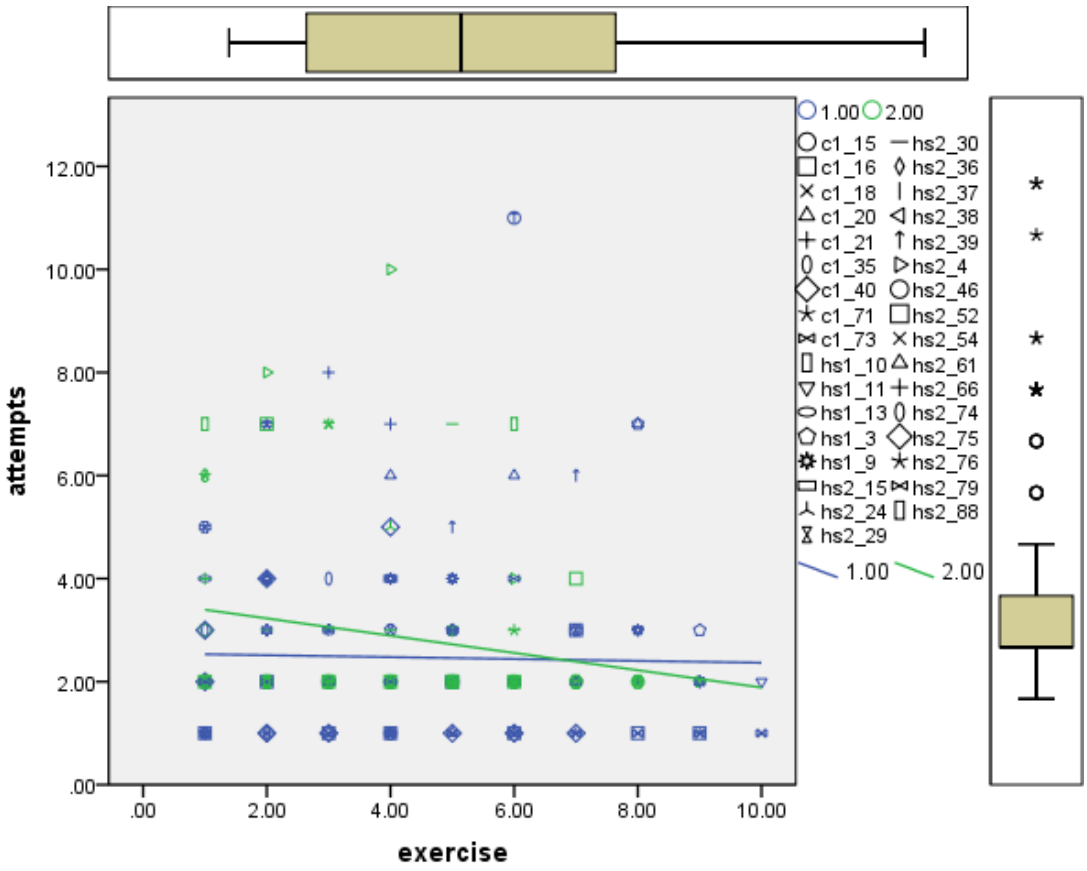


Pearson Correlation:

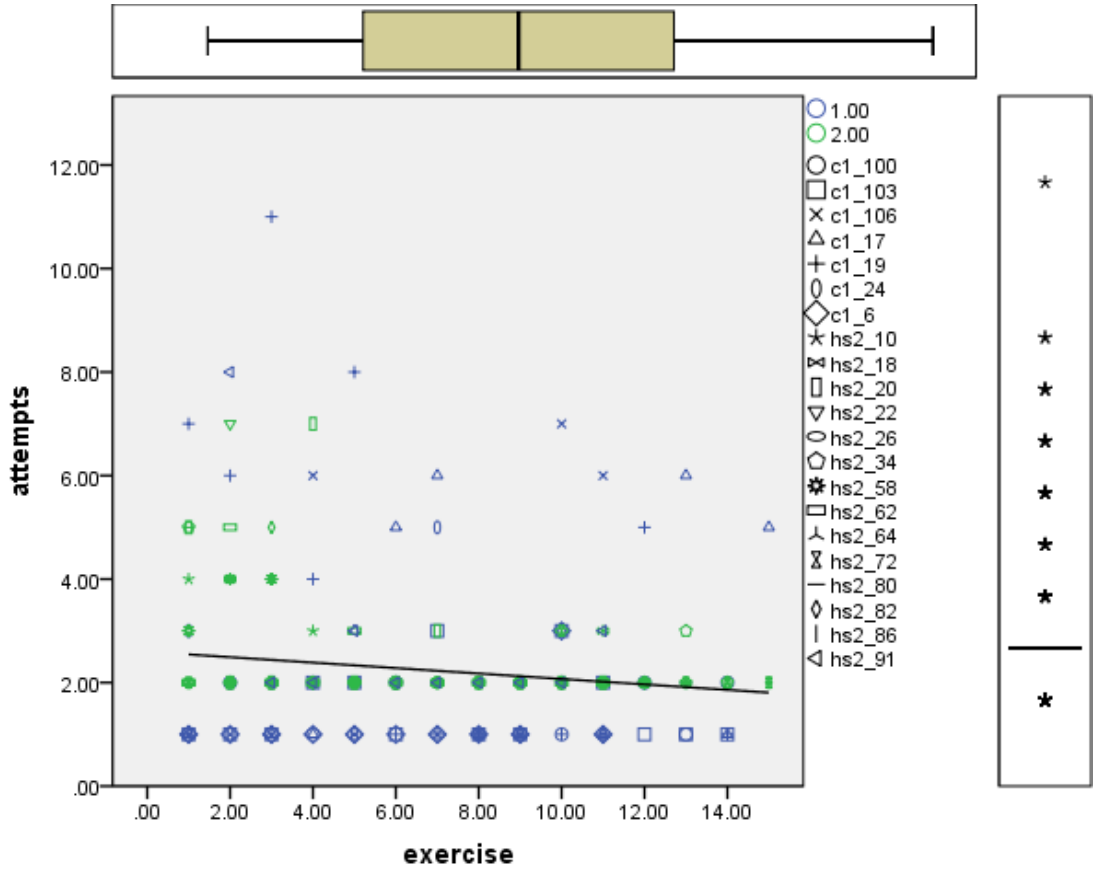
Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.095
	Sig. (2-tailed)		.161
	N	221	221
attempts	Pearson Correlation	-.095	1
	Sig. (2-tailed)	.161	
	N	221	221

Subset: Everyone, 6 to 10 exercises completed, Regression Lines by Phase



Subset: Everyone, 11 to 15 exercises completed



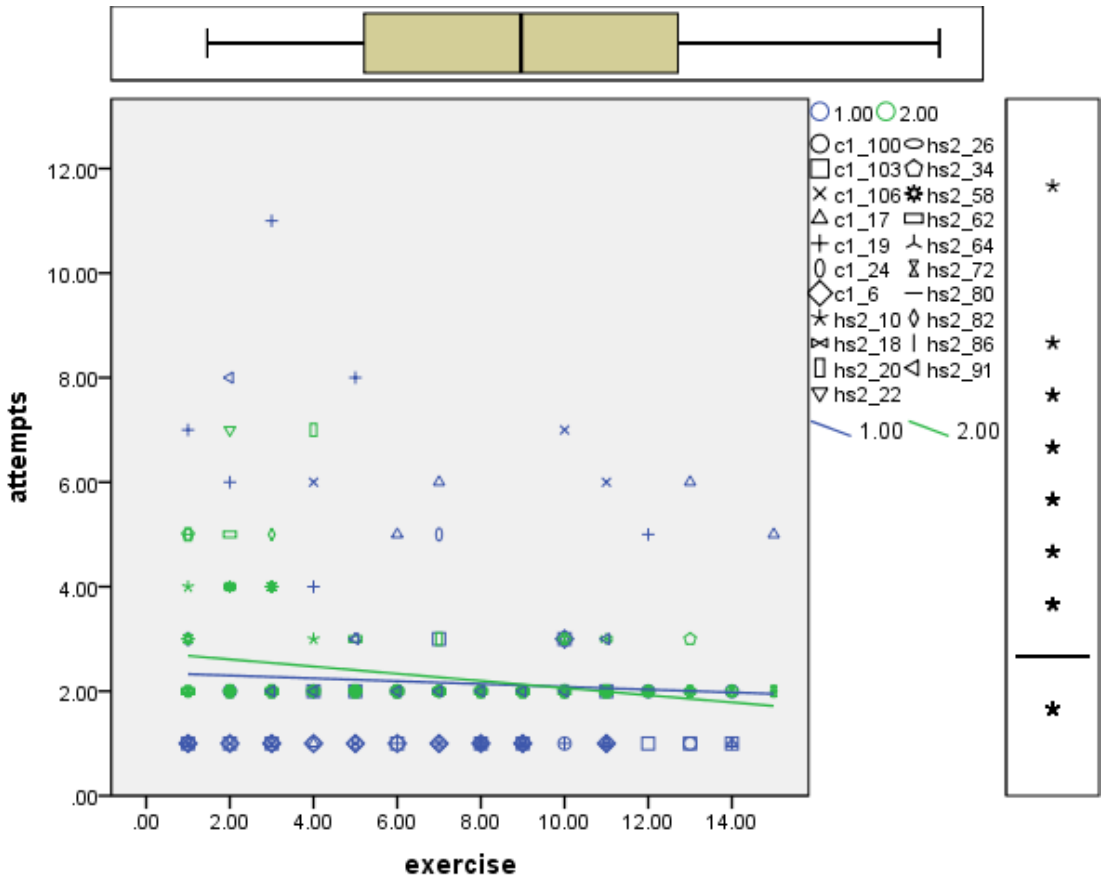
Pearson Correlation:

Correlations

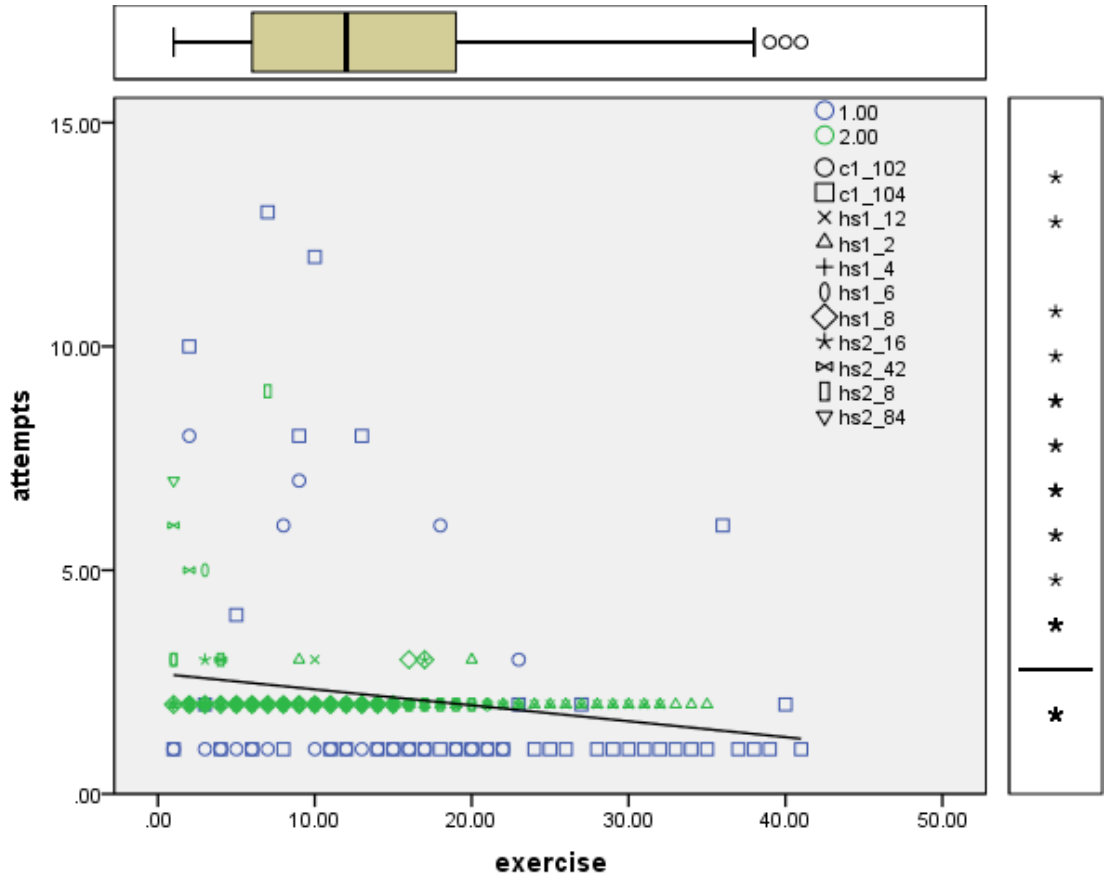
		exercise	attempts
exercise	Pearson Correlation	1	-.146*
	Sig. (2-tailed)		.017
	N	266	266
attempts	Pearson Correlation	-.146*	1
	Sig. (2-tailed)	.017	
	N	266	266

*. Correlation is significant at the 0.05 level (2-tailed).

Subset: Everyone, 11 to 15 exercises completed, Regression Lines by Phase



Subset: Everyone, 16+ exercises completed



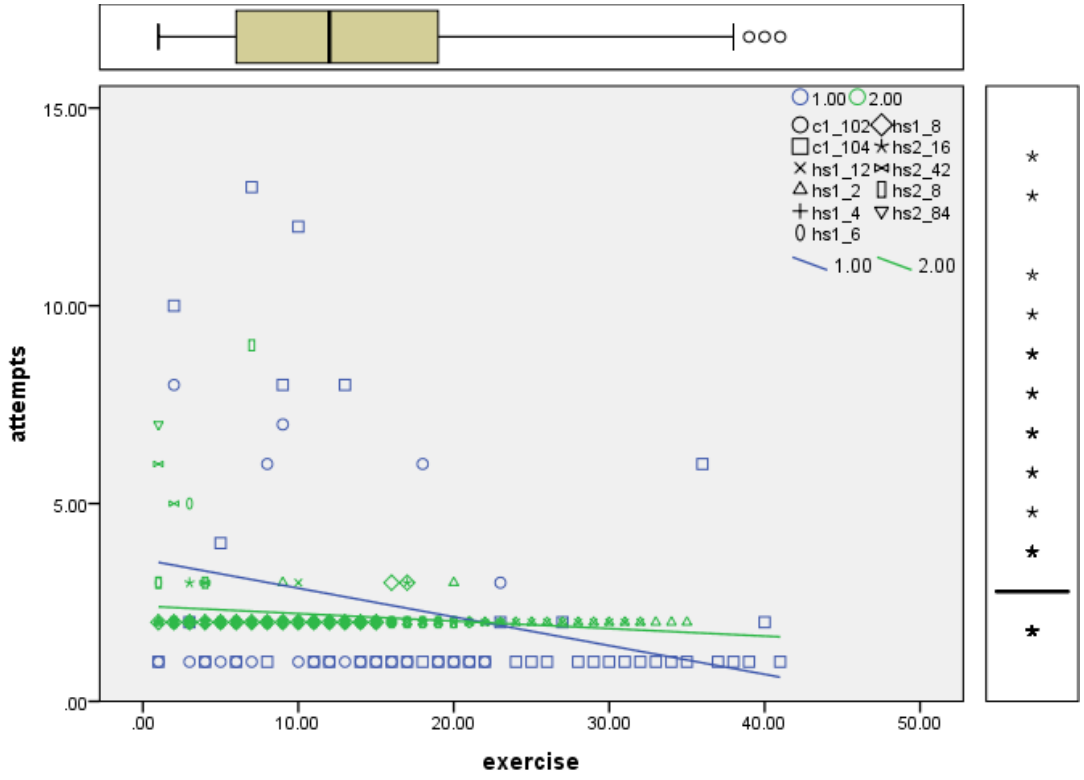
Pearson Correlation:

Correlations

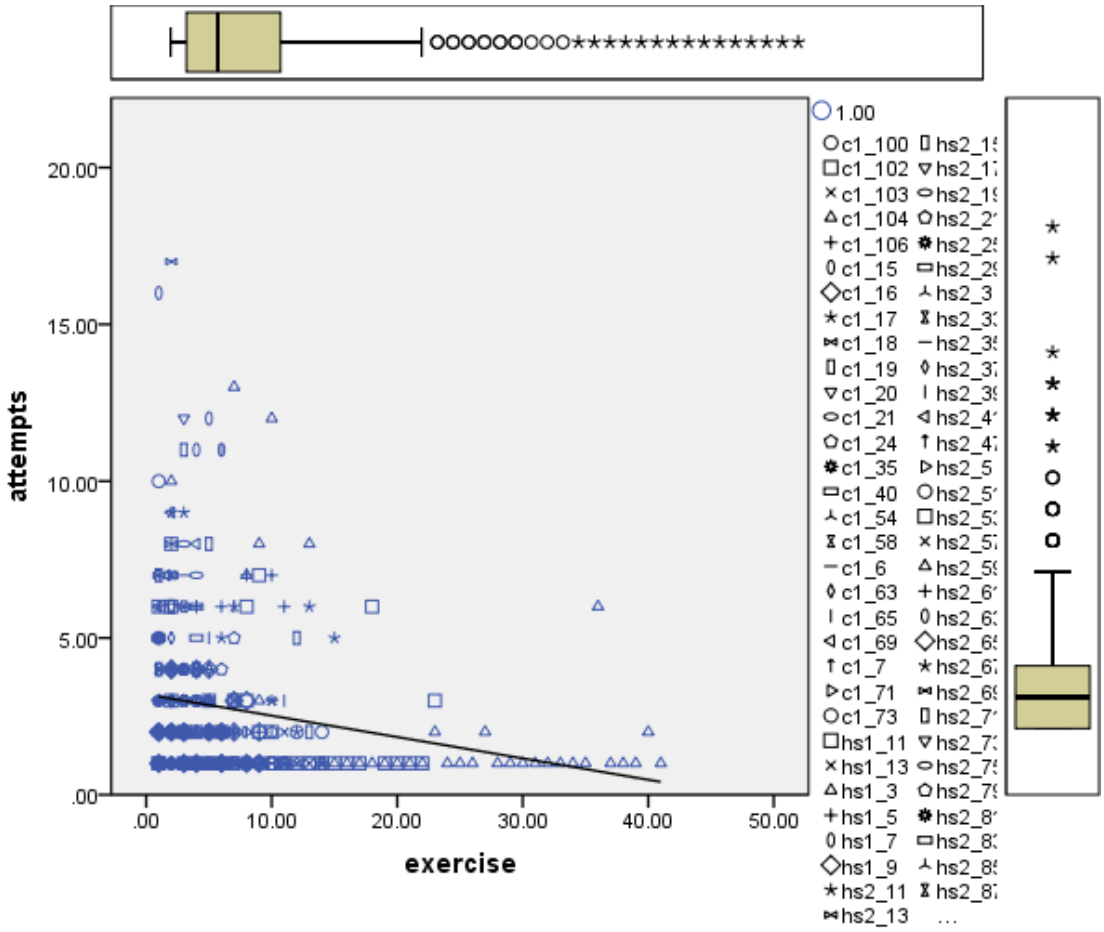
		exercise	attempts
exercise	Pearson Correlation	1	-.211**
	Sig. (2-tailed)		.001
	N	261	261
attempts	Pearson Correlation	-.211**	1
	Sig. (2-tailed)	.001	
	N	261	261

**. Correlation is significant at the 0.01 level (2-tailed).

Subset: Everyone, 16+ exercises completed, Regression Lines by Phase



Subset: Phase 1, All Participants



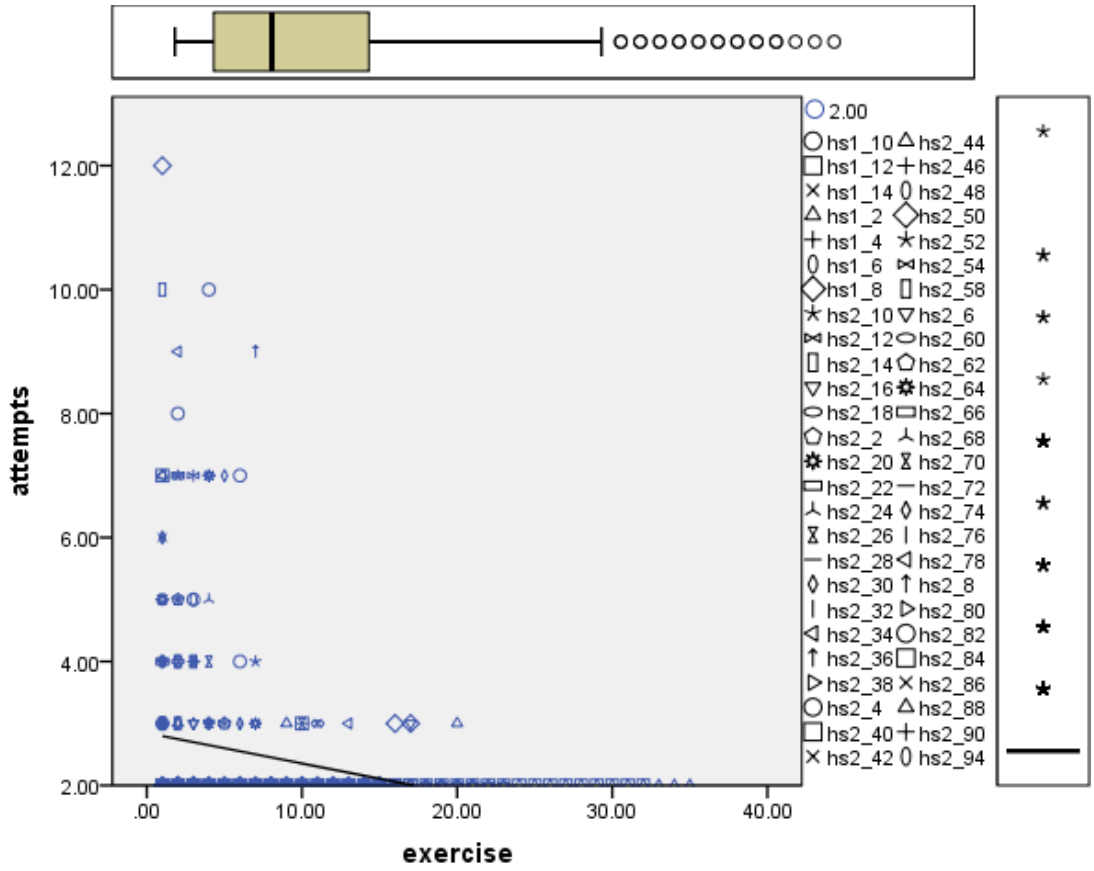
Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.194**
	Sig. (2-tailed)		.000
	N	420	420
attempts	Pearson Correlation	-.194**	1
	Sig. (2-tailed)	.000	
	N	420	420

**. Correlation is significant at the 0.01 level (2-tailed).

Subset: Phase 2, All Participants

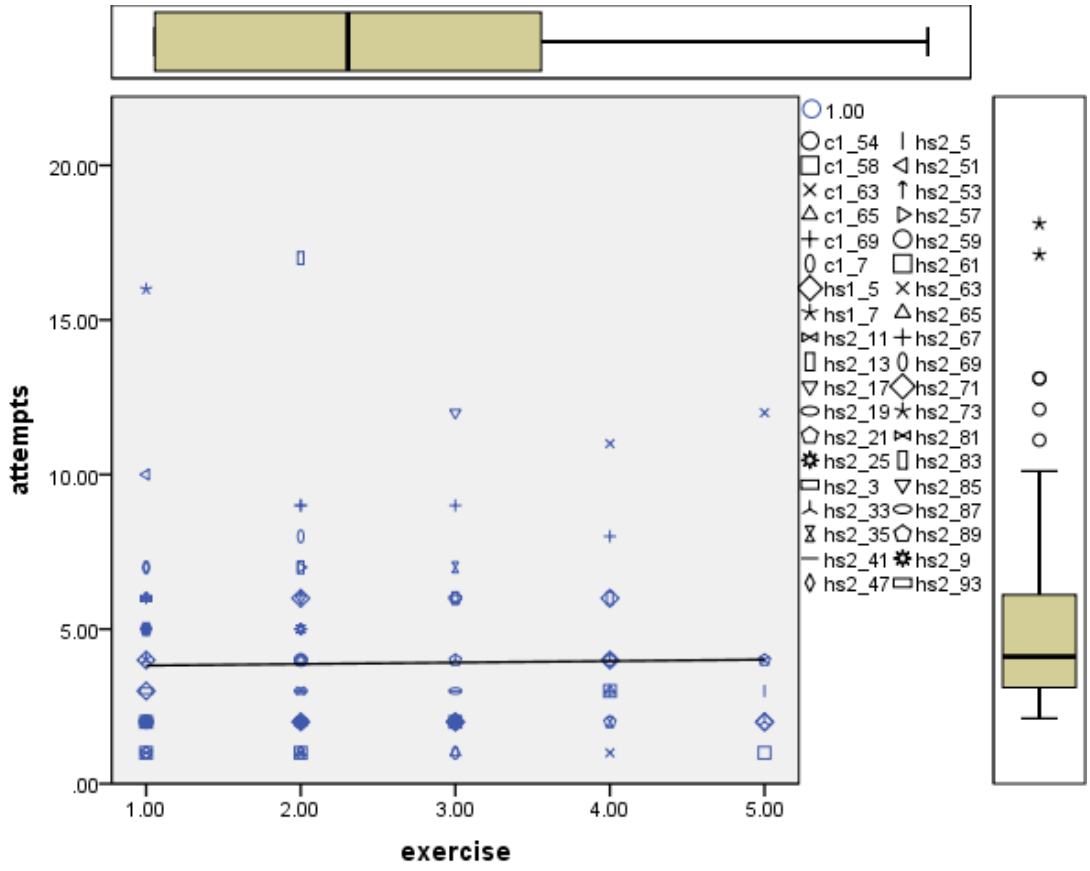


Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.261**
	Sig. (2-tailed)		.000
	N	493	493
attempts	Pearson Correlation	-.261**	1
	Sig. (2-tailed)	.000	
	N	493	493

**. Correlation is significant at the 0.01 level (2-tailed).

Subset: Phase 1, 1 to 5 Exercises Completed

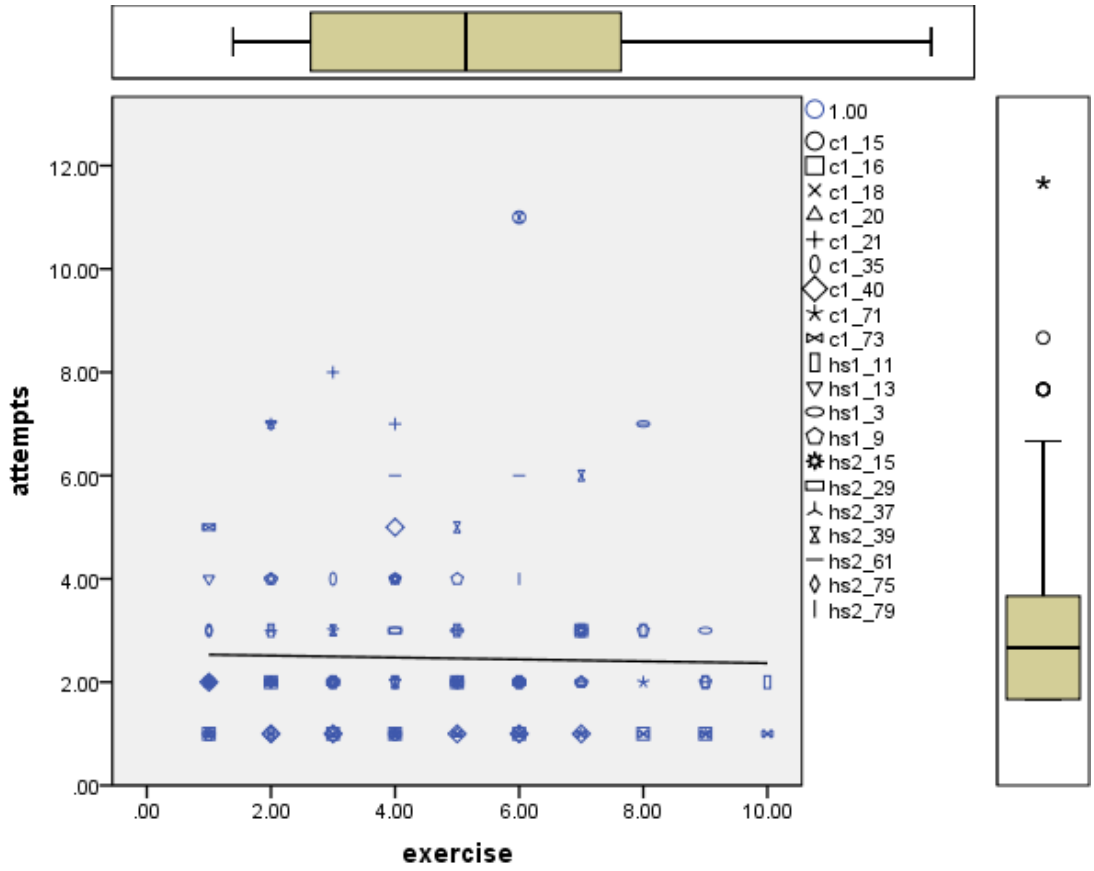


Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	.020
	Sig. (2-tailed)		.832
	N	115	115
attempts	Pearson Correlation	.020	1
	Sig. (2-tailed)	.832	
	N	115	115

Subset: Phase 1, 6 to 10 Exercises Completed

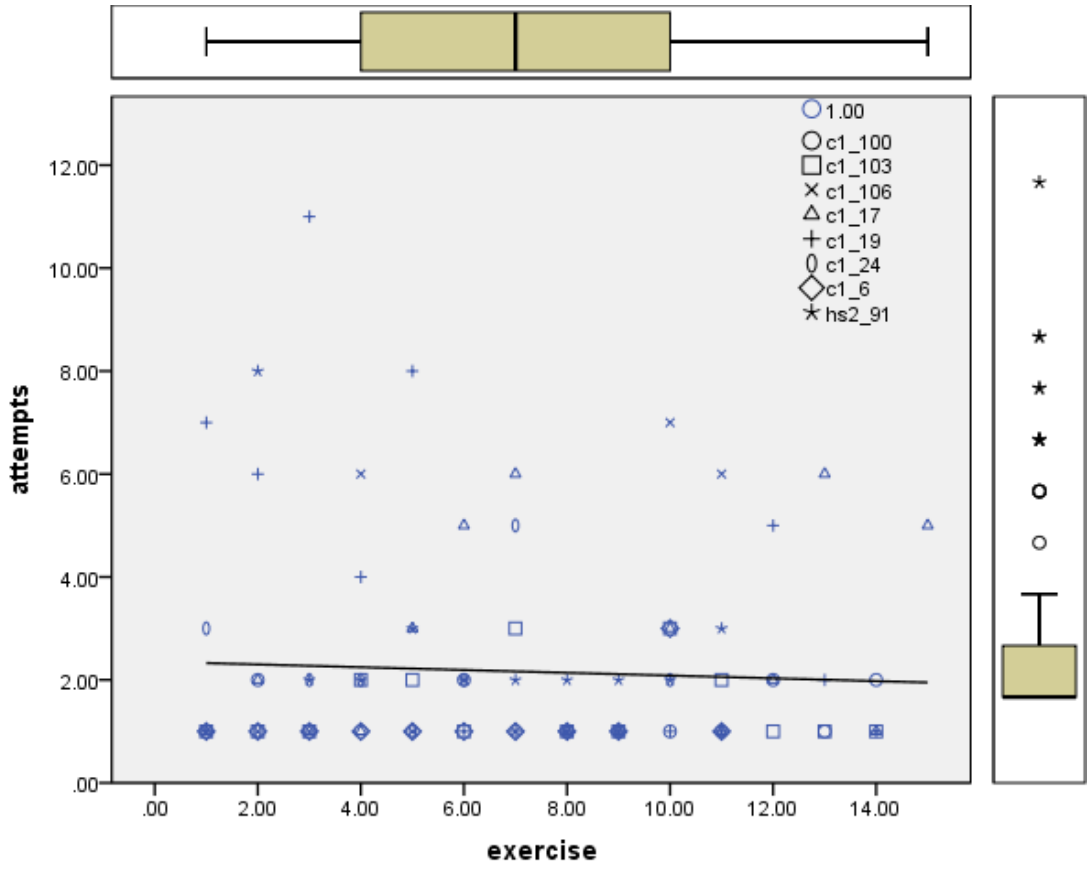


Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.024
	Sig. (2-tailed)		.775
	N	139	139
attempts	Pearson Correlation	-.024	1
	Sig. (2-tailed)	.775	
	N	139	139

Subset: Phase 1, 11 to 15 Exercises Completed

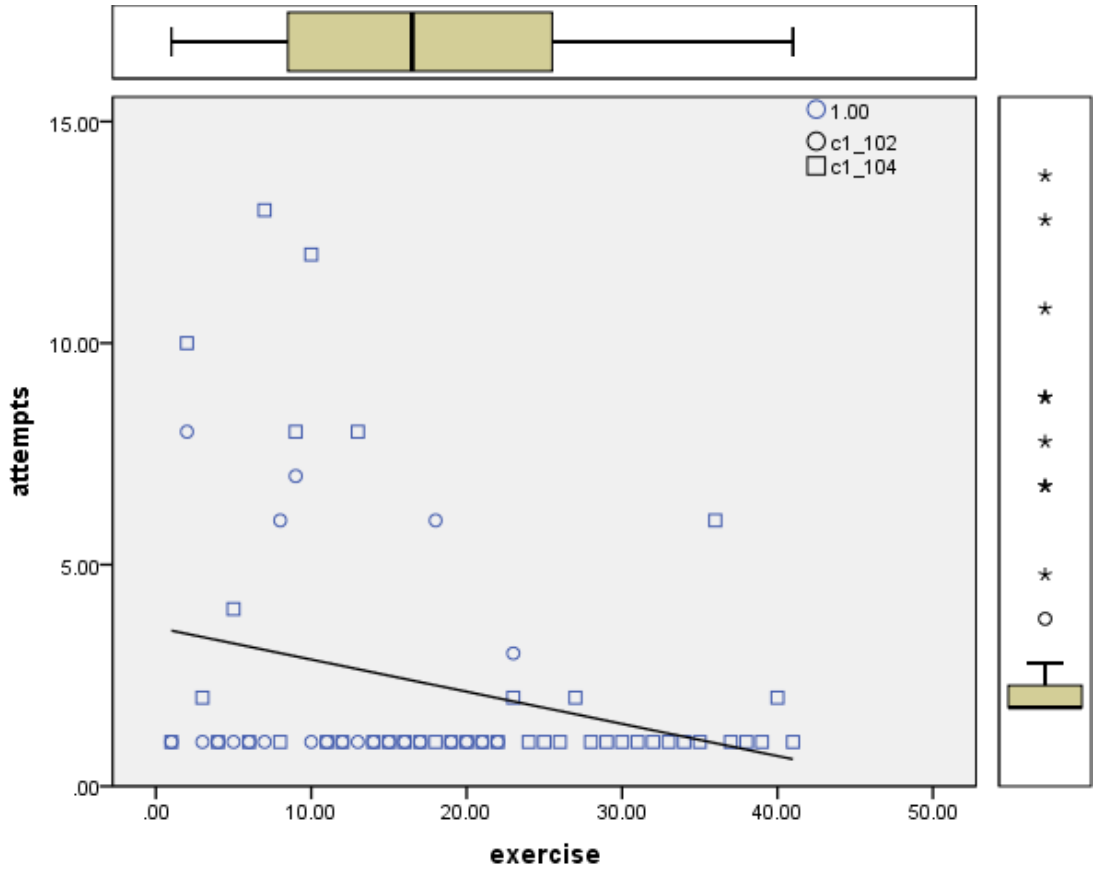


Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.053
	Sig. (2-tailed)		.600
	N	101	101
attempts	Pearson Correlation	-.053	1
	Sig. (2-tailed)	.600	
	N	101	101

Subset: Phase 1, 16+ Exercises Completed



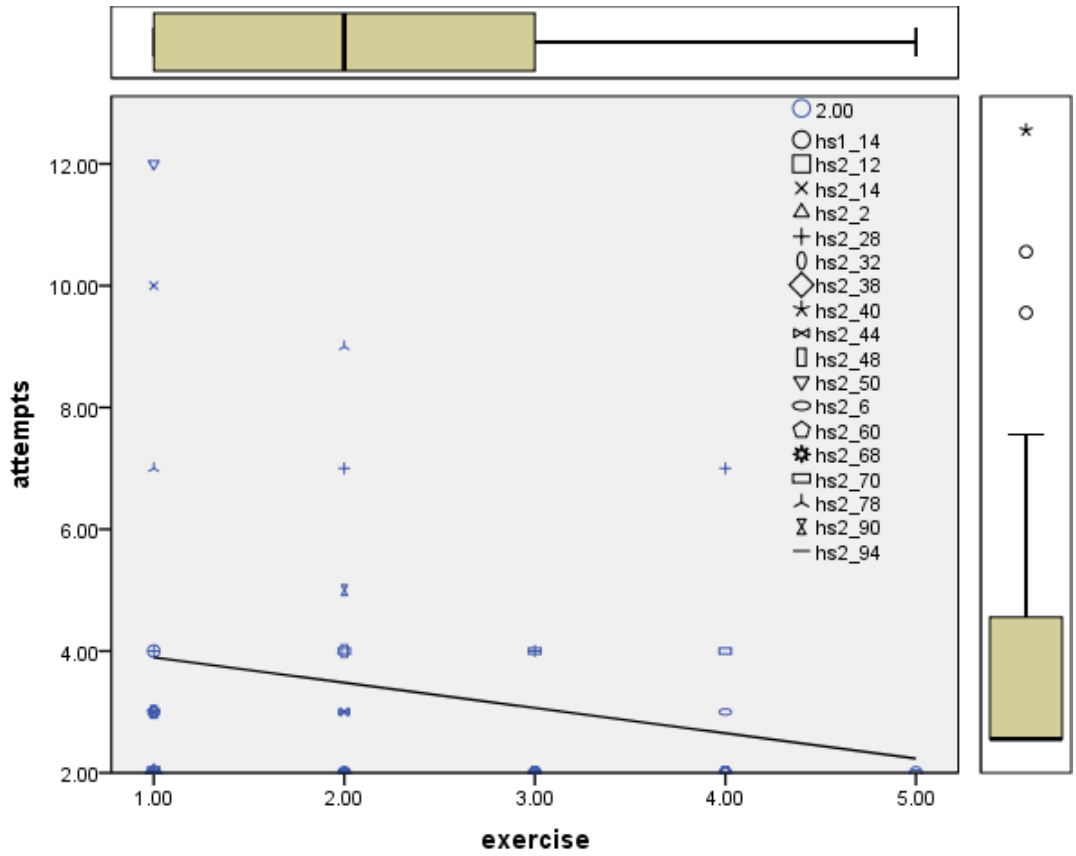
Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.285*
	Sig. (2-tailed)		.022
	N	64	64
attempts	Pearson Correlation	-.285*	1
	Sig. (2-tailed)	.022	
	N	64	64

*. Correlation is significant at the 0.05 level (2-tailed).

Subset: Phase 2, 1 to 5 Exercises Completed

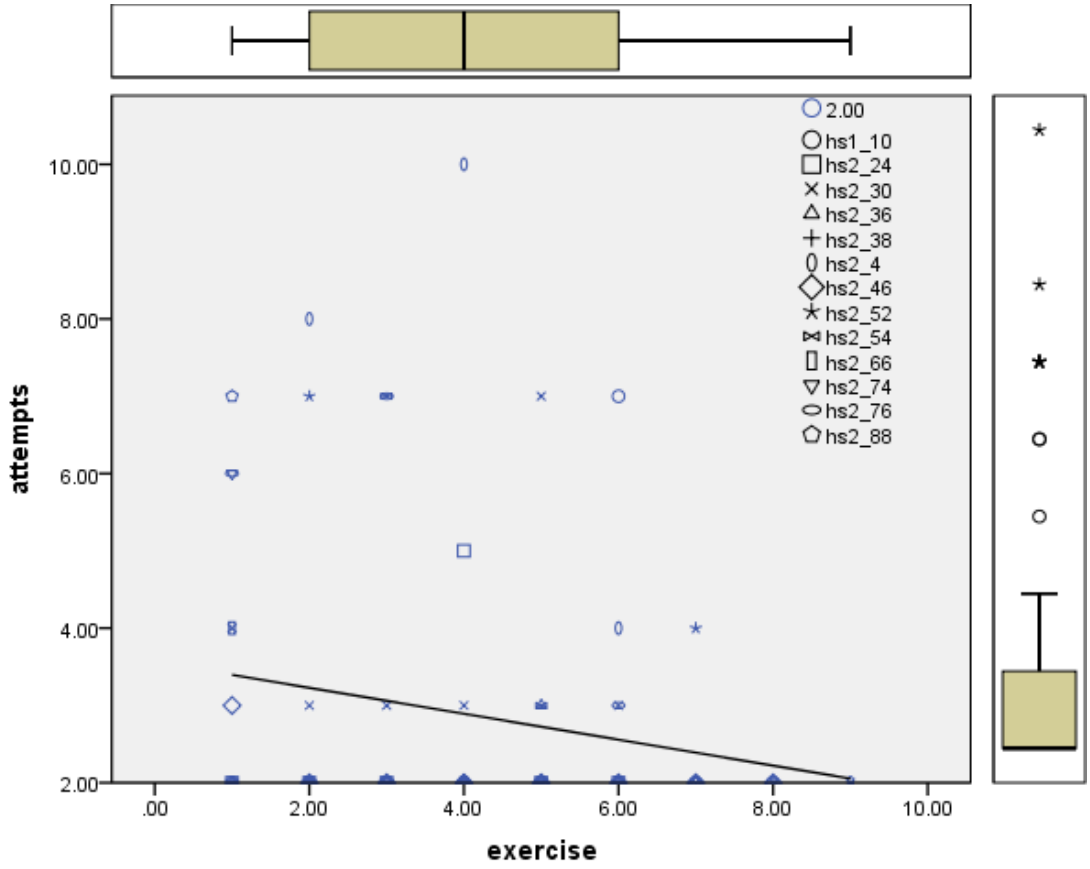


Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.223
	Sig. (2-tailed)		.123
	N	49	49
attempts	Pearson Correlation	-.223	1
	Sig. (2-tailed)	.123	
	N	49	49

Subset: Phase 2, 6 to 10 Exercises Completed

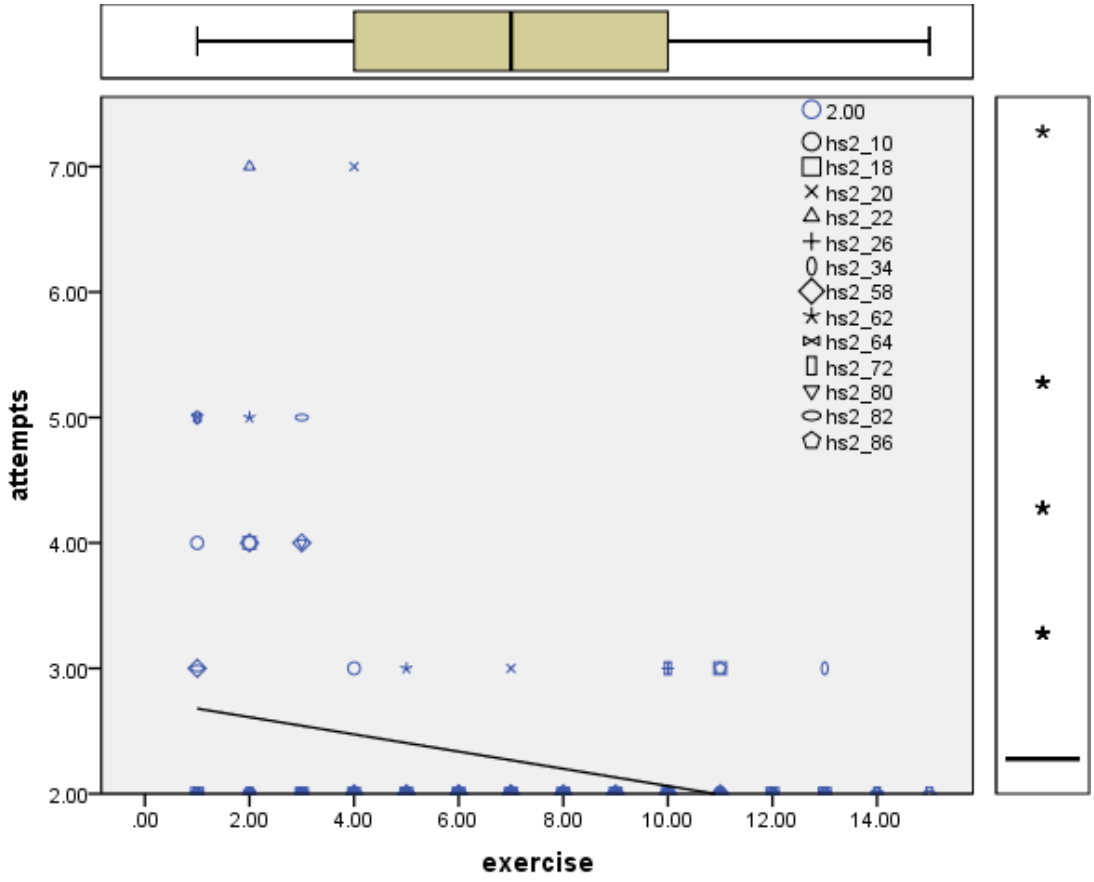


Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.214
	Sig. (2-tailed)		.053
	N	82	82
attempts	Pearson Correlation	-.214	1
	Sig. (2-tailed)	.053	
	N	82	82

Subset: Phase 2, 11 to 15 Exercises Complete



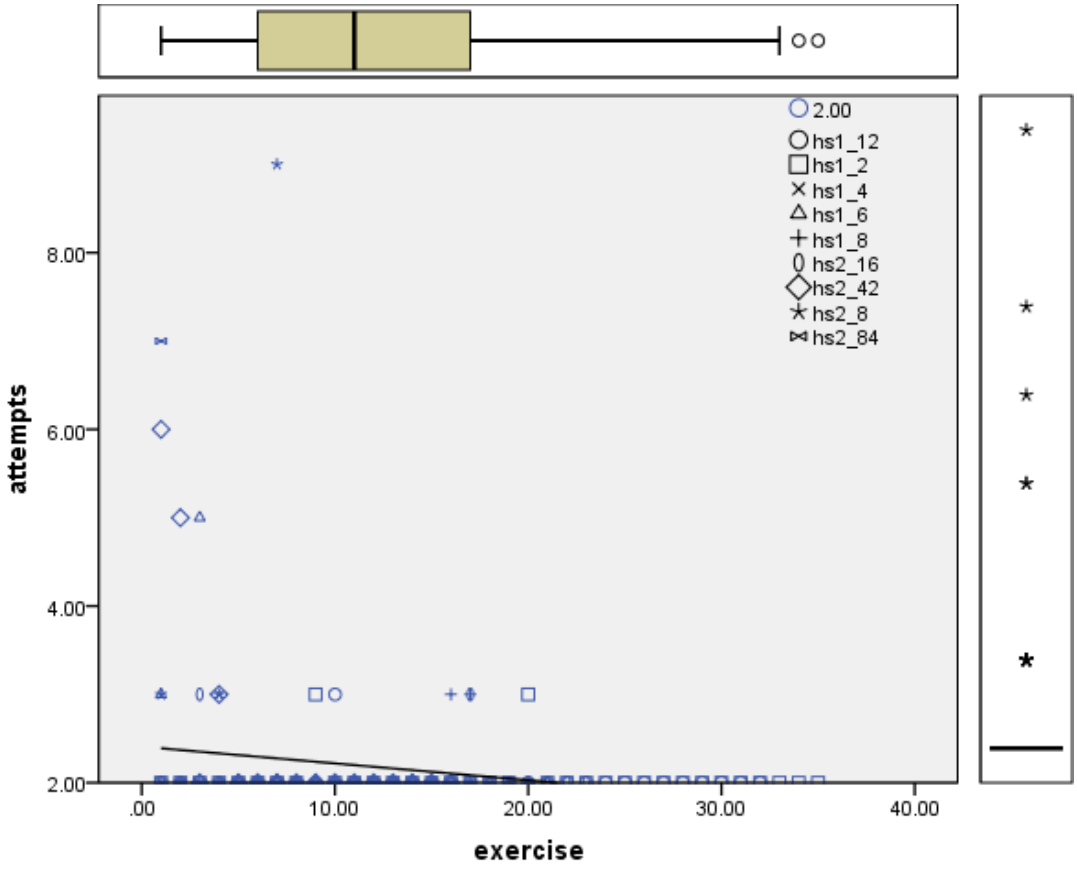
Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.315**
	Sig. (2-tailed)		.000
	N	165	165
attempts	Pearson Correlation	-.315**	1
	Sig. (2-tailed)	.000	
	N	165	165

**. Correlation is significant at the 0.01 level (2-tailed).

Subset: Phase 2, 16+ Exercises Completed



Pearson Correlation:

Correlations

		exercise	attempts
exercise	Pearson Correlation	1	-.202**
	Sig. (2-tailed)		.004
	N	197	197
attempts	Pearson Correlation	-.202**	1
	Sig. (2-tailed)	.004	
	N	197	197

**. Correlation is significant at the 0.01 level (2-tailed).

10 Vita

Elizabeth Carter was born in New Jersey in 1984 to Nancy and Randy Carter. She completed her High School education at Phillipsburg High School in 2003. She then attended The College of New Jersey where she pursued a major in Computer Science with a minor in Interactive Multimedia. While attending TCNJ she was inducted into the Computer Science Honor Society Upsilon Pi Epsilon and the honor society Phi Kappa Phi. She graduated Cum Laude in 2007 and obtained a full time position as a Technology Analyst at Merrill Lynch. In 2009 she enrolled at Lehigh University as a PhD Student, obtaining her Masters in 2011 and PhD in 2014. While studying at Lehigh she received funding from an NSF GK-12 grant and participated in Lehigh's LVSTEM program.