AN EMPIRICAL INVESTIGATION OF THE INFLUENCE OF ONLINE SOCIAL AND PROGRAMMING BEHAVIORS ON LEARNING OUTCOMES IN EARLY COMPUTING COURSES

By

ADAM SCOTT CARTER

A dissertation submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2016

© Copyright by ADAM SCOTT CARTER, 2016 All Rights Reserved
To the Faculty of Washington State University:

The members of the Committee appointed to examine the dissertation of ADAM SCOTT CARTER find it satisfactory and recommend that it be accepted.

_______________________________
Christopher D. Hundhausen, Ph.D., Chair

_______________________________
Olusola Adesope, Ph.D.

_______________________________
Anantharaman Kalyanaraman, Ph.D.

_______________________________
Andrew Begel, Ph.D.
ACKNOWLEDGEMENTS

This dissertation is motivated by the fact that learning to program is a difficult endeavor. However, having just finished writing this dissertation, I would argue that obtaining a Ph.D. is exponentially more difficult. Had I known in 2007 when I enrolled at Washington State University that it would take me nine years to finish, I might have reconsidered my decision to enter graduate school! Needless to say, I couldn't have done it without the help and support of numerous individuals.

First, I would like to thank my committee for providing great insights and suggestions. Thanks to Sola for his statistical advice and Andy for making suggestions that motivated the creation of Chapter 6. Of course, special thanks to Chris. This work has very much been a joint collaboration.

I would also like to thank Dr. Shirazi and the EECS department at WSU for offering me a faculty position years before I finished my dissertation. That generosity will always be remembered and appreciated.

Lastly, I would like to thank my great friends and everyone else that I met along the way. It's been fun.
At only 46%, computing has one of the lowest baccalaureate retention rates. This statistic is especially distressing given the upward trend in demand for computing professionals. In combination, these facts present a clear crisis for computing education. In an effort to address this problem, this dissertation uses social programming environments (SPEs) to explore the application and impact of social learning theory on students enrolled in early computing courses. Unlike traditional integrated development environments, SPEs provide students with opportunities to form learning communities and to engage other classmates in both formal and informal discussions. Even though participation within a learning community is positively linked to retention, such communities are frequently absent in early computing courses.

Our empirical investigation of SPEs yields several illuminating discoveries. First, we find that SPEs are capable of connecting students with each other in ways that help students overcome challenges when programming. Students who use SPEs to ask for and obtain help are more likely to receive higher grades on their homework assignments. Additionally, we discover that, regardless of the kinds of discussions in which they engage, students who regularly participate in the SPE's community are more likely to succeed in a course. Furthermore, we find that SPEs promote an increased sense of community, a key factor in student retention. Lastly, we learn that students' interactions within a SPE yield new insights into their programming processes. Using data collected from an SPE, we are able to identify key differences in the
programming behaviors of successful and unsuccessful students. Furthermore, we discover that these data can also be used to construct predictive models of student activity that strongly correlate with course grades. In total, these results suggest that an SPE can offer substantive benefits to a computing course.
# TABLE OF CONTENTS

| ACKNOWLEDGEMENTS | ................................................................. | iii |
| LIST OF TABLES | .................................................................... | viii |
| LIST OF FIGURES | .................................................................. | xi |

## CHAPTER

1 INTRODUCTION ........................................................................... 1
1.1 What Makes Learning to Program Difficult? .................................................. 1
1.2 Why Do Students Leave Computing Majors? .................................................... 5
1.3 What Do Instructors Know about Their Students? .......................................... 7
1.4 Thesis .................................................................................. 8
1.5 An Overview of the Dissertation .................................................................. 10

2 BACKGROUND AND RELATED WORK ........................................ 14
2.1 Theories of Learning ....................................................................... 14
2.2 The Design Space of Social Programming Environments ............................. 19
2.3 Relating Existing Work to the Construction of an SPE ................................ 32
2.4 Predictors of Success in Computing Courses ............................................. 37
2.5 Learning Analytics .......................................................................... 38
2.6 Conclusion .............................................................................. 41

3 EXPLORING THE PEDAGOGICAL USEFULNESS OF ACTIVITY STREAMS IN COMPUTING EDUCATION ................................................................. 42
3.1 Study I: Exploring Facebook Activity Streams ........................................... 42
3.2 Study II: Integrating Activity Streams into a Course's Learning Management System ........................................... 47
3.3 Study III: Exploring the Relationship between Regular Activity and Grades .................................................. 51
3.4 Summary .................................................................................. 55

4 INTRODUCING THE OSBIDE SOCIAL PROGRAMMING ENVIRONMENT ................................................................. 57
4.1 System Design ........................................................................... 58
4.2 Summative Evaluation of OSBIDE ....................................................... 82
4.3 Summary ................................................................................ 94

5 MODELING PROGRAMMING BEHAVIORS .................................. 96
5.1 Developing a Programming State Model ................................................... 98
5.2 Deriving a Model from Aggregate Transitions ......................................... 101
5.3 Deriving a Model from Transition Chains ................................................................. 108
5.4 A Comparison of Predictive Measures ................................................................. 124
5.5 Summary ............................................................................................................... 129

6 INVESTIGATING THE RELATIONSHIP BETWEEN PROGRAMMING BEHAVIORS AND
ONLINE SOCIAL PARTICIPATION .............................................................................. 131
6.1 Incorporating Social Behavior into the NPSM ..................................................... 132
6.2 Discussion .............................................................................................................. 139
6.3 Exploring the Relationship between Social and Programming Behaviors ......... 140
6.4 Summary ............................................................................................................... 155

7 CONCLUSION ........................................................................................................... 156
7.1 Contributions ....................................................................................................... 156
7.2 Limitations ........................................................................................................... 158
7.3 Implications and Future Work ............................................................................. 160
7.4 SPEs and the Future of Computing Education ..................................................... 164

APPENDIX

CONTENT CODING MANUAL ...................................................................................... 166

ONLINE SURVEY ....................................................................................................... 178

BIBLIOGRAPHY .......................................................................................................... 183
LIST OF TABLES

Table 1. Top-Level Categories .................................................................................................................. 45
Table 2. Modifier Categories ..................................................................................................................... 45
Table 3. New / Modified Categories & Subcategories ........................................................................... 49
Table 4. Definition of Social Role Participation Measure ........................................................................... 52
Table 5. Participation Level Averages by Two-Week Intervals (Standard Deviations in Parenthesis) .... 54
Table 6. Extent to Which Participation Level Predicted Course Grades .................................................. 54
Table 7. Final Grades (%) by Participation Level Quartile ...................................................................... 54
Table 8: Forms of virtuous learner engagement enabled by OSBIDE, their theoretical benefits according to Bandura’s and Lave and Wenger’s Theories, and design and pedagogical strategies promoting them. The dotted line separates observational engagement from interactional engagement ................................ 60
Table 9: OSBIDE Usage by Major Feature ............................................................................................. 83
Table 10. Posting Behavior by Course ................................................................................................... 87
Table 11: C++ Self-Efficacy Response Comparison (7pt scale) .............................................................. 91
Table 12: Classroom Community Scale Response Comparison (5pt scale) ........................................... 91
Table 13: Survey Delta Comparison ....................................................................................................... 91
Table 14: Definition of Social Role Participation Measure ...................................................................... 91
Table 15: Participation Level Averages by Two-Week Intervals (Standard Deviations in Parenthesis) .... 93
Table 16: Extent to Which Participation Level Predicted Course Grades (* = sig. at p < 0.01) .............. 93
Table 17: Extent to Which CS 1 Grade Predicted Course Grades (* = sig. at p < 0.01) ......................... 93
Table 18. Final Grades (%) by Participation Level Quartile ................................................................... 93
Table 19. Explanation of Variance in Individual Assignment Grades .................................................... 104
Table 20. Significant Predictors in NPSM for Individual Assignment Grades (no minimum time) ....... 104
Table 21. Significant Predictors in NPSM for Individual Assignment Grades (at least 1 hr of programming time) .................................................................................................................. 104
Table 22. Significant Predictors in NPSM for Assignment Average Grades ......................................... 104
Table 23. Significant Predictors in NPSM for Final Grades ................................................................. 104
Table 24: NSPM Predictive Power and Coefficients for Seven Datasets of Increasing Size (* = sig. at p < 0.05) ......................................................................................................................... 106
Table 25: Variance Explained by Overall NPSM Model, Averaged NPSM Formula, and Weighted NPSM Formula ......................................................................................................................... 107
Table 26: Frequency of Cycle by Final Course Grade (* = sig at p < 0.05) ........................................... 112
Table 27: Time (in Minutes) Spent in Cycle by Final Course Grade (* = sig at p < 0.05) ................. 112
Table 28: Significant Normalized Transitions by Final Course Grade (p < 0.05) ................................. 112
Table 29. Explanation of Variance in Individual Assignment Grades .................................................. 127
Table 30. Explanation of Variance in Average Assignment Grades ...................................................... 127
Table 31. Explanation of Variance in Final Grades .............................................................................. 127
Table 32: Definition of Participation Level Metric .............................................................................. 132
Table 33: Explanation of NPSM and SNPSM Variance in Individual Assignment Grades .................. 133
Table 34. Significant Predictors in SNPSM for Individual Assignment Grades (no minimum time) ..... 133
Table 35. Significant Predictors in SNPSM for Individual Assignment Grades (at least 1 hr of programming time) ............................................................................................................ 133
Table 36. Explanation of SNPSM Variance in Average Assignment Grades ........................................ 133
Table 37. Significant Predictors in SNPSM for Assignment Average Grades ...................................... 133
Table 38. Explanation of SNPSM Variance in Final Grades ................................................................. 133
Table 39. Significant Predictors in SNPSM for Final Grades .............................................................. 133
Table 40: SNPSM Predictive Power and Coefficients for Seven Data Sets of Increasing Size (* = sig at p < 0.05) ................................................................................................................. 135
Table 41: Variance Explained by Overall SNPSM Model, NPSM Model, Averaged SNPSM Formula, and Weighted SNPSM Formula .................................................................................................. 136
Table 42: NPSM Predictive Power and Coefficients for Final Grade using Seven Data Sets of Increasing Size (* = sig at p < 0.05) ........................................................................................................... 138
Table 43: SNPSM Predictive Power and Coefficients for Final Grade using Seven Data Sets of Increasing Size (* = sig at p < 0.05) .................................................................................................................................................................................. 138

Table 44: Final Grade Variance Explained by Overall S/NPSM Model, Averaged S/NPSM Formula, and Weighted S/NPSM Formula ............................................................................................................................................................... 139

Table 45: Programming Question Coding Scheme .................................................................................................................................................................................................... 142

Table 46: Frequency of question types posed by students .................................................................................................................................................................................................... 143

Table 47: NPSM States of Students Asking Questions .................................................................................................................................................................................................... 146
LIST OF FIGURES

Figure 1: Design Space of Social Programming Environments ......................................................... 20
Figure 2: Facebook Activity Stream .................................................................................................... 21
Figure 3: User-Centered View ............................................................................................................ 26
Figure 4: User Profile Page ................................................................................................................ 26
Figure 5: Timeline Visualization ......................................................................................................... 26
Figure 6: Code Annotation Example .................................................................................................. 29
Figure 7: Categorization of Activity Stream Posts by Course .............................................................. 46
Figure 8: Categorization of CODE Posts by Course ............................................................................ 46
Figure 9: Categorization of Activity Stream Posts .............................................................................. 49
Figure 10: Breakdown of CODE by Subcategory ................................................................................. 49
Figure 11: OSBIDE for Visual Studio ................................................................................................ 58
Figure 12: Sketch of Course Dashboard .............................................................................................. 61
Figure 13: Homework Dashboard ........................................................................................................ 62
Figure 14: User Profile Page ............................................................................................................... 63
Figure 15: Activity Feed ..................................................................................................................... 65
Figure 16: Components of the Activity Feed ..................................................................................... 65
Figure 17: User Profile View ............................................................................................................. 66
Figure 18: Activity Post Details View ................................................................................................. 67
Figure 19: User Profile View ............................................................................................................. 68
Figure 20: Adding / Removing Users from Group .............................................................................. 69
Figure 21: Asking for Help .................................................................................................................. 70
Figure 22: Chat Interface .................................................................................................................... 71
Figure 23: Working OSBIDE Prototype ............................................................................................. 72
Figure 24: Submitting Assignments through OSBIDE ..................................................................... 73
Figure 25: OSBIDE Course Page ...................................................................................................... 74
Figure 26: OSBIDE Grades Page ........................................................................................................... 74
Figure 27: Searching for an error in OSBIDE using Visual Studio ......................................................... 74
Figure 28: OSBIDE toolbar .................................................................................................................... 75
Figure 29: OSBIDE ................................................................................................................................ 77
Figure 30: Annotated Activity Feed ....................................................................................................... 78
Figure 31: Runtime Exception (top left), Ask for Help (top right), and Build Error (bottom) Details Page .................................................................................................................................................. 79
Figure 32: Exploring a Student's Compilation History ............................................................................. 79
Figure 33: Asking for Help within OSBIDE .............................................................................................. 81
Figure 34: Filtered Activity Feed Only Showing Build Events ................................................................. 81
Figure 35: OSBIDE User Profile Page ..................................................................................................... 82
Figure 36: Categorization of Activity Stream Posts .................................................................................. 86
Figure 37: Breakdown of CODE Subcategory ......................................................................................... 86
Figure 38: Programming State Transition Diagram ................................................................................ 100
Figure 39. Seven Programming Data sets of Increasing Size as a Percentage of All Programming Data 105
Figure 40: Average Number of Transitions Generated Per Day ............................................................. 116
Figure 41: Average Number of Identified Transitions Leading Up to Assignment Due Date ............... 116
Figure 42: Interaction between Letter Grade and Transition ................................................................. 117
Figure 43: Interaction between Letter Grade and Transition Four Days before Due Date ..................... 119
Figure 44: Interaction between Letter Grade and Transition Three Days before Due Date ................... 119
Figure 45: Interaction between Letter Grade and Transition Two Days before Due Date .................... 121
Figure 46: Interaction between Letter Grade and Transition One Day before Due Date ....................... 122
Figure 47: Interaction between Letter Grade and Transition on Due Date ............................................ 123
Figure 48: Final Grade Variance Explained by NPSM and SNPSM Models ............................................ 138
Figure 49: Percentage of questions that have responses, suggestions, acknowledgements, and demonstrate progress for all students (N=93) and only those whose question relates to an active coding document (N=59) ........................................................................................................................................... 144

Figure 50: Percentage of future builds that demonstrate progress based on social feedback received..... 144

Figure 51: Grades of students who received help on assignment compared with those that did not for a given assignment........................................................................................................................................... 146

Figure 52: Average score received by a student who asked for help on an assignment versus class average ........................................................................................................................................... 146

Figure 53: Proposed OSBLE+ Interface ........................................................................................................................................... 162
CHAPTER 1

INTRODUCTION

Learning to program is a difficult task that can be challenging to even the most intelligent students. This is borne out in the retention rate within the computing discipline, which falls far below the national average of 78% (National Center for Education Statistics, U.S. Department of Education, 2011). Indeed, according to recent statistics, only 46% of students who entered baccalaureate granting institutions with the intention to earn a degree in computing actually graduated within six years (National Center for Education Statistics, U.S. Department of Education, 2011). Another study found that in a comparison between science, technology, engineering, and mathematics (STEM) majors, students enrolling in computing majors were the least likely to complete their degree (Xianglei & Weko, 2009). In fact, computing has one of the lowest baccalaureate retention rates among all majors (National Center for Education Statistics, U.S. Department of Education, 2011).

This fact is especially distressing given the upward trend of computing enrollments since 2009 (Roberts, 2016) and the importance placed at the national level of promoting computing literacy (whitehouse.gov, 2016). While we cannot expect to completely solve the retention issue, it is realistic to expect the rates to rise to the levels of other fields of study. Given the current state of affairs, we pose a simple question: What makes learning to program difficult? However, in order to provide a worthwhile investigation, we must first understand the current academic climate in computing.

1.1 What Makes Learning to Program Difficult?

When asked about students who struggle, one might be tempted to point to a "geek gene" that, when present, enables success, and, when absent, guarantees failure (Lister, 2010; Seymour & Hewitt, 1996; Tinto, 1993). A strong counterargument to this claim is the work done by Seymour and Hewitt (1996), who

---

1 Throughout this dissertation, I use the first person plural pronoun "we" as a means to acknowledge that the work presented in this dissertation has been a collaborative effort between my chair, my committee, the authors of my joint publications, and myself.
interviewed undergraduate STEM majors in two categories: those who left STEM majors ("switchers"), and those who remained in STEM majors. Their multi-year, multi-institution study (n = 335) found that the majority of students who left STEM majors were just as academically capable as their non-leaving counterparts. If ability is not the cause of computing’s notoriously high attrition rate, what else might it be? Others have asked this question (Guzdial, 2004), and generally speaking, their answers can be placed into one of three categories, as described below.

1.1.1 Programming Environments / Languages

It does not take a great leap of imagination to conclude that industrial development environments such as Microsoft Visual Studio or Eclipse might not be ideally suited for teaching programming fundamentals (see, e.g. Storey et al., 2003). Industrial programming languages such as C or Java pose a similar dilemma. Research in this area generally focuses on trying to make the programming environment, language, or both more approachable for novices. Popular approaches include the pairing of abstract concepts with more concrete visual representations (e.g. Hundhausen, Farley, & Brown, 2009; Weintrop & Wilensky, 2015), reducing the number of programming commands (Guzdial, 2004), or preventing the user from writing incorrect syntax (Resnick et al., 2009). Several novice-oriented systems are described and categorized by Kelleher and Pausch (2005). Assessments of these systems vary greatly and usually include some form of subjective assessment from students and/or instructors along with examinations of students' academic performance and/or retention. While these systems appear to promote achievement among low performing students, a general lack of detailed, rigorous investigations often prevent broader claims from being made (P. Gross & Powers, 2005). However, some recent research appears to be addressing this deficiency (see, e.g. Weintrop & Wilensky, 2015).

1.1.2 Course Content

Rather than focus on a particular language or development environment, some researchers instead focus on how best to structure or present introductory course content. If this line of research were asked the question of what makes learning to program difficult, it would likely respond, "the course content is
difficult." Language selection (Boszormenyi, 1998), the ordering of content (Pears et al., 2007), and programming model (e.g. objects first) (Ehlert & Schulte, 2009) have all been presented as ways to improve learning or motivation. Another popular approach has been to structure curriculum content around a particular construct such as game development (see, e.g. Bayliss, 2009), or robot construction (see, e.g. van Delden & Wei, 2008). Perhaps even more so than the results on programming environments and languages, papers that investigate changing course content tend to be mixed, with some studies reporting positive results (see, e.g. Bayliss, 2009; van Delden & Wei, 2008) and others reporting less of an effect (see, e.g. Ehlert & Schulte, 2009; Simon et al., 2009). Regardless of the effectiveness of such treatments, restructuring course content tends to lack generalizability, as it heavily depends on the course instructor and the materials available. For example, language selection has a large impact on the types of games that can be developed or the robots that can be used. Furthermore, these implementations are not "one size fits all." While game development is known to be quite popular with male students, this is less true for female students (Gurer, 2002). Therefore, it makes sense to want to look beyond individual content adjustments by examining the larger pedagogical issues.

1.1.3 Pedagogies

Researchers interested in pedagogical research might argue that it's not what we teach, but how we teach it, that makes programming difficult to learn. To this end, pedagogical researchers often investigate augmenting or replacing the traditional lecture format with pedagogies of engagement (Smith, Sheppard, Johnson, & Johnson, 2005). Whereas lecture assumes that learning is acquired passively, pedagogies of engagement assume that learning is acquired when the learner actively engages with the material. Pedagogies of engagement vary along the dimensions of formality, scope, and collaboration. For example, Active Learning is an informal pedagogy of engagement in which students form ad-hoc groups during class to discuss lecture topics (Smith, Douglas, & Cox, 2009). At the other end of the formality and scope spectrum is Problem-Based Learning (PBL) where students work in cooperative groups for extended periods of time. Both active learning and PBL also vary on the amount of collaboration required. While all
levels of collaboration have theoretical merit, the recent trend has been towards more socially-oriented (i.e., group) pedagogies (see, e.g. Hundhausen, Agrawal, & Agarwal, 2013; Smith et al., 2005).

While there are some dissenting opinions on the effectiveness of socially-oriented pedagogies, and pedagogies of engagement in general (see, e.g. Kirschner, Sweller, & Clark, 2006), most agree that socially oriented pedagogies are more effective than passive learning models (National Research Council, 2000; Smith et al., 2005). For example, a large meta-analysis on problem based learning (PBL) within the field of medicine found that a PBL-based curriculum promotes better interpersonal skills while also improving the perceived quality of education and lessening dropout rates (Schmidt, van der Molen, & te Winkel, 2009). Similarly, in a study of the use of algorithm visualization in computer science education, Hundhausen et al. (2002) found that student engagement was the main factor in determining pedagogical effectiveness. However, in spite of these mostly positive results, adoption of socially-oriented pedagogies within computing education remains limited. We discuss possible reasons for this below.

1.1.4 Logistical Issues

Several logistical factors may inhibit the adoption of socially-oriented pedagogies by computing majors. As previously stated, these pedagogies encourage a high level of activity with the course material and other students. However, the majority of classes are taught in lecture halls which, as their name implies, are intended to focus attention towards the front of the class and generally make active engagement difficult. Similarly, students need areas outside of class that are conducive to active engagement, which is often not the case. In many cases, lab rooms are organized much like lecture halls (Patitsas, 2012), thereby making collaboration and communication feasible only with one's immediate neighbors. Nevertheless, space related logistical issues are not insurmountable. Indeed, entire buildings and departments have been restructured to accommodate active pedagogies (see, e.g. Lynch, Carbone, Arnott, & Jamieson, 2002). However, adopting such an approach in this manner is far beyond what most universities can logistically accomplish.
1.1.5 Instructor or Institutional Reluctance

Instructor or institutional reluctance to adopt new pedagogies is common (Barker, Hovey, & Gruning, 2015). Reasons for this include a perceived lack of payoff in terms of student achievement, loss of classroom control, or lack of time to fully investigate new pedagogies (Lee, 2000; Walker, 2004). Reluctance may also stem from the fact that many interventions are designed for instructors rather than with instructors (Carroll, Chin, & Rosson, 2002). Pedagogies that rely on collaboration also raise widespread fears of cheating (Sheard & Martin, 2011). Regardless of the specific concern, reluctance to adopt new pedagogies is notoriously difficult to change (Walker, 2004), thereby presenting a significant barrier to educational progress. However, this reluctance can be mitigated somewhat through a variety of means. Interventions can be developed in conjunction with instructors, thus transforming them from reluctant participants into champions of the cause (Carroll et al., 2002). Software tools can be developed to reduce any additional burden placed on instructors, and in the case of cheating, can be used as a way to monitor collaboration efforts between students.

1.1.6 The "Outside of Class" Problem

Pedagogical interventions are most successful when they are tightly integrated into a given curriculum. Instructors have a huge say in how students must interact during class time. Yet, given that most universities assume a 1:2 or 1:3 ratio between time spent in and out of class, the majority of learning time is often left to individual student discretion. Because socially-oriented pedagogies employ group-based projects, this is somewhat mitigated by the fact that students must continue working in groups outside of class, but many instructors, especially in introductory courses, often prefer individual over group assessment. In these cases, it would be incredibly valuable to provide tools to aid in allowing "student time" to be more social.

1.2 Why Do Students Leave Computing Majors?

The previous section reviewed researchers' attempts at identifying the difficult aspects of computing curricula. Implicit in this research is the assumption that the difficulty of computing is what is driving students to leave the major. Rather than identifying difficult aspects of learning to program, one might
instead actually investigate why students choose to leave computing majors. A review of the literature on retention reveals a multitude of reasons a student might leave the discipline. The following is a summary of commonly cited issues, listed in no particular order:

1. Students do not see themselves as fitting the typical, hacker-like, definition of computing professionals (Allan & Margolis, 2002; Guzdial, Ericson, McKlin, & Engelman, 2012; Ko, 2009; Ruslanov & Yolevich, 2010; Seymour & Hewitt, 1996).

2. Students experience anti-social peers, departments, and/or careers (Cassel, McGettrick, Guzdial, & Roberts, 2007; Ko, 2009).

3. Students cannot connect computing to larger world issues or have little interest in the discipline (Allan & Margolis, 2002; Cassel et al., 2007; Guzdial et al., 2012; Ruslanov & Yolevich, 2010; Seymour & Hewitt, 1996).

4. Students lack confidence (Guzdial et al., 2012) owing to a lack of self-efficacy (Bandura, 1997; M.B. Rosson, Carroll, & Sinha, 2011).

5. Students have a hard time making meaningful connections with faculty (Allan & Margolis, 2002; Seymour & Hewitt, 1996).

6. Students have misconceptions about the discipline being more about application usage rather than development (Beaubouef & Mason, 2005).

7. Students are subjected to poorly designed labs that do little to reinforce programming skills (Beaubouef & Mason, 2005).

8. Students are overloaded with too much curriculum. (Guzdial et al., 2012; Seymour & Hewitt, 1996).

Notably absent from this list is that students fail to achieve. Indeed, both Humphreys and Freeland (1992), and Seymour and Hewitt (1996) found that grade point averages between switchers (i.e., those who switched out of the major) and non-switchers (i.e., those who stayed in the major) were nearly equivalent [3.0 vs. 3.15 in (Seymour & Hewitt, 1996) and 3.10 vs. 3.07 in (Humphreys & Freeland, 1992)].
there are some students who will invariably leave the major because of poor grades, this phenomenon is not unique to computing and therefore cannot be the source of the major's higher than average attrition rates. Instead, the list is overwhelmingly composed of issues that possess a *social* component.

The foregoing review suggests we should focus on remedying social deficiencies in the learning processes traditionally promoted in computing education. Socially-oriented pedagogies of engagement would appear to offer the viable solution path; however, as noted above, pedagogies of engagement often face steep opposition for a variety of reasons. A possible solution, then, might be to adapt the social aspects of these pedagogies so as to make them more palatable to computing educators.

1.3 What Do Instructors Know about Their Students?

It is easy for a disconnect to develop between instructors and their students. For example, an instructor might gain insight into a given student's progress upon grading a recent homework assignment. However, by the time that all homework submissions have been graded and turned back to students, struggling students are likely already weeks behind on their current assignment. Furthermore, in large classes, instructors may offload grading to teaching assistants, thereby remaining ignorant to the specifics of a given student's struggles. The end result is that, due to low awareness, instructors frequently miss the opportunity to rescue students as they fall through the cracks.

Identifying at-risk students has long been of interest to computing education researchers. Indeed, investigations of attrition within the major, such as the ones listed in the prior section, are often motivated by a desire to identify and retain struggling students. However, measures used to identify students are often static. A survey provides a singular snapshot of a student, but we know that it is common for attitudes to change; students who appear to be well-suited for success at the beginning of a term might, for a variety of reasons, stumble midway through the course.

Recently, advances in technology have helped educators to address this issue. Rather than rely on static measures, technology has made it incredibly convenient for instructors gather new information from their students. For example, clicker technology and the pedagogy that surrounds it (see, e.g. Porter et al., 2016)
provide instructors with the power to get daily snapshots of both individual students and the classroom as a whole. Similarly, smartphones can be used to rapidly gather feedback from students (Foth, Fitz-Walter, Ti, Russell-Bennett, & Kuhn, 2012). While these technologies provide us with new, valuable insights into our students, they still present barriers. First, they require the use of a pedagogy that incorporates feedback and discussion into lecture. Second, identifying struggling students can require substantial investment from instructors of large classrooms. In these cases, the use of automated tools becomes appealing.

In recent years, the ease with which data can be collected, coupled with the availability of low-cost, high-power machines to store and process such data, has led to an explosion in the field of learning analytics (Wise, 2014). For example, computing education researchers have used data collected from students' development environments as they program to locate where students encounter barriers and how these issues are resolved (Altadmri & Brown, 2015). Similarly, researchers have created predictive models that relate these programming behaviors to course outcomes (Ahadi, Lister, Haapala, & Vihavainen, 2015; Watson, Li, & Godwin, 2014; Watson et al., 2014; Tabano, Rodrigo, & Jadud, 2011; Jadud, 2006a). While these models have varying levels of accuracy, they do point a way towards a future in which instructors, with minimal effort and involvement, can be made immediately aware of individual student learning processes and achievement, as well as overall class trends.

1.4 Thesis

In computing education, students spend a majority of their out-of-class time solving programming problems in integrated development environments (IDEs). While prior research has used IDEs to collect data related to students' problem solving processes, it would also appear to be an ideal venue for social learning activities. This observation yields the following research question:

RQ1. How can we design a social programming environment (SPE) that promotes peripheral awareness, group communication, and collaboration?
Because they are designed for experts, modern IDEs do not commonly support social learning experiences (and learning experiences in general). By introducing social features into the IDE, we can leverage social learning theory in order to explore a new way of learning to program, and of understanding students. To this end, we pose the following research questions:

RQ2. What impact might an SPE have on students' grades and attitudes?

RQ3. How might SPEs and data analytics be leveraged in order to accurately describe students' programming processes?

Through the investigation of these research questions, we aim to demonstrate that increased social participation is associated with more positive programming experiences. This will eventually manifest itself in the form of an increased feeling of success, academic performance—and ultimately, increased retention within the major. Furthermore, the artifacts generated during social interaction, as well data collected during regular programming activities, can be used to better inform instructors. To this end, this dissertation makes the following arguments:

1. The anti-social nature of computing education is the cause of much of the discipline's attrition. In order to address this deficiency, we need to strategically leverage the learning theories that underlie socially-oriented pedagogies of engagement. The integration of social learning into students' out-of-class learning activities, which consume a majority of a student's time, is a promising avenue for making learning to program more social.

2. Instructors need to be better equipped to identify struggling students before their negative experiences drive them away from the discipline. Advances in technology can help us better understand our students. This, in turn, can empower instructors and students to improve the effectiveness of their teaching and learning. However, the present state of learning analytics has not yet developed models that can robustly predict student performance based on their learning
processes. In order to make this a reality, we need to develop new, multi-dimensional learning analytic models that provide a more holistic view of our students.

1.5 An Overview of the Dissertation

We now provide a brief summary of the major components of this dissertation.

1.5.1 Exploring the Pedagogical Usefulness of Activity Streams

In Chapter 3, we begin our investigation into making programming more social by exploring how students appropriate activity streams—a centerpiece of social media environments—as they engage in programming assignments. To this end, we develop coding scheme that categorizes students’ posts along a variety of content dimensions. We then apply this coding scheme to two Facebook activity streams used in two separate course offerings. In both cases, we find that the majority of students' posts relate to course discussions. This analysis clearly indicates that, when given the opportunity, students naturally appropriate an activity stream for academic purposes.

In order to see if these results generalize beyond Facebook activity streams, we use the same content coding scheme to analyze an activity stream in our research lab’s OSBLE learning management system (HELP Lab, 2012). As with the Facebook analysis, we find that students overwhelmingly used the activity stream to discuss course-related issues. Furthermore, a comparison of participation rates among students indicate that students using OSBLE were much more likely to participate in course discussions: 92% of students using the OSBLE feed made at least one post versus 75% of students using Facebook. As one possible explanation for the increase in participation, we suggest that the proximity of OSBLE's activity feed to other course materials (e.g. lecture notes and homework assignments) lowers the barrier to asking questions.

Chapter 4 further explores the hypothesis that proximity to course content increases social participation. In this chapter, we examine participation rates of our SPE’s activity feed and find it to be slightly higher
than the OSBLE condition (93%). Unfortunately, confounds present in design of the SPE study prevent us from performing additional analyses and making stronger claims.

### 1.5.2 Development of a Social Programming Environment

In Chapter 4, we present the design evolution of OSBIDE, the SPE that forms the technological foundation for exploring key research questions posed by this dissertation. Grounding the design in theories of social learning, we document the development of OSBIDE through its paper prototype, high fidelity mock-up, and eventual working implementation. Along the way, we perform two empirical evaluations in which we gain valuable insights into the design of social programming environments. Lastly, we test a working version of OSBIDE in a semester long study involving approximately 130 CS2 students.

### 1.5.3 Relating Online Social Participation to Course Outcomes

In Chapters 3 and 4, we explore the relationship between students' online social participation in Facebook, OSBLE, and OSBIDE and course outcomes (i.e. homework scores and final grade). Depending on the course considered, analysis relating overall online social participation to course outcomes found either a weak or non-existent relationship between the two.

Motivated by the idea proposed by situated learning theory (Lave & Wenger, 1991; Wenger, 1998) that membership within a community is demonstrated through regular participation in the community, we perform a follow-up analysis comparing regular social participation with course outcomes. In this analysis, we codify a student's online social participation during a given homework assignment, and consider its relationship with course outcomes. While non-significant for the Facebook groups, statistical analysis reveals a significant, moderate relationship between regular online social participation and course outcomes for the OSBLE and OSBIDE groups.

### 1.5.4 Using Programming Behaviors to Predict Course Outcomes

In Chapter 5, we present the Programming State Model (PSM), which categorizes a student's current programming state based on the student's past and current programming behaviors. From the PSM, we
derive two secondary measures: the Normalized Programming State Model (NPSM), which records the normalized amount of time spent in each of the PSM's eleven states, and the Transition-Based Programming State Model (TPSM), which examines the frequency of commonly occurring PSM transitions made by students.

From the NPSM, we derive a four-factor predictive formula that accounts for up to 45% of the total variance present in students' homework grades. To give this result more context, we perform a replication study comparing the NPSM to the Error Quotient and Watwin Score, two popular predictors of student academic success in programming courses (Jadud, 2006a; Watson, Li, & Godwin, 2013). In our analysis, the predictive NPSM formula consistently accounts for more variance in assignment grades than either the Error Quotient or Watwin Score.

In building the TPMS, we identified 18 cycles within the PSM that are regularly generated by students. While most of these cycles do not coincide with a student's grade, we are able to identify cycles that act as significant differentiators between A, B, and C students. Analysis does not identify any regularly significant predictors for D and F students.

### 1.5.5 Building a Holistic Model of Student Performance Based on Online Social and IDE Behaviors

Having explored online social predictors in Chapters 3 and 4 and programing predictors in Chapter 5, Chapter 6 considers how combining such factors into a singular holistic model might relate students to course outcomes. To this end, we incorporate the Social Role measure introduced in Chapter 3 into the NPSM and reran our analysis. In all cases, the combined model accounts for more variance in course outcomes than either the Social Role or NPSM did separately. This result would seem to indicate that modeling student behavior is most effective when it incorporates multiple, disparate factors of student behavior.
1.5.6 Exploring the Impact of Online Social Behavior on Programming Behavior and Vice Versa

Having considered the relationship between online social participation, programming behaviors, and course outcomes, Chapter 6 concludes by qualitatively examining the impact that online social participation can have on a student's programming behaviors and vice versa.

Through a series of vignettes, we find that social participation can have a strong impact on students' ability to solve programming issues. In a follow-up qualitative analysis, we discover that online social participation can significantly impact future programming behaviors. Indeed, students who ask for help online, receive a suggestion, and acknowledge that suggestion are over 40% more likely to demonstrate positive progress in their future compilation attempts.

We also consider how programming behavior influences social participation. In examining when questions are asked, we find that more often than not, programming questions relate to an issue presently affecting the question author.
CHAPTER 2

BACKGROUND AND RELATED WORK

The development of a social programming environment and its use in facilitating learning analytics draws on several disparate lines of research. In this chapter, we situate the dissertation within the broader context of this research. We begin with an exploration of underlying learning theories, which is then used to construct a design space of social programming environments. In doing so, we highlight key aspects of design and consider the potential implications and tradeoffs researchers must make when constructing social programming environments. Lastly, we consider the emerging field of learning analytics and suggest how social programming environments might contribute to this research space.

2.1 Theories of Learning

In essence, any learning theory attempts to answer the question, "What causes learning?" In the process of answering this question, the learning theory also implicitly answers the question, "How can we make learning happen?" In the context of constructing a social programming environment, it makes sense for us to consider both the theories of cognitivism and constructivism.

2.1.1 Cognitivism

To a cognitivist, the human mind and learning are described in terms of models and theories. Like a computer, the inner workings of the mind are broken down into a variety of theoretical subsystems whose combination forms the human information processing system (Mayer, 2005). Each subsystem describes certain characteristics and attempts to explain why learning does or does not take place. For example, Mayer's Cognitive Theory of Multimedia Learning (2005) is based on the theories of dual channel (Paivio, 1986), limited capacity (Chandler & Sweller, 1991), and active processing (Mayer, 2009). Often accompanying each subsystem is a list of best practices. As an example, dual coding theory suggests that learning is more likely to occur when information is presented in both visual and auditory form (Paivio, 1986). From this example, we can see that when designing for learning, the cognitivist approach is to
structure content in such a way as to activate a given model's structures in the most efficient manner. Implicit in this approach is the idea that knowledge is absolute and that an instructor's goal is simply to transmit this knowledge to students. However, knowledge transmission (i.e. lecture) is widely regarded to be an ineffective method of instruction (see Smith et al., 2005). Instead, it is argued that instruction should be learner-centered (National Research Council, 2000), which requires a different theory of learning.

2.1.2 Constructivism

Constructivist theories of learning reject the belief that knowledge is an absolute entity to be transmitted, and instead see knowledge as being individually constructed by learners (Papert, 1991). Constructivists believe that knowledge is gained as learners "examine thinking and learning processes; collect, record, and analyze data; formulate and test hypotheses; reflect on previous understandings; and construct their own meaning" (Jonassen, Davidson, Collins, Campbell, & Haag, 1995). There are two main branches of constructivism: cognitive constructivism, which holds that learning occurs at the level of the individual, and social constructivism, which holds that learning takes place within a social context. The work of Piaget falls into the realm of cognitive constructivism (see, e.g. Gruber & Voneche, 1995), while the works of Vygotsky (1978), Lave and Wenger (1991), and Bandura (1986, 1997) would be considered social constructivism. The aforementioned pedagogies of engagement (see Smith et al., 2009, 2005) also fall under the constructivist umbrella. Given the positioning of an SPE as a social learning tool, it makes sense to ground the development of an SPE in the theories of social constructivism. To this end, we next consider Situated Learning Theory (Lave & Wenger, 1991), Communities of Practice (Wenger, 1998), and Social Cognitive Theory (Bandura, 1986).

2.1.3 Situated Learning Theory and Communities of Practice

Situated Learning Theory believes that learning cannot occur apart from social participation because knowledge is a social construction relative to a particular community of practice. Situated Learning differentiates accidental (i.e. unintended) learning from purposeful learning through the construct of
Legitimate Peripheral Participation (LPP). Taking the construct of LPP one word at a time, LPP defines a particular form of participation that is:

- **Legitimate**—Participation must be a means of demonstrating belonging to a community of practice.
- **Peripheral**—Participation is variable. Learning occurs as learners take on multiple and varied roles within a community of practice, ranging from peripheral to central.

Both terms refer to Communities of Practice, which can be conceptualized as a self-perpetuating groups of individuals that have great power over knowledge and learning. According to Lave and Wenger, a community's standards prioritize certain actions or beliefs over others and adoption of these attitudes demonstrates community membership, and therefore learning. However, while Lave and Wenger do an excellent job of defining how learning occurs, they intentionally, "[focus] attention on the structure of social practice rather than...pedagogy" (p. 113). Therefore, it is necessary to look elsewhere for specific pedagogical recommendations. For this, I turn to *studio-based learning* (SBL).

In many ways SBL can be thought of as formalized apprenticeship. In *The Reflective Practitioner*, Schon (1983) outlines the four characteristics of SBL:

1. Classroom assignments should be project-based.
2. Students are assigned shared studio spaces to complete their assignments.
3. Student work should be periodically evaluated both formally and informally through design critiques ("design crits").
4. Similarly, students should be required to engage in critiquing the work of others.
5. Design critiques should revolve around the artifacts typically created by the discipline.

In computing, this usually involves incorporating periodic critical reviews into the normal homework cycle (A. S. Carter & Hundhausen, 2011). Using previously defined terminology, this has been more or less an exploration of the "design crit" aspect of SBL. On the other hand, the SPE proposes to explore the
informal, "studio" aspect of SBL (item #2 from the above list) by facilitating peripheral awareness and group discussion. However, future iterations of the SPE could add support for the design crit, similar to the capabilities present in commercial software such as Code Collaborator (SmartBear Software, 2012).

2.1.4 Social Cognitive Theory and Self Efficacy

Similar to Situated Learning, Social Cognitive Theory (Bandura, 1986) states that people learn from observing others' behaviors. In this context, self-efficacy, another construct developed by Bandura (1997), is a measure of a person's belief that he or she can accomplish a previously observed behavior. Self-efficacy is known to be a contributing factor when selecting computing as a major (Papastergiou, 2008) and also plays a role in overall career selection (Bandura, Barbaranelli, Caprara, & Pastorelli, 2001). According to self-efficacy theory, an individual's self-efficacy is obtained from four sources, as described below.

**Enactive experiences** are ones in which the learner is the primary participant. For example, attempting to lift a weight is an enactive experience that will affect a person's weightlifting self-efficacy. As these events are directly experienced, they tend to be the most influential in determining self-efficacy. Early enactive experiences are of critical importance and poor results can lead to significant drops in self-efficacy. Interestingly, the positive or negative impact of an enactive experience is socially determined as most tasks do not have inherit benchmarks for success. Going back to the weightlifting example, if I struggle to lift 25 lbs. and am told that 25 lbs. is incredibly light, my weightlifting self-efficacy is likely to decrease. Conversely, if I am told that 25 lbs. is an astonishing amount of weight to lift, my weightlifting self-efficacy is likely to increase.

**Vicarious experiences** are those that are not directly undertaken by the learner. Generally, these experiences have a weaker impact on self-efficacy, but vicarious experiences have the power to override poor enactive experiences. Vicarious experiences tend to have more weight when a learner is unfamiliar with a given task and has ill-formed self-efficacy and are most effective when the person being observed is judged to have characteristics that are similar to those of the observer. For example, if I witness someone with my height, weight, and age lift a weight, I will then tend to believe that I too can lift that weight.
Conversely, watching a professional weightlifter lift a weight will not likely have an effect on my self-efficacy. Vicarious experiences are especially useful when used as a method for socially validating enactive experiences. For example, assume that I judge myself to have a poor enactive experience when lifting weights. However, I witness someone similar to me struggle when performing the same task. Having witnessed someone else struggle, I may start to think that I'm not doing as bad as I thought. Alternatively, I may start to think that the task is inherently difficult and therefore is less of a reflection of my personal skill.

**Verbal persuasion** refers to after-the-fact affirmations of faith and encouragement. As such, saying, "good job" would not count as verbal persuasion whereas, "Don't give up, you can do better!" would be counted. According to Bandura, verbal persuasion is likely to result in a boost to self-efficacy as long as the encouragement is within a reasonable distance to a person's ability. Conversely, providing unrealistic encouragement is likely to lower self-efficacy.

**Physiological state** refers to the physical and mental wellbeing of the person and can affect self-efficacy. For example, a person with a broken arm will probably not believe that he is capable of hitting a home run. Mood also plays an important factor in determining self-efficacy. Bandura (1997) cites research that indicates that being in a poor mood makes it more likely for a person to recall past failures, whereas a good mood makes it more likely to recall past triumphs.

Given that an SPE is intended to be highly social, it would seem that the largest impact that an SPE can have on self-efficacy is providing additional social validation. Recall that computing classes leave many students feeling socially isolated (Ko, 2009) and that infrequent or inaccurate social validation can be harmful to self-efficacy. By allowing students to witness other students struggling and succeeding, an SPE can appropriately situate the struggles encountered when learning how to program. Ideally, an SPE would accomplish this by allowing students to observe the work of others. Unfortunately, as previously discussed, concerns over cheating may prevent instructors from embracing a completely open programming environment in which code is freely shared. However, self-efficacy theory suggests that it may be sufficient
to merely inform students of others' struggles and successes. As we will see, our SPE incorporates the ability for students to be made aware of others' programming processes.

2.2 The Design Space of Social Programming Environments

A social programming environment seems strategically poised to support the underlying processes of Situated Learning, Communities of Practice, and Social Cognitive Theory, as well as concrete pedagogies such as Studio Based Learning. In this section, we consider the overall design space of SPEs (Figure 1) and how it relates to these theories.

2.2.1 Awareness Mechanism

We use the term awareness mechanism to denote a system that allows a user to gather information about other users without the need for direct communication. Phrased differently, an awareness mechanism allows for passive information gathering. In this section, we present two possible solutions that are suitable for an SPE.

2.2.1.1 Activity-Centered Systems

Popular in social networking sites such as Facebook and Twitter, activity streams aggregate the activity of one or more users into a stream of consciousness for general consumption (see Figure 2). On social networking sites (SNS), activity feed posts tend to be user-created and contain thoughts, feelings, or opinions (e.g., "I like ice cream."). However, some feed posts are automatically generated on behalf of the user by other programs (e.g., "John just played Scrabble."). When adapting an activity-centered approach to an SPE, there are many subdimensions to consider, which we describe below.
Figure 1: Design Space of Social Programming Environments
2.2.1.1 Event Selection

A primary concern is the selection of events that compose an individual's activity stream. Whereas traditional activity streams primarily rely on user-generated content, an SPE has the opportunity to supply a large amount of potentially useful feed items generated automatically from user's activity within the IDE. This raises questions regarding the source of a feed posts, the frequency at which feed posts occur, the content of posts that are exposed to the user, and the relatedness of the posts to the current user.

Source. As previously noted, the majority of SNS feed posts are user-generated. Often, automatically generated messages are seen as a nuisance (Facebook allows users to selectively turn off these posts), but the automatic messages generated through SPE activity may be of great interest or use to other students in
the class. For example, Student A may find it very helpful to see that Student B encountered the same compiler error. Recall that these kinds of vicarious experiences are believed to be vital in the formation of one's self-efficacy, a key factor in determining persistence (Bandura et al., 2001). Therefore, we might expect automatically generated messages to play a larger role in an SPE than they do in normal SMSs. However, this gives way to the problem of priority. Given that a single student using an SPE is capable of producing several hundred messages per day, it is likely that user-generated messages will get lost in noise. Therefore, it is likely that user-generated posts need to be placed above those that are automatically generated.

**Frequency.** The frequency with which automatically generated feed posts are injected into a person's activity stream is a key determinant in the stream's overall helpfulness. Too many posts make it difficult for people to sift through the noise, but too few posts make the feed seem static and unused.

**Content.** The logging feature of our SPE captures a wide variety of data points. Generally speaking, these data points occur while in one of three states: *editing, compiling*, and *debugging*. Depending on the course, events that occur in a particular state may be more significant. For example, CS1-type courses, learning how to properly construct code is a key roadblock for many students. In this case, students may be more interested in events that occur in the *editing* state. However, in more advanced classes, students are confident in their program construction skills and encounter higher level issues. In this case, these students may be more interested in events that occur in the *debugging* state.

**Relatedness.** Grudin (1988) states that in order for a collaborative system to succeed, it must be useful to its primary users. In the case of the SPE, the primary users are the students. In order for students of computer programming to find an activity stream useful, it must be capable of providing new insights that were not previously available. For this to occur, it is likely that feed items will somehow need to be relevant to the current user. Finding such relationships can be done in a variety of ways. For example, the SPE might present posts noting a compiler error only if the current user experienced that same error. Another possibility might be to highlight posts to which the user has written a response. Alternatively, relatedness can be user-specified through a series of filters that remove unwanted posts from the stream.
2.2.1.1.2 Event Composition

The second dimension of activity streams involves the composition of individual feed posts. Composition sub dimensions are discussed below.

Access Level. Automatically generated feed posts are created based on another user’s input. How much information the system exposes about the event to other users is an open question. For example, in a build event that contains compiler errors, is it sufficient to know just that a student received compiler errors, or should we also indicate which errors were present? Going further, the system could also provide line numbers and even sections of student solutions. Taken to its fullest, events could expose complete documents or the entire solution to other users. While providing deep hooks into the work of others is important for collaborative systems (Erickson & Kellogg, 2002), it also makes it easier for students to plagiarize. Striking a proper balance between providing access to more useful system and facilitating rampant cheating is something that will likely need to be negotiated with individual instructors.

Recommendation System. Tied closely with the relatedness subdimension is the way in which posts are recommended to the current user. We identified two general strategies that we have labeled the bandwagon and connector strategies. Used by Facebook, the bandwagon strategy highlights the relationship of the active post to the rest of the group. Using this strategy, the recommendation system would display an item like, "Bob and 10 others received the same compiler error." The purpose of this strategy is to highlight popular group issues. In contrast, the connector strategy would attempt to make connections between the active post and the current user. Using this system, the recommendation system would display a message like, "You and Bob both got the same compiler error." The purpose here is to highlight potential shared interests or goals between the current user and the rest of the group. Note that these strategies are not mutually exclusive. Such a message might read, "You, Bob, and 5 others got the same compiler error."

Verbosity. While it is completely possible from a technical standpoint to include every data point in each feed post, it is unlikely that this would be the most effective method. Twitter messages have a hard 140 character limit, while Facebook hides content that exceeds a certain length. Both systems prevent a
Likewise, determining an appropriate level of verbosity in an SPE is a key consideration.

2.2.1.2 User-Centered Awareness System

An alternative to stream-based profiles is the user-centered model. Whereas stream-based profiles promote peripheral awareness through a constant stream of incoming feed posts, a user-centered approach aggregates an individual's data, presenting a synopsis of the individual's most recent activity (Figure 3). In so doing, the user-centered approach sidesteps many of the dimensions present in activity-centered systems. Instead, a user-centered approach must concern itself with appropriate states, access level, relatedness, and verbosity.

States. Recall that the user-centered approach only displays a user's most recent action. However, given a large number of potential actions, some of which last only a few seconds, this kind of system ends up being unusable, as the constantly changing set of actions is likely to create an unreadable mess. To compensate, it makes sense to group like actions into a set of states that are less likely to change at a rapid pace. This leads to a question regarding the number of possible states. Too few will result in a static interface that gets neglected, whereas too many will run the risk of overwhelming users. Based on a task analysis of computer programming, four types of events are of potential interest: editing, compiling, executing, and debugging. These four states encompass the primary activities of programmers, which we speculate will provide sufficient awareness to other students. Additionally, to represent a user's status within the SPE, two additional states are needed: online and offline.

Access Level. Access level is used here in the same way it was used in the activity-centered approach: namely, it determines the granularity of document access granted by the system. For example, when a user is in the debugging state, the system could grant the user access to a person's breakpoints, stack traces, the active line being debugged, or the entire code document.
**Relatedness.** This dimension refers to the way in which another user's state is related to the current user. For example, the system could highlight states that match the current user's state. Alternatively, states could be grouped and ordered based on user preferences.

**Verbosity.** While a feed post must contain the user's current state, it might also contain other useful information. For example, the state could list the current document being edited or the results of the most recent compile attempt.

### 2.2.1.3 Presentation

Presently, the Facebook-style stream-based presentation model is the gold standard for presenting activity-centered content. Yet, this view may be augmented with more detailed views. For example, the system might present individual profiles (Figure 4) that include a single user's activity statistics and a summary of recent activity or a customized timeline visualization (Figure 5) of the user's activity based on a variety of selectors. Perhaps even more so than in the activity-centered "presentation" dimension, alternate presentation modes are of critical importance to a user-centered awareness mechanism. This is because, unlike the activity-centered model, a user-centered awareness mechanism does not have a notion of a history. Therefore, providing views that contain both individual and group histories becomes an important aspect to consider. From a research standpoint, these views raise the issues of content (what to present) and interactivity (manipulability of the view data).

### 2.2.1.4 Summary

Both Situated Learning Theory and the construct of self-efficacy place heavy emphasis on peripheral awareness. For Situated Learning, peripheral awareness leads to the development of an active and vibrant community. Self-efficacy sees peripheral awareness as promoting opportunities for vicarious experiences, which are important factors in shaping one's opinion of one's own abilities. Therefore, the goal of any SPE should be the promotion of peripheral awareness by providing users with knowledge of their peer's activity.
Figure 3: User-Centered View

Figure 4: User Profile Page

Figure 5: Timeline Visualization
An activity-centered approach accomplishes this through a continuously-updated activity feed where actions are streamed in real time to the rest of the group. A user-centered approach groups this raw data for each user into states. However, these approaches are not mutually exclusive. It is not hard to imagine a system that offers dual views, or perhaps offers an activity-centered system as a details-oriented overlay to a user-centered view. We speculate that this hybrid approach is likely to be the most effective as it offers the widest variety of choice to the user.

2.2.2 Communication Mechanism

Unlike awareness mechanisms, a communication mechanism facilitates direct communication between two or more users. Communication mechanisms have two dimensions, which are described below.

2.2.2.1 Mode

In the context of a generalized communication mechanism, mode refers to the rate at which users can transmit and receive messages within the system. The mode can be either synchronous, as is the case with instant messaging, or asynchronous, as is the case with message boards.

2.2.2.2 Groundedness

We use the term groundedness to denote how anchored the discussion is in the artifact being discussed. For example, a lightly grounded system would have little connection between the conversation and artifact. An example of a lightly grounded system would be the chat functionality provided by Facebook ("Facebook," 2016). In contrast, a more highly grounded system might have discussions attached to the artifacts being discussed or might even embed the discussion within the artifact itself (e.g. Suthers & Xu, 2002).

The ability to pose context-specific questions is an important aspect of SBL's design studio and critical review process. Consider the typical computing course in which an SPE is not present. In order for a student to receive help on a specific code segment, he or she must describe the situation with enough detail and context (i.e. code) that another person can replicate the issue. An SPE without a review system will likely
not fare much better as conversations would still occur outside the authentic programming context. Likewise, in a more formal review process (see Hundhausen, Agrawal, Fairbrother, & Trevisan, 2009), students are required to create and annotate printouts of their code, which must then be transferred back to its original digital form. Again, such a process would benefit from having reviews take place within the same context in which the coding takes place. In order to lower the barriers to posing context-specific programming questions, an SPE should support a review system that enables students, instructors, teaching assistants, and mentors to pose and answer programming questions directly within the contexts in which they arise (Figure 6). Highly grounded systems have three subdimensions, as described below.

Annotatability. This dimension refers to the subcomponents of an artifact that can be annotated. In the case of an SPE, the most obvious candidate is the code window. Yet, one can imagine scenarios in which it would useful to be able to annotate a number of other components of an IDE, such as compilation error windows, run-time exception windows, call stack windows, breakpoints in the code, variable watch windows, and program output windows. Furthermore, artifacts generated by the SPE itself, namely activity feed post items, also lend themselves to being annotated.

Granularity. This dimension focuses on the level at which annotations can be made for a given component. For example, assuming that an SPE supports the annotation of code, should such annotations be allowed at the level of the document, function, line, sentence, or word? Likewise, is merely annotating the variable watch window sufficient, or should the system allow annotations of specific items within the watch?
Executability. Finally, since many issues that learners encounter as they learn to program have to do with run-time behavior (Ko, Myers, & Aung, 2004), the executability sub-dimension focuses specifically on annotations of program state. In order for one to obtain help with a run-time problem, must such annotations exist in a "live" execution environment that allows anyone in the community to execute the code, or are non-executable static snapshots of the IDE at those execution points (akin to screenshots) sufficient?

2.2.2.3 Summary
While awareness systems are appropriate for promoting peripheral awareness, they do little to support the community and artifact-centered discourse deemed important by Studio Based Learning and Situated Learning Theory. For this, more dedicated communication mechanisms with the dimensions described above become necessary.

2.2.3 Pedagogy
While the previous two sections discuss how important aspects of Situated Learning Theory and Self Efficacy Theory might be incorporated into an SPE, they do not consider the practical implications of implementing Studio Based Learning or any other pedagogy based on Situated Learning Theory into the classroom. This section discusses potential issues of privacy, ways in which to construct learner
communities, and methods to encourage SPE use by students that may arise when integrating an SPE into a course.

2.2.3.1 Privacy

A central issue to the creation of an SPE lies in students' willingness to share personal information with online communities. Given the widespread use of online services that collect personal information, people are becoming increasingly conscious of what they share online (see, e.g. Dwyer, 2011). While there is evidence to suggest that younger generations are more willing to reveal personal information to others (see, e.g. Barnes, 2006; Young & Quan-Haase, 2009), others are suggesting the opposite (see, e.g. Dey, Jelveh, & Ross, 2012). This tension yields the question of how nuanced SPE privacy settings need to be. Three privacy subdimensions are a) the control learners have over the extent to which they reveal their learning activities and progress, b) the level of access learners have to others' learning activities, and c) the level of anonymity provided to members. From the outset, this subdimension will be constrained by U.S. FERPA Law (U.S. Department of Education, 2012), which stipulates that student grades may not be revealed to others without their consent, and that student work may not be revealed to those outside of the class without their consent.

2.2.3.2 Learning Community Composition

This dimension concerns the people to be included in the community that has access to a given learner's activities. This includes the size of a given learners' support community (the entire class, section, or smaller "programming circles"), how community members are selected (by the instructor, or by the learners themselves), and what mentors (key in Situated Learning Theory) are included (including teaching personnel, peer mentors from more advanced courses, and even industry professionals).

2.2.3.3 Instructor Tools

In order to gain broad acceptance in the academic community, an SPE must provide some tangible benefit to instructors. Likely, this benefit will be realized in the form of a centralized instructor dashboard. Possible elements of this dashboard include information on student activities (e.g. common partners, lines
of communication, etc.), common stumbling blocks within a programming assignment, possible attempts at cheating, and classroom management tools (e.g. assignment creation, user management features, etc.).

2.2.3.4 Incentive Mechanism

As the usefulness of social software is based on the rate at which it is used (Grudin, 1988), a key issue of any SPE is the incentive mechanism that will most effectively promote active participation among students. In formal academic settings, instructors can tie participation within the SPE to grades. Alternatively, an SPE might support gamification (Deterding, Dixon, Khaled, & Nacke, 2011)—the inclusion of game-like elements. For example, an SPE might include achievements (see, e.g. Jakobsson, 2011) for accomplishing certain tasks, or be awarded points or levels based on progression (see, e.g. "Reputation Patterns - Design Pattern Library - YDN," 2012). In this way, gamification can be used to represent community artifacts, which are known to be an important aspect in the development of a community of practice (Lave & Wenger, 1991). These artifacts serve as evidence of membership and/or progression within a community (Wenger, 1998) and are vital in the progression from legitimate peripheral participation to full participation (Lave & Wenger, 1991).

2.2.3.5 Summary

The pedagogical decisions made when implementing Situated Learning Theory have an immense impact on its overall pedagogical effectiveness. For example, the level of anonymity given to students as they complete reviews is known to affect group effectiveness (Sosik, Avolio, & Kahai, 1997). Likewise, mentorship in the form of moderation, has been found to be a key factor in the effectiveness of pedagogical code reviews (Hundhausen, Agarwal, & Trevisan, 2011). Therefore, to determine the maximum effectiveness of an SPE, it may be necessary to test the same SPE under different pedagogical conditions.
2.3 Relating Existing Work to the Construction of an SPE

Given the many choices one must make when developing an SPE, it is important to have a broad understanding of related work. In this section, we review relevant research and attempt to relate it to the construction of a social programming environment.

2.3.1 Learning Management Systems

Numerous online learning management systems (LMSs), both commercial (e.g., Blackboard, 2012) and open source (e.g., Dougimas & Taylor, 2003), have been developed to support online learning activities within formal education settings. A central feature of LMSs is their support for computer-mediated communication (see Najafi, Ellis, Cox, & Calvert, 2007). While such communication has traditionally taken the form of synchronous chat and asynchronous threaded discussion, LMSs are beginning to explore communication features that look similar to those of social networking sites. For example, Piazza (Piazza, 2012; Rusli, 2011) enables instructors to create online course communities in which students and instructors can ask and answer questions, track answer progress, and rate answer quality through a social networking-style interface. Aside from its focus on rigorously evaluating the educational effectiveness of learning technologies, this dissertation differs in two ways: (a) it makes learners' problem-solving activities (as opposed to only their products) available to the learning community; and (b) it enables learners to ask and answer questions within the specific problem-solving contexts in which they occur.

2.3.2 Novice Programming Environments

There exist numerous attempts at making the environment in which students program more conducive to learning (see Guzdial, 2004; Kelleher & Pausch, 2005). As expected, the design of these systems is usually dictated by what designers see as the primary educational barrier. Kelleher and Pausch (2005) developed a taxonomy that attempts to situate the motivations behind each system. Using this taxonomy, the work of an SPE would fall underneath the heading of network-supported social learning (section 3.1.2). Kelleher and Pausch found only three systems (out of nearly 100) that matched this description and noted in their conclusion that social learning environments were a promising direction for future researchers.
Indeed, since the time of Kelleher and Pausch's publication, several environments that utilize social interaction have been developed. Broadly speaking, these environments either attempt to support traditionally co-located activities (e.g. pair programming) over the Internet or use social data to provide students with additional programming insights.

Collabode (Goldman & Miller, 2009, 2011), JavaWIDE (Jenkins et al., 2012), and Saros (Salinger, Oezbek, Beecher, & Schenk, 2010) are all examples of systems that facilitate real-time collaboration over a networked environment. Common features include the ability to co-edit a document, see other documents being edited by other users, and view document edits made by other users. While this line of research does show promise, it is unable to support individual assignments in introductory courses. Because these systems promote the collaborative construction of programming assignments, it becomes difficult for instructors to assign individual credit. In contrast, the model utilized by an SPE encourages students to collaborate on different solutions to the same assignment. Doing so makes it possible for instructors to assign credit at the individual level.

Another approach to making learning more social has been to pair learning environments with ways to share and discuss projects online. For example, the Scratch novice programming environment (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010) supports both an online learning community where learners share their programming creations, and an online teaching community where teachers share and discuss their teaching practices (Brennan, 2009). The proposed SPE expands on this approach by enabling teachers and learners to engage in grounded discussions of not only computer programs, but also the detailed programming processes that generated those programs.

HelpMeOut (Hartmann, MacDougall, Brandt, & Klemmer, 2010) and Crowd::Debug (Mujumdar, Hallenbach, Liu, & Hartmann, 2011) are examples of systems that use the programming behavior of other students to provide code recommendations to other students within the class. Both systems record errors encountered by other students as well as the method in which the error was solved. When a student in the future encounters a similar error, the system then presents the student with a list of recommended fixes. Using this strategy, HelpMeOut was able to find useful recommendations 47% of the time while
Debug::Code was more successful with a recommendation rate of 57%. While not a novice environment, LemonAid (Chilana, Ko, & Wobbrock, 2012) is a related system that attempts to provide UI navigation tips based on the experiences of others. LemonAid achieved a much higher success rate of 90%. In contrast with these recommendation systems, the SPE views programming not as a solo activity in which learners search for help when stuck, but rather as a community activity in which learners can see what others are up to, identify others' activities related to their own, and actively participate in the learning process by asking others within the community for help, and by offering help to others.

2.3.3 Cheating

The advent of word processors and the Internet has made it much easier to share information and documents. Concern over the misuse of these resources for the purposes of cheating or plagiarism is widely reported and discussed in the educational literature (see, e.g. Fontaine, 2012; Mastin, Peszka, & Lilly, 2009). However, while it is easy to think of cheating as a modern phenomenon whose prevalence is ever-increasing, the fact is that students have been cheating for a very long time. In 1911, the Registrar of Stanford wrote, "the freshman sees the game of cheating going on as almost a matter of course" (Elliot, 1911; quoted in Sheard & Martin, 2011). Surprisingly, studies have found that cheating from online sources is just as likely as cheating from non-digital material (Selwyn, 2008), and according to one study, the amount of online cheating is actually on the decline (Sheard & Martin, 2011). Nevertheless, considerable effort has been spent on plagiarism detection (see, e.g. Fontaine, 2012; Hage, Rademaker, & Vugt, 2011) and concerns about cheating continue to be on the minds of many educators. For example, when presenting a poster on the SPE at a recent conference, we observed that many visitors' first comments were related to cheating. Indeed, the proposed design for the "code view" feature of the SPE has been constrained due to instructor's concerns over cheating.

By remaining active during students' programming activities, an SPE can make two contributions to the literature on cheating. First, because an SPE is always "on," we will be able to provide more accurate quantifications of the amount of cheating that takes place. Second, rather than relying on post-hoc cheat
detection, an SPE can perform better cheat detection through the real-time monitoring of copy and paste events as well as automatically comparing document diffs. Note that the use of SPEs to study cheating is beyond the scope of this dissertation.

2.3.4 Privacy

In a recent interview, Facebook CEO Mark Zuckerberg was quoted as saying, "People have really gotten comfortable not only sharing more information and different kinds, but more openly and with more people" (Mike Arrington with Mark Zuckerberg, 2011). This statement aligns with a commonly held sentiment that people, especially younger generations, are more comfortable with revealing personal information online (see, e.g. Barnes, 2006). While it is true that, generally speaking, people are more willing to reveal personal information online versus in-person (Christofides, Muise, & Desmarais, 2009), people also remain cautious about what gets revealed. For example, while most Facebook users provide basic information such as profile pictures, birthdays, and names, users are much less likely to reveal more sensitive information such as phone numbers (R. Gross & Acquisti, 2005; Young & Quan-Haase, 2009). Furthermore, shared information is much more likely to be viewable only by immediate friends (see, e.g. Dey et al., 2012; Young & Quan-Haase, 2009). From this arises an important question: under what circumstances are people willing to reveal their personal information?

Research suggests that, under certain circumstances, people are more willing to reveal personal information. First, people are willing to trade personal information if doing so results in a perceived benefit (Youn, 2005). Another motivator for revealing personal information for increased social capital (Christofides et al., 2009; Ellison, Steinfield, & Lampe, 2007). However, those that are less interested in gaining popularity are less likely to share information for this reason (Christofides et al., 2009). This can be tied back to the notion of perceived benefits. Those that see increased social capital as a benefit are willing to trade privacy. However, those who do not see social capital as a benefit remain unaffected. In the context of an SPE, this might mean that as long as users find the features of an SPE beneficial, they may be willing to expose their programming activities to others. It would also seem that perceived "benefits" are
somewhat subjective, and for that reason, it would be helpful for an SPE to provide several benefits in order to appeal to the broadest audience. Achievements or a point-based system might appeal best to people more interested in gaining social capital, whereas providing additional opportunities to learn would appeal best to those that are more academically motivated.

2.3.5 Social Programming Aids

Similar in purpose to social programming environments for novices, social programming aids use social data in an attempt to aid the programming process. However, in contrast to social programming environments, the focus is less on teaching how to program and more on promoting programmer efficiency and/or team collaboration.

Several systems have been developed to better support team collaboration. In many ways, Jazz (Hupfer, Cheng, Ross, & Patterson, 2004) is for professional development what an SPE is for novice programming. Jazz promotes "contextual collaboration," or the integration of collaborative elements within the IDE. To this end, Jazz supports anchored conversations and displays a document's edit history. However, the centerpiece of Jazz is the Jazz Band, a profile-based hub that shows each team member's current activity (online, document currently being edited, etc.). The Jazz Band also serves as the launch point for initiating email or chat conversations with other users. FastDash (Biehl, Czerwinski, Smith, & Robertson, 2007) is a similar system that exists outside of the IDE and provides peripheral awareness of team activities. Another approach is to support team process through an ability to collaboratively annotate and review computer code in a shared code base (see, e.g. Guzzi, Hattori, Lanza, Pinzger, & van Deursen, 2011; SmartBear Software, 2012). As with real-time collaborative learning environments, these social team systems are intended to be used by programmers as they work on a shared project. An SPE needs to facilitate collaboration across individual projects that must meet identical requirements.

Another use of social tools is to better introduce newcomers to a programming project. TeamTracks (DeLine, Czerwinski, & Robertson, 2005) uses the activities of other programmers to display the popularity and relatedness of code segments. Similarly, CARES (Guzzi & Begel, 2012) allows programmers to view
profile information (e.g. job, email, activity) and contact of any user that made recent edits to a given code segment. While the proposed SPE does not have an analog for this kind of social help system, it does provide an avenue for future research opportunities, as these ideas can be adapted for an SPE. For example, instead of highlighting popular code within a single project, an SPE might highlight common problem areas across all student projects (e.g. "50% of students encountered an error when implementing function foo").

The final family of social programming tools involves the integration of web resources directly into the IDE. Codetrail (Goldman & Miller, 2009) incorporates language documentation pulled from the web and displays it in the IDE, thereby removing the need to look for this information using other means. Similarly, Blueprint (Brandt, Dontcheva, Weskamp, & Klemmer, 2010) automatically pulls relevant code segments into the IDE from the web. Calcite (Mooty, Faulring, Stylos, & Myers, 2010) provides automatic object instantiation based on popularity. This family of tools differs from SPEs in that the tools emphasize productivity over learning.

2.4 Predictors of Success in Computing Courses

A large body of educational research has explored the extent to which various learner variables are able to predict learning outcomes or future learning behaviors. These variables include the learner's background (e.g. Bransford, Brown, & Cocking, 1999; Jeske, Stamov-Rossnagel, & Backhaus, 2014), prior knowledge (e.g. Bransford et al., 1999; Ma, Adesope, Nesbit, & Liu, 2014), cognitive abilities (e.g. Schunk, 2012), time-on-task (e.g. Slavin, 2011) and learning attitudes (e.g. Bergin, Reilly, & Traynor, 2005). In computing education, for example, Rosson et al. (2011) found strong positive correlations between a number of attitudinal variables, including self-efficacy, and a learner's orientation towards the computing discipline. While this line of research shares our interest in predicting student learning outcomes, it differs in that it relies on only a limited number of data snapshots to make its predictions. Thus, it lacks the ability to furnish predictions of student performance that are dynamic, robust and continuously updated throughout a course.

The research presented in this dissertation analyzes a continuous stream of data in order to identify patterns of learning associated with positive learning outcomes. As such, it falls within the emerging areas
of educational data mining and learning analytics (Baker & Siemens, 2014; U.S. Department of Education, Office of Educational Technology, 2012), which, in many STEM fields, have been used to gain insights into the processes that underlie student learning, and ultimately to better tailor instruction. A foundational idea is to build learner models that infer learners’ background knowledge, learning strategies, and motivations from learning process data (Ma et al., 2014). In turn, such models are used to adapt instruction to learner needs.

Within computing education, a legacy of research has studied students’ programming processes using think-aloud protocols (e.g. Kessler & Anderson, 1986; Spohrer, 1992), video analysis (e.g. Hundhausen, Brown, Farley, & Skarpas, 2006), and software logs (e.g. Goldenson & Wang, 1991; Guzdial, 1994). These studies have had a variety of goals, ranging from better understanding how novices approach programming and debugging (e.g. Ahmadzadeh, Elliman, & Higgins, 2005), to developing cognitive models of student programming knowledge (e.g. Kessler & Anderson, 1986; Spohrer, 1992), to evaluating novice programming environments (Goldenson & Wang, 1991; Guzdial, 1994; Hundhausen et al., 2006) While carrying forward its interest in studying students’ programming processes in detail, our work differs from this line of work in that it aims to make accurate advance predictions of course performance using learning analytics.

2.5 Learning Analytics

In recent years, the ease with which data can be collected, coupled with the availability of low-cost, high-power machines to store and process such data, has led to an increased interest in the field of data analytics. For example, researchers have mined data repositories to describe tool usage (Khodabandelou, Hug, Deneckere, & Salinesi, 2014) and to detect difficulty in programming tasks (J. Carter & Prasun, 2010). The application of data mining and analytics techniques to education is referred to as learning analytics (Wise, 2014).

Duval and Verbert (2012) define learning analytics as a research area that, “focuses on collecting traces that learners leave behind and using those traces to improve learning.” They then go on to define two
research approaches: one that focuses on identifying patterns of behavior and another that focuses on deriving interventions aimed at improving the learning process. In computing education, the former approach has been used to describe compilation behavior (e.g. Altadmri & Brown, 2015; Jadud & Dorn, 2015) and to identify behaviors associated with eventual success or failure in a course (e.g. Ahadi et al., 2015; A. S. Carter, Hundhausen, & Adesope, 2015; Jadud, 2006b; Watson et al., 2013). In computing education, the latter approach (developing learning interventions) has been less explored, although there are some notable exceptions (e.g. Hartmann et al., 2010; Mujumdar et al., 2011). Nevertheless, how to best approach this emerging field within the context of computing education remains an open question (Wise, 2014). As this dissertation is more aligned with research interested in identifying patterns of behavior, we devote additional consideration to learning analytics from the perspective of data collection and analysis methodology.

2.5.1 Data Collection

What can be accomplished through learning analytics depends heavily on the type and amount of data that can be collected. Data collection is often a technological constraint. For example, when storage and processing power are limited, it may only be possible for researchers to only take snapshots of a single aspect of student behavior (e.g. compilation, see Jadud, 2006b). However, when such technological restrictions are lifted, it becomes possible to collect every keystroke (e.g. Leinonen, Longi, Klami, & Vihavainen, 2016) or IDE interaction (e.g. Brown, Kolling, McCall, & Utting, 2014) made by every student. While this may seem to encompass the entirety of what researchers might collect from students, there are several additional sources of data that have not yet been explored. For example, researchers could collect mouse gestures (see Arapakis, Lalmas, & Valkanas, 2014), or behavioral data such as voice, facial expressions, or eye gaze (e.g. Busjahn et al., 2014). Alternatively, researchers may decide to move beyond the IDE as the data collection mechanism and instead consider general application usage. For example, it might be beneficial to record web searches so that we can better understand where students are getting stuck
and how they are using online resources. Furthermore, it may be of interest to log similar interactions on smartphones or other mobile devices.

2.5.2 Analytic Approach

Thus far, researchers have used either statistical analysis (see Field, 2013) or machine learning (see Bulling, Blanke, & Schiele, 2014) to gain insights into the data collected.

The majority of learning analytics research in computing education has attempted to develop statistical models that relate one or more behavioral characteristics to course outcomes. Two statistical predictors are the Error Quotient (Jadud, 2006b; Tabano et al., 2011), Watwin Score (Watson et al., 2013, 2014). Both measures focus on quantifying a student's ability to recover from compilation errors. This is accomplished by examining successive pairs of compilation attempts and awarding points based on whether later compilation attempts remove errors identified in earlier compilation attempts. In past studies, the Watwin Score has generally outperformed the Error Quotient as a predictive measure. While the Error Quotient was able to account for between 19% (Watson et al., 2014) and 25% (Jadud, 2006b) of the variance of final course grades, the Watwin score was able to account for between 36% (Watson et al., 2014) and 42% (Watson et al., 2013) of the variance in final course grades. However, a refinement of the Error Quotient published more recently appears to raise the Error Quotient's predictive power to nearly 30% (Tabano et al., 2011). In Chapters 5 and 6, we propose a new statistical model and consider its usefulness in predicting a student's course outcomes.

Recently, researchers have begun to employ machine learning techniques as a method of analysis. For example, researchers have used machine learning to investigate the relationship between course outcomes and student factors (e.g. gender, see Ahadi et al., 2015), programming activity (Ahadi et al., 2015), and patterns of typing (Leinonen et al., 2016). Similarly, researchers have used machine learning to analyze common novice mistakes when writing SQL queries (Ahadi, Behbood, Vihavainen, Prior, & Lister, 2016).

A weakness of this line of work is a lack of ground truth analysis—a technique commonly applied in human activity recognition (e.g. Kim, Helal, & Cook, 2010; Minor, Doppa, & Cook, 2015). In ground truth
analysis, the selected machine learning algorithm is trained on carefully labeled data obtained from controlled laboratory observations of participants performing tasks. Without obtaining the ground truth, approaches utilizing machine learning become difficult to interpret and often lack generalizability. For computing educators interested developing a ground truth will likely require lab observations of students working on carefully constructed programming problems. We foresee the establishment of a ground truth for learning analytics as a key contribution to the research space.

2.5.3 SPEs and Learning Analytics

Given the close proximity of an SPE to students' problem solving processes, it seems natural for an SPE to act as both a data collection mechanism and an interactive user interface for learning analytics. Indeed, we believe that the SPE is the idea venue for such ventures. To this end, we foresee a symbiotic relationship between the design of an SPE and the design of learning analytics. For example, the design of an SPE might influence choices made when constructing learning analytics mechanisms. Likewise, choices made for the purposes of developing learning analytics mechanisms will likely influence the design of an SPE. Elucidating the relationship between SPEs and learning analytics deserves its own design space exploration, which we leave for future work.

2.6 Conclusion

In this chapter, we used the theoretical frameworks of situated learning theory and social cognitive theory to develop a design space exploration of social programming environments. In this design space, we consider possible SPE implementations and their potential impact on classroom learning, privacy, and cheating. Furthermore, we consider how an SPE might contribute to the emerging field of learning analytics. In the next chapter, we consider the pedagogical usefulness of activity streams, a key aspect of the design space of social programming environments.
CHAPTER 3

EXPLORING THE PEDAGOGICAL USEFULNESS OF ACTIVITY STREAMS IN COMPUTING EDUCATION

As a first step towards the development of an SPE, we explore how the centerpiece of an SPE—the activity stream—might be appropriated within early computing courses. Because these courses traditionally discourage social interaction among students, we were curious as to whether or not students would actually use an activity stream to engage in online conversations. In this context, we were interested in the following research questions:

RQ1: To what extent do students appropriate an activity stream as they engage in computer programming assignments?

RQ2: To what degree are conversations in an activity stream educationally relevant?

RQ3: What is the relationship between social activity and class outcomes?

To investigate these questions, we present in this chapter a series of studies that compared courses in which activity streams were integrated at differing levels into the course: through an external social networking tool and through a learning management system. By establishing not only that students actively engage in educationally relevant discussions through an activity stream, but also that they are more likely to do so if the activity stream is more closely integrated with their problem-solving environment, the studies of this chapter provide the beginnings of an empirical case for the potential for social programming environments to support the kind of participatory learning envisioned by social learning theory.

3.1 Study I: Exploring Facebook Activity Streams

In Study I, we considered how students appropriate Facebook's activity stream in two separate courses offered in the School of Electrical Engineering and Computer Science at Washington State University. Facebook, being very popular among students (Arrington, 2005; Forrester, 2014), seemed like a natural starting point as most students already have an account and actively monitor updates to their activity feeds.
The first course, CptS 111 (Introduction to Algorithmic Problem Solving), took place in the (15-week) spring semester of 2011, and enrolled 39 students (31 male, 8 female). As a "CS ½" course, the course is taken by a mix of computer science majors and non-majors. The course enrolled mostly underclassmen, including 27 freshman and 5 sophomores.

CptS 121 (Program Design and Development), took place during an 8-week summer session in 2012, and enrolled 20 students (17 male, 3 female). A classic "CS 1" course, CptS 121 is the first required course for majors at Washington State University. It also fulfills degree requirements in other science and engineering disciplines. Because the course was being offered in the summer, it enrolled an unusually high number of upperclassmen looking to fulfill degree requirements. Indeed, 11 students were non-CS majors, and 14 students were either juniors or seniors.

### 3.1.1 Data Collection and Analysis Method

To obtain the data analyzed in this study, we obtained transcripts of the activity streams generated in the Facebook groups of each course considered in the study. Since the research questions posed for this study all relate to the focus and content of the activity streams, we chose to employ content analysis (see Krippendorff, 1980) as our primary method of analyzing these transcripts. Using the activity stream post as the atomic unit of analysis, we iteratively developed a content coding scheme by reading through excerpts of activity stream transcripts, adding and refining categories until the categorical definitions stabilized and no new categories emerged. The complete coding scheme is presented as an appendix.

RQ1 ("To what extent do students appropriate an activity stream as they engage in computer programming assignments?") addresses the extent to which students and instructors participate in the activity stream. To get at this, we classified each post according to whether it was written by a student or instructor.

RQ2 ("To what degree are conversations in an activity stream educationally relevant?") concerns the relevance of activity stream discussions. To that end, Table 1 briefly describes the top-level categories in our content coding scheme. These categories distinguish the key topics considered in activity stream
discussions. We viewed posts coded as CODE to be the most relevant to the courses studied. To further explore the presence of code in posts, we used the code presence categories described in Table 2.

RQ3 ("What is the relationship between social activity and class outcomes?") is addressed by considering the relationship quantity of posts made by a student and that student's final grade. We explore RQ3 again later in this chapter.

To verify the reliability of the coding scheme, two researchers independently coded a 15% sample of the corpus \((n = 99\) posts), attaining the following levels of agreement: 97.0\% (0.95 kappa) for the top-level categories (Table 1) and 96.0\% (0.86 kappa) for the code presence categories (Table 2). Having established high levels of inter-rater reliability for our coding schemes, we had each researcher code half of the remaining corpus.

### 3.1.2 Analysis

We focus our analysis around our original research questions.

#### 3.1.2.1 RQ1. To what extent do students appropriate an activity stream as they engage in computer programming assignments?

We answer our first research question by examining the categorization of activity feed posts within each class. Figure 7 presents a top-level categorical breakdown of the posts in each course and Figure 8 provides the categorical breakdown of posts categorized as CODE. In both figures, bar heights represent the percentage contribution of each category to the overall number of posts in each course, thus supporting a direct comparison of the two courses. Bars are shaded to indicate the percent contribution of student and instructor posts to each category.

As illustrated by Figure 7 and 8, students are capable of supporting the day-to-day activities of early computing courses, and that the strong majorities of students post to the activity stream (62\% in CptS 111, and 80\% in CptS 121). It was notable that student participants in CptS 121 were more than twice as active as those in CptS 111. One might explain this by observing that CptS 121 students were generally further
along in their academic careers than students in 111. Perhaps because they were more mature, CptS 121 students were more willing to use the activity stream.

Table 1. Top-Level Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Post refers to computer coding issues, problems, or questions. We distinguish three subcategories:</td>
<td>—</td>
</tr>
<tr>
<td>COMPIL</td>
<td>Post is in the service of addressing issues that arise from a specific compilation action or event (present or future).</td>
<td>“I keep getting the following error: unrecognized identifier foo”</td>
</tr>
<tr>
<td>RUNTIME</td>
<td>Post is in the service of addressing issues that arise from a specific runtime action or event (present or future).</td>
<td>“When I call foo(), why does it always return the value value 3?”</td>
</tr>
<tr>
<td>IDE</td>
<td>Post refers to an issue related to the Integrated Development Environment (IDE).</td>
<td>“How can I bring up the solution explorer in Visual Studio?”</td>
</tr>
<tr>
<td>CONCEPTUAL</td>
<td>Post addresses general conceptual coding questions, and is not in the service of addressing an issue with a specific compile or runtime event.</td>
<td>“How am I supposed to define the custom string_len() function in this assignment?”</td>
</tr>
<tr>
<td>COURSE</td>
<td>Post is relevant to course, but does not specifically address computer programming. We distinguish three subcategories:</td>
<td>—</td>
</tr>
<tr>
<td>GRADES</td>
<td>Post refers to student grades or course grading system.</td>
<td>“I didn’t get a grade for my last programming assignment.”</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>Post refers to issues with, scheduling of, or content of a particular course lecture, assignment, lab, or exam.</td>
<td>“When is Assignment #3 due?”</td>
</tr>
<tr>
<td>OTHER</td>
<td>Post relates to course in general, but does not refer to grades or assignments.</td>
<td>“Where can I obtain a copy of the course syllabus?”</td>
</tr>
<tr>
<td>COORDINATION</td>
<td>Post coordinates present or future collaboration or help between instructors and/or students.</td>
<td>“Can you take a look at my code if I e-mail it to you?”</td>
</tr>
<tr>
<td>OTHER</td>
<td>Post that is relevant to a discussion, but that does not fall into the above categories.</td>
<td>“Cool!”</td>
</tr>
<tr>
<td>OFF-TOPIC</td>
<td>Post that is not relevant to the discussion in which it appears.</td>
<td>“It rained a lot last night.”</td>
</tr>
</tbody>
</table>

Table 2. Modifier Categories

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLY</td>
<td>Post is a reply to a previously-established thread.</td>
<td>—</td>
</tr>
<tr>
<td>QUESTION</td>
<td>Post asks a question.</td>
<td>“What’s wrong with my code?”</td>
</tr>
<tr>
<td>SUGGESTION</td>
<td>Post suggests a solution, partial solution, or information that could be construed as a (partial) solution to a previously-mentioned question.</td>
<td>“Try stepping through it in the debugger.”</td>
</tr>
<tr>
<td>UPDATE</td>
<td>Post provides an update to the class. (Usually posted by instructor.)</td>
<td>“I’ve posted Assignment #2.”</td>
</tr>
<tr>
<td>GRATITUDE</td>
<td>Post expresses gratitude for a previously posted suggestion or reply, or acknowledges that a previously posted suggestion was helpful.</td>
<td>“Thanks! It worked!”</td>
</tr>
<tr>
<td>AGREEMENT</td>
<td>Post expresses agreement with a previous post.</td>
<td>“Me too?”</td>
</tr>
<tr>
<td>DISAGREEMENT</td>
<td>Post expresses disagreement with a previous post.</td>
<td>“That’s not how to solve the problem.”</td>
</tr>
</tbody>
</table>
3.1.2.2 RQ2. To what degree are conversations within the activity stream educationally relevant?

As our results indicate, in CptS 121, a solid majority of activity stream posts (57.6%) focused on computer programming issues, whereas just 18.8% of posts in CptS 111 focused on computer programming issues. We can offer at least two possible explanations for this difference. First, CptS 121 is a more academically rigorous course from a computer programming standpoint. It moves at a faster pace than CptS 111, and requires students to come to grips with low-level computing concepts (e.g., dynamic memory and pointers) that simply are not present in CptS 111. It makes sense that this level of rigor would give rise to more problems with code, and hence more activity stream posts about code. Second, the fact that CptS 121 was offered in a compressed 8-week summer session put greater pressure on students. Perhaps because of the urgency CptS 121 students felt to get assignments done quickly, they resorted more heavily to the activity stream for help.

3.1.2.3 RQ3. What is the relationship between social activity and class outcomes?

In this section, we consider the relationship between the quantity of posts made by a student and the student's course grade. A regression analysis could not detect a statistically significant relationship between these two factors ($r = 0.11, p = 0.55$ for CptS 111 and $r = 0.34, p = 0.15$ for CptS 121). However, this does
not necessarily imply that activity streams lack educational value. Later in this chapter, we consider the relationship between regular posting activity and grades.

3.1.2.4 Summary

The results of Study I indicate that, when given the opportunity, students naturally appropriate an activity stream for academic purposes. Furthermore, we see that activity streams can be used to facilitate discussions of coding artifacts. In the next section, we consider the use of an activity stream that is more tightly integrated into students' learning environment.

3.2 Study II: Integrating Activity Streams into a Course's Learning Management System

In Study II, we considered the use of an activity stream within CptS 122, the "CS2" course at Washington State University. Unlike in Study I, the activity stream resided within OSBLE (HELP Lab, 2012), the learning management system (LMS) used in the course. CptS 122 enrolled 123 students (111 male, 12 female) and was primarily composed of students majoring in electrical engineering, computer engineering, and computer science.

In this study, we again employed content analysis. However, we used a slightly modified version of the coding manual used in Study I. While all of the top-level categories remain the same, we added two additional subcategories to the top level CODE category (RESOURCES and SOFTWARE) and merged the OTHER and OFF-TOPIC categories. We did this in order to better differentiate between questions that likely relate to present coding issues and those less likely to be grounded in code. A more detailed description of the changes to the coding system can be found in Table 3.

Since we made changes to the coding system, we needed once again to verify its reliability. Two researchers independently coded a 15% sample of the corpus \( n = 125 \) posts and obtained the following levels of agreement: 95% (0.89 kappa) for the top-level categories (Table 1), 87% (0.84 kappa) for subcategories (Table 2). Having established high levels of inter-rater reliability for our coding schemes, we had each researcher code half of the remaining corpus.
3.2.1 Results

Figure 9 presents the top-level content categorizations of the posts. Again, bar heights represent the percentage contribution of each category and shading is used to distinguish between student and instructor contribution.

We begin by comparing the distribution of top-level categories with those obtained in Study I, which used Facebook. A chi-squared test revealed a significant association between course and top-level category ($\chi^2(6) = 181.61, p < 0.01$). An associated z-test revealed significant ($p < 0.05$) differences in proportions for the following categories:

1. The proportion of CODE posts in CptS 111 (Facebook) was significantly less than the proportion observed in CptS 121 (Facebook) and CptS 122 (OSBLE).
2. The proportion of COURSE posts in CptS 121 (Facebook) was significantly less than the proportion of COURSE posts observed in CptS 122 (OSBLE). The proportion of COURSE posts in CptS 122 (OSBLE) was significantly less than the proportion observed in CptS 111 (Facebook).
3. The proportion of COORDINATION posts between the three courses did not differ significantly.
4. The proportion of OTHER posts in CptS 111 (Facebook) and CptS 121 (Facebook) was greater than the proportion observed in CptS 122.

As was the case in Study I, students contributed more posts than the instructor: 70% (student) vs 30% (instructor). A chi-squared test revealed this association to be significant ($\chi^2(3) = 14.15, p < 0.01$). An associated z-test revealed significant ($p < 0.05$) differences in the frequency distributions of COURSE, COORDINATION, and OTHER categories between students and instructors. In the cases of COORDINATION and OTHER, students had a higher frequency relative to overall posting behavior. On the other hand, the instructor had a higher relative frequency for posts coded as COURSE.
We next focus on posts classified as CODE (Figure 10). Again, we compare the distribution of posts made in CptS 122 (OSBLE) to CptS 111 and CptS 121 (Facebook) courses using a chi-squared test. Because the frequencies of IDE categorizations were below the recommended level of 5 observations, we were forced to drop that subcategory from the analysis. Likewise, the number of COMPILE categorizations in CptS 111 was below the required amount. For the COMPILE categorization, we only compared CptS 121 (Facebook) and CptS 122 (OSBLE). The chi-squared test revealed the association between course and CODE subcategory to be significant ($\chi^2(4) = 42.54, p < 0.01$). An associated z-test revealed the following significant ($p < 0.05$) differences:

1. The proportion of COMPILE posts in CptS 121 (Facebook) was significantly less than CptS 122 (OSBLE). (CptS 111 was not considered).
2. The proportion of CONCEPTUAL posts in CptS 111 significantly less than CptS 121 and CptS 122.

3. The proportion of RUNTIME posts in CptS 111 and CptS 121 was significantly less than CptS 122.

Next, we consider the relationship between posting behavior and final grades. A regression analysis in which the number of posts was the predictor variable and the student's final grade was the outcome variable was found to be statistically significant ($r = 0.25, p = 0.01$).

Lastly, we consider the relationship between gender and the number of posts made. A Welsh Test of Equality of Means revealed that male students made significantly more posts than female students ($F(1, 71.01) < 0.01$). Males (N = 111) made an average of 4.8 posts (SD = 7.31) over the term whereas females (N = 12) made an average of 0.8 posts (SD = 1.48) over the same period.

### 3.2.2 Discussion: Comparing Facebook and OSBLE

The results of Study I and Study II suggest that activity streams, regardless of location, can successfully accommodate asynchronous discussion of code. Given that each study focused on successively more advanced courses in the introductory sequence, we are also able to see how the topics of activity stream posts changed over time. In the earliest course, CptS 111, we see that students were primarily interested in grades, general assignment details, and other general course activity. In CptS 121 and CptS 122, we see interest shift to coding discussion. Furthermore, between CptS 121 and CptS 122, we see another shift from discussions that focus on general concepts to those that deal with specific issues encountered during program execution.

It should also be noted that, unlike the Facebook groups, the OSBLE group had a statistically significant relationship between total posts made and final grades. Likewise, there was statistical significance in the OSBLE group between gender and posting behavior (males tended to post more). Whether this difference can actually be attributed to the OSBLE environment or merely the fact that the OSBLE group had greater statistical power due to larger sample size would require additional follow-up research.

50
In terms of average posting behavior within a given class, we see that CptS 111 averaged 6.95 posts per student, CptS 121 averaged 12.7 posts per student, and CptS 122 averaged 7.85 posts per students. If we consider the posting behavior of only those who made at least one post, we get averages of 10.0, 23.2, and 8.4 posts per student for CptS 111, CptS 121, and CptS 122 respectively. On a per-student basis, it would appear as though Facebook encourages more participation. However, if we instead consider the percentage of students who made at least one post, we see participation rates of 72%, 75%, and 92% for CptS 111, CptS 121, and CptS 122 respectively. This seems to indicate that students who use Facebook will post often, but it presents a barrier to students without Facebook or who do not check the site regularly. On the other hand, although OSBLE is less central to many students' daily behaviors, OSBLE encourages a larger percentage of the student body to participate, perhaps due to its proximity to course materials and grades.

Lastly, we consider the role of the instructor in online discussions. Prior research indicates the need for a strong presence of an expert moderator in online learning environments (Hundhausen et al., 2013). In all three courses, the instructor was the single largest contributor to the activity stream. While these studies did not consider a course in which the instructor was not a central participant, their results seem to support the notion that expert participation plays a critical role in online discourse.

3.3 Study III: Exploring the Relationship between Regular Activity and Grades

In our prior analyses of Study I, we found no relationship between the total number of posts and course outcomes in CptS 111 and CptS 121. In Study II, we found a weak relationship between posting behavior and final grades in CptS 122. However, given that both Situated Learning Theory and Communities of Practice Theory emphasize regular participation, we wondered whether a metric that considered the amount of regular participation, rather than aggregate participation, might provide more insight into the relationship between posting behavior and course outcomes. To that end, we devised a measure that
categorizes students based on the quantity of activity stream posts and replies generated in a two week period—roughly the length of any given programming assignment.

The measure we devised, Social Role, captures both the number of posts and replies a student makes in a two week period. The four levels of Social Role are presented in Table 4. Originally, the measure had thirteen different categorizations. However, preliminary analysis seemed to indicate that most of these categorizations were unnecessary and could be successfully condensed into only four without losing explanatory power. Also, it is worth noting that achieving the maximum Social Role level of 4 requires only a minimum of two posts and two replies. This ceiling was chosen because it aligned with a participation requirement in another class that will be considered later in this dissertation (see Chapter 4).

<table>
<thead>
<tr>
<th>Level</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No posts or replies for a given period</td>
</tr>
<tr>
<td>2</td>
<td>At least one post or reply, but no more than one of either for a given period</td>
</tr>
<tr>
<td>3</td>
<td>Two posts and fewer than two replies, or two replies and fewer than two posts for a given period</td>
</tr>
<tr>
<td>4</td>
<td>Two or more posts and two or more replies for a given period</td>
</tr>
</tbody>
</table>

3.3.1 Data Collection and Analysis Method

For this analysis, we consider the data generated by the same three courses considered in Study I and Study II:

- CptS 111 (Introduction to Algorithmic Problem Solving) took place in the 15-week spring semester of 2011 and enrolled 39 students (31 male, 8 female). This course used Facebook's activity stream.
- CptS 121 (Program Design and Development) took place during an 8-week summer session in 2012 and enrolled 20 students (17 male, 3 female). This course used Facebook's activity stream.
- CptS 122 (Data Structures) took place during the 15-week fall semester of 2013 and enrolled 123 students (111 male, 12 female). This course used OSBLE's (HELP Lab, 2012) activity stream.

For each class, we generated a timeline of each student's social activity. In order to generate the Social Role measure, we recorded posts and replies separately. For CptS 111 and CptS 122, the timeline was
converted into a bi-weekly Social Role number. Because CptS 121 occurred over a condensed 8-week semester, we converted a given week's activity into a Social Role number. This was done in order to match the number of Social Role numbers across all three conditions. These numbers were then averaged for the entire term and compared with each student’s final grade.

3.3.2 Results

Table 5 lists the average Social Role for each polling period for all three courses. A one-way ANOVA between courses revealed a statistically significant difference in participation levels between the courses ($F(2, 158) = 6.01, p < 0.01$). A post-hoc Bonferroni test revealed statistically significant differences between CptS 111 and CptS 121 ($p < 0.01$). The difference between CptS 111 and CptS 122 was not found to be statistically significant ($p = 0.07$). Likewise, the difference between CptS 121 and CptS 122 was not statistically significant ($p = 0.13$).

In order to determine whether or not regular participation was a reliable predictor of academic success, we considered each student’s semester-long average Social Role with respect to final grades using an ANOVA. Table 6 presents the results of these analyses. Only in CptS 122 was there a statistically significant relationship between Social Role and final grades ($F(1, 100) = 10.5, p < 0.01, \eta^2 = 0.20$).

In order to further explore differences in students’ grades vis-à-vis participation level, we partitioned students into quartiles based on their overall average participation level. Table 7 presents the mean final grade (percentage) by quartile. In examining Table 7, we see a negative relationship between participation and Social Role for CptS 111 and positive relationships between participation and Social Role for CptS 121 and CptS 122. However, a between-subjects ANOVA only detected a statistically significant difference between the quartiles for CptS 122 ($df = 3, F = 8.15, p < 0.001$). A post-hoc Bonferroni test revealed that the significant differences lay between the top two quartiles and the lowest quartile ($p < 0.01$).

3.3.3 Discussion

Interpreting the usefulness of the Social Role measure is a bit difficult, given large discrepancies in the results obtained. In CptS 122, Social Role is clearly useful as it accounts for a moderate amount of variance
in final grade. In CptS 111, we see the complete opposite: Not only is there no significant relationship between Social Role and final grade, but there actually appears to be a slightly negative relationship between participation and final grade. CptS 121 lies somewhere in the middle. The quartile splits (Table 7) indicate a positive relationship between participation and final grade. Furthermore, CptS 121 had the most regular participation out of all three courses. Given this, why aren't the results in CptS 121 significant? In this case, it seems likely that the small course size was unable to provide the necessary power to detect a statistically significant difference. In spite of this ambiguity, we posit the following explanations.

Social Role may require a critical mass

Recall that regular social participation, as quantified by Social Role, was significantly lower in CptS 111 than in 122, and that the difference between CptS 111 and CptS 122 was not statistically significant. It stands to reason, then, that in order for regular social participation to be a predictor, there might need to be

| Table 5. Participation Level Averages by Two-Week Intervals (Standard Deviations in Parenthesis) |
|-----------------|-----------------|-----------------|
| **Period**      | CptS 111        | CptS 121        | CptS 122        |
| 1               | 1.13 (0.41)     | 1.00 (0.00)     | 1.21 (0.51)     |
| 2               | 1.10 (0.31)     | 1.90 (1.07)     | 1.42 (0.73)     |
| 3               | 1.13 (0.34)     | 1.25 (0.55)     | 1.32 (0.68)     |
| 4               | 1.03 (0.16)     | 1.80 (0.15)     | 1.35 (0.72)     |
| 5               | 1.05 (0.22)     | 1.45 (0.94)     | 1.25 (0.59)     |
| 6               | 1.18 (0.45)     | 1.50 (0.89)     | 1.40 (0.77)     |
| 7               | 1.33 (0.62)     | 1.65 (0.88)     | 1.23 (0.63)     |
| 8               | 1.36 (0.58)     | 2.10 (1.29)     | 1.39 (0.74)     |
| **Avg.**        | 1.16 (0.39)     | 1.58 (0.85)     | 1.32 (0.44)     |

| Table 6. Extent to Which Participation Level Predicted Course Grades |
|-----------------|-----------------|-----------------|-----------------|
| **Final Grade** | F   | Sig. | η² | F   | Sig. | η² | F   | Sig. | η² |
| Final Grade     | 0.64 | 0.43 | 0.02 | 1.39 | 0.25 | 0.08 | 10.5 | < 0.01 | 0.20 |

| Table 7. Final Grades (%) by Participation Level Quartile |
|-----------------|-----------------|-----------------|-----------------|
| **Quartile**    | CptS 111        | CptS 121        | CptS 122        |
|                 | N   | N   | SD | N   | M   | SD | N   | M   | SD |
| Quartile 1      | 10  | 84.36 | 10.44 | 6   | 82.44 | 7.70 | 20  | 76.9 | 13.3 |
| Quartile 2      | 10  | 84.36 | 10.44 | 3   | 85.02 | 6.86 | 29  | 84.8 | 8.1  |
| Quartile 3      | 6   | 79.28 | 17.05 | 5   | 85.80 | 12.66 | 24  | 86.2 | 5.9  |
| Quartile 4      | 7   | 78.48 | 8.90  | 5   | 88.41 | 5.87  | 36  | 90.5 | 5.9  |
some minimum baseline of regular participation in the class. Fully investigating this possibility would require additional data that put its answering beyond the scope of this dissertation.

**The significance of Social Role may be course-dependent**

Another explanation is that Social Role may be context-specific for a given set of courses. Although it runs counter to social theories of learning (Bandura, 1997), it is possible that Social Role is only an effective predictor in later classes. This dovetails into the next possibility.

**Participation is only a predictor when the course is sufficiently difficult**

Based on the data presented, there is a clear relationship between the difficulty of a course and the significance of Social Role. Perhaps social participation only becomes a significant predictor when the difficulty of the course essentially forces good students to participate.

**Social Role depends on activity stream placement**

Rather than difficulty of coursework, perhaps what made Social Role in CptS 122 significant was the placement of the activity stream. Unlike CptS 111 and CptS 121, the activity stream in CptS 122 was situated in the course's learning management system. Perhaps the placement of CptS 122's activity stream in this more academically-focused environment led to participation that was more academically beneficial.

### 3.4 Summary

In this chapter, we have presented three studies that explored the pedagogical usefulness of activity streams. In Study I, we examined whether or not students would naturally appropriate a Facebook activity stream for course learning. In both cases, we found that the majority of posts related to discussions of code. In Study II, we considered an activity stream within OSBLE, a course management system. Once again, we found that students appropriated the activity stream for academic purposes.

These studies also attempted determine whether or not participation in activity streams is academically beneficial. They did so by relating the total number of posts and replies to course outcomes and found no relationship. In a follow-up study (Study III), we operationalized regular participation using the Social Role measure and found a statistically significant relationship between participation and grades in CptS 122.
While future work will need to perform a more rigorous content analysis of the activity streams, these studies lay the groundwork for the exploration of social programming environments in the chapters that follow. In particular, the correlation observed in Study III—that regular participation in an activity stream closely integrated with a student's academic environment is positively correlated with student grades—suggests that we would do well to explore integrating an activity stream even more closely within student learning and problem-solving environments.
CHAPTER 4

INTRODUCING THE OSBIDE SOCIAL PROGRAMMING ENVIRONMENT

In the prior chapter, we examined the pedagogical usefulness of activity streams in Facebook and a learning management system (OSBLE). We found that students were able to successfully appropriate an activity stream for discussing programming. However, one promising finding was that students were more likely to use the activity stream at least once in the OSBLE condition. We hypothesize that this is due to the proximity of the activity stream to other learning activities. To explore this possibility further, we developed OSBIDE, a plugin that aims to convert a traditional integrated development environment (IDE) into a more social environment for learning to program. To this end, we posed the following research question:

RQ: How can we best adapt features of social networking environments in order to build a more social programming environment?

Drawing on social theories of learning, we present OSBIDE, a social media plugin for the Visual Studio IDE (see Figure 11). We conceptualize OSBIDE as a social programming environment (SPE). OSBIDE contains key features commonly found in social media: an activity feed, a social recommender system, profile pages, and user points and achievements. In addition, through its activity stream, it allows students to participate in conversations, pose questions, and search for solutions to problems within the context of programming. In this chapter, we introduce and motivate the key design features of OSBIDE, discuss the initial results of deploying the environment in early computing courses, and identify research questions to explore in future chapters.
Throughout OSBIDE's design, we have continually sought to ground design decisions in the social learning theories introduced earlier (Bandura, 1997; Lave & Wenger, 1991). These theoretical frameworks place a heavy emphasis on allowing learners to observe the activities of others, and on promoting social interaction within a community of learners. We have attempted to embody these ideas with the following design goals:

1. Students should be able to observe others' activities and progress.
2. Students should be able to observe others' problem solving processes.
3. Students should be able to observe others' social interactions.
4. Students should be able to participate in a social learning community by
   a. Asking questions of others
   b. Answering other's questions
Table 8 explicitly connects these design goals with underlying theory and provides additional implications for design and pedagogy. The following vignettes expand upon our design goals and illustrate how these goals might manifest themselves in a social programming environment.

**Vignette 1: Observing others' activities and progress**

Jill is struggling to complete the "battleship" programming assignment. She doesn't understand how to reference a variable in one of her loops. Her code isn't compiling; she's getting a strange error. In the activity feed, Jill notices that other students recently got a similar error. This makes her feel slightly better.

**Vignette 2: Observing others' problem solving processes**

Jill decides to investigate a particular build error in further detail and clicks on the "See more..." link. She notices that one student, Jack, was able to compile his program just 5 minutes later. Tracing through the evolution of Jack's code, Jill sees that Jack fixed the error by adding an "&" before the variable. She immediately changes her code, and it successfully compiles.

**Vignette 3: Observing others' social interactions**

In the course of exploring events in the activity feed, Jill observes Jack and Jane interacting about code. She also notices that Jack, who recently had the same error she did, is asking the same kinds of questions she has wondered about. Others are helping him. Maybe if she asked, she could get the help she needed too?

**Vignette 4: Asking questions of others**

After executing her solution, Jill notices she's not getting the right answer. She highlights the code that she thinks could be the source of the problem and chooses "ask question" from the context menu. She types in "How can I get this loop to visit all coordinates?" and clicks "post." Her question, along with the code snippet she highlighted, is, posted to the activity feed, where it is immediately fielded by Dora, who just went through a similar struggle. Dora's response provides the insight Jill needs to fix the problem. She rates Dora's response as “helpful,” which positively impacts Dora's community reputation rating.
Table 8: Forms of virtuous learner engagement enabled by OSBIDE, their theoretical benefits according to Bandura’s and Lave and Wenger’s Theories, and design and pedagogical strategies promoting them. The dotted line separates observational engagement from interactional engagement.

<table>
<thead>
<tr>
<th>Type of Engagement</th>
<th>Theoretical Grounding</th>
<th>Implications for Design and Pedagogy</th>
</tr>
</thead>
</table>
| 1. Observing others’ activities and progress | **Bandura:** Observation of others’ activities, progress, and process constitutes a *vicarious experience*, which impacts the self-efficacy of the observer. This impact is strengthened if the observer perceives the observed to be of similar ability. **Lave & Wenger:** Observation of others’ activities, progress, and process constitutes a form of legitimate peripheral participation in a learning community. It forms a foundation for more central forms of participation. | - Enable learners to become aware of, and to observe, their peers’ learning activities, especially those that are similar to their own (Schunk, 2012).  
- Present activities in a positive, encouraging light, using statements like “Don’t give up! You’re almost there!” (Schunk, 2012) |
| 2. Observing others’ problem-solving processes | **Bandura:** Asking and questions are *enactive experiences*, the most powerful shaper of self efficacy. They can also increase the chances of task success, leading to more positive enactive experiences. **Lave & Wenger:** Asking and answering questions constitutes a more central form of community participation in a learning community. | - Lower the barriers to asking questions; make it easy to ask questions in the contexts in which they arise (Roscoe & Chi, 2007; VanLehn, 2011)  
- Provide incentives for answering questions to increase participation and increase chances that question-asking is perceived as helpful (Hattie & Timperley, 2007; Schunk, 2012) |
| 3. Observing others’ social interactions    | **Bandura:** Observing of others’ activities and progress, process constitutes a *vicarious experience*, which impacts the self-efficacy of the observer. This impact is strengthened if the observer perceives the observed to be of similar ability. **Lave & Wenger:** Observation of others’ activities, progress, and process constitutes a form of legitimate peripheral participation in a learning community. It forms a foundation for more central forms of participation. | - Enable learners to become aware of, and to observe, their peers’ learning activities, especially those that are similar to their own (Schunk, 2012).  
- Present activities in a positive, encouraging light, using statements like “Don’t give up! You’re almost there!” (Schunk, 2012) |

Vignette 5: Answering others’ questions

*Later on, Jill notices that Dora is struggling with the code to test whether someone has won the game. Jill figured out that conditional test earlier, and decides to respond to Dora’s query. Through multiple exchanges with Jill and Jim, a peer mentor who took the class last semester, Dora is able to understand how to formulate the conditional test properly. She rates Jill’s comments as "helpful," which gives Jill her first reputation points.*

We now turn to a discussion of the evolution of the specific OSBIDE features designed to support these vignettes.

4.1.1 Low Fidelity Prototype

As with many software designs, we began OSBIDE development with a paper prototype. We originally conceptualized OSBIDE as a dashboard-centric program that informed students’ of their peers’ behavior. In this section, we present each major feature to be implemented in OSBIDE.
Course Dashboard

We started with a sketch of a course dashboard to show recent activity related to a specific course (Figure 12). The main screen area (left) contains recent student activity. Students would have the ability to comment on each activity item and click links (underlined text) to get more information. The sidebar (right) contains the course roster. Underlining is used to denote whether or not a student is presently online. Clicking on a student's name (see Howard Johnson) would bring up additional options that would allow a user to view the student's profile, recent activity, and submitted homework files.

Homework Dashboard

The homework dashboard (Figure 13) was intended to provide an easy entry point into other students' code by highlighting potential topics of interest. Namely, it was thought that it might be interesting to know what files other students were looking at, which files students found to be the most helpful, and an all-time most
popular list. Near the bottom of the screen, one finds the same sort of activity feed present in the course dashboard. However, the homework dashboard only displays recent activity feed items that relate to the current homework. Finally, this mock-up contains an alternate right-hand screen. Instead of being sorted by student, this version is organized by course deliverable. The thought behind this implementation was that students seeking help are probably more interested in a particular homework assignment than a particular student.

**User Profile**

Figure 14 presents a sketch of a profile page for individual OSBIDE users. The top of the each user profile page contains user metrics. It is hoped that such metrics might encourage students to participate more activity within the system. Furthermore, such public metrics could be used as symbols of community
status, which often factors into various communities of practice. Below the user metrics, we again see the presence of an activity feed. This time, the feed simply displays activity generated by the single user.

**Summary**

While rough, these low fidelity prototypes facilitated discussion about what a social programming environment might look like. As is evident from the screens, our initial prototype favored a document-centric instead of a user-centric system. At each level of this prototype, documents are a focal point of SPE activity and discussion. While this approach certainly has merit, it has the potential to generate a lot of noise, as we later discovered. Furthermore, by giving students full access to other students’ code documents, this design would require a level of trust that could make both students and instructors uncomfortable.

4.1.2 **High Fidelity Prototype**
We next developed a high fidelity prototype that refined and expanded the paper mockups. Perhaps the most noticeable change was the shift of the activity feed in Visual Studio from the main programming window to a sidebar (see Figure 15). We reasoned that the activity feed is a peripheral awareness mechanism. As such, the activity feed should remain in a user's peripheral awareness while he or she completes the primary task. In the context of early computing education, the primary task is often programming. Therefore, we moved the feed to the side (by default, the same location as the solution explorer), thereby allowing students to both program and be aware of their classmates' activities. Figure 16 details the individual components of the activity stream. Each feed post has up to five components (Figure 16, items 2-6). Each feed posts begin with the post's author. Clicking on the user's name opens the profile page for that user (Figure 17). Next is the post's message. Automatically generated posts have simple messages (e.g., "Jack compiled main.c and got a compile error."), followed by how the event relates to the viewing user (e.g., "You and 2 others have gotten this error when compiling solutions to the current assignment."), the number of comments made in relation to the post, and an option to open the post's details view (Figure 18). Feed post messages were intentionally kept minimalistic for two reasons. First, C compiler errors are notoriously ambiguous; the same error can account for several programming mistakes. Therefore, it was reasoned that adding this information would not prove to be meaningful. Second, we hoped that keeping messages short would encourage deeper exploration by clicking the "See More…" link. Furthermore, this would allow us to quantify the amount of exploration by tracking the frequency of clicks to "see more." User-created posts follow a similar format, but do not contain the relatedness element. Each feed post also contains a "See More..." link that opens the details view for that post.
Figure 15: Activity Feed

Figure 16: Components of the Activity Feed

1. Space for users to create custom feed posts.
2. Clicking on a person’s name will bring the user to their profile page.
3. Example of a custom feed post.
4. System relates events to other people and current user.
5. Indicates the number of comments posted by other people that relate to this feed post.
6. Clicking will open up the details view for the feed post.
7. Opens a dialog for filtering users and feed posts.
4.1.2.1 Activity Post Details

Due to the limited screen space available for the main activity stream, it is necessary to direct users to an additional screen that contains the post's complete details. As illustrated in Figure 18, the details page presents all of a given event's information. This includes all errors, code segments, and past comments that relate to the event. It is also the place where users can add new comments to the post. When applicable, individual errors can be selected from the overall list (item #3). Selecting an error brings up the surrounding code associated with the error. In the current design, the SPE exposes the error's line number +/- 3. In production, this number was eventually increased to +/- 7 after observing that students' code often contain a superfluous amount of whitespace.
4.1.2.2 User Profile View

As previously mentioned, clicking on a person's name will open that person's profile view (Figure 19). In addition to presenting basic usage statistics (item #1), the profile also acts as a hub for receiving personalized messages from other users. Note the ability for other students to post messages directly to a user's profile view (item #2). This was intended to be used as a sort of answering machine when a user is either not online or unavailable. User-created posts are slightly differently from automatic feed posts in that there is not a "details" view. Instead, communication takes place directly in the profile's activity stream. Because user-created messages are likely to get lost in the noise of automatic items, we provided the ability to filter by type (user / automatic; #3).
4.1.2.3 Group Selection

In discussing our ideas for a social programming environment at ICER 2012, participants expressed concern over having too many students participating in a single activity stream. They reasoned that too many students would increase the signal to noise ratio, thereby limiting the usefulness of the activity stream. From this came the notion of giving students limited control over whom they see in their activity feed (Figure 20). We decided that, by default, students should be subscribed to their course section. A student's section represents their core group and cannot be removed by students. Not allowing students to remove others from their section ensures that each student's programming activity is visible to a minimum number of people and prevents the scenario in which a student is not being followed by anyone in the class. However, given that students may have friends in other sections, they should be able to add students from other sections into their group. Note that students will also be automatically subscribed to course instructors and TAs (Christopher Hundhausen in the example), and will not be able to "unfollow" them. Furthermore,
while it is common for social networking sites to list a user's friend count, we decided against including such a feature in order to maintain a more academic focus.

4.1.2.4 Asking For Help

In Chapter 3, we observed that students often use Facebook groups to ask other students specific questions about a particular code segment. In our early prototype, only code related to compile or runtime errors is inserted into the activity stream. However, the environment did not allow students to ask questions about working code. While students could paste code directly into the activity stream, it would be more convenient to provide a built-in "Ask For Help" feature (see Figure 21) as a means of asking code-specific questions. To use, students first select the block of code that they have a question about. Next, they right-click on the highlighted code and select "Ask For Help" from the context menu. This brings up a dialog in which they can provide additional information about their problem. Upon completion, this information appears in the activity stream, where it can be seen by other students.

![Figure 20: Adding / Removing Users from Group](image)
4.1.2.5 Chat

While an activity stream appears to facilitate asynchronous discussions of coding, we were worried about the stream’s ability to facilitate synchronous communications. For this reason, we added a basic chat system into our SPE prototype (Figure 22).

4.1.2.6 Summary

In our low fidelity prototype, OSBIDE was conceptualized as a separate set of screens that existed in the IDE. In contrast, in our high fidelity prototype, OSBIDE was envisioned more as a system that resided within the periphery of the IDE. Our goal was to provide a system that allowed for students to instantaneously shift between coding and being social. Indeed, many user interface design decisions were centered around how to get the most out of a relatively small amount of screen space. Having iterated sufficiently on our high fidelity prototype, we began the transition to a working prototype written in C#.

4.1.3 Working Prototype

A working prototype of OSBIDE was iteratively developed and tested during the 2013 summer and fall offerings of CptS 121 at Washington State University. The system was developed as a C# ASP.NET MVC
web application and seamlessly hosted inside Visual Studio via a browser plugin (Figure 23). This web-based approach was chosen because for the following reasons:

1. Our research group is much more familiar with web-based application development.
2. Due to their centralized nature, web applications are easier to update.
3. Switching to a web interface would more easily allow customization for different user types (e.g. student, TA, instructor, administrator).

Because we were worried about students only accessing OSBIDE through the web interface, we decided to disallow access to anyone that did not have OSBIDE open inside Visual Studio. Unfortunately we did not capture additional screenshots of this version of OSBID. However, each screen looked very much like the high-fidelity prototype presented in the previous section.

Figure 22: Chat Interface
4.1.4 Prototype Evaluation I: Summer 2013

In order to better understand students' experiences with OSBIDE, we conducted two surveys during the 8-week summer offering of CptS 121. In the first summer survey (N = 8), we were interested in seeing how students used the activity feed. While half of the students claimed to check the activity feed on a near daily basis, only one student used the feed for learning purposes (e.g. to search for solutions to build errors). Instead, most students said that they only checked the feed in order to obtain the posted lecture notes. To understand why, we performed a follow-up survey (N = 11) asking students for suggestions or features that might cause them to use OSBIDE more during their learning process. Several suggestions were proposed:

1. Make OSBIDE accessible outside Visual Studio. Students asked to be able to view OSBIDE on their phones and while not programming Visual Studio.
In our prototype, we were concerned about students not downloading OSBIDE and instead accessing lecture material solely through the web interface. To ensure that this would not be the case, we only made the web interface available to students who had Visual Studio up and running. Unfortunately, this arrangement actually prevented students from using OSBIDE on a more frequent basis. To remedy this, we introduced two changes. First, we decided to allow students to access the web interface as long as they had used OSBIDE within the past week. Second, we decided to require students to submit their assignments directly through OSBIDE (Figure 24). We believe these two changes encouraged students to install and use OSBIDE in Visual Studio. However, they also moved farther away from the previous "always on" requirement, thereby giving students freedom to check the web interface when they were not programming.

![Submit Assignment](image)

**Figure 24: Submitting Assignments through OSBIDE**

2. Further integrate course materials into OSBIDE such as the syllabus, course calendar, and assignment files.

During this evaluation, the course used OSBIDE for online discussions and OSBLE as a repository for course materials. Students disliked having to visit multiple places in order to obtain all of the course materials. As this survey occurred before the OSBLE activity stream analysis, we were not yet aware of the potential importance of placing activity streams near critical course content. To account for this, we added
basic course management functionality (Figure 25) and a gradebook (Figure 26) to the OSBIDE web interface.

3. Allow users to search for errors directly from Visual Studio.

In the working prototype, students had to search for errors using an activity feed filter. This was not only cumbersome, but also made it difficult to search for solutions to specific errors. In order to allow for easier searching, we added the ability to search for an error in OSBIDE directly inside of Visual Studio’s Error List window (Figure 27). This process automatically filtered the user’s activity feed to show only related build events.
4.1.5 Prototype Evaluation II: Fall 2013

In the fall of 2013, we deployed OSBIDE in the semester's offering of CptS 121 (N=237). The version deployed was the same as the version tested in the prior summer, as had not yet had the chance to implement the changes discussed above. This also marked the first time in which the tool was deployed in a course that was not taught by someone on the research team.

A few weeks into the semester, it became clear that students were not using OSBIDE for code discussions. To understand why, we conducted a survey (N=165) of students midway through the semester. From the survey, we learned that over 80% of respondents were unaware of a single OSBIDE feature, even though they had downloaded and installed the plugin at the start of the semester. Clearly, the tool suffered from a lack of visibility. This led us to reconsider our design decisions and resulted in the following changes to the system:

Add an OSBIDE toolbar to Visual Studio. Our first effort to increase OSBIDE's visibility within Visual Studio was to add a toolbar to the IDE and situate it directly to the right of the debugging toolbar (Figure 28). From left to right, the toolbar allows students to log into OSBIDE, access the activity feed, view the user's profile, access help documentation, and view the OSBIDE web site. Prior to the toolbar, all four commands were only accessible by navigating through a deep menu system underneath the "View" dropdown menu.

Automatically open the activity feed upon launch of Visual Studio. Prior to the introduction of the toolbar, the activity feed was only accessible through a difficult to find option under the "View" dropdown menu in Visual Studio. While adding the toolbar certainly makes opening the feed easier, we wanted the activity feed to be the first thing that students saw when they launched Visual Studio.

Figure 28: OSBIDE toolbar
Make the activity feed global. Recall that our prototype contained the idea of groups that segmented students into sections. However, given that 80% of our test group was unaware of the feed, it seemed likely that a student assigned to a given group might be the only active person within that group. In an effort to increase the likelihood of obtaining a critical mass of social activity, we decided that OSBIDE would contain a single activity feed for all users.

Add more information to posts within the activity stream. In our prototype, the activity feed was comprised of generic messages (e.g. "Adam commented on Bob's post"). This was done in order to more easily track which stream items were being investigated. However, the end result was a bland and uninspiring interface that actually discouraged exploration; we logged very few click-throughs to the details page. To make the stream livelier, we decided to include an entire post's content in the stream. Furthermore, we added the ability to view responses to posts directly within a stream; these originally required an additional click and appeared on a separate page. The resulting activity stream is presented in Figure 30.

Remove automatic IDE events from the default activity stream filter. Given the large class size of the course (N=237), it soon became apparent that including every single programming event in the activity resulted in an overwhelming amount of information. Furthermore, we noticed the tendency for programming events, which were automatically generated by the OSBIDE IDE plugin, to drown out social activity. Indeed, we observed over a 20:1 ratio between programming and social events. In order to increase the impact of social events on the learning process, we decided that the default feed settings for each user would only include social events. However, programming events could still be viewed by adjusting the feed filtering settings.

Remove chat system. Based on the limited social interaction observed in the evaluation, we noticed that students used the chat system and activity stream interchangeably – content posted to either source was similar in nature. Indeed, even the research team had difficulty in determining the type of content that would be best served by the chat room and what content would be best served by the activity stream. Furthermore, because of this ambiguity, we were worried about users having to constantly monitor two social sources. Thus, we decided to consolidate all social interaction into the activity stream.
4.1.6 Updated Version Used in Dissertation

After two prototype evaluations and numerous design meetings, we arrived at the version of OSBIDE used in the remainder of this dissertation. In this section, we present the system features as they appeared during data collection.

4.1.6.1 Activity Feed

The activity feed forms the backbone of OSBIDE activity within the IDE. Figure 29 presents OSBIDE as it would appear upon opening a project inside Visual Studio. The activity feed automatically launches and docks itself on the right edge of the IDE. This allows users to seamlessly switch between programming and participating in class discussions. Figure 30 annotates key elements in the activity stream.

![Figure 29: OSBIDE](image-url)
4.1.6.2 Details Pages

As can be seen in Figure 30, a details link is attached to each event in the activity feed. Clicking this link replaced the main IDE window with the event's details. For most events, the details view replicates the same information present in the activity feed. However, for runtime exception, build, and ask for help events (Figure 31), the details view displays additional information not present in the activity feed. For runtime exceptions, the details view provides the line on which the runtime exception occurred and the stack trace at the time of the runtime exception. Similarly, build events provide the line on which the build event occurred. In addition, the build event details page contains a link to a compilation timeline that indicates...
how the student addressed the compilation issue (Figure 32). Lastly, the details page for ask for help events displays the student's code snippet using proper formatting and highlighting.

Figure 31: Runtime Exception (top left), Ask for Help (top right), and Build Error (bottom) Details Page

Figure 32: Exploring a Student's Compilation History
4.1.6.3 Obtaining Help

OSBIDE provides two main mechanisms for obtaining help. The first mechanism allows students to ask questions about their code by highlighting a troublesome block of code, right clicking, and selecting "Ask for Help" from the context menu. The student is then given an opportunity to annotate the code selection with additional information. This question will then show up in the class' activity feed. This process is illustrated in Figure 33.

The second mechanism for obtaining help is limited to solving compilation issues. Students can either filter the activity feed to show compilation attempts (Figure 34) or they can search for an error that they are presently having from within the IDE (Figure 27). Either approach will eventually land the student at a compilation attempt's details page (Figure 31) and ultimately a student's compilation history (Figure 32).

4.1.6.4 User Profile

Each user in OSBIDE has a profile page (Figure 35). The profile page allows other users to see a given user's recent activity within the system. This includes a score that is derived from overall activity within the system, the total number of posts and replies made by the user, people with whom the user had recent interactions, and a custom feed that displays the user's recent posts and comments. For a user viewing his or her own profile, a subscriptions area is also displayed. The subscriptions area allows users to quickly jump back to particularly helpful or interesting posts at a later date.

4.1.6.5 LMS Features

As we discovered during our first prototype evaluation, students disliked having to visit multiple websites in order to obtain all of a course's digital content. In order for OSBIDE to become the central hub of all online activity in a course, we added individual course pages (Figure 25) that house course content (e.g. lecture notes, homework descriptions) and a gradebook (Figure 26).
Figure 33: Asking for Help within OSBIDE

Figure 34: Filtered Activity Feed Only Showing Build Events
Having described the design evolution of OSBIDE, we now turn to a summative field evaluation of the software. Our evaluation is based on field data collected during the spring 2014 offering of CptS 122 (CS2) at Washington State University. Focusing on the C and C++ programming languages, CptS 122 has three 50-minute lectures and one 170-minute lab period per week; three exams (two midterms and a final); and seven individual programming assignments due at roughly two week intervals. The course enrolled 140 students, 129 of whom finished the course and received a grade. 108 of these students (100 men, 8 women) consented to releasing their data.

Because participation rates were so low in our second prototype, we decided to tell students that regular participation would account for up to 5% of their final grade. For each assignment period, we told students that they needed to make at least two posts and reply to two other students’ posts. While this deceptive...
requirement may well have increased the likelihood of students using OSBIDE, it also makes it difficult to compare activity stream participation within this course and the courses discussed previously.

4.2.1 Feature Usage

We begin by exploring students' usage of OSBIDE. Usage statistics, as reported in Table 9, were aggregated from 21,963,217 activity logs generated by the students in CptS 122. In some cases, an activity record did not contain enough information to indicate the exact action that took place. In these cases, the activity record was dropped from further consideration. As such, it is possible that the numbers reported are slightly lower than actual usage.

Table 9's data is both promising and disappointing. On the positive side, every student viewed the activity feed at least once and nearly every student (93%) made at least one post to the activity feed. Likewise, nearly the same number of students posted at least one comment to another's post. Furthermore, students appear to be invested in their online personal profile, as is evident in the number of profile edits made to students' profiles.

Table 9: OSBIDE Usage by Major Feature

<table>
<thead>
<tr>
<th>Feature</th>
<th>Usage Total</th>
<th>Unique Students</th>
<th>Median Per Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Diff</td>
<td>3</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>Search Feed</td>
<td>470</td>
<td>75</td>
<td>4</td>
</tr>
<tr>
<td>Ask for Help</td>
<td>28</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>Details Views – Feed Post</td>
<td>9,828</td>
<td>97</td>
<td>14</td>
</tr>
<tr>
<td>Details View – Build</td>
<td>29</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>Details View – Runtime</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Details View – Ask for Help</td>
<td>254</td>
<td>36</td>
<td>5.5</td>
</tr>
<tr>
<td>Details View – Submit Assignment</td>
<td>228</td>
<td>38</td>
<td>3</td>
</tr>
<tr>
<td>Follow Post</td>
<td>98</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>View Activity Feed</td>
<td>40,131</td>
<td>107</td>
<td>286</td>
</tr>
<tr>
<td>Post Comment</td>
<td>2,342</td>
<td>100</td>
<td>19</td>
</tr>
<tr>
<td>Reply to Post</td>
<td>5,765</td>
<td>99</td>
<td>34</td>
</tr>
<tr>
<td>Mark Post as Helpful</td>
<td>388</td>
<td>71</td>
<td>3</td>
</tr>
<tr>
<td>Edit Profile</td>
<td>1,137</td>
<td>96</td>
<td>6</td>
</tr>
<tr>
<td>Profile Views - Self</td>
<td>4,514</td>
<td>105</td>
<td>30</td>
</tr>
<tr>
<td>Profile Views - Others</td>
<td>1,497</td>
<td>96</td>
<td>8</td>
</tr>
</tbody>
</table>
On the flip side, it is clear that other features received little attention. While many students attempted to search the feed, this rarely resulted in a student viewing another student's build errors or runtime exceptions. Indeed, only two students viewed another student's compilation history. Furthermore, we can assume that these students did not find the tool particularly useful as usage did not persist.

Somewhere in the middle, we have the "Ask for Help" feature. Relative to standard activity feed posts, this feature was used less frequently, both in terms of median student usage (2 vs 19) and unique students (15 vs 100). While one might think that this implies that students weren't using the activity feed to discuss coding issues, a sample of 790 posts revealed that 148 of these posts involved coding questions. (A more in-depth analysis of these questions is conducted in Chapter 6.) Given that students seem to be wanting to ask coding questions, but unwilling to use what is arguably the easiest way to ask these questions, one might suspect that the feature's implementation is at fault. Unlike normal feed posts, which display a post's content in its entirety, ask-for-help posts only show the question; any included code can be viewed only by clicking the post's "details" link. While this requirement appears to have improved the ratio of click-through ratio to the corresponding "details" page (9.10 vs 1.70), it is possible that this requirement discouraged the feature's usage. Indeed, in reading the activity feed's transcripts, we see such a sentiment: "Does anyone else not like that you have to hit 'Details' … rather than just having it show up when clicking 'View Conversation'?"

Similarly, another student who used the ask for help repeatedly left instructions on how to properly view his code ("Click details to see it properly" and "Also, press details for correct formatting."). This seems to indicate that the implementation tended to hinder adoption of this feature. While similar quotes do not exist for the other little-used features, it seems plausible that they too suffered from a lack of visibility. Future research should reexamine each unused feature to better understand how they can be presented so as to provide more value to students.

4.2.2 Investigating Social Activity

In the previous chapter, we employed content analysis using a custom coding manual. Using the activity stream post as the atomic unit of analysis, we iteratively developed a content coding scheme by reading
through excerpts of activity stream transcripts, adding and refining categories until the categorical definitions stabilized and no new categories emerged. The complete coding scheme is presented as an appendix. In this chapter, we apply the same content coding scheme to OSBIDE's activity stream. However, due to the volume of posts made, we employed a principled approach to sampling the content. To this end, we randomly sampled 10 students within each of five grade bands in the course (A, B, C, D, F). While we were able to sample 10 students from the A, B, and C level, only four students were available from the D level, and only two students were available from the F level. Therefore, our sample could only include 36 students, instead of the 50 that we targeted.

We begin by comparing top-level posts in the four courses studied so far in this dissertation: CptS 122 (OSBIDE), CptS 122 (OSBLE; see Chapter 3), CptS 121 (Facebook; see Chapter 3), and CptS 111 (Facebook; see Chapter 3). Figure 36 presents the relative frequencies of each class' top-level categories. A chi-squared test revealed a significant association between course and top-level category ($\chi^2$(9) = 543.02, $p < 0.01$). An associated z-test revealed significant ($p < 0.05$) differences in several frequencies between groups. Most notably, in comparison to the CptS 121 and the OSBLE groups, the OSBIDE group had a significantly smaller proportion of social activity dedicated to CODE and a significantly higher proportion dedicated to COURSE and OTHER.

We next compare the CODE subcategory of all four courses. Figure 37 provides the breakdown for each course. A chi-squared test revealed a significant association between course and top-level category ($\chi^2$(9) = 399.99, $p < 0.01$). An associated z-test revealed significant ($p < 0.05$) differences in several frequencies between groups. In comparison to the OSBLE CptS 122 group, the OSBIDE group had a significantly fewer proportion of COMPILE and RUNTIME posts and a higher proportion of CONCEPTUAL and IDE posts.
Finally, we consider average posting behavior within the course. The average student in the OSBIDE group made 75 posts throughout the semester. The bottom quartile of students made 19 posts per semester and the top quartile of students made 101.5 posts per semester. In terms of overall class participation, 93%
of students made at least one post within OSBIDE. Table 10 provides a comparison of posting behavior between all four courses.

Clearly that students in the OSBIDE condition participated significantly more \((F(3, 346) = 13.61, p < 0.01)\) than students in the other condition. Unfortunately, the presence of a posting requirement in the condition prevents us from being able to definitively attribute this increase to the tool, as it could also be attributed to the course requirement. While we examine the relationship between posting activity and coding behaviors in Chapter 6, it is worth examining the activity feed transcript in order to shed light on the present dilemma.

We begin by considering how the posting requirement might have influenced student's posting behaviors. The fact that the requirement existed was not lost on students. Indeed, many expressed concerns that students might post merely to meet the requirement. Early in the semester, one student made the comment that, "there is going to be a lot of commenting/posting just to get the points rather than actually adding anything." Later in the semester, another student commented on the fact that some students were attempting to game the system, "Why does repeating what everyone else is saying help? It is literally copy and pasted. I don't think this is what [the course instructor] was thinking of when he said to post 2 and reply to 2 posts." Indeed, there exist numerous posts that make it clear that students were gaming the system:

- "There's no shame in posting to get points, considering that is 80% of osbide."
- "[student] literally just posted 4 things in 20 seconds to get points"
- "Here have some points"

### Table 10. Posting Behavior by Course

<table>
<thead>
<tr>
<th>Course</th>
<th>Average Posts per Student</th>
<th>Overall Participation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CptS 111</td>
<td>6.95</td>
<td>72%</td>
</tr>
<tr>
<td>CptS 121</td>
<td>12.70</td>
<td>75%</td>
</tr>
<tr>
<td>CptS 122 – OSBLE</td>
<td>7.85</td>
<td>92%</td>
</tr>
<tr>
<td>CptS 122 – OSBIDE</td>
<td>75.32</td>
<td>93%</td>
</tr>
</tbody>
</table>
These posts would seem to indicate that some portion of posts were made simply to fulfill the course requirement. This would help explain the disproportionately large amount of OTHER-coded posts. On the other hand, there also exists evidence that the tool itself encouraged students to post more:

- "[The instructor] should disable OSBIDE during lectures. It's turning into a Facebook-like distraction."
- "Yeah [Student], pay more attention to lecture and less time to the computer science Facebook. See you there on Thursday."
- "[OSBIDE] was a constant disruption to his lecture"

As the comments above indicate, some students came to see OSBIDE to be a distraction in class—evidence of its pervasiveness in classroom culture. It should be noted that students had the ability to turn off OSBIDE at any point. Therefore, while previous comments might indicate a dislike for the tool, they in fact likely demonstrate a conscious decision on the student's end to remain active participants in the OSBIDE community. Interestingly, in a follow-up analysis, we found that while students did post during class times, such posts represented just 13% of total social activity.

Further evidence supporting the success of OSBIDE as a tool was evident in the course's end of term evaluations.

- "OSBIDE was a great tool. It was forced upon us but in a good way."
- "Using OSBIDE helped create a community in the class."

We also saw a single negative comment, "OSBIDE...... Biggest distraction ever." However, this supplies further evidence of OSBIDE being a pervasive tool within the classroom culture. A follow-up survey found similarly positive sentiment towards the tool:

- "OSBIDE was an awesome idea[...]the social integration helped me push forward[...]I made good friends through OSBIDE"
• "OSBIDE was definitely helpful in the sense that it broke those social barriers and really built my network of other CS friends"

• "I loved using OSBIDE in your classes. It absolutely gave a sense of community, and it was quite fun to discuss things while class was going on without being disruptive"

These comments give support to the idea that integrating social networking directly into students' problem solving environment impacts participation. However, as the coding of posts would seem to indicate, this does not necessarily correlate with a relative increase in coding discussions. Instead, we witnessed an increase in off-topic conversation. While this may initially seem not desirable, we believe that these kinds of off-topic conversation contribute to authentic communities of practice as described by Wenger (Wenger, 1998). Indeed, the OSBIDE condition is the only one that exhibited true community discourse. For example, students started a 63 post impromptu discussion on the struggle that many women face in computing. Seeing the popularity of this discussion, the instructor went on to use the next class to explore this topic in further detail.

In the next section, we consider the impact of social participation on students' attitudes on standardized measures.

4.2.3 OSBIDE's Impact on Attitudes

According to the Situated Learning Theory and Social Cognitive Theory, increasing students' social exposure is likely to increase students' self-efficacy and sense of community. To further investigate the relationship between OSBIDE and these attitudes, we performed a pre-post survey analysis between spring 2014 and fall 2014 offerings of CptS 122. The spring 2014 course used OSBIDE and the fall 2014 course used a traditional LMS (Angel). Of the 120 students in the OSBIDE condition, 108 students consented to have their responses used in this analysis. Of the 114 students in the traditional condition, 65 consented to have their responses used in this analysis.

Our survey consisted of two instruments. For quantifying self-efficacy, we used the C++ Self-Efficacy (CSE) Scale (Ramalingam & Weidenbeck, 1998). For sense of community, we used the Classroom
Community Scale (CCS) (Rovai, 2002). Students completed the survey once at the beginning of the semester and again at the end of the semester. Table 11-Table 13 present the results of an analysis of variance (ANOVA) on students' initial and final survey responses.

We first consider Table 11, which report the results of the C++ Self-Efficacy survey. Across both conditions, we see a statistically significant \((p < 0.01)\) increase in self-efficacy. Interestingly, both courses' final survey response averages are exactly the same (5.82) as is the effect size (24%). However, students in the OSBIDE condition started with a lower initial response average (4.45 vs 4.56).

We next consider Table 12, which compares mean group responses on the Classroom Community Scale. Unlike the C++ Self-Efficacy scale, we find that only the OSBIDE condition had a statistically significant \((p = 0.02)\) increase in sense of community. However, we find both the mean increase and effect size between responses to be quite small.

In both surveys, we see a difference in initial responses, with the OSBIDE group having a lower initial response rate in both surveys. To explore how this difference might affect the statistics reported in the previous paragraphs, we examined each student's delta increase between pre and post surveys. The results of this ANOVA are reported in Table 13. In both surveys, students' mean increases do not differ significantly across conditions. In other words, either condition appears to be just as likely to induce the same amount of change in students' attitudes.

It should be noted that the analysis conducted in this section suffers from several threats to validity. First, the courses were taught during different semesters (fall vs spring). Students taking CS2 in the fall are typically in their 2nd year of college whereas spring students tend to still be in their first year. Indeed, using a four-point scale (1 being freshman and 4 being senior) the average class standing of the traditional course is 2.62 whereas the average class standing of the OSBIDE group is 1.99. Second, the courses were taught by different instructors using different lecture materials, homeworks, quizzes, and exams. As with any pre-post survey design, we must also worry about attrition. It is possible that only students whose attitudes increased remained in the course. Alternatively, it is possible that students whose attitudes increased were more motivated to complete the survey. Any of these possibilities might explain any difference (or lack
thereof) reported. Unfortunately, analyzing only two groups prevents us from further investigating these potential issues; this must be left to future research.

4.2.4 Exploring the Relationship between OSBIDE usage and Course Outcomes

In Chapter 3, we explored the construct of Social Role, which captures both the number of posts and replies a student makes in a two week period. Recall that the four levels of Social Role, presented in Table 14, are based on the posting requirement for the OSBIDE class. Our original analysis, presented in detail in Chapter 3, found Social Role to be a significant predictor in an offering of CptS 122 that did not use OSBIDE. In this section, we again consider Social Role in the context of two new factors: OSBIDE and a course requirement to participate.

Table 11: C++ Self-Efficacy Response Comparison (7pt scale)

<table>
<thead>
<tr>
<th></th>
<th>Pre-Survey</th>
<th>Post-Survey</th>
<th>F-Value</th>
<th>Significance</th>
<th>$\eta^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>4.56</td>
<td>5.82</td>
<td>(1,101)= 31.07</td>
<td>&lt;0.01</td>
<td>0.24</td>
</tr>
<tr>
<td>OSBIDE</td>
<td>4.45</td>
<td>5.82</td>
<td>(1,178)= 54.91</td>
<td>&lt;0.01</td>
<td>0.24</td>
</tr>
</tbody>
</table>

Table 12: Classroom Community Scale Response Comparison (5pt scale)

<table>
<thead>
<tr>
<th></th>
<th>Pre-Survey</th>
<th>Post-Survey</th>
<th>F-Value</th>
<th>Significance</th>
<th>$\eta^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional</td>
<td>2.56</td>
<td>2.65</td>
<td>(1,101)= 00.29</td>
<td>0.59</td>
<td>N/A</td>
</tr>
<tr>
<td>OSBIDE</td>
<td>2.35</td>
<td>2.55</td>
<td>(1,178)= 46.62</td>
<td>0.02</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 13: Survey Delta Comparison

<table>
<thead>
<tr>
<th></th>
<th>Mean Delta</th>
<th>F-Value</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ Self-Efficacy</td>
<td>1.55</td>
<td>(1,126)= 0.39</td>
<td>0.54</td>
</tr>
<tr>
<td>Classroom Community Scale</td>
<td>0.24</td>
<td>(1,126)= 2.64</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 14: Definition of Social Role Participation Measure

<table>
<thead>
<tr>
<th>Level</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No posts or replies for a given period</td>
</tr>
<tr>
<td>2</td>
<td>At least one post or reply, but no more than one of either for a given period</td>
</tr>
<tr>
<td>3</td>
<td>Two posts and fewer than two replies, or two replies and fewer than two posts for a given period</td>
</tr>
<tr>
<td>4</td>
<td>Two or more posts and two or more replies for a given period</td>
</tr>
</tbody>
</table>
4.2.4.1 Participants and Method

In this analysis, we consider two offerings of CptS 122: one that uses OSBLE and one that uses OSBIDE with a required amount of minimum participation. The OSBLE group took place during the 15-week fall semester of 2013 and enrolled 123 students, 110 of which completed the course and consented to release their grade (99 male, 11 female). The OSBIDE group is the same group examined in prior sections. The OSBIDE course enrolled 140 students, 129 of whom finished the course and received a grade. 108 of these students (100 men, 8 women) consented to releasing their data.

For each class, we generated a timeline of each student's social activity. In order to generate the Social Role measure, we recorded posts and replies separately. These numbers were then averaged for the entire term and compared with each student's final grade.

4.2.4.2 Results

Table 15 presents the mean participation level for each two-week interval, as well as the averages across the entire semester. As the table indicates, students in the OSBIDE group participated more actively in the activity stream. Indeed, a non-parametric Kruskal-Wallis ANOVA showed a statistically significant difference in level of participation between the two groups ($H = 90.479, df = 1, p < 0.001$), with students in the OSBIDE group maintaining a regular participation level that was nearly twice that of the OSBLE group.

In order to determine whether or not regular participation was a reliable predictor of academic success, we considered each student's semester-long participation level average. In the case of the OSBIDE group, in which participation was mandatory, we were concerned that participation level might co-vary with prior computing grades: since there was academic incentive to participate in the discussion, higher-achieving students might be more motivated to participate. Indeed, in the OSBIDE group, a correlational analysis between a student's level of participation and prior grade in CS 1 was found to be significant ($r = .468, p < 0.001$). However, in the OSBLE group, this was not the case ($r = .12, p = 0.26$). Hence, we decided to retain prior CS 1 grade as a covariate in further analyses with the OSBIDE group (by using a MANCOVA),
whereas we used a MANOVA for the OSBLE treatment, while adding prior CS 1 grade as a separate independent variable for comparison purposes.

Table 16 and Table 17 present the results of these analyses. As Table 17 shows, CS 1 grade was not a significant predictor of student performance in the OSBLE group. In contrast, in the OSBIDE group, students' prior CS 1 grade was a significant predictor of students' exam scores and final grade. The partial eta squared ($\eta^2$) values indicate that the strength of the relationship between CS 1 grade and these two items was weak.

Table 16 tells a slightly different story: Participation level predicted student performance with respect to all graded items and the final grade in both groups. Moreover, in all cases, the partial eta squared values indicate that participation level was a stronger predictor of student grades than was prior CS 1 grade, as it accounted for roughly twice the variance.

In order to further explore differences in students' grades vis-à-vis participation level, we partitioned students into quartiles based on their overall average participation level. Table 18 presents the mean final grade (percentage) by quartile.
In both groups, a between-subjects ANOVA detected a statistically-significant difference between the quartiles (OSBLE: $df = 3, F = 8.15, p < 0.001$; Integrated: $df = 3, F = 11.62, p < 0.001$). In both groups, a post-hoc Bonferroni test revealed that the significant differences lay between the top two quartiles and the lowest quartile ($p < 0.01$). Furthermore, the OSBIDE group had significant differences between the 2nd and 4th quartile ($p < 0.045$).

In order to determine the specific impact of participation quartile on student grades, we ran a regression analysis using quartile as categorical variables. In the case of the OSBLE group, a change from one quartile to the next corresponded to a grade change of 8.5%. In the case of the OSBIDE group, a one-quartile change corresponded to a grade change of 4%. Thus, in both treatments, a one-quartile change equated to a difference of roughly one-half of a letter grade.

4.2.5 Discussion

At the end of the original discussion on Social Role in Chapter 3 section 4.3, we were undecided whether or not Social Role was a useful measure. The results just presented would seem to indicate that Social Role is indeed useful in CptS 122. However, differences between the OSBIDE and OSBLE group, namely the course requirement to post content, prevent us from answering perhaps the most interesting question: how does activity stream placement influence Social Role and participation in general? Nonetheless, we feel comfortable making the claim that introducing a posting requirement does not impact the statistical significance of the Social Role measure. It stands to reason that forcing students to artificially participate in social activity might negate the positive impacts observed in the OSBIDE group. However, our analysis clearly indicates that this is not the case: when posting was required, regular online social activity positively correlated with grades. This remained true even when we controlled for prior academic performance.

4.3 Summary

This chapter presented OSBIDE, a social programming environment that integrates social networking features into an IDE. After a series of pilot studies, OSBIDE was deployed in the spring 2014 offering of
CptS 122 at Washington State University. Based on this deployment, we performed a comprehensive mixed-methods summative evaluation that furnished valuable insights into both the tool itself and its usefulness and impact on the students who used it.

An analysis of OSBIDE's feature usage revealed clear winners and losers. The activity feed was a major success. Nearly every student made at least one post to the activity feed, and the median number of views per student was 2.5 per day. On the flip side, we found that students rarely used the search feature or the "code diff" feature. Clearly, future research should reexamine each unused feature to better understand how they can be designed so as to provide more value to students.

Having explored OSBIDE's features, we next attempted to quantify OSBIDE's impact on its users. We found that students that used OSIBDE experienced a significant, but minor, increase in sense of community. We could not detect a significant increase in sense of community in a control condition of CptS 122 that did not use OSBIDE. We also examined the relationship between regular participation in OSBIDE and course outcomes. We found that regularly participating in OSBIDE discussions significantly predicted course assignment, quiz scores, exam scores, and final grades. In the next two chapters, we will consider other ways in which a student's behavior within OSBIDE relates to course outcomes.
CHAPTER 5

MODELING PROGRAMMING BEHAVIORS

In the previous chapter, we considered the design and educational effectiveness of a social programming environment that incorporates an activity feed into a computer programming environment. An additional educational benefit of a social programming environment not considered in the previous chapter is that it facilitates the collection of detailed learning process data on computing students. Indeed, by collecting a stream of learning process data in their courses, computing educators obtain new opportunities to continuously assess their students learning processes and progress. Using techniques from the fields of educational data mining and learning analytics (Baker & Siemens, 2014; U.S. Department of Education, Office of Educational Technology, 2012), we can analyze these data in order to identify ways in which learning patterns and attitudes relate to learning outcomes. Such analyses open up new opportunities to better tailor instruction to individual learners, and ultimately improve student learning outcomes, especially among at-risk learners.

In computing education, employing educational data mining and learning analytics techniques would appear to be particularly appropriate, given computing education's "grand challenge" problem of improving student retention, especially in early computing courses (see, e.g., (Campbell & McCabe, 1984; Graham, Federick, Byers-Winston, Hunber, & Handelsman, 2013; B. C. Wilson & Shrock, 2001). If computing educators are able to identify, at an early stage in a computing course, students who are at risk of dropping out or failing the course, then they are in a better position to improve retention by tailoring or adapting their instructional approaches.

Recognizing this potential, we became interested in further analyzing the log data collected by our social programming environment as students worked on programming assignments (Altadmri & Brown, 2015). In particular, we were interested in the following research question:

RQ1: How well can log data on students programming processes predict a student's performance?
At least two lines of computing education research have developed measures to predict student learning outcomes based on programming behavior. Both of the predictive measures derived from this research—the Error Quotient (Jadud, 2006b) and the Watwin Score (Watson et al., 2013)—focus exclusively on differences between successive compilation attempts. These metrics associate improved learning outcomes with an ability to quickly remove compilation errors from a program. Given the apparent limitation of focusing solely on compilation behavior, we identified an additional research question:

RQ2: How well can a more holistic model of students' programming processes—one that considers both syntactic and semantic correctness—predict performance?

To address this research question, this chapter proposes the Programming State Model (PSM) as a more holistic and powerful characterization of students' programming processes. We explore the PSM's effectiveness using two approaches: one focused on aggregating states and another focused on quantifying transitions between states.

Furthermore, while prior work suggests that these measures are capable predictors, we notice that their narrow focus on compilation behavior ignores other programming behaviors—most notably, debugging (Ahmadzadeh et al., 2005)—that might be associated with learning success. Second, both measures have been derived within one particular programming course (CS1), language (Java) and novice programming environment (BlueJ); their predictive power has not been tested in other courses, programming languages and environments. These limitations raise a third research question:

RQ3: How is the predictive power of the Error Quotient and Watwin Score affected by different courses, languages, and programming environments?

To address these research questions, this chapter derives a new, more holistic predictive model, and then performs a replication study of the Error Quotient and Watwin Score studies, using a different student population (CS2 instead of CS1), different programming language (C/C++ instead of Java), and different programming environment (Visual Studio® instead of BlueJ).
5.1 Developing a Programming State Model

Both the Error Quotient and Watwin Score focus exclusively on students' compilation activities: students who quickly and accurately fix syntax errors in their programs are predicted to perform better than those who do not. While the ability to eliminate syntax errors from a program is an important programming skill, it is widely acknowledged that programming success also hinges on one's ability to identify, diagnose, and repair runtime (semantic) errors (see, e.g., (Ahmadzadeh et al., 2005)). Thus, one would expect that an ability to eliminate semantic errors would also correlate positively with performance in a computing course.

This observation motivates a more holistic predictive model of student performance rooted in a student's ability to develop both syntactically and semantically-correct programs. Our proposed model aims to approximate the syntactic and semantic correctness of a programming solution at any given point in time (see Table 1). Given a stream of programming data, we map a student's current programming solution to one of the four states in this 2 × 2 space. We can determine the syntactic correctness of a program based on whether the last compilation attempt yielded an error.

In contrast, semantic correctness is impossible to determine unequivocally. All we have is a rough proxy: the presence or absence of runtime exceptions in the last execution attempt. If the last execution attempt yielded a runtime exception, we classify the program as semantically incorrect. If the last execution attempt did not yield a runtime exception, we classify the program as semantically unknown. Clearly, our proxy for semantic correctness has significant limitations. For instance, a student's program could meet the assignment specification (and hence be “semantically correct” for the purpose of the assignment), but still raise a runtime exception if it encounters input data that it is not required to process. Conversely, a student's program could run without raising a runtime exception, but its output could be incorrect. Likewise, the student could have failed to test key boundary cases that would have raised runtime exceptions.

Figure 38 presents a state-transition diagram that maps our model to the stream of programming log data made available to us by Microsoft® Visual Studio® (Microsoft Corporation, 2015), the IDE used in
our study. Note that Visual Studio does not report runtime exceptions that occur outside of debug mode. Thus, our log data do not contain two potentially important transitions: those between RN/RU and YN/YU. For this reason, we are forced to determine the current state based on the results of the student's last compilation and debug attempt. In order to switch from a syntactically incorrect state (NU and NN) to a syntactically correct state (YN and YU), the student's last compilation attempt must have been free of build errors. Likewise, a student switches from a semantically unknown state (NU and NN) to a semantically incorrect state (YN and YU) only if the last debug attempt yielded a runtime exception.

Observe that intermediate execution states are also captured in this state transition diagram. If the student's program is syntactically correct, it can be executed either with or without the debugger in Visual Studio. This leads to the four left-most "Execute" states in the diagram (RN, RU, DN, DU). In contrast, if the student's program is syntactically incorrect, it is not possible to execute it in debug mode. However, in Visual Studio, it is possible to execute the last successful build of a program. This leads to the right-most "Execute" state (R/).

Two additional states are also necessary in this model. First, it is impossible to determine the state if no compilation or execution attempts have been made. To account for this situation, which commonly occurs at the beginning of a programming session, we define an additional state called "Unknown (Start) State" (UU). Second, a prolonged period of inactivity (which we established as three minutes) in any state leads to a transition to the Idle state, in which the next editing activity causes a transition back to the previous state.

Although we have developed a model for tracking a student's programming state, we have not specified how Figure 38 might be operationalized for use in a statistical model. In the following sections, we consider two possible methods of operationalization. In the first, we aggregate time spent in each state and normalize

---

²We have since learned that it is, in fact, possible, through an event log generated by the Microsoft Windows operating system, to obtain runtime exceptions raised by programs run outside of debug mode. However, at the time we ran this analysis, we were unaware of this possibility, so we did not obtain data on such runtime exceptions.
Figure 38: Programming State Transition Diagram

- All states timeout after 3 minutes of inactivity (e.g. no editor activity, compile, etc.)
- Upon new activity, transition is made to last known state.
these times relative to the total time spent programming. In the second approach, we examine a student's path through our transition diagram.

5.2 Deriving a Model from Aggregate Transitions

We begin by considering a model derived from the PSM that quantifies the amount of time spent in each state relative to the total time spent programming. In this model, each state in the PSM (e.g. YU) is treated as a predictor variable in the statistical model. Each variable is assigned a coefficient ranging from 0 (no time spent in that state) to 100 (all time spent in that state). As these values have been normalized relative to the total time spent programming, the sum of all predictor variable coefficients will equal 100. Because we employ a normalization process to our PSM, we refer to this model as the Normalized Programming State Model, or NPSM. In this section we examine the NSPM's ability to predict course outcomes for a given student in a computing course.

5.2.1 Methodology

We used the same participant cohort used to evaluate the OSBIDE tool (Chapter 4). Our data was collected during the spring 2014 offering of CptS 122 (CS2) at Washington State University. Focusing on the C and C++ programming languages, CptS 122 has three 50-minute lectures and one 170-minute lab period per week; three exams (two midterms and a final); and seven individual programming assignments due at roughly two week intervals. The course enrolled 140 students, 129 of whom finished the course and received a grade. 108 of these students (100 men, 8 women) consented to releasing their data.

Three performance indicators were used for the analysis: (1) students' grades on individual assignments; (2) students' overall assignment average, and (3) students' final grades, which were based on the grades received on programming assignments (35%), labs (10%), participation (5%), in-class quizzes (10%), midterm exams (20%), and a final exam (20%). Predictions of individual assignment grades were based exclusively on the programming log data generated while the corresponding assignment was open. (The length of each assignment varied between ten and twenty three days.) Predictions of students' overall
assignment averages and final grades, in contrast, were based on programming log data generated throughout the entire semester.

5.2.2 Evaluating Explanatory Power

To evaluate the ability of the NPSM to explain the variance in individual assignment grades, we performed a linear regression, with variables within the NPSM acting as predictor variables and individual assignment grades as the outcome variable (see Table 19). Because the NPSM relies on normalized time values, students who spend little time on an assignment are likely to skew the explanatory model. With that in mind, we ran a secondary analysis that considered only students who spent at least one hour on each programming assignment. The results of this analysis, in which 63 out of 665 data points were thrown out, are also presented in Table 19. Significant factors in the multivariate regression run on all data points are presented in Table 20, while Table 21 shows the significant factors in the multivariate regression run on only those data points corresponding to students who were active for at least an hour.

As indicated in Table 19, the NPSM was a significant but weak predictor of individual assignment scores. If we filter out data corresponding to students who spent less than an hour of programming time on an assignment, the NPSM model accounts for more variance ($r^2 = 0.11$) in assignment grades. Interestingly, setting a minimum time limit of one hour altered three of the five significant contributing factors in the NPSM.

We next aggregated an entire semester's worth of IDE data and correlated these data with students' overall assignment averages. This yielded a significant relationship between the NPSM and a student's assignment average ($F(10,84) = 7.00$, $p < 0.01$). In considering the entire semester, we also found a substantial increase in explained variance from an $r^2$ of 11% when using a single assignment's data to an $r^2$ of 39% when considering the entire data set. Significant contributors in the NPSM model are shown in Table 22.

Interestingly, when the input dataset was expanded to include all data collected throughout the semester, the number of NPSM variables that made significant contributions shrank from three (NU, UU, RU) to two
(UU, NU) (see Table 22). Moreover, both of these variables (UU and NU) were negatively correlated with performance. Recall that the UU state is used when students first begin programming. It makes sense that the longer students go without compiling or running their programs, the more likely it is that they will do poorly on the assignment. Likewise, it makes sense that students who spend large proportions of time in the NU state would tend to do worse on assignments, since students in that state are grappling with syntax errors, and may not ever be able to execute their programs.

Lastly, we consider the NPSM's ability to explain the variance in students' final course grades. As was the case with assignment averages, we used students' programming behavior over the entire semester as input to each measure. Again, we detected a significant relationship between the NPSM and final grade ($F(10,84) = 6.63, \ p < 0.01, \ r^2 = 0.36$). Significant factors in the NPSM are shown in Table 23. The NPSM appears slightly worse at accounting for the variance in final grades than at accounting for the variance in average assignment scores. We note that the two significant factors in the NPSM (UU and NU) remained the same in its correlations with assignment average and final grade.

5.2.3 Deriving a Predictive Measure

The previous section showed that the NPSM was able to account for substantially more variation in student performance than both the Error Quotient and Watwin Scores. Given this potential, it makes sense to derive a predictive measure that can be used in situ to predict performance, rather than post hoc to explain variance. We now present a follow-up study that uses the results of the previous study to derive a predictive formula rooted in the NPSM.

Given that the NPSM includes eleven predictors, the ideal sample size for achieving full statistical power when deriving a predictive measure would be approximately 220 students (e.g. Harrell, 2001; C. R. Wilson, Voorhis, & Morgan, 2007). While it is still possible to detect strong effects on a smaller sample size, running eleven predictors against our sample size of 95 students increases the probability of producing a significant model without any significant predictors. For this reason, we restricted ourselves to the
We began by examining the seven significant variables identified in our prior analysis: YN, RU, RN, R/, NU, UU, and time on task. A preliminary data analysis using datasets of varying sizes (see Figure 39) revealed a sporadic level of significance for YN, R/, and time on task. Therefore, we decided to drop these variables from further consideration, and settled on the variables UU, NU, RU, and RN for our predictive model.

For the present analysis, we evaluated the NPSM using seven input data sets whose sizes were systematically varied, as illustrated in Figure 39. The first data set consisted solely of the data collected
during the first programming assignment. The final six data sets each added an additional assignment's grades and programming data. Therefore, starting with the second, data set, the outcome variable was the average of all the programming assignment scores received up to that point in time. It follows that the final data set included all programming data from the semester, matching the dataset used in the previous study.

For each of the seven data sets, a multivariate regression was performed using UU, NU, RU, and RN as predictor variables and assignment averages as outcome variables. Table 24 provides the individual contribution of each predictive variable. The bottom two rows of the table compute the average value and weighted average value of each coefficient.

In examining the coefficients listed in Table 24, we see that the RU and RN variables are consistently significant regardless of the amount of data considered. In contrast, the UU and NU variables only become significant as the amount of data considered increases. However, in examining the standardized beta coefficients, we see that the relative contributions of RU and RN decrease as the size of the data increases, whereas the relative contributions of UU and NU increase as the size of the data increases. Finally, we see a drop in the amount of variance explained when adding data associated with the last assignment. Whether this represents a true ceiling in the NPSM's predictive capability is left for future research.

![Figure 39. Seven Programming Data sets of Increasing Size as a Percentage of All Programming Data](image-url)
Running the NPSM model with the variables UU, NU, RU, and RN across the seven overlapping data sets reveals a general trend in which the amount of variance increases with the size of the data set. We now use the coefficients from these results in order to formulate two predictive measures. The first is obtained by averaging the unstandardized beta coefficients of each predictor variable across the data sets considered. The second model is obtained by using a weighted averaged of each predictor variable's unstandardized beta coefficients. Recall that the weighted average is formulated based on the overall model's variance numbers. Using the averaged coefficient values reported in Table 24, we arrive at the following formula:

\[ \text{NPSM Score} = 69.78 + (1.75 \times \text{RN}) + (0.66 \times \text{RU}) - (0.63 \times \text{UU}) - (0.43 \times \text{NU}) \]

Using the weighted coefficient values yields a slightly different formula:

\[ \text{NPSM Score} = 73.42 + (1.58 \times \text{RN}) + (0.61 \times \text{RU}) - (0.84 \times \text{UU}) - (0.45 \times \text{NU}) \]

To verify the accuracy of each formula, we calculated the predicted score for each dataset using both formulas. Next, we performed a linear regression using this predicted score as the predictor variable and the student's actual assignment score as the outcome variable. We deemed a formula to be successful if it closely mirrored the total amount of variance explained by the overall NPSM model. The amount of variance accounted for by each formula, as well as the overall NPSM model, is listed in Table 25. Inspection of Table 25 reveals that both formulas closely mirror each other (within +/- 2%), and that both are close to the overall NPSM model in terms of explanatory power. As such, it would appear that either formula does an adequate job of transforming the NPSM data into a usable predictive measure.
5.2.4 Discussion

The results presented in this discussion would seem to indicate that the NPSM can be used as a moderately successful predictor for student achievement on homework assignments. RN (execute a semantically incorrect program) and RU (execute a semantically unknown program) were found to be positive contributors to student success. Conversely, UU (default state before first compilation/execution action is taken in a programming session) and NU (syntactically incorrect program) were found to be negative contributors to student success. This seems to indicate that toying with a program's runtime behavior, regardless of semantic correctness, is a successful programming approach. In contrast, writing large portions of code without attempting to compile (UU) is not conducive to success. It is easy to imagine that when these students finally do compile, they quickly find themselves in NU, the other state negatively correlated with performance. Furthermore, the significance of NU as an explanatory factor aligns well with the Error Quotient and Watwin Score (Watson et al., 2014), two prior predictive measures, as both can be seen as quantifying how students leave the NU state.

In comparing the explanatory power of the NPSM to prior studies that examined the Error Quotient and Watwin Score, we find that the NPSM has higher predictive capabilities: The NPSM accounts for 41% of variance in assignment scores, the Watwin Score 36%, and Error Quotient 18%. However, it should be noted that because differing data sets were used, drawing any definite conclusions from these differences would be overly presumptuous. In Section 5, we compare all three measures on the same dataset.
5.3 Deriving a Model from Transition Chains

In the previous section, we categorized student programming behavior into one of ten possible states and derived the Normalized State Programming Model (NPSM). In this section, we take an alternate approach by examining how students arrived at each state. That is, we will focus on the transitions between PSM states rather than on the times spent in the states themselves. We call this approach the Transition-Based Programming State Model (TPSM).

Analyzing transitions is a common activity in the field of Activity Recognition, which attempts to categorize a stream of log data into discrete behaviors. For example, Activity Recognition in the area of smart home environments might try to use sensor data to distinguish between making dinner and washing dishes while in the kitchen (Kim et al., 2010). In order to make these distinctions, researchers often first pilot activities in controlled laboratory settings, so that machine learning techniques can properly distinguish between two similar activities. Unfortunately, the present methodology was decided upon long after data collection occurred. This prevents us from utilizing a similar methodology; we have no lab studies on which to train a learning machine; we must turn to an alternate approach.

Rather than using machine learning techniques to categorize behavior and make predictions based on prior behavior, we have instead opted to describe programming behaviors. To this end, we identify common programming cycles, attempt to temporally place these cycles within the broader context of a given programming assignment, and examine how these cycles might relate to a student's programming performance within a given course.

5.3.1 Methodology

For this section, we again draw from the dataset used in the prior section. This data was collected during the spring 2014 offering of CptS 122 (CS2) at Washington State University. The course enrolled 140 students, 129 of whom finished the course and received a grade. 108 of these students (100 men, 8 women) consented to releasing their data.
5.3.2 Identifying Common PSM Transitions

We begin our analysis by identifying commonly occurring transition chains within the PSM. Note that while the PSM contains ten states, it is not possible to reach every other state from a given starting state, as many states only contain a single incoming edge. The sparseness of the PSM graph automatically gives rise to several inherit transitions. We break these transitions into two categories:

**Vertical Transitions** are those that cause the student to move vertically (either up or down) in the PSM. A vertical transition is an indication that a student is toggling between editing and running their application with no change in the underlying syntactical correctness of their project. We identified six vertical transitions within the PSM:

1. YU - RU - YU (running a semantically unknown program outside of debug mode)
2. YU - DU - YU (running a semantically unknown correct program in debug mode)
3. YN - RN - YN (running a semantically incorrect program outside of debug mode)
4. YN - DN - YN (running a semantically incorrect program in debug mode)
5. NU - R/ - NU (attempting to execute a syntactically incorrect program)
6. NN - R/ - NN (attempting to execute a syntactically incorrect program)

**Horizontal Transitions** are those that cause the student to move horizontally (either left or right) in the PSM. Horizontal transitions indicate a change in either syntactic or semantic correctness of the student's solution. We identified three horizontal transitions within the PSM:

1. YU - NU (failed compilation attempt)
2. YU - DU - YN (a runtime exception in debug mode occurred)
3. YN - DN - YU (a runtime exception in debug mode did not occur)

In addition to the transitions identified above, we examined programming log data obtained throughout the course for common sequence patterns. To accomplish this, we generated every possible permutation, counted its frequency in the data, and eliminated sequences unique to individual students. Furthermore, as
this process recognizes similar patterns as separate entities (e.g. YU-RU-YU-RU and RU-YU-RU-YU likely describe a similar sequence of events), we used the Needleman Wunsch algorithm (Needleman & Wunsch, 1970) to align similar sequences. After completing this process, we identified 21,215 pattern groups. Excluding the previously identified patterns, which we found to be the most popular patterns, we identified an additional eight commonly occurring (those that with at least 1,000 occurrences) programming patterns. This selection was based on frequency of occurrence and all eight patterns occurred at least 1,000 times throughout the semester. The additional patterns are as follows:

1. Idle - YN - Idle
2. Idle - YU - Idle
3. NU - R/ - YU
4. Idle - NU - Idle
5. Idle - UU - Idle
6. RN - DN - YN - DN and YN - RN - DN
7. YU - RU - DU - YU
8. YN - DN - YU - DU and DN - YU – DU

Because these patterns were identified using post-hoc analysis, we are unable to assign meaning to any of the above eight patterns. It is entirely possible that a single pattern chain (e.g. Idle – YN – Idle) occurs in disparate circumstances and therefore is associated with multiple programming behaviors. Exploring this limitation will require additional lab studies and data analysis.

5.3.3 Associating Transitions with Performance

Having identified common transition chains within the PSM, we now turn to connecting these transitions to learning outcomes. To this end, we captured the number of times that a cycle occurred for each student. However, using simple counts has the disadvantage that it treats all cycles the same. For example, a short cycle (e.g. "YU-RU-YU") would receive the same weighting as a longer cycle of the same
type (e.g. "YU-RU-YU-RU-YU"). In an effort to counterbalance this effect, we decided to weight the counts by time. For example, given the cycle "YU-RU-YU", we count the time elapsed between the first transition and final transition as time spent in the general "YU-RU-YU" cycle.

Having normalized transitions by time, we next performed an ANOVA on both cycle frequency and cycle times. As is evident by the summary of results presented in Table 26 and Table 27, final course grade seems to have a significant effect on most cycles. Note that for cycles that involve the idle transition, the duration implicitly includes time not spent programming, whereas the other cycles only include time spent programming. Therefore, given that students do not spend the majority of the day programming, idle cycles will naturally have much larger times than the other cycles.

There appears to be a significant difference between most cycles with respect to the final course grade of the students who encounter them, regardless of whether considering counts or times. However, the astute reader will notice that both the total number of cycles observed (last row of Table 26) and the total time spent programming (last row of Table 27) significantly differ between groups. Due to this significance, it is possible that the observed differences are merely a result of spending more time programming and not an actual difference in programming behavior as dictated by a given programming cycle. To test this hypothesis, we normalized both data sets by factoring in total number of cycles generated (for Table 26) or the total amount of time spent programming (for Table 27). Performing a second ANOVA on the normalized data tells a much different story. Table 28 lists the transitions that remained significant after the normalization process.

A post-hoc Bonferroni analysis relating transitions to grade levels on the remaining significant transitions reveals the following:
Table 26: Frequency of Cycle by Final Course Grade (* = sig at p < 0.05)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Count - F</td>
<td>48</td>
<td>20</td>
<td>95</td>
<td>4</td>
<td>19</td>
<td>13</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>13</td>
<td>32</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Count - D</td>
<td>66</td>
<td>41</td>
<td>123</td>
<td>9</td>
<td>60</td>
<td>22</td>
<td>14</td>
<td>10</td>
<td>8</td>
<td>22</td>
<td>51</td>
<td>14</td>
<td>10</td>
<td>6</td>
<td>10</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Count - C</td>
<td>122</td>
<td>69</td>
<td>227</td>
<td>15</td>
<td>51</td>
<td>65</td>
<td>21</td>
<td>3</td>
<td>20</td>
<td>33</td>
<td>72</td>
<td>25</td>
<td>13</td>
<td>10</td>
<td>13</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Count - B</td>
<td>127</td>
<td>59</td>
<td>169</td>
<td>10</td>
<td>60</td>
<td>60</td>
<td>28</td>
<td>10</td>
<td>10</td>
<td>39</td>
<td>86</td>
<td>23</td>
<td>10</td>
<td>14</td>
<td>14</td>
<td>12</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Count - A</td>
<td>173</td>
<td>108</td>
<td>201</td>
<td>17</td>
<td>64</td>
<td>68</td>
<td>18</td>
<td>3</td>
<td>13</td>
<td>61</td>
<td>110</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>28</td>
<td>11</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Table 27: Time (in Minutes) Spent in Cycle by Final Course Grade (* = sig at p < 0.05)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Time - F</th>
<th>Time - D</th>
<th>Time - C</th>
<th>Time - B</th>
<th>Time - A</th>
</tr>
</thead>
<tbody>
<tr>
<td>YU-RU-YU*</td>
<td>438</td>
<td>473</td>
<td>1,104</td>
<td>1,112</td>
<td>1,552</td>
</tr>
<tr>
<td>YN-RN-YN*</td>
<td>156</td>
<td>338</td>
<td>458</td>
<td>373</td>
<td>802</td>
</tr>
<tr>
<td>NU-R/NU*</td>
<td>95</td>
<td>123</td>
<td>227</td>
<td>169</td>
<td>201</td>
</tr>
<tr>
<td>NN-R/NN*</td>
<td>10</td>
<td>6</td>
<td>9</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>YU-DU-YU*</td>
<td>82</td>
<td>375</td>
<td>165</td>
<td>255</td>
<td>192</td>
</tr>
<tr>
<td>YN-DN-YN*</td>
<td>81</td>
<td>134</td>
<td>224</td>
<td>269</td>
<td>234</td>
</tr>
<tr>
<td>YU-NU-YU</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>YU-DU-YU-YN</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>YN-DN-YU</td>
<td>13</td>
<td>21</td>
<td>43</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>Idle-YN-Idle*</td>
<td>12,443</td>
<td>21,521</td>
<td>20,441</td>
<td>24,431</td>
<td>27,389</td>
</tr>
<tr>
<td>Idle-YU-Idle*</td>
<td>30,538</td>
<td>51,302</td>
<td>55,778</td>
<td>58,151</td>
<td>66,512</td>
</tr>
<tr>
<td>NU-R/-YU*</td>
<td>15</td>
<td>14</td>
<td>13</td>
<td>16</td>
<td>14</td>
</tr>
<tr>
<td>Idle-NU-Idle</td>
<td>8,545</td>
<td>7,995</td>
<td>9,329</td>
<td>8,504</td>
<td>7,815</td>
</tr>
<tr>
<td>Idle-UU-Idle</td>
<td>24,278</td>
<td>28,753</td>
<td>29,397</td>
<td>26,832</td>
<td>21,976</td>
</tr>
<tr>
<td>RN-DN-YN*</td>
<td>10</td>
<td>20</td>
<td>32</td>
<td>30</td>
<td>44</td>
</tr>
<tr>
<td>YU-RU-DU*</td>
<td>15</td>
<td>17</td>
<td>31</td>
<td>32</td>
<td>39</td>
</tr>
<tr>
<td>DN-DN-YU</td>
<td>6</td>
<td>18</td>
<td>16</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>YN-DN-YU-DU</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>Total Time Programming*</td>
<td>1,863</td>
<td>2,736</td>
<td>3,444</td>
<td>3,405</td>
<td>4,763</td>
</tr>
</tbody>
</table>

Table 28: Significant Normalized Transitions by Final Course Grade (p < 0.05)

<table>
<thead>
<tr>
<th>Cycle</th>
<th>F</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle-YN (count)</td>
<td>4.0%</td>
<td>5.2%</td>
<td>4.9%</td>
<td>6.6%</td>
<td>7.1%</td>
</tr>
<tr>
<td>Idle-NU (count)</td>
<td>5.1%</td>
<td>2.5%</td>
<td>2.8%</td>
<td>2.1%</td>
<td>1.8%</td>
</tr>
<tr>
<td>Idle-UU (count)</td>
<td>3.7%</td>
<td>1.7%</td>
<td>1.6%</td>
<td>1.6%</td>
<td>1.0%</td>
</tr>
<tr>
<td>Idle-UU (time)</td>
<td>3.072</td>
<td>1.049</td>
<td>1.033</td>
<td>1.241</td>
<td>549</td>
</tr>
</tbody>
</table>
• For Idle-NU (count), post-hoc analysis was unable to detect significant differences between any groups; however, differences approaching significance were found between F and B students ($p = 0.06$) and F and A students ($p = 0.05$).

• For Idle-NU (count), post-hoc analysis revealed significant differences between F and B students (mean 5.1 vs 1.8, $p = 0.01$) and F and A students (mean 5.1 vs 1.3, $p = 0.02$).

• For Idle-UU (count), post-hoc analysis revealed significant differences between F and C students ($p = 0.01$), F and B students ($p < 0.01$), and F and A students ($p < 0.01$).

• For Idle-UU (time), post-hoc analysis revealed a significant differences between F and B students ($p = 0.04$) and F and A students ($p = 0.01$).

The results presented in this section seem to indicate little relationship between PSM cycles and overall class performance. While the frequency and duration of individual cycles differs significantly between grade level, this increase in time and frequency coincide with an overall increase in expended effort. In other words, it can be argued that the significant effect of an individual cycle is merely a proxy for overall effort. Indeed, attempting to account for this increase in effort results in only four cycles (Idle-YN, Idle-NU, and Idle-UU) that differ significantly between grade levels.

It is interesting to note that all of the significant transitions represent programming sessions focused on editing activities. Based on these results, the key difference between the programming behavior of students of differing levels of course achievement can be characterized as follows: A and B students tend to expend more of their effort editing programs that have had recent runtime exceptions, whereas F students tend to expend more of their effort editing programs that either have syntax errors, or that have never been compiled. As C and D students' behaviors do not differ significantly from either extreme (F or A/B), we might conclude that C and D students exhibit behavior similar to both groups. In all cases, editing a project without periodically exploring its compilation and runtime behavior appears, on the surface, to be a suboptimal solution strategy, and our cycle-based analysis bears this out.
5.3.4 Exploring Transition Cycles over Time

In the prior section, we found that few of the most common PSM transitions had a significant relationship to course achievement. In this section, we consider whether or not differences might exist when we also factor in the relationship between cycles and proximity to an assignment's due date. For example, it seems plausible that F students might encounter "troublesome" cycles closer to an assignment's due date, whereas A students might spend the same time testing their program. To explore this possibility, we use the data set from the prior section and count the number of times a transition occurs on a particular day. As assignment periods were of differing lengths, we recorded the frequency relative to each assignment's due date. This process resulted in 18 data points (one for each transition) for each student for the 21 days leading up to an assignment. Inspection of this dataset reveals little activity prior to the 14th day of an assignment's due date. In order to increase the clarity of our graphs, we decided to only consider activity that occurs up to 14 days before the due date.

We begin our analysis by examining the average number of transitions generated by students of each grade level (Figure 40). We notice that on most days, an A student generates more cycles than other students. We also notice similar patterns of behavior between B/C and D/F students. While all students ramp up their programming efforts as an assignment due date approaches, we see that A students are far more consistent in their programming efforts. For example, effort expended by A students eight days prior to the due date exceeds the effort expended by B-F students only three days prior to the due date.

Having identified general programming trends leading up to a homework's due date, we now turn to the identification of differing transition patterns that occur near the end of an assignment.

For each transition chain under consideration (see Table 26), we constructed 18 graphs similar to Figure 40 that track the frequency of a given transition chain as a homework's due date approaches. From these graphs, we identified eight sequences whose graphs appear to be affected by grade level: YU-RU, NU-R/, NN-R/, YU-DU, NU-R/-YU, Idle-NU, YN-DN-YU, and YU-RU-DU. In order to get a sense of how these transitions identified vary over time, we next plotted them in aggregate over time as the due date approaches.
(see Figure 41). From Figure 41, we notice that the majority of transition activity occurs during the four days leading up to an assignment’s due date. As such, we limit our statistical analysis to these four days.

Focusing on the data on the four days preceding an assignment due date, we next performed a three-way analysis of variance (ANOVA), inputting letter grade, day, and transition as predictor variables and frequency as the outcome variable. Statistical analysis revealed significant main effects on letter grade ($F(4,199) = 9.23, p < 0.01, \text{partial } \eta^2=0.01$), day ($F(4,199) = 19.17, p < 0.01, \text{partial } \eta^2=0.02$), and transition ($F(7,199) = 21.29, p < 0.04, \text{partial } \eta^2=0.04$). Bonferroni post hoc tests revealed the following:

- C students produced significantly more ($p < 0.05$) transitions ($M=23$) than F ($M=10$), D ($M=10$), and B ($M=17$) students.
- The transitions YU-RU, YU-DU, and Idle-NU were significantly ($p < 0.05$) more likely to occur than the other transitions
- The transitions NN-R/ and YU-RU-DU were significantly ($p < 0.05$) less likely to occur than the other transitions
- Days 1 and 2 were significantly ($p < 0.05$) more active than all other days
- Day 0 (the due date) was significantly ($p < 0.05$) less active than all but day 4

In addition to discovering significant main effects, statistical analysis revealed a significant two-way interaction between letter grade and transition ($F(28,199) = 16.85, p < 0.01, \text{partial } \eta^2=0.12$) and a statistically significant interaction between all three predictor variables ($F(140,199) = 1.29, p = 0.01, \text{partial } \eta^2=0.05$).
Figure 40: Average Number of Transitions Generated Per Day

<table>
<thead>
<tr>
<th>Days Until Due Date</th>
<th>Average # Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>35</td>
</tr>
<tr>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>9</td>
<td>45</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>55</td>
</tr>
<tr>
<td>12</td>
<td>60</td>
</tr>
<tr>
<td>13</td>
<td>65</td>
</tr>
<tr>
<td>14</td>
<td>70</td>
</tr>
<tr>
<td>15</td>
<td>75</td>
</tr>
<tr>
<td>16</td>
<td>80</td>
</tr>
<tr>
<td>17</td>
<td>85</td>
</tr>
<tr>
<td>18</td>
<td>90</td>
</tr>
<tr>
<td>19</td>
<td>95</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>21</td>
<td>105</td>
</tr>
<tr>
<td>22</td>
<td>110</td>
</tr>
<tr>
<td>23</td>
<td>115</td>
</tr>
<tr>
<td>24</td>
<td>120</td>
</tr>
<tr>
<td>25</td>
<td>125</td>
</tr>
<tr>
<td>26</td>
<td>130</td>
</tr>
<tr>
<td>27</td>
<td>135</td>
</tr>
<tr>
<td>28</td>
<td>140</td>
</tr>
<tr>
<td>29</td>
<td>145</td>
</tr>
<tr>
<td>30</td>
<td>150</td>
</tr>
<tr>
<td>31</td>
<td>155</td>
</tr>
<tr>
<td>32</td>
<td>160</td>
</tr>
<tr>
<td>33</td>
<td>165</td>
</tr>
<tr>
<td>34</td>
<td>170</td>
</tr>
<tr>
<td>35</td>
<td>175</td>
</tr>
<tr>
<td>36</td>
<td>180</td>
</tr>
<tr>
<td>37</td>
<td>185</td>
</tr>
<tr>
<td>38</td>
<td>190</td>
</tr>
<tr>
<td>39</td>
<td>195</td>
</tr>
<tr>
<td>40</td>
<td>200</td>
</tr>
<tr>
<td>41</td>
<td>205</td>
</tr>
<tr>
<td>42</td>
<td>210</td>
</tr>
<tr>
<td>43</td>
<td>215</td>
</tr>
<tr>
<td>44</td>
<td>220</td>
</tr>
<tr>
<td>45</td>
<td>225</td>
</tr>
<tr>
<td>46</td>
<td>230</td>
</tr>
<tr>
<td>47</td>
<td>235</td>
</tr>
<tr>
<td>48</td>
<td>240</td>
</tr>
<tr>
<td>49</td>
<td>245</td>
</tr>
<tr>
<td>50</td>
<td>250</td>
</tr>
<tr>
<td>51</td>
<td>255</td>
</tr>
<tr>
<td>52</td>
<td>260</td>
</tr>
<tr>
<td>53</td>
<td>265</td>
</tr>
<tr>
<td>54</td>
<td>270</td>
</tr>
<tr>
<td>55</td>
<td>275</td>
</tr>
<tr>
<td>56</td>
<td>280</td>
</tr>
<tr>
<td>57</td>
<td>285</td>
</tr>
<tr>
<td>58</td>
<td>290</td>
</tr>
<tr>
<td>59</td>
<td>295</td>
</tr>
<tr>
<td>60</td>
<td>300</td>
</tr>
<tr>
<td>61</td>
<td>305</td>
</tr>
<tr>
<td>62</td>
<td>310</td>
</tr>
<tr>
<td>63</td>
<td>315</td>
</tr>
<tr>
<td>64</td>
<td>320</td>
</tr>
<tr>
<td>65</td>
<td>325</td>
</tr>
<tr>
<td>66</td>
<td>330</td>
</tr>
<tr>
<td>67</td>
<td>335</td>
</tr>
<tr>
<td>68</td>
<td>340</td>
</tr>
<tr>
<td>69</td>
<td>345</td>
</tr>
<tr>
<td>70</td>
<td>350</td>
</tr>
<tr>
<td>71</td>
<td>355</td>
</tr>
<tr>
<td>72</td>
<td>360</td>
</tr>
<tr>
<td>73</td>
<td>365</td>
</tr>
<tr>
<td>74</td>
<td>370</td>
</tr>
<tr>
<td>75</td>
<td>375</td>
</tr>
<tr>
<td>76</td>
<td>380</td>
</tr>
<tr>
<td>77</td>
<td>385</td>
</tr>
<tr>
<td>78</td>
<td>390</td>
</tr>
<tr>
<td>79</td>
<td>395</td>
</tr>
<tr>
<td>80</td>
<td>400</td>
</tr>
<tr>
<td>81</td>
<td>405</td>
</tr>
<tr>
<td>82</td>
<td>410</td>
</tr>
<tr>
<td>83</td>
<td>415</td>
</tr>
<tr>
<td>84</td>
<td>420</td>
</tr>
<tr>
<td>85</td>
<td>425</td>
</tr>
<tr>
<td>86</td>
<td>430</td>
</tr>
<tr>
<td>87</td>
<td>435</td>
</tr>
<tr>
<td>88</td>
<td>440</td>
</tr>
<tr>
<td>89</td>
<td>445</td>
</tr>
<tr>
<td>90</td>
<td>450</td>
</tr>
<tr>
<td>91</td>
<td>455</td>
</tr>
<tr>
<td>92</td>
<td>460</td>
</tr>
<tr>
<td>93</td>
<td>465</td>
</tr>
<tr>
<td>94</td>
<td>470</td>
</tr>
<tr>
<td>95</td>
<td>475</td>
</tr>
<tr>
<td>96</td>
<td>480</td>
</tr>
<tr>
<td>97</td>
<td>485</td>
</tr>
<tr>
<td>98</td>
<td>490</td>
</tr>
<tr>
<td>99</td>
<td>495</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
</tr>
</tbody>
</table>

Figure 41: Average Number of Identified Transitions Leading Up to Assignment Due Date
The interaction graph between letter grade and transition is presented in Figure 42. Post hoc pairwise analysis revealed the following significant ($p < 0.05$) relationships:

- YU-RU was significantly more likely associated with A students
- YU-DU was significantly more likely associated with B students
- NU-R/-YU was significantly more likely associated with B and C students
- Idle-NU and YU-RU-DU were significantly more likely associated with C students

Finally, we consider significant differences in the three-way interaction by performing post hoc pairwise analysis for the four days leading up to an assignment due date. Because this analysis produced 200 interaction points (8 transitions * 5 grade levels * 5 days), we provide a descriptive analysis of the data in the following subsections.

### 5.3.4.1 Patterns of Behavior Four days before Due Date
We begin our descriptive analysis by considering the interaction between letter grade and transitions generated on the day farthest from the due date (Figure 43). We see similar levels of activity from all grade levels with the exception of YU-RU, YU-DU, Idle-NU, and YU-RU-DU:

- For YU-RU, A students generated significantly more \((p < 0.05)\) cycles than any other grade level.
- For YU-DU, B students generated significantly more \((p < 0.05)\) cycles than any other grade level.
- For Idle-NU, C students generated significantly more \((p < 0.05)\) cycles than any other grade level.
- For YU-RU-DU, C students generated significantly more \((p < 0.05)\) cycles than any other grade level.

These findings would seem to indicate that A, B, and C students are already at the testing phase. However, unlike A and B students, we also see that C students are spending a lot more time working on an uncompiled project, as indicated by the number of Idle-NU cycles generated. Interestingly, we see that B students are apparently using the debugger on a much more frequent basis than A students. Investigating the cause of this difference would be a worthwhile focus for future research.

### 5.3.4.2 Patterns of Behavior Three Days before Due Date

We next consider interaction between letter grade and transitions generated three days prior to the due date (Figure 44). Unlike the previous day, we see marked differences between the transitions generated by each grade level. Furthermore, we begin to see increased activity from F students, who generated the 2\textsuperscript{nd} most YU-RU cycles. Statistical analysis discovered the following differences \((p < 0.05)\) between the groups:

- For YU-RU, A students again generated significantly more transitions than all other grade levels.
  F students generated significantly more YU-RU transitions than B and D students.
- For YU-DU, B students again generated significantly more transitions than all other grade levels.
- For both Idle-NU and YU-RU-DU, C students generated significantly more transitions than all other grade levels.
Figure 43: Interaction between Letter Grade and Transition Four Days before Due Date

Figure 44: Interaction between Letter Grade and Transition Three Days before Due Date
It would seem as though the patterns of cycle generation identified in the prior section continue. Again, we see A students adopting a strategy comprised mainly of YU-RU cycles, B students with YU-DU, and C students with Idle-NU and YU-RU-DU. Unlike the previous day, we see an increase in YU-RU cycles generated by F students. However, other cycle activity remains minimal.

5.3.4.3 Patterns of Behavior Two Days before Due Date

As illustrated in Figure 45, behavioral trends for A, B, and C students remains steady. That is, A students generate the most YU-RU transitions, B students generate the most YU-DU transitions, and C students generate the most Idle-NU and YU-RU-DU transitions. However, the overall increase of activity makes it more difficult for statistical tests to detect significant ($p < 0.05$) differences between certain groups:

- YU-RU transitions continue to be significantly more likely to be generated by A students
- B students are statistically more likely to generate YU-DU transitions when compared to A, D, and F students. There was no statistically significant difference between B and C students.
- B students are statistically more likely to generate NU-R-YU transitions when compared to A, C, and F students. There was no statistically significant difference between B and D students.
- C students are significantly more likely to generate Idle-NU transitions when compared to A, B, and F students. There was no statistically significant difference between C and D students.
- C students are significantly more likely to generate YU-RU-DU transitions when compared to A, B, and F students. There was no statistically significant difference between C and D students. This is likely due to the relative low number of D students in the sample.

Again, we see well-defined patterns of behavior for A, B, and C students. In contrast, while D and F students’ transition frequencies have increased, they lack a definite common pattern, which prevents us from identifying transitions unique to their problem solving behaviors.
5.3.4.4 Patterns of Behavior on Day before Due Date

Once again, we witness the three peaks of activity characteristic of A, B, and C students (Figure 46). Statistical analysis discovered the following significant ($p < 0.05$) findings:

- A and F students are more likely to generate YU-RU transitions than C and B students. Analysis could not detect differences between A, F, and D students.
- B students continue to produce more YU-DU transitions than any other grade level.
- Additionally, B students produce more Nu-R/-YU transitions than A and C students. There was not statistical difference between B, D, and F students.
- Likewise, C students continue to produce more Idle-NU transitions than any other grade. D students produce more than A students.
- Again, C students produce the most YU-RU-DU transitions.
While we continue to see the same pattern of cycle generation from A, B, and C students, we also witness a slight reduction in B's generation of NU-R/-YU cycles. Two days prior to the due date, B students generated an average of 67 of these transitions whereas on the day before the due date, B students generated 44. This might indicate that B students have transitioned from working through both compiler and runtime issues and are focusing more attention on runtime behavior.

5.3.4.5 Patterns of Behavior on Due Date

Lastly, we examine the frequency of transitions on the assignment's due date (Figure 47). As assignments were due in the early morning, we see an overall decline in transition frequency. Nevertheless, the transition patterns between grade levels identified on the previous four days remain. Indeed, tests of statistical significance reveal the following significant ($p < 0.05$) differences:

- A and C students produced significantly more YU-RU cycles
• B students produced more YU-DU cycles than F students. Statistical differences were not found between A-D students.

• C students continue to produce significantly more Idle-NU cycles

5.3.4.6 Discussion

In examining eight popular transitions derived from the PSM, we see clear differences in behavior between A, B, and C students. Namely, A students appear to favor a strategy that involves transitioning between YU and RU PSM states, B students favor an approach that involves transitioning between YU-DU and NU-R/-YU, and C students tend to generate the most Idle-NU and YU-RU-DU transitions. Other than the fact that D and F students generated significantly fewer transitions than the other groups, no distinctive behavior for these groups could be identified. Given this, it would appear as though a simple time on task measure would be sufficient to differentiate between students A, B, and C students from D and F students.

Figure 47: Interaction between Letter Grade and Transition on Due Date
It is also worth mentioning that frequent debugging, a behavior widely known to be beneficial, is most commonly associated with B students. Interestingly, the frequency of debugging behavior for A students does not differ much from C, D, and F students. Instead, A students often run without debug mode, making it more difficult to diagnose issues that occur during runtime. In future work, we would like to further investigate this finding.

Like A and B students, we find that C students also frequently run their code. However, in contrast to A and B students, we find that C students spend more time working on code that does not compile (Idle-NU). This might suggest that C students encounter more compiler errors than A or B students, perhaps due to their underdeveloped knowledge of programming constructs. Again, we leave this as an open question for future work.

Lastly, we would like to express caution in interpreting these results. It is likely that, for example, A students’ high frequency of YU-RU represents several disparate problem solving activities. In future work, we would like to conduct lab studies so that we can better connect NPSM transitions with definite patterns of behavior.

### 5.4 A Comparison of Predictive Measures

In the prior sections, we have presented two attempts at converting our PSM into a measure that relates student behavior to course achievement. The NPSM produces a predictor that is derived from time spent in each programming PSM state. In contrast, the TPSM investigates transitions between PSM cycles. However, unlike the NPSM, the TPSM itself does not produce an output that can be used to easily relate programming behavior to academic performance. In this section, we compare three measures capable of generating such a predictor variable: the NPSM discussed earlier the Error Quotient and Watwin Score, two prior measures of student performance.
5.4.1 Methodology

Again, we draw data from the dataset used in the prior sections. This data was collected during the spring 2014 offering of CptS 122 (CS2) at Washington State University. The course enrolled 140 students, 129 of whom finished the course and received a grade. 108 of these students (100 men, 8 women) consented to releasing their data.

Data for all measures was collected using OSBIDE (see Chapter 4). We hired a professional programmer to write the software for computing the Error Quotient and Watwin Scores. Because of a key difference between BlueJ and Visual Studio (BlueJ only displays one build error at a time, whereas Visual Studio displays all build errors), we had the programmer create two versions of the Watwin Score: One that mimics the originally-published algorithm by considering only one error at time, and another that considers all build errors. Both versions of the Watwin Score are considered in the analysis that follows.

Three performance indicators were used for this analysis: (1) students' grades on individual assignments; (2) students' overall assignment average, and (3) students' final grades, which were based on the grades received on programming assignments (35%), labs (10%), participation (5%), in-class quizzes (10%), midterm exams (20%), and a final exam (20%). Predictions of individual assignment grades were based exclusively on the programming log data generated while the corresponding assignment was open. (The length of each assignment varied between ten and twenty three days). Predictions of students' overall assignment averages and final grades, in contrast, were based on programming log data generated throughout the entire semester.

5.4.2 Results

To evaluate the ability of the three measures to explain the variance in individual assignment grades, we performed a linear regression, with measure (Error Quotient, Watwin Score, NPSM) as the predictor variable and individual assignment grades as the outcome variable. Note that the results for the NPSM are carried over from the analysis performed in Section 3. Furthermore, recall that we created two datasets for the NPSM: one that considers all students and another that considers only students who worked on the
assignment for at least an hour. In order to present the most robust comparison, we carry this distinction into the present analysis.

As indicated in Table 29, all measures were significant but weak predictors of individual assignment scores. If we filter out data corresponding to students who spent less than an hour of programming time on an assignment, the NPSM model accounted for the most variance ($r^2 = 0.11$) in assignment grades.

We next consider how each measure relates to a student's overall assignment average. Results for each measure are presented in Table 30. By considering an entire semester's worth of data, two of the three predictive measures improved. The Error Quotient's explanatory power decreased, whereas the Watwin Score increased its explanatory power by a factor of five, and the NPSM increased its explanatory power by more than a factor of three. In absolute terms, the NPSM was a substantially better predictor than the other two measures, nearly quadrupling the explanatory power of its closest rival (the Watwin Score).

Lastly, we consider each measure's ability to explain the variance in students' final course grades. As was the case with assignment averages, we used students' programming behavior over the entire semester as input to each measure. The results of this analysis are presented in Table 31. The Error Quotient was the only measure that did not significantly correlate with students' final grades. The Watwin Score appears to be slightly better at explaining the variance in final grades than it was at explaining the variance in assignment scores. In contrast, the NPSM appears slightly worse at accounting for the variance in final grades than at accounting for the variance in average assignment scores. However, as before, the NPSM did a substantially better job in absolute terms, furnishing three times the explanatory power of the Watwin Score, its closest competitor.
5.4.3 Discussion

On this particular dataset, the NPSM is a drastically better predictor than the Error Quotient or Watwin Score. As mentioned previously, the calculations performed by both the Error Quotient and Watwin measures are based on the least weighted significant contributor in the NPSM model: NU. Interestingly, performing a linear regression with NU as the sole predictor variable explains more variance than either the Error Quotient or Watwin Score for both assignment average, F(1,93 = 15.06), \( p < 0.01, \text{Adj.} R^2 = 0.13 \), and final grade, F(1,93 = 23.676), \( p < 0.01, \text{Adj.} R^2 = 0.19 \). This strongly suggests that any measurement based on programming behaviors would do well to look beyond compilation behavior. We explore this possibility further in the next chapter.

The results for the Error Quotient and Watwin Score presented in this section differ drastically from the results presented in prior research. For example, a recent study of both the Error Quotient and Watwin measures accounted for 18% and 36% of the variance in students' final grades (Watson et al., 2014) as

<table>
<thead>
<tr>
<th>Measure</th>
<th>( df )</th>
<th>( F )</th>
<th>( p )</th>
<th>( \text{Adj.} R^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Quotient</td>
<td>1,679</td>
<td>53.65</td>
<td>(&lt; 0.01)</td>
<td>0.07</td>
</tr>
<tr>
<td>Watwin Score (one)</td>
<td>1,662</td>
<td>13.61</td>
<td>(&lt; 0.01)</td>
<td>0.02</td>
</tr>
<tr>
<td>Watwin Score (all)</td>
<td>1,662</td>
<td>14.63</td>
<td>(&lt; 0.01)</td>
<td>0.02</td>
</tr>
<tr>
<td>NPSM (no min. time)</td>
<td>11,653</td>
<td>8.92</td>
<td>(&lt; 0.01)</td>
<td>0.08</td>
</tr>
<tr>
<td>NPSM (&gt;1hr)</td>
<td>10,591</td>
<td>8.92</td>
<td>(&lt; 0.01)</td>
<td>0.11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Measure</th>
<th>( df )</th>
<th>( F )</th>
<th>( p )</th>
<th>( \text{Adj.} R^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Quotient</td>
<td>1,94</td>
<td>7.16</td>
<td>(&lt; 0.01)</td>
<td>0.06</td>
</tr>
<tr>
<td>Watwin Score (one)</td>
<td>1,94</td>
<td>11.50</td>
<td>(&lt; 0.01)</td>
<td>0.10</td>
</tr>
<tr>
<td>Watwin Score (all)</td>
<td>1,94</td>
<td>10.93</td>
<td>(&lt; 0.01)</td>
<td>0.10</td>
</tr>
<tr>
<td>NPSM</td>
<td>10,84</td>
<td>7.00</td>
<td>(&lt; 0.01)</td>
<td>0.39</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Measure</th>
<th>( df )</th>
<th>( F )</th>
<th>( p )</th>
<th>( \text{Adj.} R^2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Quotient</td>
<td>1,94</td>
<td>3.68</td>
<td>0.06</td>
<td>0.03</td>
</tr>
<tr>
<td>Watwin Score (single error)</td>
<td>1,94</td>
<td>14.26</td>
<td>(&lt; 0.01)</td>
<td>0.12</td>
</tr>
<tr>
<td>Watwin Score (all errors)</td>
<td>1,94</td>
<td>14.40</td>
<td>(&lt; 0.01)</td>
<td>0.12</td>
</tr>
<tr>
<td>NPSM</td>
<td>10,84</td>
<td>6.63</td>
<td>(&lt; 0.01)</td>
<td>0.36</td>
</tr>
</tbody>
</table>
compared to merely 3% and 12% in our original study. How can we account for this large discrepancy? We offer two possible explanations.

First, differences in the instructional emphasis of the courses studied might have contributed to the differences in the Error Quotient and Watwin Score observed across the studies. In previous studies in which the Error Quotient and Watwin Score were calculated, student homework was worth just 25% of the overall grade. In contrast, in our study, student homework accounted for 35% of the overall grade.

Second, the discrepancies in Error Quotient and Watwin Score measures might be related to key differences in the programming environments and languages used in the studies. Previous studies focused on the BlueJ (Kölling, Quig, Patterson, & Rosenberg, 2003) and the Java programming language. In contrast, we collected data through Microsoft Visual Studio and the C++ programming language. Both the Error Quotient and Watwin Score rely on the processing of compilation error messages. Given that C++ compilers tend to produce terser and more obtuse compilation error messages, it seems plausible that differences could have occurred with respect to students' compilation behaviors in the two environments. For example, forgetting a semi-colon in BlueJ and Java results in the error message, "error: ';' expected," followed by the exact line on which a semi-colon is missing. In contrast, forgetting a semi-colon in Visual Studio and C++ results in nine error messages. The first message is a red-herring referencing an illegal usage of a type as an expression. For the actual cause, the user must look to the second error message, which states, "syntax error: missing ';' before identifier <x>," with <x> being the line below the statement on which a semi-colon is missing.

Of these two explanations, we find the second one to be the most compelling. Recall that both the Error Quotient and Watwin Score assign penalty points when subsequent compilation attempts either result in more errors, or contain the same error messages as previous compilation attempts. Given that Visual Studio and C++ generate more error messages per compilation, it stands to reason that the Error Quotient and Watwin would artificially inflate the base penalty assigned to students for each failed compilation. Furthermore, in Visual Studio/C++, the possibility that both the Error Quotient and Watwin Score will generate false positives (matched compilations that have the same error message but for different reasons)
increases. In contrast, the coarser approach taken by the NPSM is not affected by these differences: an error state is an error state, regardless of whether a student generated one or one hundred errors in a given compilation.

Even though the amount of variance accounted for by the Error Quotient and Watwin Score in this section is much lower than what has been previously reported, it is still possible to make comparisons with prior studies. For example, in their first study, Watson et al. (2013) found that the predictive power of both the Error Quotient and Watwin Score increased as a function of the size of the input data. When they considered only a single assignment's worth of data (roughly 2-3 weeks), the variance explained by both Error Quotient and Watwin was fairly low: 10% for Error Quotient and 6% for Watwin. However, by the end of the term, the variance explained by Error Quotient and Watwin had increased to 19% and 42% respectively. Using relative magnitudes, we see that, in their study, Error Quotient increased by a factor of two and Watwin by a factor of seven. These results are somewhat consistent with the results of this study, which found that the Error Quotient performed best with smaller data sets, and that the Watwin Score performed best with larger data sets. However, unlike in previously published studies, the variance explained by the Error Quotient in our study actually decreased as the size of the input data set increased. That the relative trend in the amount of variance explained by the Watwin Score is similar across studies, whereas the relative trend in the amount of variance explained by the Error Quotient is not, lends further credence to the idea that these predictive measures do not perform consistently when applied to different programming environments and languages. In order to increase our confidence in this claim, we would need to conduct additional studies of the Error Quotient, Watwin Score, and NPSM using a variety of programming environments, languages, and class levels.

5.5 Summary

We began this chapter by asking how data generated during a student's programming session relates to grades received on programming assignments. To answer that question, we developed the Programming State Model, which tracks a student's programming behavior through a series of states. From this, we
derived two measures, the NPSM and the TPSM. The NPSM, a predictive model based on the time spent in a set of programming states derived from a program's syntactic and semantic correctness, was found to correlate highly with student achievement. While the TPSM does not yield a predictive measure, it does provide evidence that programming behaviors between A, B, and C students differ significantly.

Our second research question asked whether or not a holistic model of student performance can predict academic achievement. The results presented in Section 3 and 5 suggest that a holistic model, as represented by the NPSM, is indeed capable of predicating academic achievement. Furthermore, we find that, at least on the data set considered in our analysis, such a holistic model performs better than models that merely focus on one aspect of programming behavior (compilation). In the next chapter, we consider additional holistic models that consider factors beyond students' programming behaviors.
CHAPTER 6

INVESTIGATING THE RELATIONSHIP BETWEEN PROGRAMMING BEHAVIORS AND ONLINE SOCIAL PARTICIPATION

In Chapter 4, we investigated the conversations centered around programming assignments, and explored correlations between students’ level of participation in these conversations and their course outcomes. In performing this analysis, we found promising connections between student social participation and sense of community and between regular social participation and course grades. In Chapter 5, we turned our attention to students' programming behaviors. We developed the Normalized State Programming Model (NPSM) to categorize programming activities and found a significant connection between NPSM stats and course grades. While both approaches have yielded interesting results, we wondered whether a combined approach—one that considers both social and programming behavior—might provide further insight into relationships between student learning processes and outcomes. To this end, we pose the following research question:

RQ1: How well can a statistical model comprised of both social and programming measures predict students’ classroom performance?

In addition to developing statistical models, we were interested in the interplay between social behavior and programming activities. Do students talk about coding issues that are currently confronting? If so, how do social interactions influence future programming decisions? This interest raises the following research questions:

RQ2: How does coding behavior influence social behavior?

RQ3: How does social behavior influence coding behavior?

In this chapter, we address RQ1 by conducting a follow-up study to the ones conducted in Chapter 5. To address RQ2 and RQ3, we perform a new analysis using measures developed in the previous chapters.
6.1 Incorporating Social Behavior into the NPSM

In Chapter 5, we attempted to predict student performance based solely on a student's programming behavior within the IDE. In Chapters 3 and 4, we explored relationships between students' social behavior and academic performance within a computing course. From this line of inquiry, we derived the Social Role measure, which categorizes the level of participation for a given student over a two week period (roughly the length of an assignment) into one of four levels (Table 32). Recall that Social Role accounted for a significant amount of the variance present in both homework grades and end of term grades.

For the present analysis, we incorporate Social Role into the NPSM and investigate the impact that the inclusion of Social Role has on the NPSM's predictive capabilities. Note that the same dataset from Chapters 4 and 5 are used again in this analysis.

Table 32: Definition of Participation Level Metric

<table>
<thead>
<tr>
<th>Level</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No posts or replies</td>
</tr>
<tr>
<td>2</td>
<td>At least one post or reply, but no more than one of either</td>
</tr>
<tr>
<td>3</td>
<td>Two posts and fewer than two replies, or two replies and fewer than two posts</td>
</tr>
<tr>
<td>4</td>
<td>Two or more posts and two or more replies</td>
</tr>
</tbody>
</table>

6.1.1 Examining the Impact of Social Role on the NPSM

We begin by adding Social Role as an additional predictor in the NPSM model and rerunning the regression analysis performed in Chapter 5. In doing so, we isolate the impact of Social Role on the overall model.

Table 33 compares the explanatory power of the original NPSM and the explanatory power of the model combining Social Role and the NPSM (SNPSM) for individual assignments. Table 34 and Table 35 report the significant predictors in the SNPSM model with no minimum time and with a minimum of one hour of programming activity. As is evident in Table 33, the SNPSM accounts for 2% more variance than the NPSM regardless of time limit. Furthermore, as indicated in Table 34 and Table 35, Social Role is a significant predictor in both cases.
Next, we compare the SNPSM to the SNPSM on overall assignment average. Again, the results in Table 36 indicate that the SNPSM explains 4% more of the variance when compared to the standard NPSM. In Table 37, we see that Social Role is again a significant predictor in the SNPSM model.

Finally, we compare the effectiveness of the SNPSM on final course grades (Table 38). Here, we see the largest difference: the SNPSM explains 12% more variance than the standard NPSM. Indeed, Table 39 shows Social Role as being the strongest predictor variable in the model. This analysis seems to suggest that, when available, it would be beneficial to incorporate a social metric such as Social Role into any model that attempts to account for a student's academic performance in a computing course.
6.1.2 Deriving a Predictive Measure Using the SNPSM

As was the case with the standard NPSM, it makes sense to derive a predictive measure of the NPSM that can be used to model a student's performance on programming assignments. The previous section identified eight variables as candidates for inclusion in a predictive model: UU, NU, YN, RU, RN, R/, time on task, and Social Role. As was the case with the standard NPSM formula, an initial exploration of the data revealed sporadic significance for YN, R/, and time on task. This left UU, NU, RU, RN, and Social Role as viable predictors. Note that best practices suggest having at least 100 independent observations (Harrell, 2001; C. R. Wilson et al., 2007) for a 5-predictor model. Given that the current data set has only 96 observations, it is possible that the following analysis did not have enough power to successfully detect significance for one or more predictor variables.

For this analysis, we used the same sample set and evaluated the SNPSM in the same manner as we did the NPSM described in Chapter 5. Again, we evaluated the SNPSM using seven input data sets whose sizes were systematically varied, as illustrated in Figure 39. The first data set consisted solely of the data collected during the first programming assignment. The final six data sets each added an additional assignment's grades and programming data. Therefore, starting with the second, data set, the outcome variable was the average of all the programming assignment scores received up to that point in time. Similarly, for the first data set, the Social Role value entered into the regression is simply the student's Social Role for that assignment. However, starting with the second data set, the Social Role value entered into the regression is an averaged value for all weeks included in the data set. It follows that the final data set included all programming data from the semester, matching the dataset used in the previous section.

6.1.2.1 Results

For each of the seven data sets, a multivariate regression was performed using UU, NU, RU, RN, and Social Role as predictor variables and assignment averages as outcome variables. Table 40 provides the individual contribution of each predictive variable. The bottom two rows of the table compute the average value and weighted average value of each coefficient.
In examining the coefficients listed in Table 40, we see that RU is consistently significant regardless of the amount of data considered. In contrast, RN is significant for all but the last data set, NU is significant for data sets 4-7, UU is significant for data sets 5-7, and Social Role is sporadically significant. Similarly to the NPSM analysis performed in Chapter 5, we again see a drop in explanatory power between the sixth and seventh data set. We leave the exploration of whether or not this represents a true ceiling in both the NPSM and SNPSM for future work.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>( r^2 )</th>
<th>Constant</th>
<th>Unstandardized ( \beta ) Coefficients</th>
<th>Standardized ( \beta ) Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>12%</td>
<td>46.38</td>
<td>0.15 -0.25</td>
<td>0.87* 2.28* 2.02 0.35 -0.84 0.25* 0.31* 0.06</td>
</tr>
<tr>
<td>A1-A2</td>
<td>19%</td>
<td>43.56</td>
<td>0.70 -0.20</td>
<td>0.73* 2.24* 5.54* 0.26 -0.77 0.25* 0.31* 0.23*</td>
</tr>
<tr>
<td>A1-A3</td>
<td>21%</td>
<td>57.86</td>
<td>-0.03 -0.47</td>
<td>0.74* 2.00* 2.82 -0.01 -0.19 0.27* 0.30* 0.18</td>
</tr>
<tr>
<td>A1-A4</td>
<td>29%</td>
<td>65.28</td>
<td>-0.37 -0.54*</td>
<td>0.58* 1.40* 3.58 -0.13 -0.23* 0.25* 0.25* 0.18</td>
</tr>
<tr>
<td>A1-A5</td>
<td>39%</td>
<td>65.68</td>
<td>-0.91* -0.46*</td>
<td>0.58* 1.41* 3.67* -0.27* -0.20* 0.25* 0.24* 0.18*</td>
</tr>
<tr>
<td>A1-A6</td>
<td>46%</td>
<td>72.86</td>
<td>-1.29* -0.40*</td>
<td>0.53* 1.17* 2.34 -0.40* -0.18* 0.23* 0.19* 0.12</td>
</tr>
<tr>
<td>A1-A7</td>
<td>44%</td>
<td>69.39</td>
<td>-1.01* -0.46*</td>
<td>0.41* 0.93 3.75* -0.36* -0.20* 0.19* 0.16 0.19*</td>
</tr>
<tr>
<td>Average</td>
<td>N/A</td>
<td>60.14</td>
<td>-0.39 -0.40</td>
<td>0.63 1.63 3.39 -0.08 -0.37 0.24 0.25 0.16</td>
</tr>
<tr>
<td>Weighted Average</td>
<td>N/A</td>
<td>64.09</td>
<td>-0.65 -0.42</td>
<td>0.58 1.44 3.37 -0.19 -0.29 0.24 0.23 0.17</td>
</tr>
</tbody>
</table>

### 6.1.2.2 Deriving a Predictive Formula

Running the SNPSM model with variables UU, NU, RU, RN, and Social Role across the seven overlapping data sets reveals a general trend in which the amount of variance increases with the size of the data set. We now use the coefficients from these results to derive a predictive measure. The first formula is derived using averaged unstandardized beta coefficients of each predictor variable. A second formula is generated using weighted averages of the unstandardardized beta coefficients of each predictor variable. Recall that averages are weighted relative to the amount of overall variance explained by a given data set. Using the averaged coefficient values reported in Table 40, we arrive at the following formula:

\[
SNPSM\; Score = 60.14 + (3.39 \times Social Role) + (1.63 \times RN) + (0.63 \times RU) - (0.40 \times NU) - (0.39 \times UU)
\]

Using the weighted coefficient values yields a slightly different formula:

\[
SNPSM\; Score = 64.09 + (3.37 \times Social Role) + (1.44 \times RN) + (0.58 \times RU) - (0.65 \times UU) - (0.42 \times NU)
\]

To verify the accuracy of each formula, we calculated the predicted score for each dataset using both formulas. Next, we performed a linear regression using this predicted score as the predictor variable and the student’s actual assignment score as the outcome variable. We deemed a formula to be successful if it
closely mirrored the total amount of variance explained by the overall SNPSM model. The amount of variance accounted for by each formula, as well as the overall SNPSM model, is listed in Table 41. Inspection of Table 41 reveals that both formulas closely mirror each other (within +/- 2%), and that both are close to the overall SNPSM model in terms of explanatory power. As such, it would appear that either formula does an adequate job of transforming the SNPSM data into a usable predictive measure. Furthermore, with the exception of a 1% difference in favor of the NPSM model with the first dataset, the SNPSM model outperforms the NPSM model by 1-4% depending on the dataset considered.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>SNPSM Model</th>
<th>NPSM Model</th>
<th>Averaged Coefficient Formula</th>
<th>Weighted Coefficient Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>12%</td>
<td>13%</td>
<td>14%</td>
<td>12%</td>
</tr>
<tr>
<td>A1-A2</td>
<td>19%</td>
<td>15%</td>
<td>20%</td>
<td>18%</td>
</tr>
<tr>
<td>A1-A3</td>
<td>21%</td>
<td>20%</td>
<td>23%</td>
<td>21%</td>
</tr>
<tr>
<td>A1-A4</td>
<td>29%</td>
<td>26%</td>
<td>31%</td>
<td>31%</td>
</tr>
<tr>
<td>A1-A5</td>
<td>39%</td>
<td>37%</td>
<td>40%</td>
<td>42%</td>
</tr>
<tr>
<td>A1-A6</td>
<td>46%</td>
<td>45%</td>
<td>43%</td>
<td>46%</td>
</tr>
<tr>
<td>A1-A7</td>
<td>44%</td>
<td>41%</td>
<td>42%</td>
<td>44%</td>
</tr>
</tbody>
</table>

### 6.1.3 Evaluating Explanatory Power of SNPSM with Respect to Final Grades

In the prior section, we considered how data collected throughout the course could be used to predict performance on programming assignments. However, given the SNPSM's marked improvement in explanatory power with respect to final grade, we decided to perform an additional analysis using the same dataset with respect to student's final grade in the course.

We begin by investigating the variance explained by both NPSM and SNPSM given a single assignment's worth of data. We then progressively add in data from each additional assignment. This process yielded the same seven datasets (see Figure 39) used in prior chapters. When selecting predictor variables, we began with the same variables used in the predictive models for assignment average. This seemed reasonable, as the same two predictor variables (UU, NU) were significant in both assignment average and final grade when considering the entire dataset. However, initial analysis revealed that, with the exception of a single dataset for the variable RU, the other two variables (RU, RN) were not significant.
in any of the models. Hence, we decided to exclude both RU and RN from further analysis. This left us with a simpler model: UU, NU for NPSM and UU, NU, and Social Role for SNPSM.

Significance values for each NPSM variable as well as the total variance explained by the model is presented in Table 42. Significance values for SNPSM variables, as well as the total variance explained for each dataset, are presented in Table 43. Figure 48 provides a visual comparison between the two predictive models. In all cases, the SNPSM explains between 11 and 16 percent more variance than the standard NPSM model. Likewise, UU only becomes a significant predictor with larger datasets. In contrast, NU and Social Role (for SNPSM) are significant predictors regardless of the size of the dataset.

Using just the first dataset, the SNPSM explains an impressive 28% of the variance in final grades. In comparison, the NPSM model requires five assignments worth of data in order to achieve this level of explanatory power. In practical terms, this means that educators could use the SNPSM model to drive educational interventions very early in the semester—before students have a chance to become severely discouraged.

6.1.4 Deriving Predictive Measures

As was done in previous sections, we used the data presented in Table 42 and Table 43 to derive a predictive formula for determining a student's final grade in the course. Again, we derived two predictive formulae for each set: one based on averaged values and one based on a weighted average. The averaged formulae are as follows:

\[
\begin{align*}
NPSM \ Score &= 87.98 - (0.36 \times NU) - (0.36 \times UU) \\
SNPSM \ Score &= 74.53 + (4.11 \times SocialRole) - (0.33 \times NU) - (0.16 \times UU)
\end{align*}
\]

The weighted formulae are:

\[
\begin{align*}
NPSM \ Score &= 89.20 - (0.46 \times UU) - (0.38 \times NU) \\
SNPSM \ Score &= 74.66 + (4.15 \times SocialRole) - (0.33 \times NU) - (0.21 \times UU)
\end{align*}
\]
Table 42: NPSM Predictive Power and Coefficients for Final Grade using Seven Data Sets of Increasing Size (* = sig at p < 0.05)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Variance Explained</th>
<th>Constant</th>
<th>Unstandardized β Coefficients</th>
<th>Standardized β Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>UU</td>
<td>NU</td>
</tr>
<tr>
<td>A1</td>
<td>12%</td>
<td>84.97</td>
<td>-0.25</td>
<td>-0.23*</td>
</tr>
<tr>
<td>A1-A2</td>
<td>15%</td>
<td>85.40</td>
<td>-0.07</td>
<td>-0.28*</td>
</tr>
<tr>
<td>A1-A3</td>
<td>18%</td>
<td>85.12</td>
<td>-0.01</td>
<td>-0.38*</td>
</tr>
<tr>
<td>A1-A4</td>
<td>20%</td>
<td>85.97</td>
<td>-0.12</td>
<td>-0.43*</td>
</tr>
<tr>
<td>A1-A5</td>
<td>28%</td>
<td>90.24</td>
<td>-0.58*</td>
<td>-0.37*</td>
</tr>
<tr>
<td>A1-A6</td>
<td>37%</td>
<td>92.20</td>
<td>-0.71*</td>
<td>-0.42*</td>
</tr>
<tr>
<td>A1-A7</td>
<td>38%</td>
<td>91.96</td>
<td>-0.75*</td>
<td>-0.43*</td>
</tr>
<tr>
<td>Average</td>
<td>N/A</td>
<td>87.98</td>
<td>-0.36</td>
<td>-0.36</td>
</tr>
<tr>
<td>Weighted Average</td>
<td>N/A</td>
<td>89.20</td>
<td>-0.46</td>
<td>-0.38</td>
</tr>
</tbody>
</table>

Table 43: SNPSM Predictive Power and Coefficients for Final Grade using Seven Data Sets of Increasing Size (* = sig at p < 0.05)

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Variance Explained</th>
<th>Constant</th>
<th>Unstandardized β Coefficients</th>
<th>Standardized β Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>UU</td>
<td>NU</td>
</tr>
<tr>
<td>A1</td>
<td>28%</td>
<td>76.48</td>
<td>0.06</td>
<td>-0.29*</td>
</tr>
<tr>
<td>A1-A2</td>
<td>29%</td>
<td>74.76</td>
<td>0.02</td>
<td>-0.31*</td>
</tr>
<tr>
<td>A1-A3</td>
<td>33%</td>
<td>73.52</td>
<td>0.05</td>
<td>-0.33*</td>
</tr>
<tr>
<td>A1-A4</td>
<td>35%</td>
<td>73.45</td>
<td>-0.03</td>
<td>-0.34*</td>
</tr>
<tr>
<td>A1-A5</td>
<td>42%</td>
<td>67.11</td>
<td>-0.33</td>
<td>-0.32*</td>
</tr>
<tr>
<td>A1-A6</td>
<td>48%</td>
<td>78.32</td>
<td>-0.43*</td>
<td>-0.35*</td>
</tr>
<tr>
<td>A1-A7</td>
<td>49%</td>
<td>78.09</td>
<td>-0.46*</td>
<td>-0.36*</td>
</tr>
<tr>
<td>Average</td>
<td>N/A</td>
<td>74.53</td>
<td>-0.16</td>
<td>-0.33</td>
</tr>
<tr>
<td>Weighted Average</td>
<td>N/A</td>
<td>74.66</td>
<td>-0.21</td>
<td>-0.33</td>
</tr>
</tbody>
</table>

Figure 48: Final Grade Variance Explained by NPSM and SNPSM Models
Table 44 shows the amount of variance accounted for by all four formulae. As is the case in previous analysis, both the averaged and weighted formulae are equally appropriate for the SNPSM. However, for the NPSM, the averaged formula appears to provide a slightly better fit.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>NPSM Model</th>
<th>AVG NPSM</th>
<th>WGT NPSM</th>
<th>SNPSM Model</th>
<th>AVG SNPSM</th>
<th>WGT SNPSM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>12%</td>
<td>11%</td>
<td>09%</td>
<td>28%</td>
<td>28%</td>
<td>28%</td>
</tr>
<tr>
<td>A1-A2</td>
<td>15%</td>
<td>15%</td>
<td>11%</td>
<td>29%</td>
<td>29%</td>
<td>28%</td>
</tr>
<tr>
<td>A1-A3</td>
<td>18%</td>
<td>13%</td>
<td>12%</td>
<td>33%</td>
<td>33%</td>
<td>32%</td>
</tr>
<tr>
<td>A1-A4</td>
<td>20%</td>
<td>17%</td>
<td>16%</td>
<td>35%</td>
<td>36%</td>
<td>35%</td>
</tr>
<tr>
<td>A1-A5</td>
<td>28%</td>
<td>28%</td>
<td>28%</td>
<td>42%</td>
<td>42%</td>
<td>42%</td>
</tr>
<tr>
<td>A1-A6</td>
<td>37%</td>
<td>37%</td>
<td>38%</td>
<td>48%</td>
<td>47%</td>
<td>48%</td>
</tr>
<tr>
<td>A1-A7</td>
<td>38%</td>
<td>37%</td>
<td>38%</td>
<td>49%</td>
<td>48%</td>
<td>49%</td>
</tr>
</tbody>
</table>

### 6.2 Discussion

The purpose of the analysis conducted in this section was to quantify how a statistical model composed of both social and programming measures could be used to model students' classroom performance. To this end, we combined the Social Role measure explored in Chapters 3 and 4 with the NPSM developed in Chapter 5. Using the same methodology employed in prior chapters, we reran our analysis using this combined model. Incorporating Social Role into the NPSM (abbreviated SNPSM) resulted in an increased variance of up to 2% for assignment grades. However, the increase in variance accounted for in final grades increased up to 16%. Equally impressive, introducing Social Role as a predictor for final grades allowed us to account for 28% (vs 11% without) of the variance using just a single assignment's worth of data.

While it is difficult to compare statistical results across studies, the 49% of variance explained by the SNPSM exceeds the Watwin Score's 42% variance figure reported by Watson et al. (2014). Furthermore, the 28% variance explained after just a single assignment dwarfs the Watwin Score's 7% on roughly the same amount of data (see Watson et al., 2013). These results seem to reinforce Chapter 5's claim that modeling student behavior is most effective when it incorporates multiple, disparate factors of student behavior. In the future, we plan to develop and incorporate other social measures into our statistical modeling.
6.3 Exploring the Relationship between Social and Programming Behaviors

In the previous section, we explored how social behavior as defined by the Social Role measure might be incorporated into statistical models of a student's performance. While the results were positive, the analysis did not consider the interaction between coding and social behavior. To this end, we now present a follow-up study to investigate the relationship between these two factors.

6.3.1 Methodology

To investigate RQ2 ("How does coding behavior influence social behavior") and RQ3 ("How does social behavior influence coding behavior"), we again used the data collected during the spring 2014 semester offering of CS2 at Washington State University. We used the same content analysis coding scheme and sample used in Chapter 4. Recall that this sample included a sample of 36 randomly-selected students: 10 who received A's, 10 who received B's, 10 who received C's, four who received D's, and 2 who received F's. (The six students who received D's and F's represented all participating students who received those two grades.)

To perform this analysis, we first identified 2,352 instances in which the students in our sample were involved in some kind of social activity—a post or reply on the activity stream. Since our research questions were focused on interplay between social and programming behaviors, we opted to focus only on social activity to posts or replies that had to do with programming activities. This refinement yielded 461 such instances. Finally, in order to focus in even more, we reduced the sample further, so that it included only posts and replies that were part of a thread that was initiated by a programming question. This resulted in a final tally of 93 instances of social activity.

For each of these 93 instances, we recorded the following information:

1. Whether or not the question had any responses
2. Whether or not the question had any viable coding suggestions
3. Whether or not the author acknowledged the responses or suggestions
4. A code to categorize the type of programming question being asked
5. Whether or not the question related to the student's current build state
6. Whether or not a future build addresses the student's original question

While the content coding scheme used in Chapter 4 was sufficient for identifying this sample, we decided to refine the coding scheme so that we could better group related programming questions. To this end, we developed a new scheme for coding programming questions. This updated coding scheme is presented in Table 45. To verify the reliability of the new coding scheme, two researchers independently coded a 20% sample of the corpus (n = 19 posts), attaining an overall 88% agreement (0.83 kappa). Having established high levels of inter-rater reliability for the new coding scheme, a single researcher coded the remaining 76 posts.

Data points 5 and 6 in the above list also required further operationalization. In order to determine whether or not a question related to the student's current build state, we examined the build that came immediately before or immediately after the question was posed. In these builds, we required positive evidence connecting the question to code. For example, if a student inquired about substrings, his build must have contained code that uses, attempts to use, or infers the intended usage of functions related to substrings. In cases in which both builds occurred more than a day before or after a question was posed, we coded the question as not relating to the current build state.

In order to determine whether or not a future build addressed the student's question, we again looked for positive evidence demonstrating that the student's code had moved towards the correct solution. As we were interested in examining the impact of social interactions on programming behaviors, we limited our investigation to two days of compilation behavior following the student's last post in the discussion. Since some students compiled their code on an infrequent basis, we added the additional requirement to consider a minimum of five compilations per social interaction. In some cases, requiring a minimum of five compilations led to the investigation of compilations beyond our two-day window.
Because we insisted on concrete evidence of change, our operationalization of items 5 and 6 are likely conservative. For example, it would have been possible for students to have asked a related coding question before they started an actual code implementation. Likewise, it would have been possible for students to have made positive strides towards a correct solution outside of our two-day, five-build window. Thus, it is entirely possible that our conservative analysis approach failed to identify some relationships between programming and social behavior. This should be borne in mind in interpreting the analysis that follows.

**Table 45. Programming Question Coding Scheme**

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile</td>
<td>The question relates to an issue encountered during program compilation.</td>
<td>After I debug my code, I got this error, can anyone explain it to me? thank you Error 1 error LNK2019: unresolved external symbol […]</td>
</tr>
<tr>
<td>IDE</td>
<td>The question is related to the operation of Visual Studio</td>
<td>Okay, does anybody else consistently get the problem where cin and cout are underlined with red and VS gives you the error &quot;cout is ambiguous&quot;?</td>
</tr>
<tr>
<td>Implementation</td>
<td>The question is asking for tips on how to best implement an algorithm or function. This is often, but not always, related to the requirements of a given lab or homework.</td>
<td>If both the player and the computer draw the same type of hand (say two pairs), who wins? Are we expected to go off the rules on the wiki page or can we just say draw? Any help is appreciated! For the lab do we have to use any classes or can we just do the whole operation in main?</td>
</tr>
<tr>
<td>Language</td>
<td>The student asks a question about the C/C++ language or encounters a programming issue related to the misunderstanding of syntax.</td>
<td>Does anyone know if the strtok() keeps the old string or does it fully erase it?</td>
</tr>
<tr>
<td>Resources</td>
<td>The student is asking for external programming resources or tips.</td>
<td>What did you guys use to create your UML diagrams?</td>
</tr>
<tr>
<td>Runtime</td>
<td>The question is related to an issue encountered during the runtime execution of a student's code</td>
<td>Does anyone know what &quot;vector subscript out of range&quot; means and to fix it?</td>
</tr>
</tbody>
</table>
6.3.2 Results

We begin our analysis by performing a quantitative analysis on our dataset.

6.3.2.1 Quantitative Analysis

Table 46 presents the top-level categorical breakdown of the questions in our sample. We found the majority of questions to be related to language (26) and implementation (43). The least common questions were related to compile (4) and IDE (4) issues. The second column in Table 46 refines these numbers by considering only questions that were connected to the student's active coding solution. We see that questions related to runtime, compile, and language were most commonly associated with a student's active coding solution. Conversely, we see that none of the questions asking for resources was related to an active coding solution. This makes sense given that by definition, resource-type questions ask for external programming resources or tips.

Table 46: Frequency of question types posed by students

<table>
<thead>
<tr>
<th>Question Type</th>
<th>Frequency</th>
<th># Related to current code solution</th>
<th>Percent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile</td>
<td>4</td>
<td>3</td>
<td>75%</td>
</tr>
<tr>
<td>IDE</td>
<td>4</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>Implementation</td>
<td>43</td>
<td>27</td>
<td>63%</td>
</tr>
<tr>
<td>Language</td>
<td>26</td>
<td>19</td>
<td>73%</td>
</tr>
<tr>
<td>Resources</td>
<td>7</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>Runtime</td>
<td>9</td>
<td>8</td>
<td>89%</td>
</tr>
<tr>
<td>Total</td>
<td>93</td>
<td>59</td>
<td>63%</td>
</tr>
</tbody>
</table>

Figure 49 compares response rates and likelihood for future progress for all questions with only questions that relate to a student's active coding solution. We see similar rates of response, suggestion, and acknowledgement between the entire dataset and the subset of questions related to the active coding solution. Indeed, a chi-squared test could not find a significant difference in frequencies between groups ($\chi^2(2) = 0.14, p = 0.93$).

While Figure 49 indicates that most students are able to make noticeable progress towards a correct solution, it does not disambiguate between students who make progress on their own and students who receive a helpful suggestion via OSBIDE. Figure 50 teases this apart according to whether a student (a) did not receive a suggestion, (b) received a suggestion but did not acknowledge the response, or (c) received a
suggestion and acknowledged the response. We see a clear increase in success rates with only 42% of students making progress when a suggestion was not made versus a success rate of 85% when a post contains both a suggestion from another student and an acknowledgement from the author. A chi-squared test revealed a significant association between the type of feedback received and whether or not a future build demonstrated progress towards a correct solution ($\chi^2(2) = 7.80, p = 0.02$). A post-hoc z-test revealed a significant difference ($p < 0.05$) between students who did not receive a suggestion and those who received and acknowledged a suggestion. The middle condition—students who received a response but did not acknowledge the response—did not differ significantly from the other two categories.

![Figure 49: Percentage of questions that have responses, suggestions, acknowledgements, and demonstrate progress for all students (N=93) and only those whose question relates to an active coding document (N=59)](image)

![Figure 50: Percentage of future builds that demonstrate progress based on social feedback received](image)
We next consider the relationship between discussions anchored in code and course outcomes, in order to further explore whether talking about code was a general indicator of success within the classroom. We begin by examining the relationship between the number of code-centric discussions initiated by a student and both the student's final grade and assignment average. In both cases, we did not find a significant relationship between the amount of code-centric discussions initiated by a student and the student's final grade \( F(1,18) = 0.2, p = 0.66 \) or assignment average \( F(1,18) = 0.2, p = 0.66 \). Next, we compared students who post coding questions to those who do not. Again, we did not find a significant relationship between these groups and final grade \( F(1,105) = 0.03, p = 0.87 \) or assignment average \( F(1,86) < 0.01, p = 0.96 \).

Clearly, the data indicate that simply posting a question about code has no relationship to a student's overall class performance. However, it is possible that receiving help on a given assignment might positively impact the grade received for that assignment. To investigate this possibility, we examined the 23 instances in which a student asked a question, received a suggestion, and acknowledged the suggestion with a response. For each instance, we compared the grade the student received on that assignment to the average grade of students that did not receive help on the assignment. Figure 51 depicts this relationship. Out of these 23 observations, only three observations were below the class average. When taken as a whole, we see that the average score received by this group was, on average, 10% higher than the class average (Figure 52). A two-sample t test with equal variance not assumed found this difference to be statistically significant \( t(23) = 2.65, p < 0.01 \).

Next, we investigate the relationship between programming state, as defined by our NPSM model (see Chapter 5), and the type of coding question asked by students. Table 47 provides a matrix that breaks down questions asked by NPSM state.

Table 47 suggests some logical relationships between NPSM states and questions. For example, we see student in state DN (debugging a semantically incorrect program) ask a question about runtime behavior. Likewise, we find that students in state NU (compilation error) asked directly about compile questions and language questions, which are likely tied to compilation issues. While these associations are promising, we find the YU state (editing semantically correct, semantically unknown document) to dominate the matrix.
over half of all questions were asked in this state. We consider the implications of this finding later in the discussion subsection.

### 6.3.2.2 Vignettes

The prior section demonstrated a clear statistical relationship between online discussions of code and course outcomes. In this section, we consider a series of vignettes that capture the interplay between student's coding behavior and social interactions. In exploring these vignettes, we see how rich discussions centered around coding can positively impact a student's coding progress. Likewise, these vignettes give us insights of what might happen when these discussions fail to materialize.

![Figure 51: Grades of students who received help on assignment compared with those that did not for a given assignment](image1)

![Figure 52: Average score received by a student who asked for help on an assignment versus class average](image2)

**Table 47: NPSM States of Students Asking Questions**

<table>
<thead>
<tr>
<th>Question Code</th>
<th>--</th>
<th>UU</th>
<th>DU</th>
<th>DN</th>
<th>NU</th>
<th>NN</th>
<th>R/</th>
<th>R?</th>
<th>YU</th>
<th>YN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>IDE</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Implementation</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>Language</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>Resources</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>Runtime</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>51</td>
<td>12</td>
</tr>
</tbody>
</table>
Vignette 1: Multiple Inheritance

The concept of inheritance in object oriented programming has recently been introduced in class and James would like to use inheritance in his game of Battleship. James would like to build a subclass that inherits from multiple parents. He can get his program to compile when inheriting from a single parent, but cannot figure out the syntax for multiple inheritance. He asks for help with the issue on OSBIDE:

James: How do I make a class that is composed of two other classes? This is giving me an error:

    class Player : public Board : public Stats

Seeing James' question, Sharon provides a suggestion on how to accomplish multiple inheritance:

    Sharon: Never done it myself, but this might help... Try class Player : public Board, public Stats

James sees Sharon's question, modifies his class definition as suggested. His project compiles and he reports back his progress:

    James: That's it. Crisis averted!

Timothy, who is also toying with multiple inheritance, sees James' conversation and confirms Sharon's suggestion:

    Timothy: This works. I have for instance, 5 different boat classes, all of which are inherited from a mothership parent class...

Once again, James thanks Timothy for his confirmation. In the final post of the conversation, Steven, an upperclassman, offers a cautionary message regarding multiple inheritance:

---

3 All names used in these vignettes are fictitious, in order to protect the anonymity of the participants.

147
Steven: While you are all correct and multiple inheritance does work in C++. It can lead to a lot of errors, and stuff like the diamond problem…

Less than three hours after posing the question, James was back on track.

**Vignette 2: Understanding Requirements**

Jessica has started work on her homework assignment, but her unfamiliarity with the game of poker is preventing her from making progress on how to properly implement a required function. Unable to make headway, she uses OSBIDE to ask the class how about how the function should operate:

Jessica: In highCard(), it returns one card which has the highest rank and the highest suit, right? I just want to make sure since I am unfamiliar with [poker]

Stacie responds first:

Stacie: Your function wants to return the highest rank over the highest suit. However, if you had, say 2 jacks in your hand, it would return the Jack of Spades, before the Jack of hearts. But say you had an Ace no matter what the suit is it would return the ace over the jacks. Hope this makes sense.

Fred, who is also working on his poker assignment asks a clarifying question:

Fred: Isn’t high card just for a no hand combination? Meaning 2 jacks would make a pair, high card only returns in the instance of you have no hand right?

Barry responds to Fred:

Barry: It would only be relevant if you had a jack and the dealer had a jack. Suits are only there to be compared if there is a tie if both have the same high card …and flushes of course.
Fred replies to Barry:

Fred: I took the said suit function out, but I'm putting it back in [...] 

To close out the conversation, Jessica thanks the discussion participants:

Jessica: Thanks guys

Jessica continues to work diligently, and a day later, she has a working implementation of her function. However, not only did Jessica help herself, she also helped Fred, who might otherwise have been too shy to ask a question.

Vignette 3: Learning a new Concept

In the latest homework assignment, Beau is required to implement a basic factory pattern for creating different employee types. As this concept was recently introduced, Beau has no experience writing factories and is thoroughly confused. He uses OSBIDE to reach out for help:

Beau: Any ideas to write studentFromString, staffFromString, facultyFromString? How do those function help the fromString function?

Justin responds with an explanation of how the factory pattern works:

Justin: The fromString function determines whether the employee is student, staff, or faculty then calls studentFromString, staffFromString, facultyFromString respectively. In those functions it sets all the information it gets from the string then returns it back to the fromString function. The fromString function then returns that.

Apparently, other students are, like Beau, also struggling to understand how to implement a factory. Jessica writes:
Jessica: So a majority of the code should be in the Factory and not in main? […] What I'm having trouble with is bringing it back into main [partial code snippet]

Steven responds that in his case, the majority of his code is indeed inside the factory. James follows this with a code snippet of how he "brought it back into main":

Stephen: I just use: Employee *employ; and then in the while loop, just write: employee = factory.fromString(temp); cout<<employee->toString()<<endl; Hopefully, it can help you.

Sandwiched between the discussion between Justin, Jessica, and Steven, Beau thanks the students for their suggestions. At the time Beau originally posed his question, he had not written any factory-related code. Less than a day later, Beau had implemented a fully-operational factory.

Vignette 4: Unacknowledged Question #1

Like Beau from Vignette 3, Rachel is having trouble implementing a factory for her homework assignment. Also like Beau, Rachel decides to seek help on OSBIDE. Unfortunately, she accesses OSBIDE a day after Beau's discussion. Therefore, the help she seeks has fallen from the first page of the activity feed. Unaware of the previous discussion, Rachel makes her own post asking for implementation strategies:

Rachel: I'm not sure what I should do in employeeFactory class, anyone can give me some ideas? Thank you.

Unfortunately, Rachel's post does not receive the same attention as Beau's post; she receives no responses. Without help, Rachel has to go it alone. Over the course of the next two days (the maximum window of observation), Rachel makes progress towards correctly implementing a factory. However, at the end, she has still not implemented a fully working solution.
**Vignette 5: Unacknowledged Question #2**

Sean encounters an issue related to IFNDEF / DEFINE preprocessor directives in his code. For some reason, large chunks of his code are being categorized as an "inactive preprocessor block." Confused, he poses his question on OSBIDE:

Sean: So, in one of my header files I accidentally hit some key that made everything within that ifndef an "Inactive Preprocessor Block" (when I minimize it says that)... so, what key did I hit and how do I reactivate it?

Unfortunately, nobody responds to his post. In an effort to solve his problem, Sean alters his preprocessor directives in a way that fixes his immediate error, but also introduces a potential bug that may cause issues in the future.

**Vignette 6: Unacknowledged Response**

Jon would like to create an array of pointers to use in his homework implementation, but he cannot figure out how to properly initialize the data structure. Looking for guidance, Jon poses his question on OSBIDE:

Jon: I'm trying to use an Employee ** employee1 in my main function to hold all the employees in my .csv file. However there is no way of initializing that. So how would I hold an array of employees in main so I can access them whenever I want (i.e. for the paycheck)?

Tyler responds with a code snippet demonstrating how he had initialized his array of pointers:

Tyler: I did employee1 = new Employee*[100] to initialize my array of employees

Jon never responds. Inspecting Jon's code, one finds that he appears to have implemented a different solution strategy. Future builds of Jon's solution do not include code related to 2D array initialization.
6.3.3 Discussion

We begin the discussion by considering the results in the context of our original research questions.

6.3.3.1 How does coding behavior influence social behavior?

We can address this question from two perspectives. First, what is the relationship between a question and the student's most recent programming solution? Broken down by question type (refer to Table 46), we see that in all but a single case, at least half of all coding questions relate to the author's most recent programming solution at the time the question was posed.

We find that questions relating to language, compile, and runtime issues are most commonly associated with an active programming solution; 73%, 75%, and 89% of these types of questions relate to the author's most recent programming solution. This makes sense, as each of these question types are likely to arise as these issues are encountered by students. For example, it seems unlikely that a student would ask a random question relating to runtime behavior if he had not recently encountered such an error.

We also find that not a single question in which the author asked for coding resources was related to the student's active programming solution. Again, this makes sense because students who are looking for programming resources are probably still formulating the problem and have not yet transitioned to writing actual code.

Second, what is the relationship between the NPSM state (see Chapter 5) of the student at the time a question was posed and the type of question posed by the student? As indicated by Table 47, there might be a relationship between certain states and question types (e.g. DN and runtime). Unfortunately, the fact that the YU state dominates most question types prevents us from performing any sort of further statistical analysis. This implies that, at present, the NPSM is not sufficiently sensitive to provide insight into the present analysis.

It is possible that the NPSM's lack of sensitivity may be related to certain rules within the NPSM. For example, RU, RN, and R/ states revert back to YU after a 3 minute timeout. Perhaps we would see the proportion of questions relating to these states increase if we increase the timeout limit. Similarly, debug
states (DN, DU) revert back to YU or YN once the student exits debug mode. It seems probable that students ask a question related to these debug states on OSBIDE only after they exit debug mode. Such explanations may account for the prominence of the YU state within Table 47. Determining whether or not this is the case, whether some other factor inhibits the NPSM, or whether the NPSM is simply unsuitable for the present analysis is left for future work.

6.3.3.2 How does social behavior influence coding behavior?

As suggested by the vignettes, social activity can influence coding behavior. In cases in which there was much social activity, we found that question askers were likely to ultimately fix their issue. Furthermore, we saw that discussions often attracted students with similar issues. In this respect, we can see social activity as a mechanism for disseminating knowledge and creating bonds. The results presented in Chapter 4, where students using OSBIDE exhibited a significant increase in sense of community (the only cohort to do so), provide additional empirical support for this observation.

Our qualitative results seem to suggest that social behavior can have a significant influence on future coding behavior. When we examined conversations that contained a suggestion and an acknowledgement by the post’s author, we saw the likelihood of the author's future build making progress towards a correct solution increase significantly. Without a suggestion, only 42% of question askers' future compilations demonstrated evidence of progress. With an unacknowledged suggestion, this figure increased to 63%. Finally, this number increased to 85% when the author received and acknowledged a suggestion. A chi-squared test revealed a significant difference between students who did not receive a suggestion and those who received and acknowledged a suggestion ($\chi^2(2) = 7.80, p = 0.02$). This result implies that students who notice a helpful suggestion are likely to incorporate the suggestion into their own code, and that by doing so, they increase their chances of developing a correct solution.

In addition to helping students make positive solution progress, asking questions and receiving help also led to marked improvement in homework grades—at least for students who asked a question, received a suggestion, and acknowledged the suggestion. Of the 23 observations that met this criteria, we found that
20 of these assignments received a grade higher than the class average—a statistically-significant difference ($t(23) = 2.65, p < 0.01$). Thus, we discovered that asking for and receiving help on an assignment not only leads students towards a correct solution in the immediate sense; it also increases the likelihood of receiving a good grade on the assignment.

### 6.3.3.3 Implications for Pedagogical Tools

Having addressed our original research questions, we close by assessing the implications these results might have for future pedagogical tools. Given the positive association between social interaction and programming behavior we identified, one might conclude that providing a space for students to discuss coding problems should be a key feature of any pedagogical tool. Moreover, the results presented in this chapter provide guidance on how social discussions should be supported. As illustrated by our quantitative analysis, code-centric social interaction becomes more effective when a student receives a suggestion. The impact becomes significant when author acknowledges the suggestion. Therefore, it would make sense to ensure that the system specifically highlights unanswered questions. For example, a revised version of OSBIDE might have a section of the activity feed that is solely dedicated to unanswered questions. To ensure that those who asked for help actually read answers to their questions, an effective system would need to somehow bring potential solutions to the attention of question authors. To this end, the system might contain a notifications area that is updated whenever a new solution has been submitted to a question. Along these lines, it would be helpful to incorporate a mechanism for both marking a potential solution and for marking the "best" solution, similar to what is currently provided by StackOverflow (2012).

As highlighted by Vignettes 3 and 4, students may pose similar questions. Due to the nature of an activity feed, it seems likely that students are unaware that their current question might have already been asked and answered. Clearly, there is a need for students to be made aware of prior questions. This may be accomplished by providing an ability to search past question, perhaps by hashtags common on social media sites. Alternatively, the tool might recommend other posts based on the contents of the question (e.g. "The following posts may be related to your current question"). Another avenue might include leveraging the
nature of OSBIDE's IDE logging to relate conversations. For example, for a question related to a compiler error, OSBIDE might be able to automatically suggest a discussion that was created by another student under similar circumstances. These implementations will need to be explored further in future research.

6.4 Summary

In this chapter, we explored the relationship between students' programming behaviors and social activity from two perspectives. First, we explored the potential contribution social activity can make to statistical models of student performance based on IDE activity. We found that incorporating some measure of social activity into the model increases the amount of explained variance in homework scores by up to 4%. For a model constructed to predict final grades, a model containing a measure of social activity was able account for up to 16% more variance than a model that focuses solely on IDE behaviors. In the future, we plan to expand upon this work by developing a more fine-grained measure than the Social Role measure considered in this chapter. We expect such a measure to allow us to account for even more variance in a statistical model.

In the second part of this chapter, we explored the relationship between programming activities and code-centric activity feed discussions. We began with a quantitative analysis demonstrating that questions posed by students are likely to be related to a current coding issue. We also discovered that students who receive and acknowledge a suggestion to their question were significantly more likely to fix their issue. Furthermore, these students were likely to turn in assignments whose scores were significantly higher than the class average. These differences were then illuminated through a series of vignettes that demonstrate how social interaction can influence eventual success, how these conversations can positively impact other students, and what happens when such a discussion does not occur. In future work, we plan to broaden our analysis by expanding our selection criteria to include questions that occur as a sub-thread of another post. Additionally, we would like to explore how code-centric discussions impact students other than the author. This includes both students who participate in the discussion and students who merely "lurk."
CHAPTER 7

CONCLUSION

This dissertation was motivated by the observation that, relative to other disciplines, computer science has abnormally high attrition. In surveying the state of computing education, we hypothesized that this high attrition was related to two independent factors: a deficiency in the sense of community among classmates and an inability for instructors to identify struggling students early enough in a course to intervene and make a meaningful correction in a student's class performance.

As a means to investigate the lack of community among students, we created a social programming environment that integrated common social networking activities directly into students' development environments. Survey results present in Chapter 4 indicate that providing an activity feed had a positive impact on students' sense of community. Furthermore, we discovered that regular social participation within the SPE was positively correlated with course outcomes. In Chapter 6, we also learned that providing an activity stream allows students to obtain the exact help they need directly from other students. The fact that asking for and providing help are core activities in communities of practice (Wenger, 1998) provides further evidence that the SPE can indeed be used to foster online communities for computing education courses.

In an attempt to better identify struggling students, we used the SPE collect data on students' programming processes and online social behavior. In combining both programming and online social behaviors, we developed a statistical measure of student success. On our dataset, this holistic model outperformed prior statistical measures (e.g. Jadud, 2006b; Watson et al., 2013) as well a measure we developed that was based solely on programming behaviors.

In this final chapter, we recap our dissertation's primary contributions, discuss the limitations of our work, explore our work's implications for the research community, and identify future research directions.

7.1 Contributions

We now discuss three primary contributions made by this dissertation.
7.1.1 Social Programming Environments

We have introduced the concept of a social programming environment and presented an exploration of its design space. Central to the SPE is the activity stream. In Chapter 3, we confirmed that activity streams can be successfully used by students to discuss programming topics. In Chapter 4, we presented evidence that class-wide usage of an SPE leads to an increased sense of community among students. Furthermore, Chapter 4 also provided evidence that regular participation in the SPE's social community is positively correlated with course outcomes.

7.1.2 Predictive Modeling

This dissertation makes two contributions to the growing body of research related to predictive models of academic success within a computing course.

First, we performed a replication study of two existing predictive models: the Error Quotient (Jadud, 2006b) and the Watwin Score (Watson et al., 2013, 2014). Confirming other research (e.g. Ahadi et al., 2015), our study indicated that these measures' predictive capabilities vary widely based on setting. In our particular configuration of programming language (C++), development environment (Visual Studio), and course (CS2), these measures performed much worse than what has been reported in the literature.

Our second contribution relates to the modeling of student behaviors. Prior research has focused on relating low-level characteristics such as keystrokes (Ahadi et al., 2015), typing behavior (Leinonen et al., 2016), and compilation behavior (Jadud, 2006b; Watson et al., 2014) with performance. Rather than continuing this line of work, this dissertation instead constructed a theoretical model of students' programming activities. To this end, we developed the Programming State Model, which describes students' problem solving behavior by placing a student in one of eleven possible programming states. From the PSM, we derived a predictive measure that outperformed both the Error Quotient and Watwin Score on our dataset. Additionally, we used the PSM as a means to identify common patterns of programming behavior. This inquiry led to the identification of significant differences in programming behaviors between A, B, and C-level students.
In the final chapter of this dissertation, we extended the PSM by incorporating an additional measure derived from a student's online social participation. This extended model proved to be more accurate at predicting students' course grades.

7.1.3 Exploring the Relationship between SPE Social Participation and Programming Behaviors

A final contribution is the first-ever detailed investigation into the interplay between a students' online social behavior and their programming activities. We were able to perform this investigation because we are the first, to our knowledge, to be able access to the interleaving of students’ programming activities and their online posts within a common learning (programming) environment. First, we investigated the state of students programming solutions when they asked questions through our SPE. We discovered that students do indeed tend to ask programming questions that relate to their code. Next, we examined how social participation within our SPE influences future programming behaviors. We discovered that students who ask for help on a question, receive a suggestion, and acknowledge the help are significantly more likely to demonstrate positive progress in their programming solution. Put more plainly, our research appears to suggest that asking questions and receiving help strongly influences a student's ability to succeed. While this finding may seem obvious, we are the first to provide concrete empirical evidence of it within the context of computing education.

7.2 Limitations

While we have tried to minimize the amount of confounds in our research, the nature of quasi-experimental research prevents us from completely controlling our experiments. In this section, we discuss potential threats to validity using the methodology outlined by Shadish, Cook, & Campbell (2002).

7.2.1 Statistical Conclusion Validity

This dissertation likely suffers from an unreliable treatment implementation, meaning that not all students interacted with the SPE in equal proportion. Indeed, our results indicate that usage varied
significantly between students. However, in order to assure a minimum level of usage among all students, we required students to install the SPE and to make a minimum number of posts and replies in the system. Yet, in doing so, we lowered our study's internal validity.

7.2.2 Internal Validity

As mentioned in the prior section, requiring students to install and use the SPE introduces a potential confound when we draw relationships between SPE usage and academic performance. A critic might argue that any relationship involving SPE usage is merely a proxy for a student's motivation to succeed in the class. To address this criticism, we included prior CS1 grade in our analysis, our proxy for motivation, as a covariate.

Of the 140 students enrolled in the primary course considered by this dissertation, 11 students withdrew and 21 did not sign our study's informed consent. Therefore, our analysis was conducted using 77% of the possible student body. Given these figures, attrition is likely to be a threat to our internal validity. As such, it could be argued that any observable gains in outcome variables are simply a result of lower-performing students dropping out of the course. Unfortunately, investigating the impact of attrition would require additional studies that are beyond the scope of a single dissertation.

7.2.3 External Validity

This work suffers from threats to external validity. In particular, there exists a data collection bias as the instructor of most of the courses considered in this dissertation were taught by the primary author. Furthermore, as our data suggests, existing predictive measures have demonstrated a lack of generalizability. Therefore, it is remains a possibility that the statistics developed throughout this dissertation will not generalize beyond our sample of WSU students enrolled the spring 2014 offering of CS2, the C++ language, or Visual Studio. The only way to refute this limitation is to perform additional replication studies with a different student population.
7.3  Implications and Future Work

We now consider the implications of our contributions for computing pedagogy and future research, including our specific plans for future work.

7.3.1  Implications for the Broader Research Community

As we argued in the introduction, computing education needs to become more social. The studies conducted and data presented in this dissertation provide evidence for this argument. Social networks, such as the one provided to students through an SPE, allow students to interact with their classmates and instructor on a more consistent basis, and to pose questions relevant to their learning. Our research indicates that providing such an online social environment likely leads to increases in students' sense of community. Furthermore, our research provides guidance on how to obtain maximum benefit from an online community. As discussed in Chapter 6, students who engage in online discussions of problem solving are more likely to make positive strides in their work. Therefore, educators should encourage students to engage in a cyclical pattern of asking for help, offering help to others, and acknowledging help that was received. Indeed, we believe that an effective online system should naturally support this cycle.

In addition to providing insights into online learning communities, this dissertation contributes to the literature on learning analytics for computing education. Our statistical model of programming behaviors suggests that future researchers would do well to look beyond basic programming behaviors (e.g. keystrokes) and instead develop more intricate models of student learning. While we consider social participation in our model, we believe that future models might benefit equally from the incorporation of other student characteristics.

7.3.2  Future Work

We offer the following avenues for addressing remaining open questions and expanding the research presented in this dissertation.

7.3.2.1  Addressing Research Limitations
In order to increase our confidence in the claims made in this dissertation, we would need to replicate our analysis under differing conditions. In particular, we would like to investigate:

- Students' SPE usage in the absence of a posting requirement
- The impact of an SPE on students' sense of community when taught by a different instructor
- The reliability of the PSM under differing instructors, student populations (e.g. CS1, CS3, etc.), programming languages, development environments, and academic institutions.
- The reliability of the Social Role (Chapters 3 & 4) as a predictor across instructors, student populations, programming languages, development environments, and academic institutions.

### 7.3.2.2 Refining the SPE

As discussed in Chapter 4, some SPE features that we believe would aid students in their problem solving process went unused. We hypothesize that this is likely due to either poor visibility of the features, a poor usability of the features, or the fact that such features were not actually useful to students. We would like to employ a user-centered design process, including detailed laboratory studies of feature usage, so that we may better understand why these features failed to gain traction with students.

In future work, we would also like to further explore the design space of social programming environments. In Chapter 6, we discuss a common problem with activity stream-based systems: namely, that posts frequently drop out of students' consciousness once a post drops off of the front page of the activity stream. How might we best promote and highlight popular and useful discussions? How can we best design an environment that promotes the question and answer cycle previously discussed? For example, it might be worthwhile to create a dedicated space both for unanswered questions and for answered questions.

While OSBIDE awarded students points based on activity, the feature and its impact on learning went unexplored in this dissertation. In future work, we would like to examine more closely how public recognition, such as achievements and points, influences student learning. To this end, we would like to
apply techniques popular in behavioral economics (e.g. Thaler & Sunstein, 2009) in order to improve learning outcomes in our SPE.

Lastly, we would like to overhaul the user interface of the SPE. The version of the SPE presented in this dissertation can best be described as spartan. We believe that it would be worthwhile to hire a professional graphical designer to overhaul the user interface of our SPE so that it more closely matches the look and feel of popular social networking websites. The recent integration of OSBIDE into a new version of OSBLE called OSBLE+ has already begun to make such changes (see Figure 53).

![Figure 53: Proposed OSBLE+ Interface](image)

### 7.3.2.3 Further Investigation of the Programming State Model

While the Programming State Model provided new insights into students' programming behaviors and demonstrated the potential of holistic models, we left some interesting questions unanswered. Recall that analysis of the NPSM in Chapters 5 and 6 revealed what might be a hard ceiling in the explanatory power of the model. We would like to run a replication study under similar circumstances to see if a ceiling is again observed. Similarly, we were hesitant to assign meaning to the differences the TPSM observed in programming behaviors between A, B, and C students. In the future, we would like to perform ground truth
analysis on the PSM so that we can more confidently assign meaning to these observed behavioral differences. In addition to addressing these questions, we would like to consider how the PSM might be further extended.

In future research, we would like to investigate additional states and factors might influence the PSM's predictive powers. At present, the PSM only describes programming behaviors. Yet, given the social data collected by the SPE, we wonder how incorporating social participation into the PSM states might affect the model. In doing so, we might discover, for example, that students who are in the NU state and ask questions transition to a more productive state (e.g. YU) more quickly than students who don't ask questions. Likewise, we wonder if we can add more descriptive power to the PSM by splitting existing states based on other factors. For example, the work by Carter and Prasun (2010) might allow us to better categorize editing states based on programming difficulties that are not currently detectable by the PSM.

Lastly, we would like to investigate the PSM's usefulness in machine learning algorithms. Given the success observed using simple measures such as keystrokes (e.g. Leinonen et al., 2016), we wonder if our holistic model might serve as a better basis for prediction.

7.3.2.4 Developing Pedagogical Implementations

Lastly, we would like to begin applying the discoveries presented in this chapter in ways that improve computer science pedagogy. To this end, we have started to design a dashboard that presents learning analytics to students and instructors (see Olivares, 2015). We believe that such dashboards can be used to increase an instructor's class awareness. For example, a dashboard might inform the instructor when a low percentage of students are successfully able to utilize a new concept (e.g. loops) or when a student had submitted an assignment or test case. On the student's end, we could leverage learning analytics to inform students on their own processes (i.e. metacognition, "you spent 50% of your time writing function X") and increase their peripheral awareness (e.g. 5 other students are working on their homework). Furthermore, learning analytics could serve as the basis a "nudges" for improving programming behaviors (see Thaler &
Sunstein, 2009). For example, we could nudge students to encourage students to ask a question, make a structural change in their code, or debug their code.

7.4 SPEs and the Future of Computing Education

Social Programming Environments provide a clear path for improving the quality of computing education. SPEs promote true communities of practice in which students ask for and provide help to other. Indeed, participation within an SPE's community of practice is positively correlated with course outcomes. Furthermore, the rich data collected from students' interactions within the SPE show great potential for improving both student and instructor awareness. Whereas other innovative pedagogical interventions have had their widespread adoption hindered due to differences in institution, department, or pedagogical approach, the SPE's lack of reliance on such factors make it equally suitable for nearly every setting. Ultimately, we envision a day in which all computing courses will utilize an SPE.
APPENDICES
APPENDIX A

CONTENT CODING MANUAL

In this section, we present the content coding manual used in Chapters 3, 4, and 7 to categorize online social participation.

Speaker Code

Use the initials of the person who made the post (e.g., Scott Carter = SC).

Speaker Type

This can be either Instructor or Student.

Content Categories

There exist three general content categories for coding posts. Always code a post into the category that is the best fit, given the focus and/or emphasis of the post. However, some posts may not have a clear emphasis or focus. In such cases in which a post could be coded into multiple categories, always code into the category with the highest priority.

Notes:

- Forward inference may be necessary in order to accurately code a post.
- A post that clearly refers to a previous post either because it uses an indexical term (e.g. "this", "that", "here", "there"), or because the context of the post clearly relates it to previous post (e.g., "Thanks!" or "No problem!"), should be coded into the same category as the previous post unless the post introduces new content that could be coded into a higher-priority category.

Below, we have listed the categories in decreasing order of priority.

Reflection
Any post made by a student that has been tagged as a reflection essay on their homework assignment. These posts usually include "#reflection" in the content.

**Code**

Any post that includes code or asks code-related questions (including questions about UML or other code-related notations) should be coded into the Code category. Examples include, "How do I use an enum?" or, "Why do I keep getting this error?" All items coded with this category must also be categorized into one of the following subcategories:

**Conceptual**

A post is conceptual if it does not directly deal with a specific compile or runtime event that is explicitly or implicitly referenced in the post. General code questions fall into this category, such as, "How do I write a sorting algorithm?" or, "What function should I use to accomplish this task?" In addition, specific questions or comments about coding resources found on the web and elsewhere would fall into this category.

In general, there are at four types of conceptual issues:

- **Conceptual Design**: "How do I write this procedure?" "How do I write a program to do x?"
- **Conceptual Syntax**: "What's the syntax for creating a new list in Python?" or "Here is the syntax for creating a new list".
- **Conceptual Behavior**: "What does this piece of code do?" "How does this piece of code work?"
- **Conceptual Requirements**: "What are the requirements of this assignment?" "Should I account for the case in which x is negative?"

All of these types of conceptual issues should be coded as "Conceptual"; there's no need to distinguish between them for the purposes of our coding system.
**Compile**

The post is in the service of dealing with issues that arise from a specific compile action or event (present or future). These should be pretty easy to spot because students will often describe the compiler error in their post.

**Runtime**

The post is in the service of dealing with issues that arise from a specific runtime action or event (present or future). An example runtime issue would be something like, "When I call foo(), why does it always return the value 3?", "I keep getting stack overflow messages", or "my code ran flawlessly."

**IDE**

The post refers to an issue related to the IDE or other procedural detail related to the IDE such as where files are located or how to perform specific IDE functions. This does not include information or issues related to (a) downloading the IDE, obtaining a license to install the IDE, or installing the IDE (see Course->Software) or (b) OSBIDE (see Course->Software below). *Note: Visual Studio has an IDE feature called Intellisense, which tries to predict compilation errors before they occur. Posts related to Intellisense error messages should be coded as Compile, not IDE.*

Some posts may ask code questions that appear to be related to a particular category, but in fact are really asking about another category. For example, suppose a student asks a question that appears to be related to compilation, but later discourse reveals his issue actually relates to a runtime issue. The post should be recorded as runtime in consideration of that later discourse.

**Course**

The course category is the most general. It should be used to code any posts that are relevant to the course, but that do not specifically address computer programming. We define the following subcategories to further distinguish the content of different types of posts in this category. Note that
these categories are listed in decreasing order of priority. In cases in which a given segment could be coded into multiple categories, always code the segment into the category with the highest priority.

**Grades**

This subcategory includes posts that refer to student grades and the course grading system. This includes discussions of how a particular assignment, quiz, lab, or other deliverable is graded. Note that questions that refer to how to access grades in an LMS should be coded as Course->Software.

**Assignment**

This subcategory includes posts that explicitly reference issues with, the scheduling of, the content of, the requirements of, or the electronic posting of, a particular course lecture, assignment, lab, or exam, *so long as the post is not related to*

- *programming* (often, code and coding resources are discussed in lecture or lab, and references to such discussions should be coded as either Code->Conceptual or Course->Resources, but see first note below).
- *programming requirements* (discussions of programming requirements bridge the gap between a textual problem description and a code solution; hence, they should be coded as Code->Conceptual).
- *the electronic handling of course documents or assignment submissions* (such posts should be coded as Course->Software).
- *how the assignment is graded, what is required to get a certain grade, or how much an assignment is worth* (such posts should be coded as Course->Grades).

Notes:
• Posts that ask about, or describe, the content of quizzes or tests (not assignments) will often refer to programming content. However, if the purpose of the post is to determine, or to identify, what that content will be—as opposed to discussing the content conceptually—then the post should be coded as Course->Assignment, and not as Code. Here are a couple of examples: "Will Friday's Quiz cover inheritance?" "Things to know for Friday's quiz: * defining characteristics of arrays, vectors and linked lists * how add/removal works in each." Be sure to use a Code Modifier code in such cases, even though the posts will be coded as Course->Assignment.

• Posts about assignment submissions may be coded into this category as long as answers to such posts do not require specific knowledge of course software. For example, answers to a question like "Did you get my assignment?" (addressed to a course instructor) would not require knowledge of course software, and hence should be coded as Course->Assignment. In contrast, answers to questions like "How do I submit my assignment" would require knowledge of the software being used in the course, and hence should be coded as Course->Software.

• Posts that discuss the (non-programming) requirements of an assignment (e.g., "How many posts and replies do we need to make each week?") without explicitly linking those requirements to specific grades (e.g., "What do I need to do to get an A?") should be coded as Course->Assignment.

Software

Posts in this subcategory have to do with issues and experiences related to the use of software used for course purposes, with the exception of the IDE when the focus is on how to use the IDE itself (see Code-IDE above). These include, for example, posts that (a) refer
to profile pictures, (b) describe issues encountered while registering for, logging into, or using the learning management or social media environment used in the course (including OSBIDE); (c) relate to questions or issues pertaining to submitting assignment solutions, or confirming such submission. Note that posts that ask about whether course content has been posted, or confirm that it has been posted, should be coded as Course->Assignment.

**Resources**

Posts in this subcategory relate to study resources, tutoring resources, instructional resources (including schedules, syllabi, physical facilities, computers, study guides, the instructors themselves and their office hours), and coding resources available to students in the course. Examples include "What websites can I use to get programming help?" "I've uploaded the coding style guide to be used in this course." "Joe is available for tutoring on Wednesdays from 5:30-6:30 pm." "When are your office hours?" "Where can I get the course syllabus?"

**Note 1:** If the post asks a specific (conceptual) question about a coding resource, the post should probably be coded as Code->Conceptual.

**Note 2:** If a post refers to a software issue related to the posting of a resource (e.g., "I think the link is broken"), then the post should be coded as Course->Software.

**Other**

Posts in this subcategory relate to the course in general, but do not refer specifically to grades, assignments, or software.

**Coordination**

Posts in this category coordinate present or future collaboration or help between instructors and/or students—and not necessarily for the purposes of the current class. Examples, "I'm in the
Is there anyone interested in getting together to work?”, “Email me your file and I'll take a look at it by class,” “Did that help?” or “Who wants to study for calculus exam with me?”

**Other**

This is a catch-all category for posts that we cannot deem to fall into any of the above categories, based either on content or context. This includes, for example,

- greetings (e.g., "Hi," "How's it going")
- posts related to events at the university (e.g., the hack-a-thon or a programming event)
- posts related to future courses (e.g., "Are the books in CptS 223 really needed?")
- posts that have nothing to do with the course (e.g., "Last weekend was crazy.")

**Category Modifiers**

These modifiers can be used in association with any top-level category. They are used to further characterize the content of each post in terms of its communicative role in the discussion. More than one modifier can apply to a given post. In such cases, list modifiers based on their precedence, as shown in the following list, which is arranged from highest to lowest precedence.

**Update**

The author (usually the instructor or TA) is providing an update to the class. Updates are usually related to course or assignment posts. Note: if the instructor replies to his own update, code it only as a REPLY (not UPDATE). Also note that if a student is providing updated information on a meeting place/time for a study session or the like (coded into "Coordination"), then you can use this modifier.

**Reply**

The author is responding to someone else's comment or is expanding or clarifying one of his/her own posts. Note that REPLlYs must respond to a previous post, which may or may not have
occurred in the same thread. If the post has an ID with a decimal (e.g. 1.1), it will almost always be a reply.

**Note**: A post that *clearly* refers to another post using an indexical term (e.g. "this", "that", "here", "there") should be coded as a Reply. Not also that, per the rule described above, such a post should be coded into the same category as the post to which it refers *unless* the post introduces new content that could be coded into a higher-priority category.

**Question**

The author asks a question within the post, or phrases a statement with a question mark in order to indicate that s/he is really asking a question. Most posts coded as Question will also be coded as Reply.

**Answer**

The author is attempting to answer, partially answer, or to provide information that might prove helpful in answering, a previously-posed question *that was explicitly coded as Question.* (usually, but not necessarily, in the same thread). Alternatively, the author is offering an answer or solution that he or she thinks may be of general interest, since the author encountered the problem or had the question previously. Note that attempts at coordination, e.g., "Let's meet at 9" or "Email me your file" are not suggestions.

**Note**: There are cases in which posts are statements that could be construed as questions or requests for help, but aren't coded as Question because they are not technically questions—i.e., they do not meet the definition of Question. Thus, if a given post appears to address a previous post, but that previous post was *not* coded as Question, then do not code the post as Answer.

How to handle replies that address these in some way?

**Gratitude or Acknowledgement**
The author is expressing gratitude for a previously posted suggestion or reply. Alternatively, the author is acknowledging that a previously posted suggestion worked or was helpful.

**Agreement**

The author agrees with a previous post, "Me too!", "I agree". Note that posts that contain "Yes," "Ya, and "Yeah" and other variants of "yes" should not be coded as Agreement, since they do not contain an explicit statement of agreement.

**Disagreement**

The author disagrees with a previous post. E.g. "I disagree", "That's not how to solve the problem"

**Code Usage**

The following modifiers can be used in association with any post coded as "Code." Only one modifier can apply to a given post.

**Code**

The post includes at least **two** lines of executable code with or without semicolons, as opposed to one or more non-executable lines of code that are used for illustrating a syntax error. Example:

- also I still haven't overcome stringt_to_array and stringt_from_array. just not sure how to convert properly: int i;

  int size = sizeof(string);

  char string_arr[BUFFER_SIZE];

  for(i = 0; i < size; i++)

  {

    string_arr[i] = string[i];
Pseudocode

The post describes some sort of algorithmic process with at least two computational statements. This includes both code solutions and strategies for writing code. Examples:

- (code solution example) "Don't use strcat at all, in combine all you do is connect the pointers. so do stringt_to_end, to make your pointer point to the end of the first string, then do first->next = second. and second->previous = first. they've now been connected. just make sure you called stringt_ rewind(second)"

- (strategy for writing code) "enum auto assigns the words you use to an integer value. I think by default it starts with the first string you use as 0. You make it start at 1 by saying addition = 1, then leave enum to label the rest."

- (description of how code works) "When I scan into a multi-dimensional array using fgets, it will scan all the letters, but if the next word is shorter than the previous, then the later characters are left in a new array."

A post is not considered to have pseudo code when it merely mentions a code segment, or contains only one computational statement. Examples:

- "Have you tried using strlen()?"

- "Simply swap the first and second items in the array." (This is one computational statement in pseudocode, even though it must be written in three statements in many programming languages.)

Bits

The post includes any of the following:
• at least one non-executable code segment, function, class name, variable name, library name, or compiler message fragment (hence it's not Code)

• two or more keywords (e.g. int, while).

• One line of executable code (hence it's not Code)

• One line of pseudocode (hence it's not Pseudocode).

Examples include the following:

• "Use strlen()," (strlen() is a function)

• "I called foo(bar) and got a result of 5" (foo(bar) is a non-executable statement)

• "I got compiler error CX432343: Invalid Operation". (Compiler message)

• "Reset your pixel count at the end of your WHILE loop that converts the current line to the array of points." (One Pseudocode statement)

• Int x = y + 1; (One Code statement)

When trying to distinguishing between Pseudocode, Code, and Bits, think about the following:

• Pseudocode usually, but not always, contains Bits and/or up to one line of executable code with an explanatory story.

• A post coded as Pseudocode will usually also be coded as Answer, since it responds to a coding question of some sort.

• If a given statement contains just one executable statement or one line of pseudocode, code it as Bits.

• If a post contains at least two executable lines of code mixed in with non-executable lines of code, it should be coded into the Code category, which has higher priority.

HelpAcknowledged
• Any post coded with a GRATTITUDE modifier should have an associated "HelpAcknowledged" code of 1 if, and only if, the post was an expression of thanks or acknowledgement that a previously-posted question or issue was resolved. If, in contrast, the post was an expression of gratitude for something that was not explicitly asked for, then the corresponding "HelpAcknowledged" column entry should be left blank. Examples of acceptable acknowledgement:

• "Ok, thanks"
• "K"
• "That helped"]
APPENDIX B

ONLINE SURVEY

The following survey was used in Chapters 3 and 4 to collect students' demographic data, self-efficacy, and sense of community.

General Demographic Data
1. Name
2. WSU ID Number
3. Age
4. Gender
5. Ethnicity
6. Class Standing
7. Are you presently a computer science major?
8. (If no) Do you plan to become a computer science major?
9. (If no) How likely are you to switch your major to computer science?
10. (If yes) How likely are you to switch to a major that is not computer science?
11. (If yes) How likely are you to switch to a major that is not computer science?
12. What is your reason for taking CptS 121?
13. How likely are you to enroll in the next computer science in this sequence (CptS 122)?

Adapted C++ Self-Efficacy Survey
Rate your confidence in doing the following C programming related tasks using a scale of 1 (not at all confident) to 7 (absolutely confident). If a specific term or task is totally unfamiliar to you, please mark 1.

1. Write syntactically correct C statements.
2. Understand the language structure of C and the usage of the reserved words.
3. Write logically correct blocks of code using C.
4. Write a C program that displays a greetings message.

5. Write a C program that computes the average of three numbers.

6. Write a C program that computes the average of any given number of numbers.

7. Use built-in functions that are available in the various C libraries.

8. Build my own C libraries.

9. Write a small C program given a small problem that is familiar to me.

10. Write a reasonably sized C program that can solve a problem that is only vaguely familiar to me.

11. Write a long and complex C program to solve any given problem as long as the specifications are clearly defined.

12. Organize and design my program in a modular manner.

13. Understand the procedural programming paradigm.

14. Identify the data types in the problem domain and declare, define, and use them.

15. Make use of a pre-written function, given a clearly labeled declaration of the function.

16. Make use of a data structure that is already defined, given a clearly labeled declaration of the data structure.

17. Debug (correct all the errors) a long and complex program that I had written, and make it work.

18. Comprehend a long, complex multi-file program.

19. Complete a programming project if someone showed me how to solve the problem first.

20. Complete a programming project if I had only the language reference manual for help.

21. Complete a programming project if I could call someone for help if I got stuck.

22. Complete a programming project once someone else helped me get started.

23. Complete a programming project if I had a lot of time to complete the program.

24. Complete a programming project if I had just the built-in help facility for assistance.

25. Find ways of overcoming the problem if I got stuck at a point while working on a programming project.
26. Come up with a suitable strategy for a given programming project in a short time.

27. Manage my time efficiently if I had a pressing deadline on a programming project.

28. Mentally trace through the execution of a long, complex, multi-file program given to me.

29. Rewrite lengthy confusing portions of code to be more readable and clear.

30. Find a way to concentrate on my program, even when there were many distractions around me.

31. Find ways of motivating myself to program, even if the problem area was of no interest to me.

32. Write a program that someone else could comprehend and add features to at a later date.

Classroom Community Scale
Below, you will see a series of statements concerning a specific course or program you are presently taking or have recently completed. Read each statement carefully and select the statement that comes closest to indicate how you feel about the course or program. There are no correct or incorrect responses. If you neither agree nor disagree with a statement or are uncertain, select the neutral area. Do not spend too much time on any one statement, but give the response that seems to describe how you feel. Please respond to all items.

1. I feel that students in this course care about each other
2. I feel that I am encouraged to ask questions
3. I feel connected to others in this course
4. I feel that it is hard to get help when I have a question
5. I do not feel a spirit of community
6. I feel that I receive timely feedback
7. I feel that this course is like a family
8. I feel uneasy exposing gaps in my understanding
9. I feel isolated in this course
10. I feel reluctant to speak openly
11. I trust others in this course
12. I feel that this course results in only modest learning
13. I feel that I can rely on others in this course
14. I feel that other students do not help me learn
15. I feel that members of this course depend on me
16. I feel that I am given ample opportunities to learn
17. I feel uncertain about others in this course
18. I feel that my educational needs are not being met
19. I feel confident that others will support me
20. I feel that this course does not promote a desire to learn

**Motivated Strategies for Learning Questionnaire**

The following questions ask about your motivation for and attitudes about this class. Remember there are no right or wrong answers, just answer as accurately as possible. Use the scale below to answer the questions. If you think the statement is very true of you, circle 7; if a statement is not at all true of you, circle 1. If the statement is more or less true of you, find the number between 1 and 7 that best describes you.

1. I believe I will receive an excellent grade in this class.
2. I'm certain I can understand the most difficult material presented in the readings for this course.
3. I'm confident I can learn the basic concepts taught in this course.
4. I'm confident I can understand the most complex material presented by the instructor in this course.
5. I'm confident I can do an excellent job on the assignments and tests in this course.
6. I expect to do well in this class.
7. I'm certain I can master the skills being taught in this class.
8. Considering the difficulty of this course, the teacher, and my skills, I think I will do well in this class.
The following questions ask about your learning strategies and study skills for this class. Again, there are no right or wrong answers. Answer the questions about how you study in this class as accurately as possible. Use the same scale to answer the remaining questions. If you think the statement is very true of you, circle 7; if a statement is not at all true of you, circle 1. If the statement is more or less true of you, find the number between 1 and 7 that best describes you.

1. When studying for this course, I often try to explain the material to a classmate or friend.
2. I try to work with other students from this class to complete the course assignments.
3. When studying for this course, I often set aside time to discuss course material with a group of students from the class.


193


Pears, A., Seidman, S., Malmi, L., Manila, L., Adams, E., Bennedsen, J., … Patterson, J. (2007). A survey of literature on the teaching of introductory programming. In Working group reports on ITiCSE on Innovation and technology in computer science education (pp. 204–223). Dundee, Scotland: ACM.


