HOW STUDENTS MEASURE UP: AN ASSESSMENT INSTRUMENT FOR INTRODUCTORY COMPUTER SCIENCE

by

Adrienne Decker

May 1, 2007

A dissertation submitted to the
Faculty of the Graduate School of
The State University of New York at Buffalo
In partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Department of Computer Science and Engineering

Acknowledgments

To borrow from (and modify) Julie Andrews in the Sound of Music, "when a door closes, a window opens." That is exactly what happened one day in December 2003 when I walked into Bill Rapaport's office looking for a new advisor. I was pleased when Bill agreed to take me on as a challenge. Soon I discovered that the window I had jumped through would change everything. My experience working on this dissertation was very different than the dreary picture of drudgery and turmoil that I had in my mind. In fact, Bill taught me more about the process of a dissertation, writing, researching, and expressing my ideas than I ever thought possible. I had often heard rather half-heartedly that a dissertation was an "apprenticeship," but I can truly say that my experience has been just that. I am extremely grateful for all of the help and guidance he has given me. I can only hope that I can be as good a mentor to someone else one day **as** he has been to me.

My committee member Carl Alphonce, has provided me with valuable insight while on my committee and also as a collaborator on several projects. If not for those other projects, I might have completed this dissertation sooner. Our collaborations have been incredibly rewarding and I am most excitedly looking forward to our next steps now that this task is behind me.

The rest of my committee, Tom Shuell and Ken Regan, have been invaluable as well. Tom has provided insight and guidance for many of the education issues that have played an important part in this endeavor. Tom's assistance on the "other side" of my dissertation has been reassuring and supportive, and I am grateful for his help, even now in his retirement. Ken's insights and comments have spurned me on my way.

Chris Egert is probably the most inspiring person I have ever met. He has forgotten more about systems than I will probably ever know. He can inspire his students effortlessly. He has inspired me to do great things and to forge ahead and complete this dissertation throughout all my doubt and uncertainty. While I could not achieve the awe-inspiring size of his dissertation (in lines of code, nor length, nor the number of references), I can see his influence in how I approach my work, and my teaching. If I try hard enough, I know there is much more I can learn from him if he'll let me. Furthermore, his friendship has meant so much to me over the past seven years, and while I miss his presence at UB daily, I can take solace in the fact that he is only a phone call or short car ride away whenever I need anything. I'll continue to use the phone and car often.

Phil Ventura gave me the inspiration for this work. Had he not completed his dissertation so that I could find a problem with it, I wouldn't be here today. He also gave me my start as a TA and then further guidance as I was the instructor for the first time. I learned many things from him about teaching, educational research, statistics, and even some miscellaneous computer science topics along the way. I am grateful for all our friendship provided to me during this time.

This dissertation would not have come together without the critique and criticisms given about the exam that I created. Every person that undertook this challenge provided valuable insight on the exam and the questions contained within it. Thanks to Kevin Bierre, Stuart Shapiro, and Bina Ramamurthy. Extra special thanks to Alan Hunt, who not only reviewed the exam, but also agreed to allow his students to participate and provide me with data for analysis. I must also thank from the school of education, Scott Meier, whose class provided me with important tools for completing this dissertation. I have received a tremendous amount of support from the faculty and staff in our department. I need to especially mention and thank Peter Scott, Bharat Jayaraman, and Helene Kershner for their support and guidance.

It is cliché to say that I owe it all to the support of my family. However, I have been blessed with a biological family, a matrimonial family, and an adopted family.

My parents have given me so much throughout my life that I owe more than I can ever repay to them. They blessed me with love and support. They gave me my strength and my drive, which is the only way I could have achieved this goal. I was also blessed to be incredibly close to my Grandma and Grandpa Janusz growing up. When I went off to college and then on to graduate school, they were always there, smiling proudly. Grandma left me in the middle of this dissertation, which in some ways only made the journey that much more important. Now, she has another accomplishment to smile about even if she can no longer share that smile with me. I know she is smiling now.

My other parents, the Deckers have been in my life for many years. They have stood up and cheered as loudly as my own parents at all my accomplishments and have always been there as additional support whenever I needed it. I can only be grateful that they have welcomed me as part of their family and that we continue to share a close bond.

I have a brother, Dan, and a sister (by marriage) Nicole. They are both younger than me and at every minute have lived up to the younger sibling image. However, there are no other people that I know that would be before them in line to lend a hand or be there for me if I needed it. I thank them for that.

When I became a Decker, I also became a Rieth and I can say that I am proud to be both. In fact, I'm not so sure that the Rieths remember that there was a time when I wasn't a Rieth. I've relied on Rieth help and friendship in so many ways and I can not thank the Rieths enough for always being there. There are too many Rieths to name individually, but I want to speak to a few special Rieths. The Snees have provided a place to stay for a week or so for the last several summers as well as incredible conversation and support during this process. The Cervis made sure I had the wedding of my dreams and moral support for life outside of school. My favorite cousin Shannon has given me lots of laughs and can always say just the right things to let me know that she is thinking of me and proud of what I've done. I will never be able to repay her for that.

Most people are grateful for the set of parents they are born with. I am lucky to also have a great set of parents that I received upon marriage. However, even before that, I had a third set of parents that “adopted” me and my husband back when we were in high school. The Maslona’s (Carolyn & Gerald) always gave everything of themselves to their own children and decided along the way that I was someone else worthy of that attention. I learned so much from Carolyn about dealing with people, dealing with students, and dealing within a school that I still have yet to process it all. She left quite a legacy in her retirement with many of her students. I can only hope that I can do that as well. While I know both are extremely proud and happy for me at this time, I can only see the expressions of happiness from Carolyn because Jerry was stolen from us all too soon. I know that he is watching and smiling and I can only smile when I think of him.

I’ve adopted many undergraduate teaching assistants throughout this process (Daniel Britt, Sara (Haydanek) Britt, Mark Zorn, Clark Dever, Mark Jensen, Christopher Kozlowski, Jim Perrin, Keith Stabins, and Kyle Savage). They have been recruited to perform excellently in the classroom as well as to complete several other projects along the way. I have forged a bond with each one of them and they have forged a bond among themselves. This bond was so strong that they collaborated to nominate me for the Milton Plesur award and I didn’t even know anything about it. I am forever grateful for their support and look forward to celebrating with them now that this mission is accomplished.

I need to single out two of the UTA crowd for special mention, Benjamin Robboy and Michael Kozelsky. Luckily for me, Ben broke his ankle in the Spring 2006 semester and was able to do some data entry, grading, and number crunching for me while he could do nothing else. Also, many trips to Dairy Queen last summer helped to make the writing of this dissertation more bearable if not a little more costly on the calories. It seems like Mike has always been around asking how my dissertation is going. Sometimes I appreciated it, and sometimes not, but in any case, I knew he always was trying to be supportive. His insights into students and learning have not only helped me in this project, but in other projects as well. Even though Ben is gone and Mike is leaving me this semester, their impact on this dissertation and my teaching career has been huge and will not soon be forgotten.

No one can really say whether we adopted Brandy or she adopted us. In either case, it doesn't matter. Who can resist a constantly smiling face with a large wet tongue? Brandy has always been there to offer her best advice and a helpful paw about the dissertation. While her advice has been amazingly silent, it has been incredibly reassuring and comforting.

Lastly, but not in any way the least, I owe a huge amount of thanks for the completion of this effort to my husband, Eric. He has frequently expressed his support in my endeavor to finish this dissertation. I couldn't have done this without that support and I am forever grateful that he has agreed to be by my side through this and all our other adventures. Now that this adventure is over, we can begin out next adventure, raising a family.

For Carolyn Maslona O'Rourke, who expected good, wanted better, but always got my best.

Table of Contents

ACKNOWLEDGMENTS ..... **III**

LIST OF TABLES ..... **XVII**

LIST OF FIGURES ..... **XXI**

ABSTRACT ..... **XXIII**

CHAPTER 1 INTRODUCTION **1**

1.1 COMPUTING CURRICULA 1
1.2 PROBLEM STATEMENT 3
1.3 MOTIVATION 6
1.4 CONTRIBUTIONS AND SIGNIFICANCE OF THE DISSERTATION 10
1.5 OUTLINE OF DISSERTATION 11

CHAPTER 2 BACKGROUND **13**

2.1 METHODOLOGIES IN THE INTRODUCTORY CURRICULUM 13
2.1.1 Courses that speak to this generation 14
2.1.2 Approaches to CS1-CS2 using collaborative techniques 16
2.1.3 Approaches relying on paradigm 18
2.2 PREDICTORS RESEARCH 18
2.3 EDUCATIONAL OBJECTIVES AND OUTCOMES ASSESSMENT 23
2.4 ASSESSMENT OF PROGRAMMING SKILL FOR CS1 25
2.4.1 "The" Study (or at least the one everyone recognizes because of its failure) 25
2.4.2 Critical Eye to Assessment Practices 26
2.5 STUDY OF PERFORMANCE IN NON-MAJORS COURSE 26
2.6 ANALYSIS OF PUBLISHED ASSESSMENT INSTRUMENTS 27
2.6.1 Advanced Placement Exam 28
2.6.1.1 Reliability of the AP Exam 30
2.6.1.2 Validity of the AP Exam 31
2.6.2 Graduate Record Exam Subject Test in Computer Science 33
2.6.2.1 Reliability of the GRE Subject Test in Computer Science 35
2.6.2.2 Validity of the GRE Subject Test in Computer Science 35
2.6.3 ETS Major Field Test in Computer Science 36
2.6.3.1 Reliability of the ETS Major Field Test in Computer Science 38
2.6.3.2 Validity Estimates for the ETS Major Field Test in Computer Science 38
2.7 ANALYSIS OF AP EXAM DATA 38
2.7.1.1 Discussion 39
2.8 CONCLUSION 41

CHAPTER 3 ANALYSIS OF THE CC2001 COMPUTER SCIENCE VOLUME **43**

3.1 SUMMARY OF CC2001 43
3.1.1 Structure of CC2001 43
3.1.2 CC2001 Sections Important to this Dissertation 44
3.1.3 Chapter 5: Overview of the CS Body of Knowledge 44
3.1.4 Chapter 7: Introductory Courses 45

3.1.4.1 Programming-first Approaches....46
3.1.4.2 Advantages and Disadvantages of Programming-First Approaches....47
3.1.4.3 Non-programming-first Approaches....50
3.1.4.4 Concepts across All Approaches....52
3.1.5 Appendix A: CS Body of Knowledge and Appendix B: Course Descriptions....55
3.1.6 The rest of CC2001: Chapters 1 – 4, 6, and 8 – 13....56
3.2 ANALYSIS OF THE PROGRAMMING-FIRST APPROACHES TO THE INTRODUCTORY CURRICULUM. 61
3.2.1 Two- or Three-Semester Sequence....62
3.2.2 Justification for Programming-First....62
3.2.3 Intersection of Topics for CS1....63
3.2.3.1 Knowledge Area Analysis....64
3.2.3.2 Knowledge Unit Analysis....65
3.2.3.3 Analysis of Knowledge Units in Intersection....67
3.2.3.4 Conclusions about CS1 intersection....69
3.2.4 Intersection of Topics for CS1 and CS2....70
3.2.4.1 Knowledge Area Analysis....70
3.2.4.2 Knowledge Unit Analysis....71
3.2.4.3 Analysis of Knowledge Units in the Intersection....73
3.2.4.4 Analysis of Topics from the Knowledge Unit Intersection....74
3.2.4.5 Analysis of Hours Covered by Each Approach for each Knowledge Unit....85
3.2.4.6 Problems with Simply “Reading” the Syllabi....87
3.2.5 Resolution of Discrepancies....89
3.2.5.1 Reasons for Inconsistencies....89
3.2.5.2 All topics should be covered, not all were indicated....89
3.2.5.3 Topics should be covered, none or one were indicated....92
3.2.5.4 All approaches should have topics covered, but only two of three do....93
3.2.5.5 All approaches should have topics covered, but only one of three do....96
3.2.5.6 Non-uniform topical coverage across approaches....97
3.2.6 Revised Intersection of Knowledge Unit Topical Coverage....101
3.2.7 Comparison of the Current Intersection to CC2001 Chapter 7....110
3.2.8 Topics Included in some, but not All Programming-first Approaches....118
3.3 CONCLUSION....121

CHAPTER 4 REFINING THE TOPIC LIST....125

4.1 INTRODUCTION....125
4.2 TOPICS REMOVED....126
4.2.1 Topics Eliminated because of Time Constraints....127
4.2.1.1 Programming process....127
4.2.1.2 Concepts underlying the programming process....131
4.2.1.3 Exploring different aspects of programming....132
4.2.2 Topics Eliminated because of Deeper Coverage in Advanced Courses....133
4.2.3 Topics Eliminated because of Difficulty in Determining Material Coverage....134
4.2.4 Records....135
4.3 TOPICS REMAINING....136
4.4 CONCLUSION....136

CHAPTER 5 LEARNING OBJECTIVES....139

5.1 MINING CC2001 FOR LEARNING OBJECTIVES....139
5.2 LEARNING OBJECTIVES FROM PROGRAMMING FUNDAMENTALS....140
5.2.1 PF1. Fundamental Programming Constructs Learning Objectives....140
5.2.2 PF3. Fundamental Data Structures Learning Objectives....142
5.2.3 PF4. Recursion Learning Objectives....146
5.3 LEARNING OBJECTIVES FROM ALGORITHMS AND COMPLEXITY....147

5.3.1 AL1. Basic Algorithm Analysis Learning Objectives ... 147
5.3.2 AL3. Fundamental Computing Algorithms Learning Objectives ... 149
5.4 LEARNING OBJECTIVES FROM PROGRAMMING LANGUAGES ... 153
5.4.1 PL4. Declarations and Types Learning Objectives ... 153
5.4.2 PL5. Abstraction Mechanisms Learning Objectives ... 154
5.4.3 PL6. Object-oriented Programming Learning Objectives ... 156
5.5 CONCLUSION ... 158

CHAPTER 6 CREATION AND CRITIQUE OF EXAM ... 163

6.1 INTRODUCTION ... 163
6.2 CREATING QUESTIONS ... 163
6.3 STRUCTURE OF EXAM QUESTION GROUPS ... 168
6.3.1 Basic Syntax Questions (Group 1) ... 168
6.3.2 Fundamentals and API Programming (Group 2) ... 172
6.3.2.1 Functions and parameter passing ... 172
6.3.2.2 Arithmetic and logical expressions ... 174
6.3.2.3 Expressions and Assignment ... 175
6.3.3 Types, Declaration Models, and Parameter Passing (Group 3) ... 181
6.3.4 Data Structures (Group 4) ... 189
6.3.5 Recursion (Group 5) ... 199
6.3.6 Searching and Sorting Algorithms, and Algorithm Analysis (Group 6) ... 203
6.3.7 Object-Oriented Programming (Group 7) ... 205
6.4 CRITIQUE OF THE EXAM ... 211
6.5 CONCLUSION ... 213
6.6 CODING OF QUESTIONS ON EXAM ... 213

CHAPTER 7 EXAM ADMINISTRATION AND GRADING ... 217

7.1 GENERAL EXAM ADMINISTRATION GUIDELINES ... 217
7.2 GRADING PROCEDURE DEVELOPMENT ... 220
7.2.1 Multiple Choice Questions ... 221
7.2.1.1 Questions with Only One Answer ... 221
7.2.1.2 Questions with More than One Answer ... 221
7.2.2 Non-Multiple Choice Questions ... 222
7.2.2.1 Objective Free-Response Questions – One Answer ... 222
7.2.2.2 Objective Free-Response Questions – Complex Answer ... 223
7.2.2.3 Subjective Free-Response Questions ... 223
7.2.3 Weighting of Questions ... 224
7.2.3.1 Special Cases ... 226
7.2.4 Partial Credit (The Triage Theory of Grading) ... 228
7.3 STUDY DESIGN ... 230
7.3.1 Research Questions ... 230
7.3.2 Subjects ... 231
7.3.3 Study Protocol ... 232
7.3.4 Exam Grading for Study Participants ... 234
7.4 RATING THE RATERS ... 235
7.4.1 Questions Double Graded to Ensure Rater Consistency ... 236
7.4.2 Discussion of Rating the Raters ... 238
7.5 RECOMMENDATIONS FOR GRADING ... 240
7.5.1 Two Raters for Subjective Questions ... 240
7.5.2 Grading Simultaneously ... 241
7.5.3 Grading Anonymous Tests ... 242
7.6 CONCLUSION ... 243

CHAPTER 8 EXPERIMENTAL RESULTS AND ANALYSIS.......... 245

8.1 OVERALL EXAM STATISTICS 245
8.2 TIME 246
8.2.1 Time to Complete 246
8.2.2 Correlation with Exam Score 248
8.2.3 Analysis of students who took the full three hours to complete exam 249
8.2.3.1 Statistical Results 250
8.2.3.2 Analysis of Results 252
8.2.3.3 Additional Statistical Results & Analysis 252
8.2.3.4 Conclusions about Students who Took Three Hours to Complete 256
8.3 RELIABILITY 256
8.4 DEMOGRAPHIC INFORMATION 257
8.4.1 Gender 258
8.4.1.1 Statistical Results 258
8.4.1.2 Analysis of Results 258
8.4.2 Age 259
8.4.2.1 Statistical Results 260
8.4.2.2 Analysis of Results 261
8.4.3 Year in School 261
8.4.3.1 Statistical Results 261
8.4.3.2 Analysis of Results 262
8.4.4 Major 262
8.4.4.1 Statistical Results 263
8.4.4.2 Analysis of Results 264
8.4.5 How Courses Were Taken 265
8.4.5.1 Students Who Took Courses at Other Institutions 265
8.4.5.2 Statistical Results 265
8.4.5.3 Analysis of Results 267
8.4.6 Repeaters 267
8.4.6.1 Statistical Results 267
8.4.6.2 Analysis of Results 269
8.4.7 Previous Programming Experience 270
8.4.7.1 Statistical Results 270
8.4.7.2 Analysis of Results 272
8.4.8 First Programming Language 272
8.4.8.1 Statistical Results 273
8.4.8.2 Analysis of Results 273
8.5 GRADES IN CS1 AND CS2 (INCLUDING EXAM SCORE) 274
8.5.1.1 Statistical Results 274
8.5.1.2 Analysis of Results 278
8.6 GRADES IN CS1 AND CS2 (EXAM SCORE REMOVED) 278
8.6.1.1 Statistical Results 279
8.6.1.2 Analysis of Results 281

CHAPTER 9 DISCUSSION 283

9.1 DISCUSSION OF EXAM CREATION PROCESS 283
9.2 DISCUSSION OF ANALYSIS OF EXAM 284
9.2.1 Students who chose not to participate in study 287
9.2.1.1 Overall enrollment 287
9.2.1.2 Gender 288
9.2.1.3 Year in School 288
9.2.1.4 Declared Major 289
9.2.1.5 Grade in course 289
9.2.1.6 Conclusions about Participants 291

9.3 FUTURE WORK 292
9.3.1 Additional Student Data 292
9.3.2 Continuation of Predictors Research 293
9.3.3 Testing of curricular changes 293
9.3.4 Trends and Longitudinal Research 294
9.3.5 Multiple Languages and Multiple Forms 295
9.3.6 Multi-Institutional Analysis 296
9.3.7 Updates for Future Curricula 297

REFERENCES 299

APPENDIX A EXAM QUESTIONS 307

APPENDIX B GRADING GUIDELINE FOR EXAM 349

APPENDIX C REVIEWER QUESTIONNAIRE 373

APPENDIX D DEMOGRAPHIC QUESTIONNAIRE 375

APPENDIX E ANALYSIS OF RATERS OF EXAM 381

Question 5 381
Question 9 382
Question 10 385
Question 11 386
Question 12 387
Question 23 389
Questions 57 and 58 391
Question 59 394
Question 101 396
Question 102 398
Question 103 400

List of Tables

Table 3-1: Figure 7-1 of CC2001 describing the concepts that should be covered in an introductory curriculum ____ *53*

Table 3-2: Figure 7-2 from CC2001 knowledge units and topics that are covered by all six introductory tracks ____ *54*

Table 3-3: Knowledge Area Coverage for Programming-first CS1 courses ____ *64*

Table 3-4: Programming Fundamentals Knowledge Unit Coverage for CS1 ____ *65*

Table 3-5: Algorithms and Complexity Knowledge Unit Coverage in CS1 ____ *66*

Table 3-6: Programming Languages Knowledge Unit Coverage in CS1 ____ *66*

Table 3-7: Social and Professional Issues Knowledge Unit Coverage in CS1 ____ *66*

Table 3-8: Software Engineering Knowledge Unit Coverage in CS1 ____ *67*

Table 3-9: Percentages of Knowledge Units Covered by Intersection ____ *68*

Table 3-10: Percentages of Total Course Lecture Hours Covered by Intersection ____ *69*

Table 3-11: Knowledge Area Coverage for Programming-first CS1 & CS2 ____ *70*

Table 3-12: Programming Fundamentals Knowledge Unit Coverage for CS1 and CS2 ____ *71*

Table 3-13: Algorithms and Complexity Knowledge Unit Coverage in CS1 and CS2 ____ *71*

Table 3-14: Programming Languages Knowledge Unit Coverage in CS1 and CS2 ____ *72*

Table 3-15: Social and Professional Issues Knowledge Unit Coverage in CS1 and CS2 ____ *72*

Table 3-16: Software Engineering Knowledge Unit Coverage in CS1 and CS2 ____ *72*

Table 3-17: Percentages of Knowledge Units Covered by Intersection ____ *73*

Table 3-18: Percentages of Total Course Lecture Hours Covered by Intersection ____ *74*

Table 3-19: PF1. Fundamental Programming Constructs topics covered in programming-first CS1-CS2 *75*

Table 3-20: PF2. Algorithms and Problem-Solving topics covered in programming-first CS1-CS2 ____ *76*

Table 3-21: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2 ____ *76*

Table 3-22: PF4. Recursion topics covered in programming-first CS1-CS2 ____ *76*

Table 3-23:AL1. Basic Algorithmic Analysis topics covered in programming-first CS1-CS2 ____ *77*

Table 3-24: AL3. Fundamental Computing Algorithms topics covered in programming-first CS1-CS2 __ *77*

Table 3-25: AL5. Basic Computability topics covered in programming-first CS1-CS2 ____ *77*

Table 3-26: PL1. Overview of Programming Languages topics covered in programming-first CS1-CS2 _ *78*

Table 3-27: PL2. Virtual Machines topics covered in programming-first CS1-CS2 ____ *78*

Table 3-28: PL4. Declarations and Types topics covered in programming-first CS1-CS2 ____ *78*

Table 3-29: PL5. Abstraction Mechanisms topics covered in programming-first CS1-CS2 ____ 78
Table 3-30: PL6. Object-oriented Programming topics covered in programming-first CS1-CS2 ____ 79
Table 3-31: SP1. History of Computing topics covered in programming-first CS1-CS2 ____ 79
Table 3-32: SE1. Software Design topics covered in programming-first CS1-CS2 ____ 79
Table 3-33: SE2. Using APIs topics covered in programming-first CS1-CS2 ____ 79
Table 3-34: SE3. Software Tools and Environments topics covered in programming-first CS1-CS2 ____ 80
Table 3-35: SE5. Software Requirements and Specifications Constructs topics covered in programming-first CS1-CS2 ____ 80
Table 3-36: SE6. Software Validation topics covered in programming-first CS1-CS2 ____ 80
Table 3-37: Topics covered by all three approaches to CS1-CS2 ____ 81
Table 3-38: Topics covered by two of three approaches to CS1-CS2 ____ 82
Table 3-39: Topics covered by one of three approaches to CS1-CS2 ____ 83
Table 3-40: Topics covered by none of the three approaches to CS1-CS2 ____ 84
Table 3-41: Hours devoted to each knowledge unit for programming-first CS1-CS2 ____ 86
Table 3-42: PF1. Fundamental Programming Constructs topics covered in programming-first CS1-CS2102
Table 3-43: PF2. Algorithms and Problem-Solving topics covered in programming-first CS1-CS2 ____ 102
Table 3-44: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2 ____ 102
Table 3-45: PF4. Recursion topics covered in programming-first CS1-CS2 ____ 103
Table 3-46:AL1. Basic Algorithmic Analysis topics covered in programming-first CS1-CS2 ____ 103
Table 3-47: AL3. Fundamental Computing Algorithms topics covered in programming-first CS1-CS2 _ 103
Table 3-48: AL5. Basic Computability topics covered in programming-first CS1-CS2 ____ 104
Table 3-49: PL1. Overview of Programming Languages topics covered in programming-first CS1-CS2 104
Table 3-50: PL2. Virtual Machines topics covered in programming-first CS1-CS2 ____ 104
Table 3-51: PL4. Declarations and Types topics covered in programming-first CS1-CS2 ____ 104
Table 3-52: PL5. Abstraction Mechanisms topics covered in programming-first CS1-CS2 ____ 105
Table 3-53: PL6. Object-oriented Programming topics covered in programming-first CS1-CS2 ____ 105
Table 3-54: SP1. History of Computing topics covered in programming-first CS1-CS2 ____ 105
Table 3-55: SE1. Software Design topics covered in programming-first CS1-CS2 ____ 105
Table 3-56: SE2. Using APIs topics covered in programming-first CS1-CS2 ____ 106
Table 3-57: SE3. Software Tools and Environments topics covered in programming-first CS1-CS2 ____ 106
Table 3-58: SE5. Software Requirements and Specifications Constructs topics covered in programming-first CS1-CS2 ____ 106
Table 3-59: SE6. Software Validation topics covered in programming-first CS1-CS2 ____ 106
Table 3-60: Topics covered by all three programming-first approaches to CS1-CS2 ____ 109

Table 3-61: Topics covered by all two of three programming-first approaches to CS1-CS2 ___________ 109
Table 3-62: Topics covered by one of the three programming-first approaches to CS1-CS2___________ 109
Table 3-63: Topics covered by none of the three programming-first approaches to CS1-CS2__________ 110
Table 3-64: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2________ 114
Table 3-65:AL1. Basic Algorithmic Analysis topics covered in programming-first CS1-CS2 __________ 115
Table 3-66: AL3. Fundamental Computing Algorithms topics covered in programming-first CS1-CS2 _ 116
Table 3-67: SE1. Software Design topics covered in programming-first CS1-CS2 __________________ 117
Table 3-68: Topics covered only by imperative-first and functional-first CS1 & CS2 ________________ 119
Table 3-69: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2________ 120
Table 3-70: SE1. Software Design topics covered in programming-first CS1-CS2 __________________ 120
Table 3-71: Topics covered only by imperative-first CS1 & CS2 _________________________________ 121
Table 3-72: Topics covered only by functional-first CS1 & CS2 _________________________________ 121
Table 3-73: Final List of Intersection Topics ____ 124
Table 4-1: Revised List of Topics ____ 137
Table 5-1: Comparison of old and revised learning objectives for PF1. Fundamental Programming Constructs ____ 142
Table 5-2: Comparison of old and revised learning objectives for PF3. Fundamental Data Structures _ 145
Table 5-3: Comparison of old and revised learning objectives for PF1. Fundamental Programming Constructs ____ 147
Table 5-4: Comparison of old and revised learning objectives for AL1. Basic Algorithmic Analysis ___ 148
Table 5-5: Comparison of old and revised learning objectives for AL3. Fundamental Computing Algorithms ____ 152
Table 5-6: Comparison of old and revised learning objectives for PF1. Fundamental Programming Constructs ____ 154
Table 5-7: Comparison of old and revised learning objectives for PL5. Abstractions Mechanisms_____ 156
Table 5-8: Comparison of old and revised learning objectives for PL6. Object-oriented Programming _ 158
Table 5-9: Final List of Learning Objectives ____ 161
Table 6-1: Topics from knowledge units included in each group _______________________________ 167
Table 6-2: Categorization of Questions on Exam ____ 216
Table 7-1: Discrepancies by Question ____ 237
Table 7-2: Number of Discrepancies per Exam ____ 238
Table 8-1: Time to Complete Exam____ 247
Table 8-2: t-test for time to complete exam____ 251
Table 8-3: Levene's Test for Equality of Means for time to complete exam _________________________ 251

Table 8-4: t-test (unequal variances) for time to complete 251
Table 8-5: Mann-Whitney test for time to complete 252
Table 8-6: Conversion of Letter Grades to 4.0 Scale 253
Table 8-7: t-test for CSE 115 overall course grades 254
Table 8-8: t-test for CSE 116 overall course grade 254
Table 8-9: t-test for recalculated CSE 116 overall course grades 255
Table 8-10: t-test for averaged CSE 115 and CSE 116 overall course grades 256
Table 8-11: t-test for Gender 258
Table 8-12: Age Ranges of Participants 260
Table 8-13: t-test for Age 260
Table 8-14: t-test for Year in School 262
Table 8-15: t-test for Major (Computer Science or Computer Engineering vs. Other Majors) 263
Table 8-16: t-test for Major (Computer Science Majors vs. Computer Engineering Majors) 264
Table 8-17: t-test for Taking CS1-CS2 in consecutive semesters 266
Table 8-18: t-test for Taking CS1-CS2 in traditional academic year 266
Table 8-19: t-test for Repeaters (Students who failed CS1 vs. those who did not) 268
Table 8-20: t-test for Repeaters (Students who failed CS2 vs. those who did not) 268
Table 8-21: t-test for Repeaters (Students who failed CS1 and/or CS2 vs. those who did not) 269
Table 8-22: t-test for Prior Programming Experience 271
Table 8-23: t-test for Prior Programming (Prior Java programming) 271
Table 8-24: t-test for Prior Programming (C-derived languages) 272
Table 8-25: t-test for First Language (Java vs. not Java) 273
Table 9-1: Course grade breakdown for all CSE 116 students 290
Table 9-2: Course grade breakdown for CSE 116 students who elected not to particpate in the study 290

List of Figures

Figure 8-1: Histogram for Time to Complete Exam _____ 248
Figure 8-2: Plot of Total Points Earned versus Time Finished_____ 249
Figure 8-3: Plot of Points Earned on Exam vs. CSE 115 Overall Course Grade_____ 275
Figure 8-4: Plot of Points Earned on Exam vs. CSE 116 Overall Course Grade_____ 276
Figure 8-5: Plot of Points Earned on Exam vs. Averaged CSE 115 & CSE 116 Overall Course Grade _ 277
Figure 8-6: Plot of Points Earned on Exam vs. Revised CSE 116 Overall Course Grade _____ 279
Figure 8-7: Plot of Points Earned on Exam vs. Averaged CSE 115 & CSE 116 Overall Course Grade _ 280

Abstract

This dissertation presents an assessment instrument specifically designed for programming-first introductory sequences in computer science as given in Computing Curricula 2001: Computer Science Volume. The first-year computer science course has been the focus of many recent innovations and many recent debates in the computer science curriculum. There is significant disagreement as to effective methodology in the first year of computing, and there has been no shortage of ideas as to what predicts student success in the first year of the computing curriculum. However, most investigations into predictors of success lack an appropriately validated assessment instrument to support or refute their findings. This is presumably due to the fact that there are very few validated assessment instruments available for assessing student performance in the first year of computing instruction. The instrument presented here is not designed to test particular language constructs, but rather the underlying principles of the first year of computing instruction. It has been administered to students at the end of their first year of an introductory computer science curriculum. Data needed for analysis of the instrument for reliability and validity was collected and analyzed. Use of this instrument enables validated assessment of student progress at the end of their first year, and also enables the study of further innovations in the curriculum for the first year computer science courses.

Chapter 1

Introduction

Pedagogic innovation at the introductory level of computer science education is not new. The notion that some curricular idea "works for me" is not good enough for most people to consider adoption of that particular idea and is not sufficient for scientific exploration. A reliable and validated instrument is needed for use in scientific experimentation of new curricular advances to enable researchers to study the impact of the curricular innovation on student knowledge and skills in a particular subject area. The purpose of this work is to create a validated assessment instrument that can be used to measure student achievement at the introductory level of computer science curriculum.

1.1 Computing Curricula

Computers and computing began to emerge as a field of study in the middle of the last century. Colleges and universities began creating departments and degree programs in the 1960s. As these departments grew in number, a group of faculty from some of these colleges and universities was formed under the auspices of the Association for Computing Machinery (ACM) to explore the various issues facing these institutions while developing these programs in computing. This group produced a report outlining a curriculum for the newly emerging discipline of computer science (Committee on Computer Science Curriculum 1968). Since that time, several revisions have been made to reflect changing times and trends in the field (Committee on Computer Science Curriculum 1978; ACM/IEEE-CS Joint Curriculum Task Force Curricula 1991; Joint Task Force on Computing Curricula 2001).

The most recent of these, commonly known as CC2001 (ACM/IEEE-CS Joint Task Force on Computing Curricula 2001) is divided into several volumes, each covering a different sub-discipline of computing. These volumes are: Computer Science, Computer Engineering, Software Engineering, and Information Systems. This dissertation focuses on the Computer Science volume, which will be referred to as CC2001 for the remainder of this dissertation. CC2001 divides the computer science curriculum into fourteen knowledge areas and subdivides the curriculum into introductory, intermediate, and advanced course levels. For each level, the report recommends pedagogical approaches to the topics in each area, including many specific details that were not present in previous curricula.

Before CC2001, there was much information in the literature about the approach, assignments, lab environments, and teaching aids that were most appropriate for courses. These issues are discussed in Chapter 2. Of special interest are the CS1-CS2 introductory courses, since these are the first courses that students are exposed to. CC2001 recognizes six approaches to the introductory sequence: three programming-first approaches (Imperative-first, Objects-first, and Functional-first) and three non-programming-first approaches (Breadth-first, Algorithms-first, and Hardware-first). The report does not recommend one over the other, but rather points out their relative strengths and weaknesses.

1.2 Problem Statement

Whenever a new curricular device is conceived, its effectiveness must be determined: Does the innovation actually help students' understanding of the material? Research investigations conducted on new curricular innovations have employed measures based on lab grade, overall course grade, resignation rate, or exam grades (Cooper, Dann et al. 2003; Decker 2003; Ventura 2003).

The problem with using these types of metrics in a study is that often they are not proven reliable or valid. Reliability, or the "degree of consistency among test scores" (Marshall and Hales 1972, p. 4), and validity, the ability of a test to be "both consistent and relevant" (Marshall and Hales 1972, p. 104), are both essential whenever the results of any metric are to be analyzed.

If a metric is reliable, then the results for a particular student for that metric must be reproducible. Reliability can be assessed using a time-sampling method, a parallel-forms method, or an internal-consistency method (Ravid 1994; Kaplan and Saccuzzo 2001). The most common time-sampling method is the test-retest method, where the same subjects take an exam at two different times and scores are checked for consistency. For a parallel-forms method, two tests are created that are designed to test the same set of skills. Students then take both forms of the exam, and their results are compared for consistency. For an internal-consistency method, the test is split into two halves, and the two halves are compared for consistency. With an internal consistency method, the test is only taken once, which saves time and resources for the researcher.

However, some of the methods have drawbacks. When using a test-retest method, there can be a practice effect. The practice effect is the possibility that when students take an exam more than once, they will do better the second time simply because they have taken the exam before. This effect is not easy to address, so many researchers choose to measure reliability using some variant of the parallel-forms method or internal-consistency methods (Marshall and Hales 1972; Ravid 1994; Kaplan and Saccuzzo 2001). However, with parallel forms, there is a burden on the participants and administrators of the exam. The participants must take a very similar exam twice, and resources must be devoted to administering these two exams. To minimize practice effect, this duplicate testing should occur on the same day (Marshall and Hales 1972; Ravid 1994; Kaplan and Saccuzzo 2001).

Validity can be assessed using the methods of face validity, content-related validity, or criterion-related validity. Face-validity evidence is gathered from the appearance of validity. For example, a test screening for suitable mechanics to fix cars for a dealership should have questions about cars and their component parts and would probably not include questions about the interpretations of famous literary works. This type of validity does not include an in-depth analysis of the test, but rather a quick read of the questions to assure that the test appears to be applicable to the domain (Marshall and Hales 1972; Ravid 1994; Kaplan and Saccuzzo 2001).

Content-related validity is like face validity in that it is a logical, rather than a statistical, way of assessing validity. With content-related validity, one must determine whether the construction of the test adequately assesses the knowledge it is supposed to. Expert judgment is often called on to assess the content-related validity of a measure (Kaplan and Saccuzzo 2001).

Criterion-related validity is the assessment of how well a particular metric corresponds with a particular criterion (Kaplan and Saccuzzo 2001). For example, the Scholastic Aptitude Test (SAT) is used by most colleges and universities as an indicator of how well a student will perform after high school. In order for one to use the SAT in this way, the SAT Program Handbook provides results of criterion-validity testing to show the evidence that it is predictive of college performance (SAT Program Handbook 2006).

The investigations cited in the first paragraph of this section (Cooper, Dann et al. 2003; Decker 2003; Ventura 2003) suffer not only from using assessments which are not demonstrated valid and reliable, they suffer a further drawback in that they do not specify how a particular grade is arrived at. For example, when using overall course grade as the success marker, one should know if there was a curve placed on the grades, or even the basic breakdown of what is considered “A” work. This problem persists even when using numeric percentage grades, such as 89%. Grading standards must be well documented to be valuable for assessing the quality of the study.

1.3 Motivation

As with the previous curricula, CC2001 does not provide faculty with instructions for how to implement its suggestions and guidelines. This leaves faculty to take their own approaches to the material, and invent assignments, lab exercises, and other teaching aids for specific courses outlined in the curriculum. When faculty claim innovation in the CS1 curriculum, we need a way of assessing students' comprehension of the core CS1 material. The original goal of this dissertation was to create a reliable and validated assessment instrument that assesses the knowledge of a student who has taken a CS1 class using one of the programming-first approaches described in CC2001. However, that goal was broadened to create an assessment for the entire introductory sequence (CS1-CS2). This change was necessitated when it was discovered through the work of this dissertation that the topical coverage of CS1 as described by CC2001 did not provide a rich enough set of topics for creating an assessment that would serve across all approaches to the introductory curriculum. A detailed explanation of this process is described in Chapter 3.

An assessment that can be used to measure curricular innovation (i.e., the success of students in a course using a particular approach)[1] should be independent of the approach and the programming language used in the introductory sequence. Essentially, the assessment should not be tied to one particular language of implementation and should not be concerned with the testing of syntactic minutiae of a particular programming language. If the assessment is designed with this idea in mind, it can be used to test the results of curricular changes regardless of the language of choice in the introductory sequence or the particular approach taken.

The main motivation for this work is the fact that no such assessment instrument is available. Many forms of assessment at the end of the four years of undergraduate education are available to computer science faculty. Two such examples are the Educational Testing Service's (ETS) Graduate Record Examinations (GRE) Subject Test in Computer Science (GRE Subject Test General Description 2004) and ETS's Major Field Test in Computer Science (ETS 2003). The GRE Subject Test is designed to assess a student's ability to succeed in graduate school, while the Major Field Test is designed as an overall outcomes test for the undergraduate curriculum. In either case, since the tests are administered at the end of the student's program for the undergraduate degree, they are not practical sources of information about the students' knowledge at the end of their first year of their undergraduate career. Furthermore, careful examination of the

[1] Several approaches for the introductory sequence are discussed in CC2001. Chapter 3 of this dissertation provides more details about these different approaches.

reliability and validity of these two exams gives us a better indication of their lack of applicability to this endeavor.

The GRE subject tests across all disciplines have been shown to predict first-year graduate grade-point average moderately well and are more predictive than undergraduate grade-point averages in half the cases (GRE Score Use 2003). The subject test scores have also been used in conjunction with GRE test scores and undergraduate grade-point average to help predict performance. Unfortunately, the data provided by ETS about the GRE Subject tests does not include information specifically about the predictive value of the computer science subject exam.

The ETS Major Field Test in Computer Science was created with the help of experts in the subject area. There is no indication, however, that the test corresponds to a particular curriculum or to the recommendations of CC2001 (Major Field Test 2003). The reliability reported for the test for the academic year 2001–2002 was 0.89 (Major Field Test 2002), which is deemed acceptable. However, the exam tests all of the following topics: programming fundamentals, software engineering, computer architecture, computer organization, operating systems, algorithms, theory, computational mathematics, and certain other specialized topics in computer science (Major Field Test Content 2003). This topic list is much larger than what is covered in any CS1, whether programming-first or non-programming-first. Even though there is a sub-section of the exam that students who have completed CS1-CS2 could complete, the breadth of the concepts covered on the test would be overwhelming for students who had only completed one year of study, even in a programming-first approach.

Another test that is available is the Advanced Placement (AP) exam (AP 2003) in computer science, which students can take while in high school to show their knowledge of material in a particular subject area before entering college. This test has been shown to be an effective instrument to gauge a student's readiness and abilities in the introductory computing courses. However, this exam has shortcomings that will be discussed in §2.6. The information about the reliability and validity of the AP exam has been collected by its creators. This information only tells us the results for the AP as a measure of student's knowledge of the material tested on the AP exam. It has not been shown to be reliable or validated for any other purpose, especially not as an assessment for introductory computer science at the college level using the CC2001 guidelines.

These assessment measures, which are taken before starting CS1 or after the end of four years of study, do not help us evaluate student's understanding of the core CS1-CS2 material immediately after completion of the CS1-CS2 sequence, nor do they provide a good source of comparison of curricular innovations for CS1. In order to promote further experimentation within the development of this course, a validated and reliable assessment instrument needs to be created.

1.4 Contributions and Significance of the Dissertation

The first part of this dissertation establishes the common subject matter among the three programming-first approaches to teaching CS1-CS2 presented in CC2001. One finding is that there is a basic skill set that students who leave CS1-CS2 should have, no matter which of the seemingly disparate approaches was used to teach the course.

The second contribution is the assessment instrument. This assessment is a paper-and-pencil exam that is language and approach independent. It is the first validated and reliable means of assessing a student's understanding of the material in the programming-first CS1-CS2 sequence and should prove useful to individual course instructors. It also provides a useful benchmark for studies that focus on the relative success of different approaches to teaching the introductory sequence, CS1-CS2. This instrument will be available to test if a particular teaching technique or pedagogical advance really improves students' performance in CS1-CS2. If instructors use the instrument as a means of assessing their students' performance in a CS1-CS2 sequence, the results could indicate poor performance of a particular instructor or teaching technique. A poor result may cause the institution to reassess their current methodologies or curriculum in the CS1-CS2 sequence.

This instrument can be useful in further study of the computer science curriculum in many areas. First, it can provide a means of assessing curricular innovation and change at the introductory level. Instructors can use previous scores as a baseline for comparison of changes that have been made to the first year courses. Second, it can provide better information for studies that have looked at predictors of success in the first year courses (see §2.2). Many such studies have been published, but few, if any, report on the measure of success that has been used. Furthermore, the metrics that have been identified have not been validated before their use. This assessment provides a validated instrument to measure success in the introductory sequence.

1.5 Outline of Dissertation

The rest of this dissertation is organized as follows. Chapter 2 is an investigation of research on methodologies for the introductory curriculum, predicting success in the introductory curriculum, and assessing students within the curriculum.

Chapter 3 gives a detailed analysis of the CC2001 document and establishes a core list of topics that are common to all programming-first introductory sequences described in CC2001.

Chapter 4 shows how the list created in Chapter 3 was refined to a more manageable list of topics that could be used to to create the exam.

Chapter 5 discusses the learning objectives as they are given in CC2001 and which of those learning objectives map onto the topics chosen for inclusion in this exam.

Chapter 6 discusses the creation and format of the exam as well as the results of the reviews of the instrument by various members of the computer science community.

Chapter 7 discusses the administration procedure for the exam as well as the grading guideline for the exam. It also presents the information about the study conducted to gather the data needed to analyze the exam for validity and reliability.

Chapter 8 presents the results of the statistical analysis of the exam data collected during the study described in Chapter 7.

Chapter 2

Background

This chapter provides a look at the research that has been published within the three major categories that the work of this dissertation spans: methodology in the introductory curriculum, predictors research, and assessment issues. It also presents the results of my preliminary work in studies of object-oriented understanding in a non-majors CS2 course (§2.5) and another study of the correlation between AP exam grades and student performance in introductory computing courses. It highlights the lack of and therefore need for appropriately validated assessment instruments in each of these research areas.

2.1 Methodologies in the Introductory Curriculum

Both for years before and during the development of CC2001, there was a long debate regarding the most acceptable way to teach the introductory computer science curriculum. CC2001 does not advocate a particular approach, but rather provides a selection of six approaches for the introductory curriculum and encourages institutions to select which one they feel is best. However, even after the publication of CC2001, the debate over methodology still continues. In this section, we look at the pre-CC2001 debate.

Owens et al. (1994), Evans (1996), Fincher (1999) and Marion (1999) each give opinions about how to present the information to students in an introductory course. Each of these papers lays out many of the foundational ideas for the advocated six approaches to the introductory curriculum given in the CC2001 document. It is clear that the CC2001 committee used these ideas as guidelines for preparing the more detailed treatment of the approaches to the introductory curriculum that appear within CC2001 itself. The papers argue for introductory methodologies that concentrate on programming as well as approaches that will be labeled non-programming-first approaches in CC2001.

2.1.1 Courses that speak to this generation

There is no doubt that the current generation of students has grown up with the computer, computer games, and the Internet. The question to educators becomes whether this familiarity impacts the way students see and interact with computers. Many would argue that the exposure of these students to computers greatly influences what they believe computers should do for them.

Stein (1996) argues that introductory programming should become more interactive and more closely mimic the way that users are interacting with the machine. This way, students realize that they are creating and modifying an interactively changing system, which will parallel more closely with what software development is like in industry.

Guzdial and Soloway (2002) suggest that one reason we have a problem keeping students interested in computing is that we have an "outdated view of computing and students" and that we should be shifting our focus towards media and the use of media to drive the direction of courses. Since the first publication of these ideas, they have continued to be developed by Guzdial, who has just released a text (Guzdial and Ericson 2006) that integrates multi-media into the CS1 course as a way to engage students in the process of programming.

Another approach that utilizes the more advanced graphical capabilities of modern computers is advocated by Cooper, Dann, and Pausch (2003), who developed a programming environment called Alice. Alice uses 3D graphics and drag and drop syntax creation while interacting in an object-oriented world. Dann, Cooper, and Pausch (2006) is a text based on this that gives support materials to their environment and their view of introductory programming.

Other graphical approaches are those of Proulx, Rasala, and Fell (1996), Reges (2000), and Alphonce and Ventura (2003). These groups argue for an approach to CS1 that utilizes graphics and event-driven programming to motivate students while learning the concepts presented in CS1. However, Reges (2005, 2006) has recently abandoned this view of introductory computing in favor of the more traditional view of programming instruction (text-based, control-structure oriented) because he believes that his earlier approach was not working. The evidence he presents for this belief is anecdotal, as his belief that since his switch "back," his students are performing better than before. Reges's switch due to personal belief rather than evidence of performance points once again to the need for an instrument that can measure student understanding and that can be used as a comparison between approaches.

2.1.2 Approaches to CS1-CS2 using collaborative techniques

Approaches that are not strictly focused on programming constructs and syntactical issues have also been explored. These ideas focus on the act of programming and how to create more effective programmers using various types of collaborative techniques.

In the Applied Apprenticeship Approach (AAA), Astrachan and Reed (1995) seek to change the way the introductory courses are taught in three key areas: expectations, focus, and delivery. They expect students to read and modify programs before actually writing them from scratch. They change the type of problem that students focus on in the introductory courses, moving students away from "toy" problems that are too small to illustrate the power of computing to larger problems that really showcase the power of the discipline. They change the order of delivery of the topics presented in the course advocating not introducing a topic before the time in a course when it is needed.

Kölling and Barnes (2004) suggest an enhancement to AAA by more closely integrating the lab (programming) part of a course with the lecture portion. The problems are presented and discussed in lecture. There are perhaps partial solutions worked on in lecture that are continued by students on their own. They also advocate having students work with code that has been expertly written and modify and expand it.

In Pair Programming (Nagappan et al. (2003)), two students work together at one computer to solve a problem. One student acts as the “driver”, actually typing and using the mouse, while the other acts as a “navigator,” providing direction about what needs to be done. Nagappan et al., showed empirical evidence that pair programming in their CS1 class improved retention rates for the number of students that remained in the course, and improved their students’ perspective on working in collaborative environments. After doing pair programming, the students feel that working in a collaborative environment is more beneficial than they originally thought. The authors further conclude using the data collected from the grades of students that pair programming is in no way a deterrent to student performance.

A common theme in each of these ideas is that programming should not be taught as a necessarily singular activity and that the learning environment can be enhanced from both the students’ and educator’s perspectives using some sort of collaborative technique. These ideas shift the focus away from simply memorizing the syntax of a language and then working on problems in isolation to working with programs along with other people and using the collaboration to benefit the learning experience for all parties.

These ideas provide an interesting viewpoint about teaching this material. For some of the approaches, anecdotal evidence is suggestive of their success. Since no validated assessment instrument was used to measure the effect of any approach on student performance, the effectiveness remains in question and points to a need for an instrument to assess their effectiveness in conveying introductory concepts.

2.1.3 Approaches relying on paradigm

A programming paradigm is a view of a particular program's main unit of computation. For example, when one programs in Lisp, the main unit of computation is the function, and the paradigm is called functional programming. Another set of documented approaches relies heavily on language, but more importantly on paradigm, and on issues that arise when teaching a particular paradigm

Pattis (1993), who was teaching Pascal, was concerned about the appropriate point in the curriculum to teach subprograms. He argued, in contrast to the prevailing ideas of the times, that procedures should be taught "first" (i.e. as early as possible in the curriculum).

Moving forward a few years, we see Culwin (1999) arguing how to appropriately teach object-oriented programming, followed by a strong course outline for an Objects-first CS1 advocated by Alphonce and Ventura (2002; Ventura 2003).

For these approaches as well as others, while there may be strong anecdotal evidence to support them, little empirical evidence, aside from Ventura (2003), has been presented as to the real effect of these methodologies on learning the appropriate material for CS1.

2.2 Predictors Research

The need for accurate assessment instruments is again evident when one looks at the literature on predictors of success for CS1. Numerous studies have focused on predicting success in the first year. Success for each of these studies has been measured in various ways, none of which have been shown to be reliable or validated, nor do any of the measures of success have an ability to be reproduced exactly, because many of them involve specific assignments for a course or unpublished exam questions. Still others simply used overall course grades in a CS1 course that were computed using various weightings of course components.

Mazlack (1980) administered the IBM Programmer Aptitude Test (PAT) to study its predictive ability for students in computer science. Little information is available about the PAT. In the early 1980s, it was used by many companies (including IBM) to screen potential applicants for jobs. However, there does not seem to be any publicly available information about its validity for this purpose. Mazlack uses the results on the PAT as a potential predictor for each of quiz grades, programming assignment grades, midterm exam grade, final exam grade, and overall course grade. His results showed that PAT was not predictive of achievement in any of these areas.

Evans and Simkin (1989) studied demographic profiles, past high school achievements, prior programming experience, behavioral habits, cognitive style, and problem solving abilities to try to predict success in introductory curriculum. To measure success, Evans and Simkin used as individual measurements, homework problem scores (presumably programming problems), as well as scores on multiple choice exam questions, fill-in-the-blank exam questions, and overall exam scores. They concluded that none of the variables they studied best predicted computer proficiency in their course and that more work was needed in this area.

Hagan and Markham (2000) studied the impact of prior programming experience on student success in introductory computing. They used the amount of prior programming as a possible predictor of assignment scores (programming projects), midterm exams, and a final exam individually. They found that not only did prior programming experience help, but the more languages that a student was exposed to before entering CS1, the more their CS1 performance improved.

Cantwell-Wilson and Schrock (2001) investigated twelve possible predictive factors in their study of success in their introductory CS1 course. They concluded that “comfort level” was the best predictor for success in the course, followed by mathematics background. It is interesting that comfort level a students’ feelings about a course and their place in the course, came out as the most predictive of performance in this study. To measure comfort level, the Computer Programmer’s Self-Efficacy Scale was used, which is a validated tool for measuring aspects of self-efficacy including comfort level (Ramalingham and Wiedenbeck 1998). The measurement of success that was used was midterm course grade. Cantwell-Wilson and Schrock showed that midterm course grade was highly correlated with final course grade, so a successful midterm grade also would indicate a successful final course grade.

For each of the four previous studies mentioned, the measures of success used were all created specifically for the course. These measures were not consistent across the studies, nor are they particularly reproducible to those outside of the course because no information is publicly available about what exam questions looked like, how they were graded or exactly what the individual assignments were and how they were graded.

Kurtz (1980) used final course grade as his measurement of success in CS1 and created and administered a test of formal (abstract) reasoning ability in order to classify the students and study their performance in an introductory programming course. His classification scheme (late concrete, early formal, late formal) of abstract reasoning ability has never been validated, but he did show that students classified into one of his groups performed well in CS1 (late formal) and should be advised to attend an advanced section, while those classified in another (late concrete) performed poorly and should be discouraged from attending an advanced section.

Leeper and Silver (1982) concluded that SAT verbal score, followed by SAT math score, were the two highest predictors of success in the population of CS1 students they studied. Success was determined for this group by overall letter grade in the course as well. Other measures that were studied, but did not reveal a significant predictive factor were a student's exposure to Science, Math, and Foreign language in high school as measured by the number of units of each type of course taken.

For the final two studies, since overall course grade was used as a measure for success, the measurement included assignments as well as exams. For Leeper and Silver, the proportional weightings that were used to compute the overall course grade were not even reported. As before, even if test grades are used as a factor to compute overall course grades, the test questions, or assignment specifications are not available and no information about reliability or validity is offered about these measures.

Another factor that is unfortunate for reproducing and accurately interpreting these results is the fact that the CS1 course had not been clearly defined. We cannot be sure that the outcomes that were expected of the students in some of these studies are in line with the recommendations of CC2001, or even in line with each other. For the studies that occurred before the publication of CC2001, it must be assumed that the researchers could not have anticipated CC2001 and therefore the courses will not reflect its recommendations.

Even recent work done on a course that embraces CC2001's recommendations for an objects-first CS1 uses only measures of overall course grade, exam grades, and lab grades in its study (Ventura 2003). The predictive values of the factors studied are given, as in the other work cited above, and predictive factors have been found in this study as well (overall course average, lab (programming) assignment average, exam average and measures of effort (actually completing the assigned tasks for the course)). However, the study once again fails to convince that the measures used for students' level of success have been validated.

2.3 Educational Objectives and Outcomes Assessment

Well-defined educational objectives and outcomes assessment measures for creating new curricula in CS1 are increasingly common. Parker, Fleming et al. (2001) give a methodology for integrating assessment into the course so that it provides frequent feedback to the students (their performance) and the instructors (in seeing student performance). They also provide a methodology for creating these frequently-administered assessment instruments. This paper provides a methodology for doing this type of assessment, but does not give the actual assessments. Since we are looking for an assessment for introductory courses, not simply a methodology, this work does not solve the problem presented for this dissertation.

Neebel and Litka (2002) propose a design of a CS1 course where a student's grade in the course is based on how many learning outcomes the student has achieved. The outcomes for the course have been created before the course is taught and the students are informed of what the outcomes are. The assessment mechanism can vary from objective to objective, but students must achieve a grade of 80% on the assessment for the outcome to have it count as achieved. The student's grade is determined by how many outcomes are achieved.

One set of educational objectives that has been explored in a CS1 course is that of Bloom's Taxonomy of Educational Objectives (Lister and Leaney 2003). Lister and Leaney use Bloom's Taxonomy as a way to structure the criterion used to grade the students of the course. Students who receive the minimum passing grade in the course are expected to have successfully completed criteria that fall into the lowest two levels of Bloom's Taxonomy. Higher grades in the course are earned by completing criteria that are categorized at higher levels of the taxonomy. Missing, however, is a clear description of exactly what (if any) skills the student should come out of CS1 with. It is unclear whether students are required to understand such topics as iteration or selection to pass the course. The authors argue that, with this approach, CS2 must be modified to embrace Bloom's Taxonomy as well. When adapting a CS2 course for using Bloom's Taxonomy the outcomes expected from CS2 seem to differ from the traditional set of topics that are normally associated with CS2 by including several software engineering concepts as opposed to the typical data structures presented. These software engineering concepts include analysis, design, and synthesis of larger software systems.

Another approach to course-embedded assessment[2] is used at Slippery Rock University (Whitfield 2003). Their curriculum was designed so that each student would come out of the program having learned a well-defined set of topics and ideas. The courses at this university are designed to make sure that the appropriate material was presented to achieve these outcomes. However, the stated outcomes for the curriculum seem generalized and vague. It is difficult to see whether or not they coincide with CC2001's recommendations for CS1. Their assessment methods were not proven valid

[2] Course-embedded assessment is assessment that occurs within the course at semi-regular intervals. For example, midterm exams, graded homeworks, and quizzes all could be administered throughout the semester as course-embedded assessment instruments.

or reliable, nor did they indicate whether students were meeting the designated goal of success in their CS1 course.

2.4 Assessment of Programming Skill for CS1

2.4.1 "The" Study (or at least the one everyone recognizes because of its failure)

There has been one documented attempt at creation of an assessment for CS1. A working group from the Conference on Innovation and Technology in Computer Science Education (ITiCSE) created a programming test that was administered to students at multiple institutions in multiple countries (McCracken, Almstrum et al. 2001). The group's results indicated that students coming out of CS1 did not have the programming skills that the test assessed.

Among the positives of this attempt at assessment were that it included problems that were well thought out and that it made an attempt to define and cover all of the material that a CS1 student should have mastery of. Another positive was the fact that there were specific grading rubrics created for the problems, which helped lead to uniform scoring. The students were not restricted to a particular language or programming environment, so the students completed the exercises in whatever way was most comfortable to them.

However, the study was flawed. This was recognized even by the members of the working group. The problems given had an inherent mathematical flavor that would have disadvantaged students with mathematical anxiety. They also admit in their analysis that one of the test questions "was undoubtedly difficult for students who had never studied stacks or other basic data structures" (McCracken, Almstrum et al. 2001). They also pointed out flaws in the presentation of the problems and the instructions for administering the exercises. Therefore, even with all the positives of this study, there is still room for improvement to make an assessment instrument that could be more true to the current flavors of CS1 as described in CC2001.

2.4.2 Critical Eye to Assessment Practices

Daly and Waldron (2004) suggest that traditional written exams in computer science courses do not accurately assess students and that laboratory or practical coding exams are a better way to get a true assessment picture of student learning. They show that there is a stronger correlation between their laboratory assessment and a larger software project that students complete in their third year than between the traditional written exam and the software project. It is not clear from their publications whether the correlations are statistically significant, nor do they provide information about the reliability, validity, or grading of their laboratory assessment. Even though the correlation is stronger with their lab assessments than the more traditional written exams that they had administered previously, it does not give any information for an educator to use in their own courses.

2.5 Study of Performance in Non-Majors Course

In an earlier project, I analyzed students' retention of object-oriented concepts was conducted in the CS2 course for non-majors at the University at Buffalo, SUNY (Decker

2003). One problem that grew out of this investigation was that of how to accurately assess students' knowledge in this area. The solution that was used was one that is commonly found in the rest of the literature, to simply use exam scores as the benchmark of success. The experiment was well received by the reviewing committee for the Consortium for Computing Sciences in Colleges Eastern Conference as well as attendees at the conference as a true empirical investigation of student comprehension of basic object-oriented material in a CS1-CS2 sequence.

However, this study in itself suffers from the same problems as much of the literature in this area. The exams and tests that were administered were not proven to be reliable or valid. Furthermore, this experiment covered only the students' knowledge of object-oriented concepts, not general CS1 knowledge. In this dissertation, we seek a valid, reliable, and comprehensive assessment of CS1 knowledge that is independent of language or paradigm.

2.6 Analysis of Published Assessment Instruments

Three publicly available instruments will be considered in this section, the Advanced Placement (AP) Exam for Computer Science, the Educational Testing Services' (ETS) Major Field Test in Computer Science, and the Graduate Record Exam (GRE) Subject Test in Computer Science. For each exam, the content of the exam, the construction process, and some information about the grading of the exam will be presented, as well as information about the psychometric properties of the instruments, specifically reliability and validity measures.

Information about the reliability of these tests has been gathered mainly from the test makers themselves. This could be viewed by some as potentially problematic. Especially for test makers like the ETS, which produces the AP, Major Field Test, and GRE, reliability will have to be high for people to continue to use the tests. However, since ETS owns all of the raw data, it is difficult for independent analysis to be performed on the exams and this provides the ability for some to question these tests and pushes the test makers to provide continued data about the reliability and validity of the tests upon which so many rely.

2.6.1 Advanced Placement Exam

The Advanced Placement (AP) exam is given at the end of a high school course of study in a topic that is usually reserved for the college level. The program was designed to help high school students take college-level courses before graduating from high school. There are many different topics that one can take an advanced placement course in. Each high school can offer as many different types of AP courses as they see fit. At the end of the course, students take the AP exam. These AP grades are then passed along to the college or university of their choice, which have traditionally given some sort of credit for “good” scores on the AP exam. Each college or university sets its own standards for awarding credit based on AP score.

There are two AP courses and tests in computer science, the Computer Science A and the Computer Science AB. The Computer Science A exam covers material generally presented in CS1, while the AB exam covers material from CS1 and CS2 (AP, 2003). The AP exams are written by a team called the Development Committee (AP CS Development Committee, 2004). The committee for the Computer Science exams consists of instructors from colleges and universities that teach introductory computer science. They develop questions, which are then reviewed by content experts and the chief reader for that exam from ETS, the company that publishes the exam. After approval, questions can be added to the exam.

The AP exam was developed with the recommendations for curriculum of the ACM and IEEE, which would indicate that it should follow the CC2001 recommendations (AP 2003). However, the exam booklet does not indicate that the exam is based on any curricular models specifically, so it is not certain whether it follows CC2001.

Both the Computer Science A and AB exam are broken into a multiple choice and free response section. The multiple choice section has 40 questions, and the free response has four questions. The multiple choice section is given 75 minutes, and the free response section is given 105 minutes (AP CS A Test Description, 2004; AP CS AB Test Description, 2004).

The multiple choice sections are scored by a computer, and the free response questions are scored by outside readers using a grading guideline. For each exam, the development committee gives weighting to the sections of the exam. A final score is computed and then mapped to a 5-point system, with 5 indicating extremely qualified in this subject area, and 1 indicating not qualified in the area (AP Exam Grading, 2004).

2.6.1.1 Reliability of the AP Exam

In the AP exam data, it is a shame that the *n*[3] is not properly reported. One could infer from reading the text that accompanies the tables about reliability that the reliability estimates were made from the entirety of the population that took the AP exam for a particular year. In this case, the AP exam data is from the 2003 administration of the exam. The sample population used for this statistical analysis was high school students who took the AP exam. Even though the explicit n is not reported for these statistics, it can be assumed that the n consisted of all students who took the AP exam in the year 2003 (AP CS A Reliability 2004; AP CS AB Reliability 2004).

It is interesting to note that the reliability coefficients are lower for the Computer Science AB exam (.919) than for the A exam (.955).[4] It is hard to know why this would be the case and also not evident whether the reliability is statistically significantly lower. From a cursory examination of the numbers provided, they seem close enough to not be a statistically significant difference. In either case, the reliability coefficients are considered adequate for an instrument.

[3] In statistical reporting, n is the total number of data points used in any statistical analysis.
[4] The numbers given in reliability estimates are the results of the statistical test known as Cronbach's alpha. This is a measure of internal consistency reliability. The results of Cronbach's alpha range from 0 to 1 with 1 being 100% reliable, with a number above .7 considered minimally acceptable for consistency.

2.6.1.2 Validity of the AP Exam

For the AP exam, two types of validity information are given. The first is a comparison between AP students and non-AP students on an alternate exam developed by the AP exam creators. The alternate exam that was created contained 12 multiple choice questions and one free response question from the 1999 Computer Science A or AB exam. Equal weights were placed on both sections of the exam. Thirteen colleges and universities participated in the study for the Computer Science A exam, and 12 participated in the Computer Science AB exam (AP Validity, 2004).

The results that are reported for this study are the average scores on these sample questions for the students. For the AP CS A exam, AP students had an average score of 47, while non-AP students had an average score of 40. For the AP CS AB exam, AP students had an average score of 55, while non-AP students had an average score of 47. The fact that the average score for the AP students is higher is reported as being significant. However, there is no indication if there is a statistically significant difference between the scores of the two groups, because information was not reported about which statistical test was used or the computed values of those tests.

I have reservations about using the results from this study as any form of evidence of validity for this test. First of all, saying one number is higher than another does not show statistically that the results of one group are any different than the other. Secondly, since the students were using actual questions from the AP exam, students who took the AP courses could have the unfair advantage of having seen those questions before while taking the AP course and practicing for the exam. There could be a practice effect influencing their scores. Lastly, this study was conducted on students who are already in college. The study does not indicate when the exam was administered to the students. It could be the case that students in college would approach an exam of this type differently than the typical high school students that would normally take the AP exam. However, this evidence is presented as proof of validity for the AP exam.

The second reporting of validity information shows the overall performance of AP students versus non-AP students in higher level courses overall. The GPAs of the students were compared to show that in most cases the average GPA of an AP student was higher than the average GPA of those who did not take AP but who took lower-level college courses in that particular discipline. The report cited here worked with 21 colleges and universities and covered all of the AP exams that were given at the time of the study (Morgan and Ramist, 1998). The use of overall GPA as an outcome measure to show the validity of the AP exam is questionable at best. The validity of the AP exam for assessing student proficiency in a particular topic area would have nothing to do with a student’s overall GPA after taking the exam.

Once again, it is not clear whether the differences in student performance are statistically significant. The reporting only points out the difference between the two groups of scores, not the results of statistical tests (such as a t-test) looking for differences in the two groups. Furthermore, I am not sure if this evidence provides support for the validity of an AP exam to assess the content of a particular discipline. It can be argued that students choosing to pursue college-level work in high school are probably more motivated to succeed than those who do not, and this motivation could contribute more to their overall success in college than the AP program itself. No precautions were reportedly taken when analyzing the data reported in this study. With the lack of real statistical evidence, I once again call into question the usefulness of these results in assessing validity.

2.6.2 Graduate Record Exam Subject Test in Computer Science

For most students entering a graduate program, a GRE general test is required in order to show their suitability for graduate work. The GRE subject tests provide information about a student's overall abilities in a particular subject area (GRE Subject Test General Description, 2004). In the case of computer science, the test contains questions in the areas of software systems and methodologies, computer organization and architecture, and theory and mathematical background. It is designed to be given at the end of an undergraduate program to be used as a tool by graduate departments to assess a candidate's suitability for graduate work in Computer Science (GRE Subject Test Computer Science Description, 2004).

The test contains 70 multiple choice questions, which are scored on a scale from 20-99. The raw scores are computed by adding up the number of correct scores and subtracting ¼ of a point for each incorrect answer given. When scores are reported, they are scaled by adding a zero to the points earned, so scores could range from 200-990 on any particular subject test. This scaling is not reported as a normalizing procedure in the published GRE descriptions, although it is reasonable to assume that it could be a normalizing of the scores. Students are given two hours and fifty minutes to complete the test (GRE Subject Test Computer Science Description, 2004).

The test creation process involves professors at both the graduate and undergraduate levels at colleges and universities in the United States and Canada. The members of this committee write and review test questions that are assembled by ETS into a test. Members of the ETS testing service then also review the test for content and any potential bias. New versions of the tests are analyzed to make sure they are equivalent in content and difficulty to the older versions. Revisions to content and scope of the test are undertaken regularly to assure that test content is in line with current trends in undergraduate programs across the country (GRE Subject Test Computer Science Description, 2004).

Although this test is available for colleges and universities for administration to students at any time during their academic careers, it covers much more material than the first year of instruction and would therefore not be a suitable instrument for the purposes of this dissertation.

2.6.2.1 Reliability of the GRE Subject Test in Computer Science

There is data available for the reliability of the regular GRE as an instrument, but I could not find any published data could be found about the subject tests to determine their reliability.

2.6.2.2 Validity of the GRE Subject Test in Computer Science

The GRE subject tests provide more information about validity of its subject tests, however. Kuncel, Hezlett and Ones (2001) give information for the regular GRE as well as the subject tests. Although no subject test is studied individually, Kuncel, Hezlett and Ones provides one of the few pieces of information available about the subject tests at all. There were 1,753 independent samples studied in this work to yield the predictive validity of the GRE. The operational validity for the GRE predicting overall graduate GPA is reported to be $p = .41$[5]. The operational validity for the GRE predicting first year graduate GPA is reported to be p = .45. The operational validity for the GRE predicting results on comprehensive exam scores is reported to be p=.51. The operational validity for the GRE correlating with faculty ratings of student performance is reported to be p=.50.

Another aspect of the GRE Subject Test's validity is the effort made to ensure that the content of the test is appropriate for the departments that will eventually use the test results. In 2001-2002, a survey was conducted to assess the appropriateness of the

[5] In this case, the p that is reported is a result of the statistical testing that is done, not an indication of significance. The results reported in this study are significant.

content of the GRE Subject Test in Computer Science. 1,250 computer science departments were contacted, and there were 256 departments whose input was used to analyze the content of the test. Based on the feedback from the departments, some changes were implemented in the content of the test in 2003 (CS Content Rep Study, 2002). This evaluation of the material of the test points to a commitment to content validity of the instrument.

2.6.3 ETS Major Field Test in Computer Science

Of the three “standardized” tests described in this paper, the ETS Major Field Test is probably the least known to the average person. ETS is the creator of the AP, GRE, and the Major Field Tests. The Major Field Tests are designed to be given at the end of a four-year undergraduate curriculum. For the computer science version, students are expected to have completed an undergraduate curriculum whose major area of study is computer science. The content of the Major Field Test ranges over all four years of undergraduate study and therefore includes introductory programming, but the content is not limited to that. It was modeled after the GRE subject test, but not designed to predict success in a graduate program in computer science. Rather, it was designed to provide an assessment of the basic knowledge and understanding of senior undergraduates ready to graduate from a field of study (ETS Major Field Test Description, 2004). Another key difference between the GRE Subject Test and the ETS Major Field Test is that with the Major Field Test, schools have the opportunity to add to the test up to 50 questions that are unique to the institution. This provides a mechanism for an institution to assess the unique facets of its undergraduate program, while also using the main body of the test as a mechanism for comparing its program to other programs across the country.

This test consists of 60 multiple choice questions in four main areas: programming fundamentals; software engineering; computer architecture, organization, and operating systems; and algorithms, theory, and computation mathematics. There is also a special topics section that contains other senior-level computer science topics. Of these areas, only the first would contain information that deals with first year introductory programming topics (ETS Major Field Test Description, 2004).

Faculty from colleges and universities in the discipline of computer science are consulted for the creation of the test, and the test is revised every five years. Students are given two hours to complete the test, and institutions can choose to add additional questions to the end of the test that are specific to their students. Students receive scores in the range of 120-200, and students are given their score as well as how they ranked among their peers taking the test. Only correct answers are graded on the ETS Major Field tests. No penalty is given for incorrect or omitted answers (ETS Major Field Test Description, 2004).

Once again, this test covers much more information than needed for this dissertation and would not be appropriate to administer to students for the purpose of assessing first-year topics.

2.6.3.1 Reliability of the ETS Major Field Test in Computer Science

The reliability estimates for the Major Field test were computed using an internal consistency method and are estimated at .87 for the 60 items on the test (ETS Reliability 2004).

2.6.3.2 Validity Estimates for the ETS Major Field Test in Computer Science

Unfortunately, ETS does not provide any validity data about its Major Field Tests. This is disappointing and disconcerting. If there are schools that are using this test as part of a final-year assessment of their students, some indication of the appropriateness of this test for that task should be given, or at the very least demanded by those faculty that use it.

2.7 Analysis of AP Exam Data

An experiment was run to see if the grades students received on the AP exam correlated with their performance in the introductory courses at the University at Buffalo (UB). The results of the experiment are presented here.

We looked for a correlation between a student's AP exam score and the final letter grade the student received in CSE 115, which is the first year, first semester course for majors offered at UB, the CS1 course. The AP exam is scored on an ordinal scale of 1 to 5, with 5 being the highest grade attainable (AP 2003). Students' letter grades in CSE 115 can be A, A-, B+, B, B-, C+, C, C-, D+, D, or F. There is an option for students to resign courses during the semester; students who resign are given a grade of R. Students who resigned the course and earned a grade of R were omitted from this analysis. The Spearman rank-order correlation test was used, since we have strictly ordinal data in this analysis.

The first group examined consisted of those students who took the AP Computer Science A exam and later took CSE 115. The analysis produced a significant correlation between the students' AP exam score and their overall grade in CSE115, $r_S(49) = .42$, $p < .01$.[6]

In the second analysis, students who took the AP Computer Science AB exam and then took CSE 115 were examined. This analysis showed that there was not a significant correlation between the students' AP exam score and their overall grade in CSE 115, $r_s = .21$, $n = 27$, $p > .05$.[7]

2.7.1.1 Discussion

The results for the AP Computer Science A exam show a correlation with CS1 grade, indicating, for example, that a high grade on the AP exam will indicate a high

[6] For statistical reporting purposes, $r_S(\mathrm{F}) = \mathrm{m}$ reports the information about the correlation coefficient. r_s is the abbreviation indicating that we are using the Spearman rank-order correlation test. F represents the degree of freedom, which is one less than the total number of data points in the sample. m is the actual correlation coefficient. Positive numbers represent a direct correlation, while negative numbers represent an inverse correlation. Recall that for any statistical analysis, *p* values that are less than .05 are considered statistically significant results at the 95% confidence level. Results that have *p* values less than .01 are considered statistically significant at the 99% confidence level.

[7] For the results that are not statistically significant, the correlation coefficient is still reported in the same manner as above, but instead of reporting the degree of freedom, we simply report the size of the sample data, represented by the letter *n*.

grade in CS1. However, this type of correlation is not useful, because the AP exam is administered at the end of an academic year of high school study. It has not been proven reliable or valid except for use as a predictor for success in CS1 course, or CS1-CS2 sequence, in the case of the AB exam. We are looking for an assessment of CS1 knowledge to be administered after the completion of CS1 in a higher education setting.

Another shortcoming of the AP exam is that the questions on the exam are given using the language C++ and test student knowledge of the object-oriented paradigm. The test has recently switched languages to Java, but remains heavily focused on language. The goal for our assessment is one that is language-independent and paradigm-independent. The AP exam fails to serve our needs on both of those points.

The results of the AP Computer Science AB exam seem troubling in that they show a lack of correlation with the CS1 grade. This is likely due to the difference in emphasis of the AB exam, which focuses on material in both a CS1 curriculum as well as a CS2 curriculum as opposed to strictly a CS1 curriculum. We must recall that a non-significant result simply shows us that it is not possible to say definitively that a good grade on the AP Computer Science AB exam would lead to a good grade in CS1, or vice versa. Prediction of poor performance is also not possible. In some cases, there may be a correlation, but there is not enough significance in the trend to have a statistically significant result. It is also important to note, however, that this exam also has the shortcomings of the A exam in that the exam questions are specific to C++ and to object-oriented programming.

Overall, the results of the analysis show that one of the AP exams does correlate with student course grades in our institution's version of CS1. However, it is not a viable solution for the problem this dissertation seeks to address, due to the shortcomings of the AP exam discussed previously.

2.8 Conclusion

While there have been many investigations into both approaches to the introductory curriculum and predictors for success in the introductory courses, little has been done to address the issues of assessment of these skills for students as they move forward into the more advanced curriculum. Such assessment is necessary when looking at curricular innovation or predictors for success. An assessment instrument for this purpose needs to have a clearly defined set of objectives to assess. An assessment instrument for this purpose also needs to have evidence of reliability and validity.

There are several standardized and validated instruments available for assessment. However, none meet the criteria for applicability at the end of a CS1-CS2 sequence as described in the CC2001 document. This type of instrument is most desirable for future testing of curricular development and new exploration of predictors of success.

In the next chapter, the beginning of the process to create such an instrument will be discussed beginning with the analysis of the CC2001 document to create a list of topics from which to build an exam.

Chapter 3

Analysis of the CC2001 Computer Science Volume

This chapter analyzes relevant portions of CC2001, the curriculum document that serves as a basis for constructing my assessment instrument. Since I am most interested in the introductory curriculum for computer science, the sections of CC2001 that contain information important for that part of the curriculum will be explained in detail. Other sections of the CC2001 Computer Science volume that are related to other topics will be explained briefly for completeness and reference.

The remainder of the chapter discusses which topics will be considered for inclusion as potential topics for the assessment instrument. The process by which the topics were selected will be discussed.

3.1 Summary of CC2001

3.1.1 Structure of CC2001

CC2001 was produced by the members of the Joint Task Force on Computing Curricula 2001, created with the support of both the ACM and the IEEE Computer

Society. Three drafts of CC2001 were released in March 2000, February 2001, and August 2001. The final document was released on December 15, 2001. The main body of CC2001 is divided into thirteen chapters, two appendices, an acknowledgments section, and a bibliography.

3.1.2 CC2001 Sections Important to this Dissertation

Since this chapter of the dissertation serves as a reference to the entire CC2001 document, it is important to point out that the sections of CC2001 of greatest importance to this dissertation are Chapter 5, Chapter 7, Appendix A, and Appendix B, especially the subsections Syllabus and Units Covered.

3.1.3 Chapter 5: Overview of the CS Body of Knowledge

Chapter 5 of CC2001 begins by listing the fourteen knowledge areas making up the core computing science body of knowledge, which correspond to the knowledge focus groups discussed in Chapter 1 of CC2001 (see §3.1.6):

- Discrete Structures (DS)
- Programming Fundamentals (PF)
- Algorithms and Complexity (AL)
- Architecture and Organization (AR)
- Operating Systems (OS)
- Net-Centric Computing (NC)
- Programming Languages (PL)
- Human-Computer Interaction (HC)
- Graphics and Visual Computing (GV)
- Intelligent Systems (IS)
- Information Management (IM)
- Social and Professional Issues (SP)

- Software Engineering (SE)
- Computational Science and Numerical Methods (CN)

Each knowledge area is further broken down into knowledge units, which represent smaller topics within the more general knowledge area. The knowledge units are further broken down into specific topics. The details of this breakdown are given in the Appendices to CC2001 discussed further in §3.1.5 of this dissertation.

One of the task force's principle goals for CC2001 was to identify the fundamental core of the discipline that everyone earning a degree in computer science should have knowledge of. The designation of this core material is given at the knowledge-unit level. The report is careful to point out that simply teaching the core material does not suffice as a full curriculum and must be supplemented by other knowledge units that are identified as elective, as well as other material deemed appropriate by a particular institution.

The report also gives recommendations for the number of hours needed to cover a particular unit. These recommended hours correspond to lecture hours or actual classroom contact time, and are to be supplemented with outside classroom exercises where appropriate; they only represent a minimum recommendation for coverage.

3.1.4 Chapter 7: Introductory Courses

Chapter 7 of CC2001 discusses the proposed approaches to the introductory curriculum. CC2001 is careful not to make any recommendations about which approach is the best for the introductory curriculum, but rather to point out relative strengths and weaknesses in each of the approaches. In this way, institutions can decide which approach will work best.

Several questions are raised and answered by this chapter, including:

- Exactly where does programming fit into the introductory curriculum?
- How long should the introductory sequence be?
- How should we integrate discrete mathematics into the introductory curriculum?
- What should be our expectations of the introductory curriculum?

In an effort to address these questions, the report gives six models for the introductory curriculum: imperative-first, objects-first, functional-first, breadth-first, algorithms-first, and hardware-first.

3.1.4.1 Programming-first Approaches

The imperative-first, objects-first, and functional-first approaches are characterized as programming-first approaches to the introductory sequence. At the end of the introductory sequence using any of these three models, students are expected to be fairly proficient in programming, and the focus of their entire introductory sequence has been programming. The difference between the three programming-first approaches is what type of introductory material is presented earliest and what type of programming language is used for the introductory sequence.

Imperative-first is arguably the most traditional of the six introductory models presented. A version of this model was first proposed in Curriculum '78 (Committee on Computer Science Curriculum 1978). In this model, the focus at the beginning of the introductory sequence is placed on the "imperative aspects of a language: expressions, control structures, procedures and functions, and other central elements of the traditional procedural model" (Joint Task Force on Computing Curricula 2001: 29). The language used for this type of introductory sequence is not specified, but it should be one that enables students to explore these imperative aspects of a language before other language features. Some examples of this type of language could be Pascal, C, or the part of C++ that does not use classes.

For objects-first, the principles of object-oriented programming and design are emphasized from the beginning, with objects and inheritance introduced before the more traditional control structures (if-statements and loops). An object-oriented language is most appropriate in this model, with common choices being C++ and Java.

In functional-first, a functional language (such as Scheme or Lisp) is used. This type of course focuses on using functions as the primary unit of computation. Recursion and the use of lists as data structures are introduced early when using this approach.

3.1.4.2 Advantages and Disadvantages of Programming-First Approaches

The programming-first approaches to the introductory curriculum have benefits to the students. Since programming is such an essential skill for a computer scientist, emphasizing it as early as possible gives students plenty of exposure and experience. A programming-first approach is also an artifact of history: Many institutions adopted programming courses before having an entire computer science curriculum, and the earlier curriculum reports (Curriculum ’68 (Committee on Computer Science Curriculum 1968) and ’78 (Committee on Computer Science Curriculum 1978)) endorsed this type of introductory course.

However, CC2001’s Chapter 7 also notes that there are several shortcomings to using a programming-first approach. Limiting the focus to programming in the first year gives a rather limited view of the discipline of computer science and tends to focus on the syntax and use of a particular programming language. This focus on syntax comes at the price of the development of algorithmic skills. In order to make programming accessible to students at a basic level, many courses oversimplify the programming process and do not place enough emphasis on design, analysis, and testing of programs.

Another possible criticism of the current categorization of programming-first approaches is that it may not seem entirely clear where every programming language fits. For example, Prolog is not clearly an imperative, object-oriented, or functional language. Most would categorize Prolog as a declarative or a logic-programming language. Therefore, Prolog does not fit nicely into these categorizations.

When dealing with programming languages, one must also be cognizant of the constant evolution of programming languages and the changes in the attitude of the computing community toward one language or another at a particular time. Hayes (2006) paints an interesting picture of this in his article about how programming languages have changed over time. Even though Hayes lumps the discussion of all the languages in his discussion into the three main categories CC2001 covers (as well as Prolog in the logic category), he misses an opportunity to discuss the possibility of new paradigms being created. Hayes also does not adequately address the fact that some languages could be classified into many different categories.

This idea of a language crossing multiple paradigms impacts the CC2001 categorization of a school's curriculum as well. Many languages could be taught in different ways to illustrate different paradigms. A popular example of this is C++. One could teach C++ as a strictly imperative language or a strictly object-oriented language. A third alternative would be to teach C++ as a little of both. LISP is another language that could be taught in different ways. Although fundamentally functional, there are object-oriented extensions to LISP as well as imperative constructs available in LISP that would enable someone to teach the language using many paradigms.

However, one must remember that CC2001 only provides recommendations and details only some of the possible approaches to the curriculum. There are certainly other approaches that are valid, but the focus of CC2001 seems to be on the most popular approaches in use at institutions today.

3.1.4.3 Non-programming-first Approaches

An alternative to the programming-first approaches are the non-programming-first approaches: Breadth-first, Algorithms-first, and Hardware-first. Each of these approaches has a slightly different emphasis in the first few courses of the curriculum. Programming is certainly a part of each of these approaches, but it is not as central as it is in the programming-first approaches.

Breadth-first approaches focus on the breadth of the field of computer science, exposing students early to the many interesting facets of the field, and then introducing programming only after the first semester. This approach seeks to better integrate both programming and discrete mathematics into the introductory sequence. With this approach, students have a greater appreciation for the diversity of the field of computer science and are better prepared to decide if computer science is the field for them. The downside is that the breadth course is an additional course required at the beginning of the curriculum.

Algorithms-first approaches introduce the ideas of the introductory sequence using a non-executable pseudo-code rather than an actual programming language and environment. Students are expected to write and analyze algorithms that perform certain operations, but they will not run them on a machine to verify their results. Students move on to using an actual programming language in the second semester of study under this approach. The reported benefit of this approach is that students are not caught up with syntactic detail right from the beginning of the curriculum; rather, they build up algorithmic thinking and problem-solving skills. However, even with a pseudo-code, there is syntax that one must learn, albeit probably simpler than that of some of the modern programming languages. A disadvantage to this approach is that students do not get to see exactly what the computer can do for them, because they are focusing for the first semester on hand-tracing code and writing out programs never to be run on a computer.

Hardware-first approaches begin with students learning about computation at the machine level, using circuits, and eventually working up to registers and a working von Neumann machine. After an introduction to computing at the machine level, the second course in this sequence considers programming in a higher-level language. This course benefits those students that prefer to understand the entire process of computing down to the machine-level details up front. However, with the increased emphasis in the computing discipline on software and the detachment of programming from hardware through the use of more sophisticated virtual machines, this type of course might be better suited for a computer engineering program.

I have mentioned how the non-programming-first approaches address some of the shortcomings of the programming-first approaches. Even though these benefits are recognized in the non-programming-first approaches to the introductory curriculum, it is still more common to see institutions that follow a programming-first introductory sequence. It is for this reason that I will be focusing on the programming-first approaches for the work of this dissertation.

3.1.4.4 Concepts across All Approaches

Chapter 7 of CC2001 also summarizes the set of concepts that should be included in each introductory curriculum. These concepts are given in this chapter in Table 3-1. Also included in this chapter is Table 3-2 that shows the knowledge units that should be covered in an introductory curriculum.

Algorithmic Thinking		
Concept	*Description*	*Associated activities*
Algorithmic computation	Algorithms as models of computational processes; examples of important algorithms	Read and explain algorithms; reason about algorithmic correctness; use, apply, and adapt standard algorithms; write algorithms
Algorithmic efficiency and resource usage	Simple analysis of algorithmic complexity; evaluation of tradeoff considerations; techniques for estimation and measurement	Estimate time and space usage; conduct laboratory experiments to evaluate algorithmic efficiency
Programming Fundamentals		
Concept	*Description*	*Associated activities*
Data models	Standard structures for representing data; abstract (described by a model) and concrete (described by an implementation) description	Read and explain values of program objects; create, implement, use, and modify programs that manipulate standard data structures
Control structures	Effects of applying operations to program objects; what an operation does (described by a model); how an operation does it (described by an implementation)	Read and explain the effects of operations; construct programs to implement a range of standard algorithms
Order of execution	Standard control structures: sequence, selection, iteration; function calls and parameter passing	Make appropriate use of control structures in the design of algorithms and then implement those structures in executable programs
Encapsulation	Indivisible bundling of related entities; client view based on abstraction and information-hiding; implementer view based on internal detail	Use existing encapsulated components in programs; design, implement, and document encapsulated components
Relationships among encapsulated components	The role of interfaces in mediating information exchange; responsibilities of encapsulated components to their clients; the value of inheritance	Explain and make use of inheritance and interface relationships; incorporate inheritance and interfaces into the design and implementation of programs
Testing and Debugging	The importance of testing; debugging strategies	Design effective tests; identify and correct coding and logic errors
Computing environments		
Concept	*Description*	*Associated activities*
Layers of abstraction	Computer systems as a hierarchy of virtual machines	Describe the roles of the various layers in the virtual machine hierarchy
Programming languages and paradigms	Role of programming languages; the translation process; the existence of multiple programming paradigms	Outline the program translation process; identify at least two programming paradigms and describe their differences
Basic hardware and data representation	Rudiments of machine organization; machine-level representation of data	Explain basic machine structure; show how different kinds of information can be represented using bits
Tools	Compilers, editors, debuggers, and other components of programming environments	Use tools successfully to develop software

Table 3-1: Figure 7-1 of CC2001 describing the concepts that should be covered in an introductory curriculum

Units for which all topics must be covered:
DS1. Functions, relations, and sets DS2. Basic logic DS4. Basics of counting DS6. Discrete probability PF1. Fundamental programming constructs PF4. Recursion PL1. Overview of programming languages PL2. Virtual machines PL4. Declarations and types PL5. Abstraction mechanisms SP1. History of computing
Units for which only a subset of the topics must be covered: DS3. Proof techniques - The following topics should be covered: The structure of formal proofs; proof techniques: direct, counterexample, contraposition, contradiction; mathematical induction PF2. Algorithms and problem-solving – The following topics should be covered: Problem solving strategies; the role of algorithms in the problem-solving process; the concept and properties of algorithms; debugging strategies PF3. Fundamental data structures – The following topics should be covered: Primitive types; arrays; records; strings and string processing; data representation in memory; static, stack, and heap allocation; runtime storage management; pointers and references; linked structures AL1. Basic algorithmic analysis – The following topics should be covered: Big O notation; standard complexity classes; empirical measurements of performance; time and space tradeoffs in algorithms AL3. Fundamental computing algorithms – The following topics should be covered: Simple numerical algorithms; sequential and binary search algorithms; quadratic and O(N log N) sorting algorithms; hashing; binary search trees AR1. Digital logic and digital systems – The following topics should be covered: Logic gates; logic expressions PL6. Object-oriented programming – The following topics should be covered: Object-oriented design; encapsulation and information-hiding; separation of behavior and implementation; classes, subclasses, and inheritance; polymorphism; class hierarchies SE1. Software design – The following topics should be covered: Fundamental design concepts and principles; object-oriented analysis and design; design for reuse SE2. Using APIs – The following topics should be covered: API programming; class browsers and related tools; programming by example; debugging in the API environment SE3. Software tools and environments – The following topics should be covered: Programming environments; testing tools SE5. Software requirements and specifications – The following topics should be covered: Importance of specification in the software process SE6. Software validation – The following topics should be covered: Testing fundamentals; test case generation

Table 3-2: Figure 7-2 from CC2001 knowledge units and topics that are covered by all six introductory tracks

For each of the approaches presented in Chapter 7 of CC2001, details are given about their relative strengths and weaknesses. Sample course syllabi are available for all of the approaches in Appendix B of CC2001.

3.1.5 Appendix A: CS Body of Knowledge and Appendix B: Course Descriptions

Appendix A of CC2001 starts out by reiterating the ideas behind the knowledge areas given previously in the report. Each knowledge area is then broken down into its knowledge units, and each knowledge unit is broken further into topics. For each knowledge area, a general description is given for what the area is and why it is considered important to the field.

For each knowledge unit, it is indicated whether they are core knowledge units and how much time (in hours) is needed to cover the core material. The topics are listed that make up the knowledge unit. Also given are a set of learning objectives that correspond with the knowledge unit’s topics. In this chapter of this dissertation, §3.2, discusses how this time information and topics were used to create a set of core topics that was used as the set of core topics for the exam. Chapter 5 of this dissertation discusses the how the learning objectives given in Appendix A of CC2001 map onto the final set of topics used to create the exam.

Appendix B of CC2001 gives the sample syllabi for courses described in the introductory, intermediate, and advanced courses chapters. Each course is given a number and title. Within each course description there is a brief explanation of the course and its prerequisites. Then the sample syllabus for the course is given followed by the knowledge units that are covered and the number of hours that the course should use to cover those knowledge units.

3.1.6 The rest of CC2001: Chapters 1 – 4, 6, and 8 – 13

Chapter 1 of CC2001 describes the mission of the CC2001 committee and explains that the document is the first in a series of curricular models for computing. The Computer Science volume serves as a model curriculum for computer science degree programs. Three other curricular models have been developed: computer engineering, information systems, and software engineering. An information technology curriculum is currently in draft form. As stated earlier, for the purposes of this dissertation, “CC2001” refers only to the computer science volume of the curriculum document.

This chapter also describes the process that was undertaken to revise the curriculum. The task force felt it important and necessary to involve many from the computer science community to gain perspective and expertise from a large range of individuals. This involvement from the larger community was facilitated by the creation of fourteen knowledge focus-groups, one for each of the knowledge units contained in the final version of CC2001. These were each charged with the creation of a document that would help the task force prepare the complete computer science body of knowledge. There was also a pedagogy focus group (PFG), whose responsibility was to “consider curricular issues across computer science as a whole" (Joint Task Force on Computing Curricula 2001: 3).

Chapter 2 discusses the history of past curriculum efforts and how the task force used community reaction to the last curriculum (CC1991) to update and create the new curriculum. The three main reactions were:

- Knowledge units are not as useful as course or curriculum designs.
- There is strong support for a more concrete definition of a minimal core.
- Curriculum reports should pay greater attention to accreditation criteria for computer science programs.

Chapter 3 examines how changes in the world and in technology since the last curriculum report impact this curriculum. The chapter points out that such things as the growth of the World Wide Web and applications associated with it require changes to the curriculum. Also included is how cultural changes across the world, such as the increase in the number of homes using computers and having Internet access, have made computing a much different discipline now than when CC1991 was created.

Chapter 4 of CC2001 discusses the principles that guided the work of the task force. These principles, reprinted here from pages 12-13 of CC2001 are:

1. Computing is a broad field that extends well beyond the boundaries of computer science.
2. Computer science draws its foundations from a wide variety of disciplines.

3. The rapid evolution of computer science requires an ongoing review of the corresponding curriculum.
4. Development of a computer science curriculum must be sensitive to changes in technology, new developments in pedagogy, and the importance of lifelong learning.
5. CC2001 must go beyond knowledge units to offer significant guidance in terms of individual course design.
6. CC2001 should seek to identify the fundamental skills and knowledge that all computing students must possess.
7. The required body of knowledge must be made as small as possible.
8. CC2001 must strive to be international in scope.
9. The development of CC2001 must be broadly based.
10. CC2001 must include professional practice as an integral component of the undergraduate curriculum.
11. CC2001 must include discussions of strategies and tactics for implementation along with high-level recommendations.

Chapter 6 of CC2001 introduces the implementation strategies for model curricula. The curriculum is divided into three course levels: introductory, intermediate, and advanced. For the introductory level, six different implementation strategies are proposed and discussed later in the report (but earlier in this dissertation; see §3.1.4).

For the intermediate level, four different approaches are proposed: topic-based, compressed, systems-based, and web-based. The advanced level consists of courses that are designed to complete the curriculum.

The CC2001 report proposes that a curriculum can be developed using any of the introductory course models, followed by any of the intermediate course models, and finishing with the advanced courses. Chapter 6 of CC2001 ends with two examples of selecting an introductory and intermediate approach and how those approaches will help cover the computer science core.

Chapter 8 of CC2001 discusses approaches to the intermediate curriculum and also gives sample course syllabi (in Appendix B of CC2001) for the different approaches. These approaches are:

- A traditional approach in which each course addresses a single topic
- A compressed approach that organizes courses around broader themes
- An intensive systems-based approach
- A web-based approach that uses networking as its organizing principle

Chapter 9 of CC2001 discusses additions to the base curriculum presented in Chapters 7 and 8 to complete the curriculum. Topics that are discussed to “fill out” a curriculum are mathematical rigor, the scientific method, familiarity with applications, communications skills, and working in teams. Several sample courses are presented that help to “fill out” the topics given above that also fit into the knowledge areas and cover some of the elective knowledge units. Another type of course that is discussed and recommended here is a project course that forces students to complete a large-scale computing project, usually working in teams.

Chapter 9 of CC2001 also gives a few complete sample curricula. The first example is for a research university in the United States. The second is a discipline-based model, used primarily in countries outside of the United States and Canada, where students do not take a large portion of their coursework at the university-level as general education requirements, but rather focus almost entirely on their field of study. The third is a model for a small department, one that has less than five or six faculty. The last is a model for two-year colleges, whose students are expected to transfer to a four-year institution for completion of a bachelor's degree.

Chapter 10 of CC2001 discusses the integration of professional practice into the curriculum. It has become increasingly evident that employers need and want certain skills out of recent college and university graduates, and it is necessary to try to incorporate some of these skills into the educational process. Some current models for incorporating these ideas into the curriculum are presented in Chapter 10, as well as discussion of how a department can support professional practice within the curriculum, and assess whether its students are incorporating these ideas appropriately in their work.

Chapter 11 of CC2001 discusses the general characteristics that students with a computer science degree should possess. This includes their capabilities and skills, as well as their ability to cope with this ever-changing field. The last element presented in this chapter is a set of standards for benchmarking a student's level of achievement with the curricular goals. These standards give minimum standards (called threshold standards) that all graduates should meet as well as more advanced standards (called modal standards) to encourage achievement beyond the minimum. For example, a threshold standard is "Demonstrate a requisite understanding of the main body of knowledge and theories of computer science", while a modal standard is "Demonstrate a sound understanding of the main areas of the body of knowledge and the theories of computer science, with an ability to exercise critical judgment across a range of issues”.

Chapter 12 of CC2001 seeks to give suggestions for how computing and computing ideas can be presented to students across all academic disciplines as well as computer science’s place in the field of academics.

Chapter 13 of CC2001 is a concluding chapter about how this report should be used by a local institution and gives suggestions for what types of resources (both machinery and personnel) are needed to make any implementation a success.

3.2 Analysis of the Programming-First Approaches to the Introductory Curriculum

This section shows the intersection of topics of the three programming-first approaches to the introductory curriculum. First examined are the sample syllabi for the programming-first CS1 courses and then the syllabi for both CS1 and CS2. The CS1 courses have a small overlap; there is a larger overlap in topics when both CS1 and CS2 are considered. As a consequence, our assessment instrument addresses both CS1 and CS2.

I conclude with a discussion of inconsistencies between the wording of the course descriptions and the descriptions of the topical coverage contained elsewhere in CC2001.

3.2.1 Two- or Three-Semester Sequence

Appendix B of CC2001 gives course descriptions for both two-semester and three-semester versions of introductory courses in both the programming-first and non-programming-first approaches. There are both two- and three-semester models for imperative-first and objects-first. However, for functional-first, there is only a two-semester model. Therefore, my efforts focused only on two-semester course models.

3.2.2 Justification for Programming-First

Of the six approaches to the introductory curriculum endorsed by CC2001 (three programming-first approaches (imperative-first, objects-first, functional-first) and three non-programming-first approaches (breadth-first, algorithms-first, and hardware-first)), I have chosen to look for commonalities among only the programming-first approaches and create an assessment for these types of courses. There are two main reasons for this.

The first reason is that many institutions use the programming-first model for their introductory sequence of courses and will continue to do so for the foreseeable future. This model has proven extremely durable and long-lasting as a model for the introductory curriculum.

The second reason is the difference in emphasis of the programming-first approaches from the non-programming-first approaches. The non-programming-first approaches each emphasize a different aspect of the computing discipline (hardware, algorithms, or breadth coverage). These three approaches are discussed in depth in CC2001 and summarized in §3.1.4.3. The three foci of the non-programming-first approaches do not overlap with each other and none have a significant programming component. Focusing only on the programming-first approaches yields a better intersection of topic coverage and, with the popularity of the programming-first approaches, an assessment that will be widely applicable at various institutions.

Lastly, the programming-first approach is the approach used at the University at Buffalo for CS1 and CS2. Since the University at Buffalo is the most readily accessible population to serve as test subjects for the exam, it makes the most sense to create an exam that can be administered to those students.

3.2.3 Intersection of Topics for CS1

The intersection of topics among the CS1 courses described by CC2001 is small and does not yield the amount of coverage that should be present in a true assessment of a semester’s worth of work.

This is an unfortunate result because an assessment of strictly CS1 could be useful in a number of contexts. First, those seeking to look at changes made to a particular CS1 course could tell if the changes had an impact on student performance. Second, those interested in predictors of success often focus strictly on CS1. A validated assessment for the end of a CS1 course provides a metric by which to measure success. However, as will be discussed in this section, a CS1 assessment is simply not possible with amount of topic coverage common to all three programming-first approaches.

3.2.3.1 Knowledge Area Analysis

Let us first look at the knowledge areas covered by each of the three programming-first CS1 courses. The knowledge areas covered by a specific course are given in Appendix B's course descriptions labeled *Units covered*. These findings are summarized in Table 3-3. An **X** in a row indicates that the knowledge area is covered in that course.

Knowledge Area[8]	Imperative-first[9] CS1	Objects-first CS1	Functional-first CS1
Algorithms and Complexity (AL)	**X**	**X**	**X**
Programming Fundamentals (PF)	**X**	**X**	**X**
Programming Languages (PL)	**X**	**X**	**X**
Social and Professional Issues (SP)	**X**	**X**	**X**
Software Engineering (SE)	**X**	**X**	**X**
Graphics and Visual Computing (GV)	**X**	**X**	
Architecture and Organization (AR)	**X**		
Discrete Structures (DS)			**X**
Operating Systems (OS)			**X**
Computational Science (CN)			
Human-Computer Interaction (HC)			
Information Management (IM)			
Intelligent Systems (IS)			
Net-Centric Computing (NC)			

Table 3-3: Knowledge Area Coverage for Programming-first CS1 courses

It is interesting to note that the knowledge area for Architecture and Organization is present in the imperative-first approach and no others, while the knowledge area for

[8] Any information contained in this chapter that refers to CC2001 has been taken verbatim from the document. However, the order has been changed so that topics covered by all three approaches are listed first, followed by topics covered in only two approaches, etc. All spellings, abbreviations, titles, and capitalization have been copied from that document.
[9] Note that for the remainder of the tables in this chapter, Imperative-first, Objects-first, and Functional-first will be abbreviated as IF, OF, and FF respectively.

Operating Systems is present for functional-first and no others. The reasons come right from the course descriptions given in Appendix B. The imperative-first approach puts emphasis on the machine representation of data as part of CS1, as well as discussion of the von Neumann architecture. The other two approaches do not include such topics. The functional-first approach includes information about concurrency, because many functional languages have built-in mechanisms for handling concurrency that can therefore easily be discussed early.

Table 3-3 shows us that the common knowledge areas in CS1 across all three approaches are: Programming Fundamentals, Algorithms and Complexity, Programming Languages, Social and Professional Issues, and Software Engineering.

3.2.3.2 Knowledge Unit Analysis

Each knowledge area is divided into more detailed knowledge units. Table 3-4 through Table 3-8 summarize, for each knowledge area, which knowledge units are covered by the CS1 courses. Once again, this information is in the *Units covered* section of each sample syllabus in Appendix B of CC2001.

Knowledge Unit	IF CS1	OF CS1	FF CS1
PF1. Fundamental programming constructs	**X**	**X**	**X**
PF2. Algorithms and problem-solving	**X**	**X**	**X**
PF3. Fundamental data structures	**X**	**X**	**X**
PF4. Recursion		**X**	**X**
PF5. Event-driven programming			

Table 3-4: Programming Fundamentals Knowledge Unit Coverage for CS1

Knowledge Unit	IF CS1	OF CS1	FF CS1
AL3. Fundamental computing algorithms	**X**	**X**	**X**
AL5. Basic computability	**X**	**X**	**X**
AL1. Basic algorithmic analysis			**X**
AL2. Algorithmic strategies			**X**
AL4. Distributed Algorithms			
AL6. The complexity classes P and NP			
AL7. Automata theory			
AL8. Advanced arithmetic analysis			
AL9. Cryptographic algorithms			
AL10. Geometric algorithms			
AL11. Parallel algorithms			

Table 3-5: Algorithms and Complexity Knowledge Unit Coverage in CS1

Knowledge Unit	IF CS1	OF CS1	FF CS1
PL4. Declarations and types	**X**	**X**	**X**
PL5. Abstraction mechanisms	**X**	**X**	**X**
PL1. Overview of programming languages	**X**		**X**
PL6. Object-oriented programming	**X**	**X**	
PL7. Functional programming			**X**
PL2. Virtual machines			
PL3. Introduction to language translation			
PL8. Language translation systems			
PL9. Type systems			
PL10. Programming language semantics			
PL11. Programming language design			

Table 3-6: Programming Languages Knowledge Unit Coverage in CS1

Knowledge Unit	IF CS1	OF CS1	FF CS1
SP1. History of computing	**X**	**X**	**X**
SP5. Risks and liabilities of computer-based systems		**X**	
SP2. Social context of computing			
SP3. Methods and tools of analysis			
SP4. Professional and ethical responsibilities			
SP6. Intellectual property			
SP7. Privacy and civil liberties			
SP8. Computer crime			
SP9. Economic issues in computing			
SP10. Philosophical frameworks			

Table 3-7: Social and Professional Issues Knowledge Unit Coverage in CS1

Knowledge Unit	IF CS1	OF CS1	FF CS1
SE1. Software design	**X**	**X**	**X**
SE3. Software tools and environments	**X**	**X**	**X**
SE2. Using APIs		**X**	
SE5. Software requirements and specifications	**X**		
SE6. Software validation	**X**		
SE4. Software processes			
SE7. Software evolution			
SE8. Software project management			
SE9. Component-based computing			
SE10. Formal methods			
SE11. Software reliability			
SE12. Specialized systems development			

Table 3-8: Software Engineering Knowledge Unit Coverage in CS1

3.2.3.3 Analysis of Knowledge Units in Intersection

Ten knowledge units are covered by all three programming-first approaches:

- PF1. Fundamental programming constructs
- PF2. Algorithms and problem-solving
- PF3. Fundamental data structures
- AL3. Fundamental computing algorithms
- AL5. Basic computability
- PL4. Declarations and types
- PL5. Abstraction mechanisms
- SP1. History of computing
- SE1. Software design
- SE3. Software tools and environments

Each of these knowledge units is covered in all three course models. What still needs to be investigated is whether these knowledge units are covered to the same degree in each course and whether these topics make up a large enough portion of the courses to be considered an appropriate assessment of the topics covered in those courses.

A first approximation to determining the degree of coverage is to consider the percentage of the common knowledge units to the total. These results are summarized in Table 3-9.

	IF CS1	OF CS1	FF CS1
Total number of Knowledge Units Covered in CS1	17	15	17
Number of Knowledge Units in Intersection	10	10	10
Percentage Covered by Intersection	59%	67%	59%
Percentage Not Covered by Intersection	41%	33%	41%

Table 3-9: Percentages of Knowledge Units Covered by Intersection

We see that there is a range of 59% - 67% of total knowledge-unit coverage for the three approaches. These figures have been computed using only the actual number of knowledge units covered, not the time spent on each knowledge unit or the actual sub-topics from each knowledge unit covered by the courses. Let us consider these percentages next.

Each course has been designed for forty course-lecture-hours. The breakdown of hours per knowledge unit is given in the *Units covered* section of the sample course syllabi in Appendix B. Recall from the information given in CC2001 that these lecture hours are given to represent actual in-class time in a lecture-style course. The results of this analysis are summarized in the Table 3-10.

	IF CS1	OF CS1	FF CS1
Total number of lecture hours in CS1	40	40	40
Number of lecture hours covered by knowledge units in Intersection	28	24	21
Percentage Covered by Intersection	70%	60%	53%
Percentage Not Covered by Intersection	30%	40%	47%

Table 3-10: Percentages of Total Course Lecture Hours Covered by Intersection

The amount of hours varies more widely than the knowledge units. The intersection only makes up a little more than half of the lecture time for functional-first, while it is more than two-thirds of the lecture time for imperatives-first.

3.2.3.4 Conclusions about CS1 intersection

If we focus only on the intersection in CS1, as much as 41% of the knowledge units presented in a course can be missing and as much as 47% of the course lecture hours can be missing from both the intersection of topics and an exam built from that intersection. Analysis did not proceed down to the topic level for CS1. Given the percentage of knowledge units missing from the intersection and the fact that the topics are simply refinements of a knowledge unit, such an analysis would not have yielded any better results for an intersection. An assessment instrument that tests for only a little more than half of the course content does not seem to be an appropriate test of a student's abilities with the course material.

3.2.4 Intersection of Topics for CS1 and CS2

Given the fact that this intersection is not really large enough to make a meaningful assessment for each of these three approaches to the introductory curriculum, let us next consider an intersection of topics for the entire first year of introductory material, CS1 and CS2.

Since it is important to focus only on implementations that all three programming-first approaches share, and there is no three-semester implementation for functional-first, we will only consider the two-semester sequences of the programming-first approaches while looking for the intersection in CS1 and CS2.

3.2.4.1 Knowledge Area Analysis

Table 3-11 shows the knowledge areas common to the first two semesters (CS1 & CS2) of all three programming-first approaches.

Knowledge Area	IF CS1 & CS2	OF CS1 & CS2	FF CS1 & CS2
Algorithms and Complexity (AL)	**X**	**X**	**X**
Programming Fundamentals (PF)	**X**	**X**	**X**
Programming Languages (PL)	**X**	**X**	**X**
Social and Professional Issues (SP)	**X**	**X**	**X**
Software Engineering (SE)	**X**	**X**	**X**
Discrete Structures (DS)	**X**		**X**
Graphics and Visual Computing (GV)	**X**	**X**	
Architecture and Organization (AR)	**X**		
Human-Computer Interaction (HC)		**X**	
Operating Systems (OS)			**X**
Computational Science (CN)			
Information Management (IM)			
Intelligent Systems (IS)			
Net-Centric Computing (NC)			

Table 3-11: Knowledge Area Coverage for Programming-first CS1 & CS2

Although a few new knowledge areas are now included, the intersection is identical to that of CS1 above.

3.2.4.2 Knowledge Unit Analysis

Since the knowledge area intersection was the same, it is important to determine which knowledge units are covered by all three approaches in both CS1 and CS2. These results can be seen in Table 3-12 through Table 3-16.

Knowledge Unit	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
PF1. Fundamental programming constructs	**X**	**X**	**X**
PF2. Algorithms and problem-solving	**X**	**X**	**X**
PF3. Fundamental data structures	**X**	**X**	**X**
PF4. Recursion	**X**	**X**	**X**
PF5. Event-driven programming		**X**	**X**

Table 3-12: Programming Fundamentals Knowledge Unit Coverage for CS1 and CS2

Knowledge Unit	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
AL1. Basic algorithmic analysis	**X**	**X**	**X**
AL3. Fundamental computing algorithms	**X**	**X**	**X**
AL5. Basic computability	**X**	**X**	**X**
AL2. Algorithmic strategies		**X**	**X**
AL4. Distributed Algorithms			
AL6. The complexity classes P and NP			
AL7. Automata theory			
AL8. Advanced arithmetic analysis			
AL9. Cryptographic algorithms			
AL10. Geometric algorithms			
AL11. Parallel algorithms			

Table 3-13: Algorithms and Complexity Knowledge Unit Coverage in CS1 and CS2

Knowledge Unit	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
PL1. Overview of programming languages	X	X	X
PL2. Virtual machines	X	X	X
PL4. Declarations and types	X	X	X
PL5. Abstraction mechanisms	X	X	X
PL6. Object-oriented programming	X	X	X
PL3. Introduction to language translation	X		
PL7. Functional programming			X
PL8. Language translation systems			
PL9. Type systems			
PL10. Programming language semantics			
PL11. Programming language design			

Table 3-14: Programming Languages Knowledge Unit Coverage in CS1 and CS2

Knowledge Unit	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
SP1. History of computing	X	X	X
SP5. Risks and liabilities of computer-based systems		X	
SP2. Social context of computing			
SP3. Methods and tools of analysis			
SP4. Professional and ethical responsibilities			
SP6. Intellectual property			
SP7. Privacy and civil liberties			
SP8. Computer crime			
SP9. Economic issues in computing			
SP10. Philosophical frameworks			

Table 3-15: Social and Professional Issues Knowledge Unit Coverage in CS1 and CS2

Knowledge Unit	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
SE1. Software design	X	X	X
SE2. Using APIs	X	X	X
SE3. Software tools and environments	X	X	X
SE5. Software requirements and specifications	X	X	X
SE6. Software validation	X	X	X
SE4. Software processes			
SE7. Software evolution			
SE8. Software project management			
SE9. Component-based computing			
SE10. Formal methods			
SE11. Software reliability			
SE12. Specialized systems development			

Table 3-16: Software Engineering Knowledge Unit Coverage in CS1 and CS2

Notice that there are now 18 instead of only 10 knowledge units common to all three programming-first approaches. The new knowledge units included for the CS1-CS2 intersection are:

- PF4. Recursion
- AL1. Basic algorithmic analysis
- PL1. Overview of programming languages
- PL2. Virtual machines
- PL6. Object-oriented programming
- SE2. Using APIs
- SE5. Software requirements and specifications
- SE6. Software validation

3.2.4.3 Analysis of Knowledge Units in the Intersection

Once again, it is important to consider the percentage of coverage these knowledge units represent. These results are summarized in the Table 3-17.

	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Total number of Knowledge Units Covered in CS1	23	23	23
Number of Knowledge Units in Intersection	18	18	18
Percentage Covered by Intersection	78%	78%	78%
Percentage Not Covered by Intersection	22%	22%	22%

Table 3-17: Percentages of Knowledge Units Covered by Intersection

These results are promising, showing that in fact there are an equal number of knowledge units covered by each of the three programming-first CS1s and CS2s. Furthermore, the intersection comprises over three-quarters of the knowledge units covered in the courses.

The analysis of how many lecture hours are covered by the knowledge units in the intersection is even more promising. The results of this analysis are presented in Table 3-18.

	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Total number of lecture hours in CS1	80	80	80
Number of lecture hours covered by knowledge units in Intersection	69	70	65
Percentage Covered by Intersection	86%	88%	81%
Percentage Not Covered by Intersection	14%	12%	19%

Table 3-18: Percentages of Total Course Lecture Hours Covered by Intersection

With this new intersection of knowledge units from CS1 and CS2, there is very little lecture time for each course that is devoted to topics outside of the intersection. In fact, for all three approaches, over 80% of the lecture time is spent on material in the intersection. This intersection seems to be a much stronger basis from which to extract topics for the assessment instrument.

In fact, one notices that objects-first leads in the amount of course coverage devoted to the intersection of topics that are common to all three approaches. This could be used as an argument in favor of objects-first, because it appears to have the most time devoted to topics that CC2001 deems to be the core of CS1-CS2.

3.2.4.4 Analysis of Topics from the Knowledge Unit Intersection

There is one more level of description that defines a knowledge area: the topics included in each knowledge unit. It is important to look at these topics to ensure that the three approaches are not covering vastly different topics within the same knowledge unit. Appendix A of CC2001 gives the listing of topics that should be covered for each of the knowledge units.

The first attempt to identify the topics covered in each of the courses involved looking at the section labeled *Syllabus* in the sample course descriptions given in Appendix B of CC2001. The syllabus is described as a “bulleted list providing an outline of the topics covered” (Joint Task Force on Computing Curricula 2001: 159). Quickly browsing these topics, they seem to correspond with the topics listed for each of the knowledge units.

For each knowledge unit in the intersection, I show which topics from the knowledge unit are covered by each of the three programming-first CS1-CS2 sequences (see Table 3-19 through Table 3-36).[10] As an alternative view, the next set of tables (Table 3-37 through Table 3-42) show which topics are covered by all three approaches, which topics are covered by only two of the three approaches, which topics are covered by only one of the approaches, and which topics are covered by none of the approaches.

PF1. Fundamental Programming Constructs Topics	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Basic syntax and semantics of a higher-level language	**X**	**X**	**X**
Conditional and iterative control structures	**X**	**X**	**X**
Functions and parameter passing	**X**	**X**	**X**
Variables, types, expressions, and assignment	**X**	**X**	**X**
Simple I/O	**X**		**X**
Structured decomposition	**X**		**X**

Table 3-19: PF1. Fundamental Programming Constructs topics covered in programming-first CS1-CS2

[10] Please note that since these tables were created using this technique of skimming the syllabus sections of CC2001, several of the tables may seem to be missing topics. The discussion of this fact and resolutions of some of the apparent ambiguities are discussed in §3.2.5 – 3.2.8.

PF2. Algorithms and problem-solving Topics	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
The concept and properties of algorithms	**X**	**X**	**X**
Implementation strategies for algorithms	**X**	**X**	**X**
Problem-solving strategies	**X**	**X**	**X**
Debugging strategies	**X**		**X**
The role of algorithms in the problem-solving process			**X**

Table 3-20: PF2. Algorithms and Problem-Solving topics covered in programming-first CS1-CS2

PF3. Fundamental Data Structures	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Arrays	**X**	**X**	**X**
Linked structures	**X**	**X (use of, not implementation)**	**X**
Strings and string processing	**X**	**X**	**X**
Implementation strategies for graphs and trees	**X**	**X (introduction)**	
Implementation strategies for stacks, queues, and hash tables	**X**	**X (use of, not implementation)**	
Pointers and references	**X**		**X**
Primitive types	**X**		**X**
Records	**X**		**X**
Strategies for choosing the right data structure	**X**		**X**
Data representation in memory	**X**		
Static, stack, and heap allocation	**X**		
Runtime storage management	**X**		

Table 3-21: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2

PF4. Recursion	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
The concept of recursion	**X**	**X**	**X**
Implementation of recursion	**X**	**X**	
Divide-and-conquer strategies	**X**		**X**
Recursive backtracking	**X**		**X**
Recursive mathematical functions	**X**		**X**
Simple recursive procedures	**X**		**X**

Table 3-22: PF4. Recursion topics covered in programming-first CS1-CS2

AL1. Basic algorithmic analysis	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Asymptotic analysis of upper and average complexity bounds	**X**		**X**
Big O, little o, omega, and theta notation	**X (Big O only)**		**X (Big O only)**
Empirical measurements of performance	**X**		**X**
Standard complexity classes	**X**		**X**
Identifying differences among best, average, and worst case behaviors			
Time and space tradeoffs in algorithms			
Using recurrence relations to analyze recursive algorithms			

Table 3-23:AL1. Basic Algorithmic Analysis topics covered in programming-first CS1-CS2

AL3. Fundamental computing algorithms	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Sequential and binary search algorithms	**X**		**X**
Binary search trees	**X**		
Hash tables, including collision-avoidance strategies	**X**		
O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)	**X**		
Quadratic sorting algorithms (selection, insertion)	**X**		
Simple numerical algorithms			**X**
Depth- and breadth-first traversals			
Minimum spanning tree (Prim's and Kruskal's algorithms)			
Representations of graphs (adjacency list, adjacency matrix)			
Shortest-path algorithms (Dijkstra's and Floyd's algorithms)			
Topological sort			
Transitive closure (Floyd's algorithm)			

Table 3-24: AL3. Fundamental Computing Algorithms topics covered in programming-first CS1-CS2

AL5. Basic computability	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Tractable and intractable problems	**X**		**X**
Uncomputable functions	**X**		**X**
Context-free grammars			
Finite-state machines			
The halting problem			
Implications of uncomputability			

Table 3-25: AL5. Basic Computability topics covered in programming-first CS1-CS2

PL1. Overview of programming languages	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Brief survey of programming paradigms: Procedural languages, Object-oriented languages, Functional languages, Declarative, non-algorithmic languages, Scripting languages	**X**		**X**
History of programming languages	**X**		**X**
The effects of scale on programming methodology			

Table 3-26: PL1. Overview of Programming Languages topics covered in programming-first CS1-CS2

PL2. Virtual machines	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
The concept of a virtual machine	**X**		**X**
Hierarchy of virtual machines	**X**		**X**
Intermediate languages	**X**		**X**
Security issues arising from running code on an alien machine			**X**

Table 3-27: PL2. Virtual Machines topics covered in programming-first CS1-CS2

PL4. Declarations and types	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
The conception of types as a set of values with together a set of operations			
Declaration models (binding, visibility, scope, and lifetime)			
Overview of type-checking			
Garbage collection			

Table 3-28: PL4. Declarations and Types topics covered in programming-first CS1-CS2

PL5. Abstraction mechanisms	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Procedures, functions, and iterators as abstraction mechanisms			**X**
Parameterization mechanisms (reference vs. value)			
Activation records and storage management			
Type parameters and parameterized types			
Modules in programming languages			

Table 3-29: PL5. Abstraction Mechanisms topics covered in programming-first CS1-CS2

PL6. Object-oriented programming	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Classes and subclasses	**X**	**X**	**X**
Collection classes and iteration protocols	**X**	**X**	**X**
Inheritance (overriding, dynamic dispatch)	**X**	**X**	**X**
Object-oriented design	**X**	**X**	**X**
Class hierarchies	**X**		**X**
Encapsulation and information-hiding	**X**		**X**
Polymorphism (subtype polymorphism vs. inheritance)	**X**		**X**
Separation of behavior and implementation	**X**		**X**
Internal representations of objects and method tables			

Table 3-30: PL6. Object-oriented Programming topics covered in programming-first CS1-CS2

SP1. History of computing	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Prehistory – the world before 1946			
History of computer hardware, software, networking			
Pioneers of computing			

Table 3-31: SP1. History of Computing topics covered in programming-first CS1-CS2

SE1. Software design	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Design patterns	**X**	**X**	**X**
Fundamental design concepts and principles	**X**	**X**	**X**
Structured design	**X**		**X**
Design for reuse		**X**	
Object-oriented analysis and design		**X**	
Component-level design			
Software architecture			

Table 3-32: SE1. Software Design topics covered in programming-first CS1-CS2

SE2. Using APIs	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
API programming			**X**
Class browsers and related tools			**X**
Debugging in the API environment			**X**
Programming by example			**X**
Introduction to component-based computing			

Table 3-33: SE2. Using APIs topics covered in programming-first CS1-CS2

SE3. Software tools and environments	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Programming environments	**X**		**X**
Testing tools			**X**
Configuration management tools			
Requirements analysis and design modeling tools			
Tool integration mechanisms			

Table 3-34: SE3. Software Tools and Environments topics covered in programming-first CS1-CS2

SE5. Software requirements and specifications	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Requirements elicitation			
Requirements analysis modeling techniques			
Functional and nonfunctional requirements			
Prototyping			
Basic concepts of formal specification techniques			

Table 3-35: SE5. Software Requirements and Specifications Constructs topics covered in programming-first CS1-CS2

SE6. Software validation	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Testing fundamentals, including test plan creation and test case generation			**X**
Validation planning			
Black-box and white-box testing techniques			
Unit, integration, validation, and system testing			
Object-oriented testing			
Inspections			

Table 3-36: SE6. Software Validation topics covered in programming-first CS1-CS2

Topic	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
PF1. Basic syntax and semantics of a higher-level language	**X**	**X**	**X**
PF1. Variables, types, expressions, and assignment	**X**	**X**	**X**
PF1. Conditional and iterative control structures	**X**	**X**	**X**
PF1. Functions and parameter passing	**X**	**X**	**X**
PF2. Problem-solving strategies	**X**	**X**	**X**
PF2. Implementation strategies for algorithms	**X**	**X**	**X**
PF2. The concept and properties of algorithms	**X**	**X**	**X**
PF3. Arrays	**X**	**X**	**X**
PF3. Strings and string processing	**X**	**X**	**X**
PF3. Linked structures	**X**	**X (use of, not implementation)**	**X**
PF4. The concept of recursion	**X**	**X**	**X**
PL6. Object-oriented design	**X**	**X**	**X**
PL6. Classes and subclasses	**X**	**X**	**X**
PL6. Inheritance (overriding, dynamic dispatch)	**X**	**X**	**X**
PL6. Collection classes and iteration protocols	**X**	**X**	**X**
SE1. Fundamental design concepts and principles	**X**	**X**	**X**
SE1. Design patterns	**X**	**X**	**X**

Table 3-37: Topics covered by all three approaches to CS1-CS2

Topic	IF CS1–CS2	OF CS1-CS2	FF CS1-CS2
PF1. Simple I/O	**X**		**X**
PF1. Structured decomposition	**X**		**X**
PF2. Debugging strategies	**X**		**X**
PF3. Primitive types	**X**		**X**
PF3. Records	**X**		**X**
PF3. Pointers and references	**X**		**X**
PF3. Implementation strategies for stacks, queues, and hash tables	**X**	**X (use of, not implementation)**	
PF3. Implementation strategies for graphs and trees	**X**	**X (introduction)**	
PF4. Implementation of recursion	**X**	**X**	
PF3. Strategies for choosing the right data structure	**X**		**X**
PF4. Recursive mathematical functions	**X**		**X**
PF4. Simple recursive procedures	**X**		**X**
PF4. Divide-and-conquer strategies	**X**		**X**
PF4. Recursive backtracking	**X**		**X**
AL1. Asymptotic analysis of upper and average complexity bounds	**X**		**X**
AL1. Big O, little o, omega, and theta notation	**X (Big O only)**		**X (Big O only)**
AL1. Standard complexity classes	**X**		**X**
AL1. Empirical measurements of performance	**X**		**X**
AL3. Sequential and binary search algorithms	**X**		**X**
AL5. Tractable and intractable problems	**X**		**X**
AL5. Uncomputable functions	**X**		**X**
PL1. History of programming languages	**X**		**X**
PL1. Brief survey of programming paradigms:Procedural languages, Object-oriented languages, Functional languages, Declarative, non-algorithmic languages, Scripting languages	**X**		**X**
PL2. The concept of a virtual machine	**X**		**X**
PL2. Hierarchy of virtual machines	**X**		**X**
PL2. Intermediate languages	**X**		**X**
PL6. Encapsulation and information-hiding	**X**		**X**
PL6. Separation of behavior and implementation	**X**		**X**
PL6. Polymorphism (subtype polymorphism vs. inheritance)	**X**		**X**
PL6. Class hierarchies	**X**		**X**
SE1. Structured design	**X**		**X**
SE3. Programming environments	**X**		**X**

Table 3-38: Topics covered by two of three approaches to CS1-CS2

Topic	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
PF3. Data representation in memory	X		
PF3. Static, stack, and heap allocation	X		
PF3. Runtime storage management	X		
AL3. Quadratic sorting algorithms (selection, insertion)	X		
AL3. O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)	X		
AL3. Hash tables, including collision-avoidance strategies	X		
AL3. Binary search trees	X		
SE1. Object-oriented analysis and design		X	
SE1. Design for reuse		X	
PF2. The role of algorithms in the problem-solving process			X
AL3. Simple numerical algorithms			X
PL2. Security issues arising from running code on an alien machine			X
PL5. Procedures, functions, and iterators as abstraction mechanisms			X
SE2. API programming			X
SE2. Class browsers and related tools			X
SE2. Programming by example			X
SE2. Debugging in the API environment			X
SE3. Testing tools			X
SE6. Testing fundamentals, including test plan creation and test case generation			X

Table 3-39: Topics covered by one of three approaches to CS1-CS2

Topic
AL1. Identifying differences among best, average, and worst case behaviors
AL1. Time and space tradeoffs in algorithms
AL1. Using recurrence relations to analyze recursive algorithms
AL3. Representations of graphs (adjacency list, adjacency matrix)
AL3. Depth- and breadth-first traversals
AL3. Shortest-path algorithms (Dijkstra’s and Floyd’s algorithms)
AL3. Transitive closure (Floyd’s algorithm)
AL3. Minimum spanning tree (Prim’s and Kruskal’s algorithms
AL3. Topological sort
AL5. Finite-state machines
AL5. Context-free grammars
AL5. The halting problem
AL5. Implications of uncomputability
PL1. The effects of scale on programming methodology
PL4. The conception of types as a set of values with together a set of operations
PL4. Declaration models (binding, visibility, scope, and lifetime)
PL4. Overview of type-checking
PL4. Garbage collection
PL5. Parameterization mechanisms (reference vs. value)
PL5. Activation records and storage management
PL5. Type parameters and parameterized types
PL5. Modules in programming languages
PL6. Internal representations of objects and method tables
SP1. Prehistory – the world before 1946
SP1. History of computer hardware, software, networking
SP1. Pioneers of computing
SE1. Software architecture
SE1. Component-level design
SE2. Introduction to component-based computing
SE3. Requirements analysis and design modeling tools
SE3. Configuration management tools
SE3. Tool integration mechanisms
SE5. Requirements elicitation
SE5. Requirements analysis modeling techniques
SE5. Functional and nonfunctional requirements
SE5. Prototyping
SE5. Basic concepts of formal specification techniques
SE6. Validation planning
SE6. Black-box and white-box testing techniques
SE6. Unit, integration, validation, and system testing
SE6. Object-oriented testing
SE6. Inspections

Table 3-40: Topics covered by none of the three approaches to CS1-CS2

3.2.4.5 Analysis of Hours Covered by Each Approach for each Knowledge Unit

In each of the sample Syllabus sections for each approach, there is an indication of how many core hours should be covered for each knowledge unit presented. Recall that topics that are indicated as core topics in CC2001 are considered the foundational core of the discipline. CC2001 gives a recommendation of classroom time that should be devoted to core topics throughout the curriculum. Therefore, for purposes of this dissertation core hours are classroom hours that should be devoted to a particular topic. This information is important in giving further indication of how many hours are devoted to each knowledge unit by each of the introductory approaches. Table 3-41 summarizes the number of hours covered for each knowledge unit as given Appendix B of CC2001.

Knowledge Unit	Total Hours in Knowledge Unit	Imperative-first CS1-CS2 Hours[11]	Objects-first CS1-CS2 Hours	Functional-first CS1-CS2 Hours
PF1. Fundamental Programming Constructs	9	9 + 0 = 9 (100%)	7 + 2 = 9 (100%)	3 + 6 = 9 (100%)
PF2. Algorithms and Problem Solving	6	3 + 0 = 3 (50%)	2 + 2 = 4 (67%)	2 + 1 = 3 (50%)
PF3. Fundamental Data Structures	14	6 + 6 = 12 (86%)	3 + 8 = 11 (79%)	6 + 5 = 11 (79%)
PF4. Recursion	5	0 + 5 = 5 (100%)	2 + 3 = 5 (100%)	5 + 0 = 5 (100%)
AL1. Basic Algorithmic Analysis	4	0 + 2 = 2 (50%)	0 + 2 = 2 (50%)	2 + 0 = 2 (50%)
AL3. Fundamental Computing Algorithms	12	2 + 4 = 6 (50%)	3 + 3 = 6 (50%)	4 + 2 = 6 (50%)
AL5. Basic Computability	6	1 + 0 = 1 (17%)	1 + 0 = 1 (17%)	1 + 0 = 1 (17%)
PL1. Overview of Programming Languages	2	1 + 1 = 2 (100%)	0 + 2 = 2 (100%)	1 + 1 = 2 (100%)
PL2. Virtual Machines	1	0 + 1 = 1 (100%)	0 + 1 = 1 (100%)	0 + 1 = 1 (100%)
PL4. Declarations and Types	3	1 + 2 = 3 (100%)	2 + 1 = 3 (100%)	1 + 2 = 3 (100%)
PL5. Abstraction Mechanisms	3	2 + 1 = 3 (100%)	1 + 2 = 3 (100%)	1 + 2 = 3 (100%)
PL6. Object-Oriented Programming	10	3 + 7 = 10 (100%)	8 + 4 = 12 (120%)	0 + 8 = 8 (80%)
SP1. History of Computing	1	1 + 0 = 1 (100%)	1 + 0 = 1 (100%)	1 + 0 = 1 (100%)
SE1. Software Design	8	2 + 2 = 4 (50%)	2 + 2 = 4 (50%)	1 + 3 = 4 (50%)
SE2. Using API's	5	0 + 2 = 2 (40%)	1 + 1 = 2 (40%)	0 + 2 = 2 (40%)
SE3. Software Tools and Environments	3	1 + 2 = 3 (100%)	2 + 0 = 2 (67%)	1 + 1 = 2 (67%)
SE5. Software Requirements and Specifications	4	1 + 0 = 1 (25%)	0 + 1 = 1 (25%)	0 + 1 = 1 (25%)
SE6. Software Validation	3	1 + 0 = 1 (33%)	0 + 1 = 1 (33%)	0 + 1 = 1 (33%)

Table 3-41: Hours devoted to each knowledge unit for programming-first CS1-CS2

[11] Hours in this table are given in the form CS1Hours + CS2 Hours = Total hours for sequence

3.2.4.6 Problems with Simply "Reading" the Syllabi

There are numerous problems with this "shallow" reading approach to the topics. Simply using the method of "shallow" reading the descriptions of the *Syllabus* sections of the course descriptions does not seem to give proper results for topical coverage. Taking for example just the last table in the analysis (Table 3-36), the only approach that has an X in any of the rows is functional-first. However, according to the table of number of hours covered (Table 3-41), each of the three approaches has coverage for this knowledge unit.

The following inconsistencies have been discovered when simply using a "shallow" approach to creating the topic intersection and each point to a need for a deeper reading of the syllabi and an analysis that is deeper than simply reading terms and topics.

- The number of hours covered indicates that the knowledge unit is covered in its entirety; however, not all of the topics from the knowledge unit are indicated in the intersection. For example, PF4 is supposed to be covered in full by all three approaches. However, Table 3-22 reveals that not all knowledge units are included for all three approaches.

- A knowledge unit is supposed to be covered in the courses, but there are no topics marked for that knowledge unit for any of the approaches. For example, SE5 is supposed to have coverage in the intersection, but according to Table 3-35, none of the topics are indicated for any of the approaches.

- One approach does not have any topics indicated for a particular knowledge unit, while the other two approaches have topics that seem to give an accurate picture of the coverage of that unit. For AL1 (Table 3-23), no coverage is indicated for objects-first, while the other two approaches have coverage indicated that corresponds with the 50% coverage indicated in Table 3-41.

- Only one approach has topics indicated for a particular knowledge unit, while the others have no topics indicated. For example, in Table 3-29 for PL5, only functional-first has knowledge units indicated, but all three need to have knowledge units indicated in this knowledge area.

- There is simply a general mismatch within the topic. For example, in Table 3-34 for SE3, two-thirds or more of the topics should be covered for each approach. Only one topic (out of five) is indicated for imperative-first, none indicated for objects-first, and two indicated for functional-first.

Of all of these, the last point is the least bothersome. It would be reasonable to expect that not all of the approaches cover the exact same material in these knowledge units. However, it is important to look at this type of mismatch to make sure there are no topics that should be included in this analysis.

3.2.5 Resolution of Discrepancies

3.2.5.1 Reasons for Inconsistencies

While it is difficult to determine the exact causes for the discrepancies between the *Syllabus* and *Units covered* sections, there is definitely a lack of uniformity between the language of the sample syllabi and the language in the knowledge-unit topic descriptions. Given how CC2001 was constructed (by various subcommittees), it is easy to postulate that while some subcommittees followed the language of the knowledge units while creating the syllabi, some did not.

The biggest offenders appear to be the objects-first syllabi. The “shallow” reading approach leaves many holes in the topical coverage, even in section PL6-Object-oriented programming (see Table 3-30), which the objects-first model focuses most heavily upon.

The other two sets of syllabi suffer from some of the same language issues, but not to the extent of the objects-first topics. For the purposes of creating this assessment instrument, the topics included in the intersection using both a “shallow” reading of the *Syllabus* sections of the course descriptions as well as the more in-depth analysis presented in §3.2.6 – 3.2.8.

3.2.5.2 All topics should be covered, not all were indicated

In some knowledge units, Table 3-41 indicates that all core hours of a topic will be covered in the introductory sequence, but, in the “shallow” read of the syllabus topics, not all of the topics in that knowledge area received an X. This happens for the following knowledge units:

- PF1. Fundamental Programming Constructs
- PF4. Recursion
- PL1. Overview of Programming Languages
- PL2. Virtual Machines
- PL4. Declarations and Types
- PL5. Abstraction Mechanisms
- PL6. Object-oriented Programming
- SP1. History of Computing

For each of these knowledge units, there is a pattern of incomplete descriptions in the syllabus topics. For example, in SP1, History of Computing, all three syllabi indicate in a broad fashion that the history of computing should be covered and that it should be covered for one full course hour. However, there is no indication in the syllabus of the specific listing of the topics for SP1 as given in Appendix B. In this case, we can resolve this discrepancy by assuming that there has simply been a lack of attention to detail by the CC2001 committee that led to this oversight.

For PF1, Fundamental Programming Constructs, simple I/O and structured decomposition are the topics not indicated in the objects-first approach. Since the entire knowledge unit is supposed to be covered, I again assume the oversight to be a lack of attention to detail.

For PF4, Recursion, again we notice missing topics in the objects-first column. It is unreasonable to think that, if one is talking about the “implementation of recursion” in a course then “simple recursion” would not be included in that discussion. Similar arguments can be made for each of the topics in this knowledge unit, so they should all be included in the intersection. The topic “implementation of recursion” is also missing for functional-first. This is also unreasonable given that all of the rest of the topics are covered. Also, given the fact that functional languages rely heavily on recursion as a base in the language, it is not reasonable to assume that implementation of recursion would be ignored in this approach. Therefore, this topic should be included for the functional-first approach.

For PL1, Overview of Programming Languages, there is a similar situation as with SP1, History of Programming: the syllabi indicate coverage, but there is no mention of some of the specific topics. Since the entire core hours should be covered, these topics should be restored to the intersection.

For PL2, Virtual Machines, the topic “security issues arising from running code on an alien machine” is missing in imperative-first and no indication of coverage of any topic in the objects-first approach. Once again, this seems to be an oversight. For objects-first, many of the newly popular object-oriented languages, especially Java, use virtual machines extensively. Therefore, it would be a natural part of the course to explain how the language works. All topics from this knowledge unit are included in the intersection.

For PL4, Declarations and Types, and PL5, Abstraction Mechanisms, very few topics have been included. In fact, for PL4, no topics have been included for any of the approaches. However, for both of these sections, this current set of topics does not make sense. For example, in PL4, one of the topics should cover such ideas as binding, visibility, scope, and lifetime. This is clearly a part of any introductory sequence when discussing local variables and should be included in the set of topics. For these topics, it seems to once again to be an oversight and lack of detail in the syllabus topics. Therefore, all topics for these two knowledge units will be included.

3.2.5.3 Topics should be covered, none or one were indicated

We can resolve the discrepancies for two of the knowledge units, SE5, Software Requirements and Specifications, and SE6, Software Validation. For each of these knowledge units, an amount of coverage greater than zero is indicated for each approach, however, no topics are indicated on the grid for SE5 and only one for SE6.

For SE5, each approach should have 1 course hour devoted to it, which accounts for only 25% of the time for that knowledge unit in the curriculum. Unfortunately, the syllabi do not give us a good indicator of what topics should be the focus of the coverage for this knowledge area. Therefore, it is important to decide which topics seem to be most appropriate for introductory courses and the amount of time that should be spent on these topics. In this case, it is most appropriate to include the topics of requirements elicitation and functional and nonfunctional requirements for each of the approaches.

Each approach should also have 1 course hour devoted to SE6, which will account for 33% of the time for that knowledge unit in the curriculum. There is some indication in the functional-first approach that the topic of testing fundamentals is covered in that approach. This topic, which also includes test-plan generation and test-case generation, seems to be a prime candidate for the other two approaches as well, because it is common to see some sort of testing taught during the first year. Therefore, we include this topic for all three approaches. Also, due to the inclusion of a large portion of the topics of PL6, Object Oriented Programming, by each of the approaches, it is logical to include the topic of object-oriented testing in the intersection.

Both SE5 and SE6 are in the knowledge area of Software Engineering. Since the focus for this dissertation is topics that can and should be introduced at the introductory level, these topics should be broad in scope and those that are most immediately important to students building their first computer programs. Therefore, the inclusion of testing techniques is appropriate. The topic of requirements elicitation should be viewed in its most general sense of "what does this program have to do?" It is not reasonable to include more formal requirements-elicitation techniques, but rather to have the students experiment with how to find out what their projects should be capable of by asking questions about the assignments given in class. In regard to the functional and nonfunctional requirements, students at this level should be exposed to the general ideas about the differences between what a program does and how it looks, sounds, etc.

3.2.5.4 All approaches should have topics covered, but only two of three do

From the tables, there are three knowledge units where topics are indicated for only two of the three approaches.

For AL1, Basic Algorithmic Analysis, no topics are given for objects-first, while the other two approaches include the exact same topics. For each of the three approaches, two course hours should be devoted to this knowledge area. Therefore, it must be decided which topics to include for objects-first. It would stand to reason that quite possibly all three approaches should cover the same material in these two hours. Looking at the wording of the syllabus for objects-first, there is an indication that there should be an "Introduction to basic algorithm analysis" (Joint Task Force on Computing Curricula 2001: 177). Therefore, in the intersection, the topics covered for objects-first will be made the same as the other two approaches.

For AL5, Basic Computability, the imperative-first and functional-first approaches have the exact same topics indicated, while no topics are indicated for objects-first. Objects-first is supposed to cover one course hour of this knowledge unit, which is the same amount of time as the other approaches. The other two approaches cover the topics of tractable and intractable problems, as well as uncomputable functions. These two topics are basic introduction-to-computability topics that are appropriate for an introductory sequence. The other topics in this knowledge unit would fit better in a slightly more advanced course looking at issues of computability and not necessarily focusing on programming. Therefore, the topics for objects-first will be the same as those for the other two approaches.

For SE3, Software Tools and Environments, no topics are indicated for objects-first. The other two approaches indicate the topic of programming environments. This topic makes sense for objects-first as well. The further description of the objects-first approach says that “Many courses that adopt an objects-first approach will do so in an environment that supports a rich collection of application programmer interfaces or APIs” (Joint Task Force on Computing Curricula 2001: 176). Therefore, it would seem reasonable that with a large number of APIs, programming environments become important. Even without the APIs, any time one creates a computer program, there is an environment that one is working with to create that program. An introduction to that environment should certainly be given in the introductory courses.

However, one problem for this knowledge unit is the amount of course coverage time allotted for it. For imperatives-first, it is indicated that 100% of the unit should be covered. For the other two approaches, it indicates two-thirds of the hours to be given in the introductory sequence. Therefore, it would be most appropriate for imperative-first to have all topics indicated for this knowledge unit. For functional-first, the topic of testing tools is indicated as covered. It also makes sense that the objects-first approach covers testing, because it is just as important in objects-first as the other approaches. It is also reasonable to assume that students are exposed to some sort of design modeling tool (e.g. flow charts, CRC cards, UML) in the introductory sequence. An emphasis is placed on design in all of the approaches, and while students are being instructed on design, they will be shown some sort of tool that helps them use a particular design technique. Therefore, the topic of design modeling tools should be included in the intersection.

3.2.5.5 All approaches should have topics covered, but only one of three do

For SE2, Using APIs, topical coverage is indicated only in the functional-first approach. However, in §3.2.5.4, we concluded that APIs should be included in the objects-first model. This once again seems to be a case of lack of detail, given the syllabus's topics-covered section. For objects-first, there is a general indication of using APIs, but no details are given about which specific topics from that knowledge unit are covered (Joint Task Force on Computing Curricula 2001: 175). Indications are also given for imperative-first in the topic section, where it states that one should present "Using a graphics API" (Joint Task Force on Computing Curricula 2001: 166). Therefore, there are broad indications that this topic is presented in all three approaches.

In order to answer which topics are covered by the approaches, let us look at the amount of course coverage hours for this knowledge unit. Two hours are indicated for all approaches for this knowledge unit, encompassing 40% of the total coverage for the knowledge unit. The functional-first approach says that four out of the five topics should be covered in that time. It does not seem likely that the four topics could be covered with significant depth in that time; however, a general introduction could be given to each of the topics. The only topic not indicated is an introduction to component-based computing, which is not appropriate for any of the approaches at this level. However, a general introduction to the other topics is appropriate, and we have included those topics in our intersection.

3.2.5.6 Non-uniform topical coverage across approaches

With the previous problems in the intersection, there were significant gaps in topical coverage that needed to be addressed, such as entire knowledge units that should be covered having no topics indicated. The remaining topics do not have such glaring omissions but are not uniform in topic coverage. However, each topic must be considered, to decide if the currently indicated topic coverage is appropriate or if there are in fact omissions in the topical coverage that should be included in our final intersection.

For PF2, Algorithms and Problem Solving, some omissions seem to be due to lack of attention to detail. For imperative-first and objects-first, the topic of the role of algorithms in the problem-solving process is not indicated. However, if you are implementing algorithms, as indicated by both approaches, you will use them in their role in the problem-solving process. Therefore, this topic should be included. In objects-first, the topic of debugging strategies is also not indicated. This, too, seems like an oversight. It is not reasonable to assume that an introductory course does not talk about debugging. Therefore, we will also include that topic. For this knowledge unit, all approaches indicate all topics covered, but the amount of hours of coverage ranges from 50-66.67%. This would indicate that further coverage of these topics will also be needed beyond the first year of courses.

For PF3, Fundamental Data Structures, no topics are indicated for objects-first, while imperative-first indicates that all topics are covered even though not all of the time allocated for this knowledge unit has been covered. It should therefore be assumed that some of the topics indicated for the imperative-first approach are not covered in their entirety or that they are only introduced.

For functional-first, some of the omissions of topics do not make sense. Since knowledge unit DS5, Graphs and Trees, is included in CS1 according to the syllabus for functional-first in CC2001, implementation of graphs and trees should also be discussed. Also, list structures are indicated as being discussed, so it would be natural to assume that stacks and queues would be included in that discussion. Consequently, these topics will be included in our intersection.

For objects-first, primitive types are omitted. While the focus of objects-first is objects, most languages have primitive types and many object-oriented languages use the primitive types in the basic control structures, so it is reasonable to assume that this topic should be covered. Since there is an indication of discussions of stacks and queues, omitting linked structures seems like an oversight, so it will be included. Also, the syllabus explicitly indicates that implementation of data structures is not covered. This does not seem likely; however, given that the syllabus is so explicit, it will not be included[12]. Lastly, the omission of the last topic "strategies for choosing the right data structure" is an oversight. When talking about data structures in any form, it is most appropriate to discuss how to choose one data structure; especially given that the objects-

[12] Look to §3.2.8 for further discussion of this issue.

first approach does not discuss implementation, the topic of choosing the correct data structure should most certainly be discussed.

For AL3, Fundamental Computing Algorithms, there is once again a case where objects-first has no topics indicated. For this knowledge unit, each approach should cover six hours of topics, or 50% of the total time indicated for the knowledge unit. It seems as though the imperative-first approach fulfills this requirement nicely, except for what seems to be an oversight of the topic of simple numerical algorithms[13]. If the courses will contain sorting and searching, simple numerical algorithms will most likely be covered.

For functional-first, the first two topics are indicated. There is also indication in the syllabus for functional-first that sorting algorithms are covered, but it does not provide specific details about which ones (Joint Task Force on Computing Curricula 2001: 180). However, it seems reasonable to assume that this approach should cover the standard set of quadratic and O(N log N) sorting algorithms; consequently, we will include those topics. Also, the syllabus indicates that hierarchical data should be covered in CS1 using this approach (Joint Task Force on Computing Curricula 2001: 178). This would seem to indicate coverage of trees and possibly graphs. This is further supported by the CS1 coverage of knowledge unit DS5, Graphs and Trees. Even though no other approach seems to cover graphs, it is reasonable to assume that functional-first does in part.

[13] See §4.3.1.1 for more discussion of this term.

For objects-first, it is slightly more difficult to decide which topics to pick from this knowledge unit. There is just a lack of indication on the syllabi as to which topics are covered. However, there is no indication that graphs are covered in this approach. Since the amount of time is the same as imperative-first, coverage will be given to topics that are similar to the imperative-first model for this knowledge unit.

For PL6, Object-Oriented Programming, there are some shocking omissions from the objects-first column. These omissions must be the result of oversight. It is unreasonable to assume that the objects-first introductory sequence would not include polymorphism or class hierarchies, when those topics are foundational to object-oriented programming itself. Also, given that Table 3-41 indicates that 120% of the core hours should be covered in this area, it is fairly safe to assume that these topics will be covered. The only topic that is being left off is the last, internal representations of objects and method tables. None of the approaches indicate that this topic should be covered; however two of the three approaches indicate full coverage of the hours for this knowledge unit. It seems reasonable that with only 80% of coverage time allotted for these topics for functional-first, this last topic may be left off. Given that, it will not be included in the intersection. Whether the other two approaches cover it is in question, so this topic will be left off entirely.

SE1, Software Design, indicates the same amount of course coverage time for all three approaches. The first two topics are indicated for all approaches. The third topic, software architecture, is indicated for none of the approaches. It seems reasonable to postpone this topic for a course that is more focused on software engineering. The fourth topic, structured design, is not indicated for objects-first. However, since object-oriented design may not be viewed as structured design proper, it can remain empty for this purpose. The fifth topic, object-oriented design and analysis, should be included for all three approaches, because even the imperative-first and functional-first spend effort in the CS2 course on object-oriented concepts. The coverage in the non-objects-first approaches will be less due to their original emphasis on structured design, while there will be more of this topic emphasized in objects-first, balancing out the lack of structured design coverage. The sixth topic is component-level design. Components may not be covered by all approaches and should therefore be covered elsewhere in the curriculum. The last topic, design for reuse, is indicated for objects-first, but not the other approaches. Reuse is an important theme in object-oriented methodology. Other methodologies do not rely as heavily on this idea. Given the limited exposure to object-oriented programming in the other two approaches, this topic will not be included.

3.2.6 Revised Intersection of Knowledge Unit Topical Coverage

Table 3-42 through Table 3-63 indicate a more realistic view of the intersection for the programming-first approaches.

PF1. Fundamental Programming Constructs Topics	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Basic syntax and semantics of a higher-level language	**X**	**X**	**X**
Conditional and iterative control structures	**X**	**X**	**X**
Functions and parameter passing	**X**	**X**	**X**
Simple I/O	**X**	**X**	**X**
Variables, types, expressions, and assignment	**X**	**X**	**X**
Structured decomposition	**X**		**X**

Table 3-42: PF1. Fundamental Programming Constructs topics covered in programming-first CS1-CS2

PF2. Algorithms and problem-solving Topics	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Problem-solving strategies	**X**	**X**	**X**
The role of algorithms in the problem-solving process	**X**	**X**	**X**
Implementation strategies for algorithms	**X**	**X**	**X**
Debugging strategies	**X**	**X**	**X**
The concept and properties of algorithms	**X**	**X**	**X**

Table 3-43: PF2. Algorithms and Problem-Solving topics covered in programming-first CS1-CS2

PF3. Fundamental Data Structures	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Arrays	**X**	**X**	**X**
Implementation strategies for graphs and trees	**X**	**X (introduction)**	**X**
Implementation strategies for stacks, queues, and hash tables	**X**	**X (use of, not implementation)**	**X**
Linked structures	**X**	**X (use of, not implementation)**	**X**
Primitive types	**X**	**X**	**X**
Strategies for choosing the right data structure	**X**	**X**	**X**
Strings and string processing	**X**	**X**	**X**
Pointers and references	**X**		**X**
Records	**X**		**X**
Data representation in memory	**X**		
Runtime storage management	**X**		
Static, stack, and heap allocation	**X**		

Table 3-44: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2

PF4. Recursion	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
The concept of recursion	**X**	**X**	**X**
Recursive mathematical functions	**X**	**X**	**X**
Simple recursive procedures	**X**	**X**	**X**
Divide-and-conquer strategies	**X**	**X**	**X**
Recursive backtracking	**X**	**X**	**X**
Implementation of recursion	**X**	**X**	**X**

Table 3-45: PF4. Recursion topics covered in programming-first CS1-CS2

AL1. Basic algorithmic analysis	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Asymptotic analysis of upper and average complexity bounds	**X**	**X**	**X**
Big O, little o, omega, and theta notation	**X (Big O only)**	**X (Big O only)**	**X (Big O only)**
Empirical measurements of performance	**X**	**X**	**X**
Standard complexity classes	**X**	**X**	**X**
Identifying differences among best, average, and worst case behaviors			
Time and space tradeoffs in algorithms			
Using recurrence relations to analyze recursive algorithms			

Table 3-46:AL1. Basic Algorithmic Analysis topics covered in programming-first CS1-CS2

AL3. Fundamental computing algorithms	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Binary search trees	**X**	**X**	**X**
O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)	**X**	**X**	**X**
Quadratic sorting algorithms (selection, insertion)	**X**	**X**	**X**
Sequential and binary search algorithms	**X**	**X**	**X**
Simple numerical algorithms	**X**	**X**	**X**
Hash tables, including collision-avoidance strategies	**X**	**X**	
Representations of graphs (adjacency list, adjacency matrix)			**X**
Depth- and breadth-first traversals			
Minimum spanning tree (Prim's and Kruskal's algorithms)			
Shortest-path algorithms (Dijkstra's and Floyd's algorithms)			
Topological sort			
Transitive closure (Floyd's algorithm)			

Table 3-47: AL3. Fundamental Computing Algorithms topics covered in programming-first CS1-CS2

AL5. Basic computability	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Tractable and intractable problems	X	X	X
Uncomputable functions	X	X	X
Context-free grammars			
Finite-state machines			
Implications of uncomputability			
The halting problem			

Table 3-48: AL5. Basic Computability topics covered in programming-first CS1-CS2

PL1. Overview of programming languages	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
History of programming languages	X	X	X
Brief survey of programming paradigms:Procedural languages, Object-oriented languages, Functional languages, Declarative, non-algorithmic languages, Scripting languages	X	X	X
The effects of scale on programming methodology	X	X	X

Table 3-49: PL1. Overview of Programming Languages topics covered in programming-first CS1-CS2

PL2. Virtual machines	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
The concept of a virtual machine	X	X	X
Hierarchy of virtual machines	X	X	X
Intermediate languages	X	X	X
Security issues arising from running code on an alien machine	X	X	X

Table 3-50: PL2. Virtual Machines topics covered in programming-first CS1-CS2

PL4. Declarations and types	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
The conception of types as a set of values with together a set of operations	X	X	X
Declaration models (binding, visibility, scope, and lifetime)	X	X	X
Overview of type-checking	X	X	X
Garbage collection	X	X	X

Table 3-51: PL4. Declarations and Types topics covered in programming-first CS1-CS2

PL5. Abstraction mechanisms	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Procedures, functions, and iterators as abstraction mechanisms	X	X	X
Parameterization mechanisms (reference vs. value)	X	X	X
Activation records and storage management	X	X	X
Type parameters and parameterized types	X	X	X
Modules in programming languages	X	X	X

Table 3-52: PL5. Abstraction Mechanisms topics covered in programming-first CS1-CS2

PL6. Object-oriented programming	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Object-oriented design	X	X	X
Encapsulation and information-hiding	X	X	X
Separation of behavior and implementation	X	X	X
Classes and subclasses	X	X	X
Inheritance (overriding, dynamic dispatch)	X	X	X
Polymorphism (subtype polymorphism vs. inheritance)	X	X	X
Class hierarchies	X	X	X
Collection classes and iteration protocols	X	X	X
Internal representations of objects and method tables			

Table 3-53: PL6. Object-oriented Programming topics covered in programming-first CS1-CS2

SP1. History of computing	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Prehistory – the world before 1946	X	X	X
History of computer hardware, software, networking	X	X	X
Pioneers of computing	X	X	X

Table 3-54: SP1. History of Computing topics covered in programming-first CS1-CS2

SE1. Software design	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Design patterns	X	X	X
Fundamental design concepts and principles	X	X	X
Object-oriented analysis and design	X	X	X
Structured design	X		X
Design for reuse		X	
Component-level design			
Software architecture			

Table 3-55: SE1. Software Design topics covered in programming-first CS1-CS2

SE2. Using APIs	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
API programming	**X**	**X**	**X**
Class browsers and related tools	**X**	**X**	**X**
Programming by example	**X**	**X**	**X**
Debugging in the API environment	**X**	**X**	**X**
Introduction to component-based computing			

Table 3-56: SE2. Using APIs topics covered in programming-first CS1-CS2

SE3. Software tools and environments	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Programming environments	**X**	**X**	**X**
Requirements analysis and design modeling tools	**X**	**X (Modeling tools)**	**X (Modeling tools)**
Testing tools	**X**	**X**	**X**
Configuration management tools	**X**		
Tool integration mechanisms	**X**		

Table 3-57: SE3. Software Tools and Environments topics covered in programming-first CS1-CS2

SE5. Software requirements and specifications	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Functional and nonfunctional requirements	**X**	**X**	**X**
Requirements elicitation	**X**	**X**	**X**
Basic concepts of formal specification techniques			
Prototyping			
Requirements analysis modeling techniques			

Table 3-58: SE5. Software Requirements and Specifications Constructs topics covered in programming-first CS1-CS2

SE6. Software validation	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Object-oriented testing	**X**	**X**	**X**
Testing fundamentals, including test plan creation and test case generation	**X**	**X**	**X**
Black-box and white-box testing techniques			
Inspections			
Unit, integration, validation, and system testing			
Validation planning			

Table 3-59: SE6. Software Validation topics covered in programming-first CS1-CS2

Topic	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
PF1. Basic syntax and semantics of a higher-level language	**X**	**X**	**X**
PF1. Variables, types, expressions, and assignment	**X**	**X**	**X**
PF1. Simple I/O	**X**	**X**	**X**
PF1. Conditional and iterative control structures	**X**	**X**	**X**
PF1. Functions and parameter passing	**X**	**X**	**X**
PF2. Problem-solving strategies	**X**	**X**	**X**
PF2. The role of algorithms in the problem-solving process	**X**	**X**	**X**
PF2. Implementation strategies for algorithms	**X**	**X**	**X**
PF2. Debugging strategies	**X**	**X**	**X**
PF2. The concept and properties of algorithms	**X**	**X**	**X**
PF3. Primitive types	**X**	**X**	**X**
PF3. Arrays	**X**	**X**	**X**
PF3. Strings and string processing	**X**	**X**	**X**
PF3. Linked structures	**X**	**X (use of, not implementation)**	**X**
PF3. Implementation strategies for stacks, queues, and hash tables	**X**	**X (use of, not implementation)**	**X**
PF3. Implementation strategies for graphs and trees	**X**	**X (introduction)**	**X**
PF3. Strategies for choosing the right data structure	**X**	**X**	**X**
PF4. The concept of recursion	**X**	**X**	**X**
PF4. Recursive mathematical functions	**X**	**X**	**X**
PF4. Simple recursive procedures	**X**	**X**	**X**
PF4. Divide-and-conquer strategies	**X**	**X**	**X**
PF4. Recursive backtracking	**X**	**X**	**X**
PF4. Implementation of recursion	**X**	**X**	**X**
AL1. Asymptotic analysis of upper and average complexity bounds	**X**	**X**	**X**
AL1. Big O, little o, omega, and theta notation	**X (Big O only)**	**X (Big O only)**	**X (Big O only)**
AL1. Standard complexity classes	**X**	**X**	**X**
AL1. Empirical measurements of performance	**X**	**X**	**X**
AL3. Simple numerical algorithms	**X**	**X**	**X**
AL3. Sequential and binary search algorithms	**X**	**X**	**X**
AL3. Quadratic sorting algorithms (selection, insertion)	**X**	**X**	**X**
AL3. O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)	**X**	**X**	**X**
AL3. Binary search trees	**X**	**X**	**X**
AL5. Tractable and intractable problems	**X**	**X**	**X**
AL5. Uncomputable functions	**X**	**X**	**X**

PL1. History of programming languages	X	X	X
PL1. The effects of scale on programming methodology	X	X	X
PL2. The concept of a virtual machine	X	X	X
PL2. Hierarchy of virtual machines	X	X	X
PL2. Intermediate languages	X	X	X
PL2. Security issues arising from running code on an alien machine	X	X	X
PL4. The conception of types as a set of values with together a set of operations	X	X	X
PL4. Declaration models (binding, visibility, scope, and lifetime)	X	X	X
PL4. Overview of type-checking	X	X	X
PL4. Garbage collection	X	X	X
PL5. Procedures, functions, and iterators as abstraction mechanisms	X	X	X
PL5. Parameterization mechanisms (reference vs. value)	X	X	X
PL5. Activation records and storage management	X	X	X
PL5. Type parameters and parameterized types	X	X	X
PL5. Modules in programming languages	X	X	X
PL6. Object-oriented design	X	X	X
PL6. Encapsulation and information-hiding	X	X	X
PL6. Separation of behavior and implementation	X	X	X
PL6. Classes and subclasses	X	X	X
PL6. Inheritance (overriding, dynamic dispatch)	X	X	X
PL6. Polymorphism (subtype polymorphism vs. inheritance)	X	X	X
PL6. Class hierarchies	X	X	X
PL6. Collection classes and iteration protocols	X	X	X
SP1. Prehistory – the world before 1946	X	X	X
SP1. History of computer hardware, software, networking	X	X	X
SP1. Pioneers of computing	X	X	X
SE1. Fundamental design concepts and principles	X	X	X
SE1. Design patterns	X	X	X
SE1. Object-oriented analysis and design	X	X	X
SE2. API programming	X	X	X
SE2. Class browsers and related tools	X	X	X
SE2. Programming by example	X	X	X
SE2. Debugging in the API environment	X	X	X

SE3. Programming environments	**X**	**X**	**X**
SE3. Requirements analysis and design modeling tools	**X**	**X (Modeling tools)**	**X (Modeling tools)**
SE3. Testing tools	**X**	**X**	**X**
SE5. Requirements elicitation	**X**	**X**	**X**
SE5. Functional and nonfunctional requirements	**X**	**X**	**X**
SE6. Testing fundamentals, including test plan creation and test case generation	**X**	**X**	**X**
SE6. Object-oriented testing	**X**	**X**	**X**

Table 3-60: Topics covered by all three programming-first approaches to CS1-CS2

Topic	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
PF1. Structured decomposition	**X**		**X**
PF3. Records	**X**		**X**
PF3. Pointers and references	**X**		**X**
AL3. Hash tables, including collision-avoidance strategies	**X**	**X**	
SE1. Structured design	**X**		**X**

Table 3-61: Topics covered by all two of three programming-first approaches to CS1-CS2

Topic	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
PF3. Data representation in memory	**X**		
PF3. Static, stack, and heap allocation	**X**		
PF3. Runtime storage management	**X**		
SE3. Configuration management tools	**X**		
SE3. Tool integration mechanisms	**X**		
SE1. Design for reuse		**X**	
AL3. Representations of graphs (adjacency list, adjacency matrix)			**X**

Table 3-62: Topics covered by one of the three programming-first approaches to CS1-CS2

Topic
AL1. Identifying differences among best, average, and worst case behaviors
AL1. Time and space tradeoffs in algorithms
AL1. Using recurrence relations to analyze recursive algorithms
AL3. Depth- and breadth-first traversals
AL3. Shortest-path algorithms (Dijkstra’s and Floyd’s algorithms)
AL3. Transitive closure (Floyd’s algorithm)
AL3. Minimum spanning tree (Prim’s and Kruskal’s algorithms
AL3. Topological sort
AL5. Finite-state machines
AL5. Context-free grammars
AL5. The halting problem
AL5. Implications of uncomputability
PL6. Internal representations of objects and method tables
SE1. Software architecture
SE1. Component-level design
SE2. Introduction to component-based computing
SE5. Requirements analysis modeling techniques
SE5. Prototyping
SE5. Basic concepts of formal specification techniques
SE6. Validation planning
SE6. Black-box and white-box testing techniques
SE6. Unit, integration, validation, and system testing
SE6. Inspections

Table 3-63: Topics covered by none of the three programming-first approaches to CS1-CS2

3.2.7 Comparison of the Current Intersection to CC2001 Chapter 7

One more comparison must be made in order to ensure that the intersection is complete and aligned with the goals of CC2001. Consider again Table 3-2, which describes the knowledge units and topics that are covered by all six of the introductory tracks. It is important for us to compare the results just achieved with the guidelines presented in this table. This will help uncover any omissions and also may help to confirm some of the decisions made during the more in-depth analysis of the intersection topics described in §3.2.5 – 3.2.6.

It is important to remember that because Table 3-2 from CC2001 represents knowledge units covered by all six introductory tracks, both programming-first and non-programming-first, there will most likely be topics included in the intersection presented in the previous section that are not included in the table. Topics that are inherently more programmatic in nature may be covered extensively in the programming-first approaches, but may not be covered at all in the non-programming-first approaches. One such example of this is discussion of data structures such as stacks, queues, trees, and graphs. These topics are not included in the table, but are included in the intersection as part of the topical coverage.

First, Table 3-2 presents knowledge units for which all topics must be covered. These include:

- DS1. Functions, relations, and sets
- DS2. Basic logic
- DS4. Basics of counting
- DS6. Discrete probability
- PF1. Fundamental programming constructs
- PF4. Recursion
- PL1. Overview of programming languages
- PL2. Virtual machines
- PL4. Declarations and types
- PL5. Abstraction mechanisms
- SP1. History of computing

One notices right away that DS1, DS2, DS4, and DS6 do not appear anywhere in the analysis of the programming-first approaches. This would at first seem to indicate that the intersection just created is totally incorrect. However, looking at all three of the introductory tracks and their suggested syllabi, none mention any of DS1, DS2, DS4, or

DS6. Therefore, the question that comes to mind is, how can these topics be included by CC2001 as covered by all six of the introductory tracks? Looking back to Chapter 7 of CC2001, the answer can be found in section 7.4, Integrating discrete mathematics into the introductory curriculum.

CC2001 advocates exposure to the concepts of discrete mathematics early, possibly in the first year of study. CC2001 indicates two possible ways to achieve this goal. The first is a separate discrete mathematics course taken concurrently with the introductory sequence of courses. The second is integrating the discrete mathematics into the introductory sequence, which is demonstrated in the three-semester model for the Breadth-first approach.

The analysis performed was on programming-first courses that must be complemented by a separate discrete mathematics course during the first year. The topics indicated in Table 3-2 from the discrete structures knowledge area would be covered in that course. The rest of the knowledge units indicated are covered in their entirety by the intersection that was created.

Table 3-2 also presents knowledge units for which only certain topics should be covered. Considering each one individually, notice that the intersection presented gives proper coverage to these areas.

- DS3. Proof techniques: The structure of formal proofs; proof techniques; direct, counterexample, contraposition, contradiction; mathematical induction

For DS3, there is the same problem as with the other knowledge units from the Discrete Structures area. Since these topics would also be a part of a first-year discrete mathematics course, they will not be included in the intersection.

- PF2. Algorithms and problem-solving: Problem solving strategies; the role of algorithms in the problem-solving process; the concept and properties of algorithms; debugging strategies

In Table 3-43, all of these topics are indicated as part of the intersection. This subset of topics also includes the topic of the role of algorithms in the problem-solving process, which was initially omitted from some of the approaches. This comparison of our current intersection with CC2001 further validates the decision to make that topic part of the intersection.

- PF3. Fundamental data structures: Primitive types; arrays; records; strings and string processing; data representation in memory; static, stack, and heap allocation; runtime storage management; pointers and references; linked structures

In Table 3-44, the topics that are indicated as part of the intersection do not coincide with the list of topics given here. Missing from objects-first are the topics of: records; data representation in memory; static, stack, and heap allocation; runtime storage management; pointers and references. Missing from functional-first are the topics of: data representation in memory; static, stack, and heap allocation; runtime storage management. It is not unusual for these topics to be found in an introductory sequence, and in fact are indicated to be a part of the imperative-first courses. This comparison with CC2001 Chapter 7 indicates that these topics should be included.

With this list of topics for this knowledge unit, support is given for the decisions that were made in §3.2.5.6 about the inclusion of primitive types and linked structures in the intersection.

Table 3-64 gives the finalized picture of the topics that should be included in the intersection for the PF3 knowledge unit based upon all of the information available to us.

PF3. Fundamental Data Structures	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Primitive types	**X**	**X**	**X**
Arrays	**X**	**X**	**X**
Records	**X**	**X**	**X**
Strings and string processing	**X**	**X**	**X**
Data representation in memory	**X**	**X**	**X**
Static, stack, and heap allocation	**X**	**X**	**X**
Runtime storage management	**X**	**X**	**X**
Pointers and references	**X**	**X**	**X**
Linked structures	**X**	**X (use of, not implementation)**	**X**
Implementation strategies for stacks, queues, and hash tables	**X**	**X (use of, not implementation)**	**X**
Implementation strategies for graphs and trees	**X**	**X (introduction)**	**X**
Strategies for choosing the right data structure	**X**	**X**	**X**

Table 3-64: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2

- AL1. Basic algorithmic analysis: Big O notation; standard complexity classes; empirical measurements of performance; time and space tradeoffs in algorithms

In Table 3-46, all but the topic of time and space tradeoffs in algorithms are indicated as included in the intersection. In §3.2.5.4, it was argued that all three approaches should have the same coverage for this knowledge unit and the topic list from Table 3-2 supports this idea. However, it has also included the topic of time and space tradeoffs in algorithms, which was not originally included in the intersection. For completeness and in following with the suggested guidelines given in this figure, that topic will be included.

Table 3-65 shows the updated listing of topics for this knowledge unit based upon all of the analysis of the CC2001 document.

AL1. Basic algorithmic analysis	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Asymptotic analysis of upper and average complexity bounds	**X**	**X**	**X**
Big O, little o, omega, and theta notation	**X (Big O only)**	**X (Big O only)**	**X (Big O only)**
Empirical measurements of performance	**X**	**X**	**X**
Standard complexity classes	**X**	**X**	**X**
Time and space tradeoffs in algorithms	**X**	**X**	**X**
Identifying differences among best, average, and worst case behaviors			
Using recurrence relations to analyze recursive algorithms			

Table 3-65:AL1. Basic Algorithmic Analysis topics covered in programming-first CS1-CS2

- AL3. Fundamental computing algorithms; Simple numerical algorithms; sequential and binary search algorithms; quadratic and O(N log N) sorting algorithms; hashing; binary search trees

In Table 3-47, all of these topics are indicated as being part of the intersection except for hashing, which is not indicated in the functional-first approach. This helps once again strengthen the arguments presented in §3.2.5.6 for the inclusion of additional topics that were not clear from the initial "shallow" reading of the syllabus descriptions. Originally, hashing was not included in the functional-first topics because there is no indication of hashing in the sample syllabi.

However, given the intent of the introductory courses to discuss hashing, it will be included in the intersection and Table 3-66 shows the finished intersection for this knowledge unit.

AL3. Fundamental computing algorithms	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Simple numerical algorithms	**X**	**X**	**X**
Sequential and binary search algorithms	**X**	**X**	**X**
Quadratic sorting algorithms (selection, insertion)	**X**	**X**	**X**
O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)	**X**	**X**	**X**
Hash tables, including collision-avoidance strategies	**X**	**X**	**X**
Binary search trees	**X**	**X**	**X**
Representations of graphs (adjacency list, adjacency matrix)			**X**
Depth- and breadth-first traversals			
Shortest-path algorithms (Dijkstra's and Floyd's algorithms)			
Transitive closure (Floyd's algorithm)			
Minimum spanning tree (Prim's and Kruskal's algorithms)			
Topological sort			

Table 3-66: AL3. Fundamental Computing Algorithms topics covered in programming-first CS1-CS2

- AR1. Digital logic and digital systems: Logic gates; logic expressions

This knowledge unit is not covered at all in any of the programming-first introductory sequences. It is not part of the Discrete Structures knowledge area; however, sample syllabi for the discrete mathematics course, found in Appendix B of CC2001, shows this knowledge unit as part of the coverage. Therefore, these topics will not be included in the intersection, but left for inclusion in such a discrete mathematics course.

- PL6. Object-oriented programming: Object-oriented design; encapsulation and information-hiding; separation of behavior and implementation; classes, subclasses, and inheritance; polymorphism; class hierarchies

Table 3-53 shows that all of the topics presented for this knowledge unit are included in the intersection. Also note that the topic of internal representations of class and method tables is not presented in this listing. This supports the decision to leave it out of the intersection. No changes are required of the topic list for this knowledge unit.

- SE1. Software design: Fundamental design concepts and principles; object-oriented analysis and design; design for reuse

In Table 3-55, the topics indicated for the intersection do not include design for reuse. It was argued in §3.2.5.6 that perhaps the reason this topic is not given as part of some of the approaches is because it is not as important to non-object-oriented programming. Clearly, the inclusion of this topic for all introductory sequences weakens this point, and gives the indication that this topic should be in the intersection. This change in topic inclusion is given in Table 3-67.

SE1. Software design	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Design for reuse	**X**	**X**	**X**
Design patterns	**X**	**X**	**X**
Fundamental design concepts and principles	**X**	**X**	**X**
Object-oriented analysis and design	**X**	**X**	**X**
Structured design	**X**		**X**
Component-level design			
Software architecture			

Table 3-67: SE1. Software Design topics covered in programming-first CS1-CS2

- SE2. Using APIs: API programming; class browsers and related tools; programming by example; debugging in the API environment

Table 3-56 indicates that all of the topics presented for this knowledge unit are included in the intersection. The inclusion of all of these topics was argued for in §3.2.5.5 and this comparison reaffirms that inclusion.

- SE3. Software tools and environments: Programming environments; testing tools

Table 3-56 shows that these topics are included in the intersection. In §3.2.5.4, it was argued that testing tools as well as modeling tools should be included. While modeling tools are not included in this list, they still seem appropriate for the programming-first approaches to the introductory sequence and there is no need to change the topics for this knowledge unit.

- SE5. Software requirements and specifications: Importance of specification in the software process

This topic is not one that is listed as a topic for this knowledge unit. However, this explanation does seem to coincide with the argument for how the topics in this section should be presented that was given in section 1.2.3.7. This discrepancy in language makes it hard to determine the exact topics, but since it does not indicate an absence of an important topic to the introductory sequence, the current topics will remain as listed in Table 3-57.

- SE6. Software validation: Testing fundamentals; test case generation

Table 3-58, shows an inclusion of these topics as well as the additional topic of object-oriented testing. These topics will remain unchanged.

3.2.8 Topics Included in some, but not All Programming-first Approaches

Only a few topics are covered by only one or two of the introductory sequences. Table 3-68 presents three topics that are covered by imperative-first and functional-first, but not objects-first. No topics are covered by imperative-first and objects-first only, or by objects-first and functional-first only.

PF3. Fundamental Data Structures

- Implementation of stacks, queues, and hash tables
- Implementation of trees and graphs

SE1. Software Design

- Structured design

Table 3-68: Topics covered only by imperative-first and functional-first CS1 & CS2

Based on these findings, it is necessary to re-examine these two units to see if these topics should actually be included in the final intersection of topics.

For PF3, it seems logical that implementation of data structures would be presented in a CS2 course, even in the objects-first style. In the discussion of these data structures in a course, the implementation will be discussed at some level.

Perhaps it is the case that for objects-first, the focus of programming assignments for the course is not to implement the data structures, but rather use them in the large-scale programming projects that the sample curriculum suggests. However, this does not preclude an instructor from introducing the implementation and discussing the implementation issues with the students. Therefore, this topic will be included in the final intersection of topics, and these changes are reflected in Table 3-69.

PF3. Fundamental Data Structures	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Primitive types	X	X	X
Arrays	X	X	X
Records	X	X	X
Strings and string processing	X	X	X
Data representation in memory	X	X	X
Static, stack, and heap allocation	X	X	X
Runtime storage management	X	X	X
Pointers and references	X	X	X
Linked structures	X	X	X
Implementation strategies for stacks, queues, and hash tables	X	X	X
Implementation strategies for graphs and trees	X	X	X
Strategies for choosing the right data structure	X	X	X

Table 3-69: PF3. Fundamental Data Structures topics covered in programming-first CS1-CS2

For SE1, the topic of structured design is included in two of the three courses. Since the focus of the objects-first curriculum is object-oriented design, it is reasonable to assume that any other types of design would not be discussed. However, one could view object-oriented design as a type of structured design. It certainly provides the student with a structure for their programs and prevents the so-called "spaghetti code" problem. Therefore, it will be included in the final intersection of topics; this change is reflected in Table 3-70.

SE1. Software design	IF CS1-CS2	OF CS1-CS2	FF CS1-CS2
Design for reuse	X	X	X
Design patterns	X	X	X
Fundamental design concepts and principles	X	X	X
Object-oriented analysis and design	X	X	X
Structured design	X	X	X
Component-level design			
Software architecture			

Table 3-70: SE1. Software Design topics covered in programming-first CS1-CS2

Table 3-71 presents a single topic that is only covered in imperatives-first and Table 3-72 presents a single topic that is only covered in functional-first. There are no topics that are only covered by the objects-first approach. Since these topics are not represented in all three introductory sequences, they are not included in the final set of topics of the intersection of the three programming-first approaches.

SE3. Software Tools and Environments
• Requirements analysis

Table 3-71: Topics covered only by imperative-first CS1 & CS2

AL3. Fundamental Computing Algorithms
• Representations of graphs (adjacency list, adjacency matrix)

Table 3-72: Topics covered only by functional-first CS1 & CS2

3.3 Conclusion

This chapter began by discussing the overall structure and content of the CC2001 document and has finished by creating a formal intersection of topics for the programming-first approaches to the introductory curriculum. This list of topics is made up from the topics in eighteen knowledge units. These knowledge units are presented in §3.2.3.1. The formal listing of topics is presented in Tables 3-42, 3-43, 3-45, 3-48, 3-49, 3-50, 3-51, 3-52, 3-53, 3-54, 3-56, 3-57, 3-58, 3-59, 3-65, 3-66, 3-69, and 3-70.

Several observations can be made about the consistency of the language of the CC2001 document. The names of knowledge areas and knowledge units are consistently used throughout the document. However, when one reaches the topic coverage level, the consistency begins to break down. This is especially evident when reading the sample syllabi for the introductory courses that are given in CC2001's Appendix B. Most notably, the objects-first syllabi did not use the correct names of the topics for the knowledge units. This made it difficult to decide which topic areas to include. Also, in some cases, entire sets of topics were not given a sufficient level of detail, but simply lumped under the category of the knowledge unit name. One was left to assume that this meant all of the topics would be covered. There were also several instances where entire sets of topics were simply left out of the course descriptions.

The list of common topics that was created forms the basis for the work on the assessment instrument. This list of topics covered in the CS1-CS2 sequence by all three programming-first approaches is presented in Table 3-73. This list represents the results of combining information gathered from the syllabus topic descriptions, the number of knowledge units covered, the amount of time devoted to each knowledge unit, the information in Chapter 7 of CC2001, and topics that were included in two of the three approaches.

PF1. Fundamental Programming Constructs
- Basic syntax and semantics of a higher-level language
- Variables, types, expressions, and assignment
- Simple I/O
- Conditional and iterative control structures
- Functions and parameter passing
- Structured decomposition

PF2. Algorithms and Problem-Solving
- Problem-solving strategies
- The role of algorithms in the problem-solving process
- Implementation strategies for algorithms
- Debugging strategies
- The concept and properties of algorithms

PF3. Fundamental Data Structures

- Primitive types
- Arrays
- Records
- Strings and string processing
- Data representation in memory
- Static, stack, and heap allocation
- Runtime storage management
- Pointers and references
- Linked structures
- Stacks, queues, and hash maps
- Graphs and trees
- Strategies for choosing the right data structure

PF4. Recursion

- The concept of recursion
- Recursive mathematical functions
- Simple recursive procedures
- Divide-and-conquer strategies
- Recursive backtracking
- Implementation of recursion

AL1. Basic Algorithmic Analysis

- Asymptotic analysis of upper and average complexity bounds
- Big O notation
- Standard complexity classes
- Empirical measurements of performance
- Time and space tradeoffs in algorithms

AL3. Fundamental Computing Algorithms

- Simple numerical algorithms
- Sequential and binary search algorithms
- Quadratic sorting algorithms (selection, insertion)
- O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)
- Hash tables, including collision avoidance strategies
- Binary search trees

AL5. Basic Computability

- Tractable and intractable problems
- Uncomputable functions

PL1. Overview of programming languages

- History of programming languages
- Brief survey of programming paradigms: Procedural languages, Object-oriented languages, Functional languages, Declarative, non-algorithmic languages, Scripting languages
- The effects of scale on programming methodology

PL2. Virtual Machines

- The concept of a virtual machine
- Hierarchy of virtual machines
- Intermediate languages
- Security issues arising from running code on alien machine

PL4. Declarations and Types

- The conception of types as a set of values together with a set of operations
- Declaration models (binding, visibility, scope, and lifetime)
- Overview of type checking
- Garbage collection

PL5. Abstraction Mechanisms
- Procedures, functions, and iterators as abstraction mechanisms
- Parameterization mechanisms (reference vs. value)
- Activation records and storage management
- Type parameters and parameterized types
- Modules in programming languages

PL6. Object-oriented Programming
- Object-oriented design
- Encapsulation and information-hiding
- Separation of behavior and implementation
- Classes and subclasses
- Inheritance (overriding, dynamic dispatch)
- Polymorphism (subtype polymorphism vs. inheritance)
- Class hierarchies
- Collection classes and iteration protocols

SP1. History of Computing
- Prehistory – the world before 1946
- History of computer hardware, software, networking
- Pioneers of computing

SE1. Software design
- Fundamental design concepts and principles
- Design patterns
- Structured design
- Object-oriented analysis and design
- Design for reuse

SE2. Using APIs
- API programming
- Class browsers and related tools
- Programming by example
- Debugging in the API environment

SE3. Software Tools and Environments
- Programming environments
- Design modeling tools
- Testing tools

SE5. Software Requirements and Specifications
- Requirements elicitation
- Functional and nonfunctional requirements

SE6. Software Validation
- Testing fundamentals, including test plan creation and test case generation
- Object-oriented testing

Table 3-73: Final List of Intersection Topics

Chapter 4

Refining the Topic List

4.1 Introduction

Looking at Table 3-73 showing the topics included in the intersection, one sees that the amount of material covered by the introductory sequence is quite substantial. More than 75 topics are in the intersection, several of them encompassing multiple sub-topics. This number is too large for careful evaluation by one examination.

In this chapter, we will eliminate some of the topics in order to create a more manageable assessment instrument. During this analysis, if a topic is covered extensively in the introductory sequence, we keep it as part of the topics to be used to create the assessment instrument. If a topic is covered in the introductory sequence, but also covered, possibly in more depth, in an upper-level course, it is eliminated. Because this assessment focuses on the programming-first approaches to the introductory sequence, topics that are more closely related to programming and program design issues are given higher priority than those issues that are not as closely related. The result is a smaller set of topics that is more manageable for testing by an assessment instrument.

Learning objectives were also considered among criteria for eliminating topics. They will be discussed in more detail in Chapter 5.

4.2 Topics Removed

The topics discussed in this section are those that were removed from the topic list for the assessment instrument. It is important that those interested in the introductory computer science sequence understand that these topics are not irrelevant to the curriculum nor should they be removed from the course content in the first year; rather, student's abilities in these areas will not be assessed by this instrument. If assessment of these issues is needed, it must be gathered using other methods. Knowledge units that did not have topics removed are not discussed in this section.

For some of the topics, the decision to remove them from the assessment was not an easy one. All of the topics in the intersection are topics that should be covered in any introductory sequence. However, some of the topics present challenges to assessment by exam. Among them are topics that deal with the process of design or debugging. These topics are not easy to assess with a traditional time-limited paper-and-pencil exam (which this exam is). Therefore, topics that fell into this category were generally eliminated, even though their importance to student understanding cannot be overstated. This exam does not assess every possible topic in an introductory sequence, so instructors will need to supplement this exam with assessment throughout the introductory sequence that will show student proficiency with some of the missing topics. In the next sections, we will look at the knowledge unit topics that will not be assessed by our exam organized by reason for their elimination from this assessment instrument.

4.2.1 Topics Eliminated because of Time Constraints

There are many topics in the original intersection list that require a significant amount of questions to be asked in order to discern a student's understanding or would require questions that can take a considerable amount of thinking and preparation before a student can effectively answer. The proper amount of time that needs to be given for exploration of a topic as well as synthesis of a solution can be achieved in many ways. Some suggestions and ideas are given in §4.2.1.1 – 4.2.1.3.

These topics are further broken down into three categories: topics assessing the process of programming and program development, topics assessing student understanding of concepts underlying programming and program development, and topics concerned with students exploring programming through the use of advanced programming techniques, algorithm analysis, or development tools and environments.

4.2.1.1 Programming process

These topics deal primarily with the process of creating a program or the process of designing a solution to a particular problem:

- Structured decomposition (from PF1)
- Problem-solving strategies (from PF2)
- Implementation strategies for algorithms (from PF2)
- Debugging strategies (from PF2)

- Object-oriented design (from PL6)
- Fundamental design concepts and principles (from SE1)
- Design patterns (from SE1)
- Object-oriented analysis and design (from SE1)
- API programming (from SE2)
- Programming by example (from SE2)
- Debugging in the API environment (from SE2)
- Testing fundamentals, including test plan creation and test case generation (from SE6)
- Object-oriented testing (from SE6)

Assessing student knowledge in any of the above areas can be done in a laboratory setting. Students can obviously demonstrate their abilities in these areas through the completion of programming projects requiring them to design and implement a solution. These programs then can be assessed on design characteristics as well as correct functionality so that a student can demonstrate understanding of design techniques and strategies. This will also allow problems of sufficient complexity to be given to the student to allow them to use more sophisticated design techniques without the time constraints this exam will have.

Another possible way to assess the design abilities of a student is assessment through controlled observation. An assessor could be brought in to watch a student construct a solution to a problem. The student could be asked to describe the process they are using, or the assessor could ask questions about the design decisions the student has made while they have developed the solution to the problem. The assessor can then assign a grade to the student based on techniques used for solving a problem, rather than the actual solution.

One could argue, however, that these ideas will be tested in the assessment any time that a student is asked to generate a piece of code. This in many ways is true. However, the questions that ask the student to create code to solve a particular problem also assess their ability to create a solution for that problem, not their general problem-solving or design abilities. Therefore, it will not be claimed that this assessment demonstrates a student's ability to problem solve in general, or that a student is competent in any particular design technique, but rather that the student has demonstrated the ability to solve problems within a specific topic area in this discipline.

Another argument that could be raised about the exclusion of these topics is that when one is dealing with languages as complex as many of the popular CS1 languages, the idea of the API (Application Programmers Interface[14]) and its place in learning a new language cannot be overemphasized. Since it would be unreasonable to assume that students memorize all the methods from the multitude of classes that could be referenced by this exam, it is possible that an API will be used as a means to provide students supporting information about code examples used in the exam.

However, the API is only being used as a tool to help the students in solving some other problem. The notable example from the exam that was created for this dissertation is the use of the API for the String class in Java. This class is fairly large and provides many useful methods that can be used by a student for string processing tasks. String

[14] Similar to the idea of a language reference manual, an API provides a developer with information about a language and the libraries and library methods contained within it. APIs can also be provided for libraries and packages developed as external projects in a language to help developers learn about its features.

processing is a topic that is included on the assessment, and giving the student access to the API for the String class is a way to ensure that the student has access to methods that will help them process strings and not have to be concerned with memorizing names of methods before the exam. See Chapter 6 for an in-depth discussion of specific exam questions. This use of APIs is not clearly indicated by the topics given in the SE2 knowledge unit, so the topics are not included as formal topics for exam creation.

Last, an argument could be raised against the exclusion of software testing from the exam. Testing can obviously be assessed using laboratory exercises where students are asked to test code or create test cases for code they have written. However, students could be asked to do the same thing on an exam as well, even if the testing can never be executed. There are two reasons that this method of assessment was not used in this exam. The first is a time concern. Asking students to create proper test cases for a problem will require that they have studied a problem for at least some time. The second is a concern over coverage of testing methodology. There are several testing methodologies available, and an instructor is free to choose whatever methodology fits more appropriately into their course. Therefore, it could be the case that test cases or test plans were never formally discussed, and asking the student to write one would not be feasible due to lack of experience with that type of testing. It is for both of these reasons that the topics have not been included on this assessment.

4.2.1.2 Concepts underlying the programming process

These topics are still concerned with the process of creating a program or designing a solution, but are not necessarily issues of process, rather, they are concepts and ideas that underlie the actual process of programming. These topics are:

- The role of algorithms in the problem-solving process (from PF2)
- The concept and properties of algorithms (from PF2)
- Time and space tradeoffs in algorithms (from AL1)
- The effects of scale on programming methodology (from PL1)
- Procedures, functions, and iterators as abstraction mechanisms (from PL5)
- Separation of behavior and implementation (from PL6)
- Design for reuse (from SE1)

These topics require a student to possess not only an understanding of the process of programming and design, but also to understand the "why" of those processes. Asking a question about a "why" forces an explanation. Explanations must be given in the form of natural language and usually involve one or more sentences and time for the answerer to prepare their thoughts about the subject. This pushes questions of "why" into a category of discussion topics or essays. All of this requires an amount of time not available in this exam.

Assessment of these topics could be achieved through a graded and guided in-class discussion or debate about practices of good design and design ideas, or even through assigned essays or position papers about these ideas. Both methods would allow students to express themselves free of the time pressure this particular exam and would also allow students time to reflect on these ideas and their importance.

4.2.1.3 Exploring different aspects of programming

These topics are topics that are related to programming activities, most specifically certain kinds of algorithms or algorithm analysis that could be performed at the introductory level. Also included are topics dealing with programming environments or tools. These topics are:

- Recursive backtracking (from PF4)
- Empirical measurements of performance (from AL1)
- Collision avoidance strategies for hash tables (from AL3)
- Class browsers and related tools (from SE2)
- Programming environments (from SE3)
- Design modeling tools (from SE3)
- Testing tools (from SE3)

The first three topics are topics that deal with specific algorithms or techniques. For a student to demonstrate an understanding of recursive backtracking, a problem of sufficient size must be given to the student for them to solve using this method. However, problems of this size will require much time in the design stage of the problem-solving process. The topic of empirical measurements could be better assessed as a laboratory exercise that engages the students in determining the empirical measurement of the runtime of algorithms and asks them to compare that to their knowledge of the Big-O running time. The topic of collision avoidance strategies for hash tables contains within it numerous strategies for collision avoidance not all of which would be covered by all instructors. Therefore, instructors are encouraged to expose the students to various collision avoidance strategies and compare them in some form of laboratory or other programming exercise.

The final four topics deal with tools and environments. Since all instructors are free to use whatever environment and tools they wish, it is impractical to try to assess students on their abilities with these tools in a uniform way. Assessment of these skills must once again be done via laboratory exercises in individual courses.

4.2.2 Topics Eliminated because of Deeper Coverage in Advanced Courses

Many topics in the original intersection list will be discussed at the introductory level to some degree, but will be discussed to a greater degree in subsequent courses in the curriculum. Due to institutional differences, the depth at which many of these topics are covered can vary. Therefore, finding an adequate level to assess these topics poses a great challenge. Therefore, it is left to individual instructors to assess these topics in homework, problem sets, or in-class examinations to the level at which the topics were covered and to subsequent courses to assess when a more thorough treatment of the topic is undertaken. These topics are either more advanced programming languages topics, systems topics, theory of computation topics, or more advanced software engineering topics. These topics are:

- Static, stack, and heap allocation (from PF3)
- Runtime storage management (from PF3)

- Recursive mathematical functions (from PF4)
- Tractable and intractable problems (from AL5)
- Uncomputable functions (from AL5)
- The concept of a virtual machine (from PL2)
- Hierarchy of virtual machines (from PL2)
- Intermediate languages (from PL2)
- Security issues arising from running code on alien machine (from PL2)
- Garbage collection (from PL4)
- Activation records and storage management (from PL5)
- Modules in programming languages (from PL5)
- Requirements elicitation (from SE5)
- Functional and nonfunctional requirements (from SE5)

4.2.3 Topics Eliminated because of Difficulty in Determining Material Coverage

Many topics in the original intersection list are very broad overviews of topics or topics that involve history of the discipline. These topics allow an instructor to have a large amount of freedom in what will be covered within particular topic and thus pose a serious problem for this assessment instrument. Since there are no explicitly given standards within these areas, it is almost impossible to ensure uniform coverage across institutions. Therefore, these topics will not be included in the assessment. These topics are:

- History of programming languages (from PL1)
- Brief survey of programming paradigms: Procedural languages, Object-oriented languages, Functional languages, Declarative, non-algorithmic languages, Scripting languages (from PL1)
- Prehistory – the world before 1946 (from SP1)
- History of computer hardware, software, networking (from SP1)
- Pioneers of computing (from SP1)

The type of assessment of these topics will need to be determined in a large part by the individual instructors based on what depth and type of coverage these topics are given. Because of their general, broad nature, it is even difficult to give good suggestions. However, the idea of essays or exploratory research in the areas of history is an option that instructors may wish to pursue.

4.2.4 Records

The topic of records has presented a unique set of challenges. Records are a clear programming-language-specific construct, and, because not all languages support records, choice of language would greatly determine how assessment could proceed with this topic.

However, records also represent the idea of composite types in a more general sense, without the compulsion to delve into object-oriented ideas. Therefore, imperatives-first courses would most definitely discuss them in the introductory sequence. Since objects are simply another type of composite type, the concept of composite types is present in all of the introductory approaches being studied. Due to this fact and our time constraints, the topic of records proper is not being included in this assessment; however, the notion of composite types will be present with any question that involves objects.

4.3 Topics Remaining

After the elimination of the topics discussed above, the topics that are left present a tighter picture of topical coverage of this assessment instrument. This new list of topics is presented in Table 4-1: Revised List of Topics.

4.4 Conclusion

It is most important to reiterate that the intent of this chapter is to make the assessment instrument easier to construct and to provide a picture of the common core material to all programming-first CS1-CS1 courses, not to imply that the topics eliminated should not be covered or assessed in the introductory sequence. For many of the eliminated topics, alternative suggestions for assessment were given. As stated before, it is most important not to consider this assessment instrument as the only form of assessment applicable to students in the introductory sequence. Students should be assessed in multiple ways to have a complete picture of student performance in the introductory sequence

PF1. Fundamental Programming Constructs
- Basic syntax and semantics of a higher-level language
- Variables, types, expressions, and assignment
- Simple I/O
- Conditional and iterative control structures
- Functions and parameter passing

PF3. Fundamental Data Structures
- Primitive types
- Arrays
- Strings and string processing
- Data representation in memory
- Pointers and references
- Linked structures
- Stacks, queues, and hash maps
- Graphs and trees
- Strategies for choosing the right data structure

PF4. Recursion
- The concept of recursion
- Simple recursive procedures
- Divide-and-conquer strategies
- Implementation of recursion

AL1. Basic Algorithmic Analysis
- Asymptotic analysis of upper and average complexity bounds
- Big O notation
- Standard complexity classes

AL3. Fundamental Computing Algorithms
- Simple numerical algorithms
- Sequential and binary search algorithms
- Quadratic sorting algorithms (selection, insertion)
- O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)
- Hash tables
- Binary search trees

PL4. Declarations and Types
- Overview of type checking
- The conception of types as a set of values together with a set of operations
- Declaration models (binding, visibility, scope, and lifetime)

PL5. Abstraction Mechanisms
- Parameterization mechanisms (reference vs. value)
- Type parameters and parameterized types

PL6. Object-oriented Programming
- Encapsulation and information-hiding
- Classes and subclasses
- Inheritance (overriding, dynamic dispatch)
- Polymorphism (subtype polymorphism vs. inheritance)
- Class hierarchies
- Collection classes and iteration protocols

Table 4-1: Revised List of Topics

Chapter 5

Learning Objectives

5.1 Mining CC2001 for Learning Objectives

The previous two chapters analyzed the CC2001 document to produce a set of topics satisfying two constraints: each topic is covered by all three programming-first approaches to the introductory curriculum, and each lends itself to assessment by a paper and pencil time-limited exam. We now turn to an examination of the learning objectives that incorporate the topics from the intersection and that provide a context for what skills students should have after completing instruction in one of the topics from the knowledge units.

To some, this could seem to be the reverse of what is typically done to create a course, where learning objectives are usually outlined before the course is created. While this is certainly true, the unique structure of the CC2001 led to some problems with approaching the creation of the assessment in this way. The sample syllabi and other course materials are given with topical coverage, but not learning objective coverage. Furthermore, the learning objectives are not given explicit associations with topics.

Therefore, looking at the list of learning objectives gives all learning objectives for that knowledge unit, not learning objectives for each topic.

Hence, to avoid looking at learning objectives for knowledge units that would not be included in the assessment at all and to further focus only for the appropriate learning objectives for the topics that would actually appear on the exam, the learning objectives were assembled after the list of topics was finalized.

Appendix A of CC2001, which includes the knowledge units and topics for each knowledge unit, also lists learning objectives for each knowledge unit. This chapter will show which of these learning objectives correlate with topics in the intersection previously defined. Learning objectives that correlate with topics that are not included in the intersection are eliminated. Topics that are in the intersection that do not correlate with a specific learning objective are noted, and new learning objectives are recommended to fill these gaps.

5.2 Learning Objectives from Programming Fundamentals

5.2.1 PF1. Fundamental Programming Constructs Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- Basic syntax and semantics of a higher-level language
- Variables, types, expressions, and assignment
- Simple I/O
- Conditional and iterative control structures

- Functions and parameter passing

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in left column of Table 5-1. All of the learning objectives for this knowledge unit are included in our list of learning objectives for the assessment, except for 5, because structured decomposition is not being included in the assessment.

The meaning of the phrase *analyze and explain* given in learning objective 1 above and elsewhere is not clearly defined either in the learning objective or elsewhere in CC2001 itself. Therefore, the terms will be defined for our purposes as the ability to describe the inputs, outputs, and the procedures used to compute the output from the input. For example, given a program and an input, students should be able to state what output would be produced, and articulate in words the functionality of a particular piece of code. It is important to note that if the program contains more than one method or function, students should be able to state what the responsibilities for each method or function are.

In the second learning objective, the phrase *modify and expand* is also vague. The definition of this phrase will be considered to be the ability of students, when given a simple program, to add elements to it, to change its functionality based on directions given, or to generalize it in some way.

Table 5-1 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

Original Learning Objectives	Revised Learning Objectives
1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs covered by this unit. 2. Modify and expand short programs that use standard conditional and iterative control structures and functions. 3. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions. 4. Choose appropriate conditional and iteration constructs for a given programming task. 5. Apply the techniques of structured (functional) decomposition to break a program into smaller pieces. 6. Describe the mechanics of parameter passing.	1. For simple programs involving the fundamental programming constructs covered by this unit, describe the inputs, outputs, and the procedures used to compute the output from the input. 2. When given a short program that uses standard conditional and iterative control structures and functions, demonstrate the ability to add elements to it, to change its functionality based on directions given, or to generalize in a way described by the directives. 3. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions. 4. Choose appropriate conditional and iteration constructs for a given programming task. 5. Describe the mechanics of parameter passing.

Table 5-1: Comparison of old and revised learning objectives for PF1. Fundamental Programming Constructs

5.2.2 PF3. Fundamental Data Structures Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- Primitive types
- Arrays
- Strings and string processing
- Data representation in memory
- Pointers and references
- Linked structures
- Stacks, queues, and hash tables
- Graphs and trees
- Strategies for choosing the right data structure

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in the left column of Table 5-2. The only learning objective that will not be included in the final list of learning objectives is 2 because the topics about allocation and runtime storage management are not included in our intersection for the exam.

Again, there are imprecise terms that I propose definitions for. In learning objective 4, we see the use of the word *implementation*. It is being used in the computer science sense of the student's ability to create source code that when run, will instruct the computer to perform some task. In this case, we want the student to be able to create source code that defines data structures.

In learning objective 5, the definition of *compare* will be similar to the definitions of *describe and discuss*; that is, when students are asked about the different implementations of data structures, they should be able to articulate in words the similarities and differences between them.

In learning objective 7, we see the terms *compare and contrast*. The ability to compare and contrast is an extension of a student's ability to *compare*, so the student should be able to articulate in words similarities as well as differences among the two ways to implement data structures.

Two topics from PF3 are not explicitly covered by the learning objectives: string processing and the data structures graphs and trees (see learning objective 6).

An intuitive definition of string processing would be any computation that involves strings, i.e. sequences of characters. However, exactly which computations are we most concerned with in CS1 and CS2? This question is not answered by this knowledge unit.

Therefore, let us consider the string processing topics covered in a number of the more popular texts for CS1. The texts we will look at are not an exhaustive sample; however, they do represent all three of the programming-first approaches (Harvey and Wright 1999; Hanly and Koffman 2003; Dietel and Dietel 2004; Dietel and Dietel 2005; Dietel and Dietel 2005; Horstmann 2006; Savitch 2006; Lewis and Loftus 2007). These texts were chosen based on their popularity as evidenced by discussion of them at conferences, promotion on various publisher's websites, and sales rankings. Searching the table of contents, indexes, and chapters that deal with strings in these texts leads to a commonality in the string processing topics that are covered. There is always a part of the chapters explaining what a string is, how to create a string, and how assignment works with strings, which for the purposes of this topic and learning objective are already covered by learning objectives 1 and 2 of this section.

The other common operations consist of creating substrings (which can also be categorized as parsing), using substrings, string concatenation (combining or joining end-to-end two or more separate strings together to create one larger string), and string comparison. Also, each text makes note of string operations that are built into the specific programming language. Therefore, the new learning objective for this section will consist of those common operations (see item 9 in the right column of Table 5-2).

Resolving the omission of graphs and trees from learning objective 6 is fairly trivial. We can simply add these two other data structures to the list, and remove records, which are not covered in our intersection.

Table 5-2 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

Original Learning Objectives	Revised Learning Objectives
1. Discuss the representation and use of primitive data types and built-in data structures.	1. Discuss the representation and use of primitive data types and built-in data structures.
2. Describe how the data structures in the topic list are allocated and used in memory.	2. Describe common applications for each data structure in the topic list.
3. Describe common applications for each data structure in the topic list.	3. Create executable source code for the user-defined data structures in a high-level language.
4. Implement the user-defined data structures in a high-level language.	4. Articulate the similarities and differences among alternative implementations of data structures with respect to performance..
5. Compare alternative implementations of data structures with respect to performance.	**5. Write programs that use each of the following data structures: arrays, strings, linked lists, stacks, queues, hash tables, trees, and graphs.**
6. Write programs that use each of the following data structures: arrays, records, strings, linked lists, stacks, queues, and hash tables.	6. Articulate the similarities and differences between dynamic and static data structure implementations, focusing especially on the costs and benefits of each.
7. Compare and contrast the cost and benefits of dynamic and static data structure implementations.	7. Choose the appropriate data structure for modeling a given problem.
8. Choose the appropriate data structure for modeling a given problem.	**8. Demonstrate ability to parse, concatenate, and compare strings, use substrings, and describe the various types of operations that are built into a high-level programming language for use with strings.**

Table 5-2: Comparison of old and revised learning objectives for PF3. Fundamental Data Structures

5.2.3 PF4. Recursion Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- The concept of recursion
- Simple recursive procedures
- Divide-and-conquer strategies
- Implementation of recursion

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in the left column Table 5-3. For this knowledge unit, the topics of recursive mathematical functions and recursive backtracking are not included in our intersection but are part of this knowledge unit in CC2001. None of the learning objectives for this section address the topic of recursive mathematical functions. The only learning objective that talks about recursive backtracking is 7 and will be removed.

Table 5-3 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

Original Learning Objectives	Revised Learning Objectives
1. Describe the concept of recursion and give examples of its use.	1. Describe the concept of recursion and give examples of its use.
2. Identify the base case and the general case of a recursively defined problem.	2. Identify the base case and the general case of a recursively defined problem.
3. Compare iterative and recursive solutions for elementary problems such as factorial.	3. Compare iterative and recursive solutions for elementary problems such as factorial.
4. Describe the divide-and-conquer approach.	4. Describe the divide-and-conquer approach.
5. Implement, test, and debug simple recursive functions and procedures.	5. Implement, test, and debug simple recursive functions and procedures.
6. Describe how recursion can be implemented using a stack.	6. Describe how recursion can be implemented using a stack.
7. Discuss problems for which backtracking is an appropriate solution.	7. Determine when a recursive solution is appropriate for a problem.
8. Determine when a recursive solution is appropriate for a problem.	

Table 5-3: Comparison of old and revised learning objectives for PF1. Fundamental Programming Constructs

5.3 Learning Objectives from Algorithms and Complexity

5.3.1 AL1. Basic Algorithm Analysis Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- Asymptotic analysis of upper and average complexity bounds
- Big O notation
- Standard complexity classes

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in the left column Table 5-4. Learning objectives 1 and 2 will be modified to only discuss Big O notation because the only notation included in our intersection is Big O.

Learning objective 4 and learning objective 5 will be eliminated, because both are concerned with the topic of recurrence relations, which is not included in the intersection of topics used to create our exam.

The only topic that does not seem to be adequately covered by the learning objectives is standard complexity classes, so a learning objective will be added for recognition of the standard complexity classes.

Table 5-4 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

Original Learning Objectives	Revised Learning Objectives
1. Explain the use of Big O, omega, and theta notation to describe the amount of work done by an algorithm. 2. Use Big O, omega, and theta notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms. 3. Determine the time and space complexity of simple algorithms. 4. Deduce recurrence relations that describe the time complexity of recursively defined algorithms. 5. Solve elementary recurrence relations.	**1. Explain the use of Big O notation to describe the amount of work done by an algorithm.** **2. Use Big O notation to give asymptotic upper bounds on time and space complexity of algorithms.** 3. Determine the time and space complexity of simple algorithms. **4. Identify the standard complexity classes and arrange them in order of growth rate.**

Table 5-4: Comparison of old and revised learning objectives for AL1. Basic Algorithmic Analysis

5.3.2 AL3. Fundamental Computing Algorithms Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- Simple numerical algorithms
- Sequential and binary search algorithms
- Quadratic sorting algorithms (selection, insertion)
- O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)
- Hash tables
- Binary search trees

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in the left column of Table 5-5. Learning objectives 3 and 4 concern hash tables and collision avoidance strategies, which do not appear in the intersection of topics and are not needed for this list of learning objectives. Learning objective 6 should be eliminated, because it includes the topics on graphs and graph algorithms that are not included in the intersection of topics for our exam.

Topics that are *not* covered by these learning objectives that *are* included in the intersection are: simple numerical algorithms, sequential and binary search algorithms, and binary search trees.

"Simple numerical algorithms" does not have a clear definition in this knowledge unit. Cormen, Leiserson, and Rivest (1990), one of the leading books in the area of algorithms, has no chapter titles or section headings for "simple numerical algorithms", nor does the index provide an entry for "simple numerical algorithms" The first chapter of the text does not provide a definition for this term, either. A search of the texts for

CS1 that were used in section 5.2.2 to define string processing did not bring forth any definition of this term, either.

A Google search of “simple numerical algorithms” (including quotes) brought several references back to the posting of the CC2001 document on the web! Also included in the result set were course web pages that have that exact term in them. Unfortunately, these courses simply use the term and the other topics from this knowledge unit as part of their syllabus with no further definition of it.

It would appear that most instructors have an intuitive notion of what this term means without a clear definition. Two course websites, elaborate on the term a bit more. The first site, a curriculum document for the Programming I course in a high school, gives as examples of “simple numerical algorithms” counting, summing, averaging, and rounding (Dade Computer Programming I Description 2001). The second gives an assignment, in a class entitled “Object-Oriented Programming”, whose stated purpose is “To program a simple numerical algorithm in a C++ class” (Bond 2004). This assignment asks the student to implement equations for linear regression. It seems reasonable to include not only counting, summing, averaging, and rounding, but also the ability to translate into code any simple mathematical formula, such as simple subtraction, multiplication, division, and modulus, as well as slightly more complicated formulae like geometric area or computation of the discriminant, and even more complicated formulae like linear regression, Newton’s method, or Simpson’s rule. This topic might well be described by the blanket statement of simply translating formulae into programs.

A combination of all of these will be considered “simple numeric algorithms.” One other consideration that needs to be made is that, in some languages, many simple mathematical functions are already implemented, and the students should be able to use those in their computation as well. The learning objective that will encompass this idea will include the ability to implement these simple numeric algorithms in code. Therefore, we will add learning objective 8 to this section (see Table 5-5).

There is a brief mention of searching algorithms in one of the learning objectives, but it is also important to include the ability to implement these algorithms, which is not included in any of the learning objectives. To rectify this situation, searching has been added to learning objective 1.

Binary search trees are mentioned as a topic in this knowledge unit and appear in the intersection of topics created, but are not given any mention in the learning objectives. This knowledge unit is not concerned with implementation of a binary search tree, because that topic is included in PF3, Fundamental data structures. Because this is the algorithms and complexity knowledge unit, binary search trees should not be considered in an implementation in source code way, but rather how a binary search tree is useful in many of the algorithms discussed in this section, such as searching and sorting. It is important to note that learning objective 5 mentions “application-specific patterns in the input data.” Patterns in data directly affect how a binary search tree is constructed and could therefore affect its efficiency. Therefore, we will include a more specific mention of binary search trees in learning objective 4 and expect that issues with binary search trees will also be discussed for learning objective 5.

Table 5-5 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

Original Learning Objectives	Revised Learning Objectives
1. Implement the most common quadratic and O(N log N) sorting algorithms. 2. Design and implement an appropriate hashing function for an application. 3. Design and implement a collision-resolution algorithm for a hash table. 4. Discuss the computational efficiency of the principle algorithms for sorting, searching, hashing. 5. Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data. 6. Solve problems using the fundamental graph algorithms, including depth-first and breadth-first search, single-source and all-pairs shortest paths, transitive closure, topological sort, and at least one minimum spanning tree algorithm. 7. Demonstrate the following capabilities: to evaluate algorithms, select from a range of possible options, to provide justification for that selection, and to implement the algorithm in programming context.	1. Implement the most common **searching algorithms as well as the most common** quadratic and O(N log N) sorting algorithms. 2. Discuss the computational efficiency of the principle algorithms for sorting, searching **(including binary search trees), and** hashing. 3. Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data. 4. Demonstrate the following capabilities: to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in programming context. **5. Implement simple numerical algorithms, such simple arithmetic (addition, subtraction, multiplication, division, modulus), as well as known mathematical formulae (geometric area, discriminant, linear regression, Simpson's rule, etc.) in programs, using both user-defined and any language-provided mathematical functions needed.**

Table 5-5: Comparison of old and revised learning objectives for AL3. Fundamental Computing Algorithms

5.4 Learning Objectives from Programming Languages

5.4.1 PL4. Declarations and Types Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- The conception of types as a set of values together with a set of operations
- Declaration models (binding, visibility, scope, lifetime)
- Overview of type checking

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in the left column of Table 5-6. We omit, learning objective 1, which touches on the concept of declaration models but also includes the topic of programming-in-the-large, because those topics are not included in our intersection. Learning objectives 5 and 6 are also eliminated for the same reason.

The only topic in our intersection that appears to be missing from the learning objectives is the idea that a type is a set of values together with a set of operations. The idea of value is mentioned in learning objective 2, but only in the context of a variable, not a type specifically. The problem with this topic is that it is basically a definition, which is applied in the ideas tested by learning objectives 2 and 3. More specifically, one of the main reasons that type compatibility issues arise is because the values and operations that can be performed on one type may not be the same as another. Even though not explicit, this definition of type is incorporated into the learning objectives.

Table 5-6 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

Original Learning Objectives	Revised Learning Objectives
1. Explain the value of declaration models, especially with respect to programming-in-the-large. 2. Identify and describe the properties of a variable such as associated address, value, scope, persistence, and size. 3. Discuss type incompatibility. 4. Demonstrate different forms of binding, visibility, scoping, and lifetime management. 5. Defend the importance of type-checking in providing abstraction and safety.	1. Identify and describe the properties of a variable such as associated address, value, scope, persistence, and size. 2. Discuss type incompatibility. 3. Demonstrate different forms of binding, visibility, scoping, and lifetime management.

Table 5-6: Comparison of old and revised learning objectives for PF1. Fundamental Programming Constructs

5.4.2 PL5. Abstraction Mechanisms Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- Parameterization mechanisms (reference vs. value)
- Type parameters and parameterized types

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in the left column of Table 5-7. Learning objectives 1, 3, and 4 are all concerned with topics that are not included in our intersection and will be omitted.

The only intersection topics that seems to be missing from the learning objectives are type parameters and parameterized types. These are concerned with parametric polymorphism, which can be characterized as “a special type of polymorphism[15] in which type expressions are parameterized” (Sethi, 1996: 359). This type of polymorphism refers to the ability of a function to have parameters which are given generic types. The type given in the function definition is simply a named place holder. When the function is actually run, the type of the actual parameter passed in will become the type of the parameter for that function call.

This type of polymorphism is available in many languages. C/C++ calls this ability “templates”; Java 1.5 has just added this ability in the form of generics; ML supports this feature internally through the typing mechanism built into the language. Parameterized type expressions can be used for many parts of a program, including: the types for the parameters of functions, return types, and the type of elements stored in abstract data types. For this knowledge unit, it is important to add a learning objective requiring that students be familiar with the use of parameterized types in their introductory language (see objective 5 in Table 5-7).

Table 5-7 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

[15] Polymorphism can be defined as the “ability of subclasses to respond differently to the same messages” (vanDam et al., 1997:67). An example of polymorphism is the following. Many things in this world can fly (helicopters, 747s, ducks, and robins, to name a few). However, each of these objects flies in decidedly different ways. Each of these objects could derive from a common superclass (or interface) and inherit (or implement) the ability to fly. A program could be written that helps simulate air traffic conditions. This program only deals with elements that can fly (i.e. have that capability), so all of the objects mentioned qualify. During the course of the execution of the program, each object is told to fly. The program does not know which object it is speaking to, but each object will receive the message and “fly” appropriately.

Original Learning Objectives	Revised Learning Objectives
1. Explain how abstraction mechanisms support the creation of reusable software components. 2. Demonstrate the difference between call-by-value and call-by-reference parameter passing. 3. Defend the importance of abstractions, especially with respect to programming-in-the-large. 4. Describe how the computer system uses activation records to manage program modules and their data.	1. Demonstrate the difference between call-by-value and call-by-reference parameter passing. **2. Demonstrate the ability to use parameterized types in programs.**

Table 5-7: Comparison of old and revised learning objectives for PL5. Abstractions Mechanisms

5.4.3 PL6. Object-oriented Programming Learning Objectives

The following topics from this knowledge unit are included in the intersection:

- Encapsulation and information-hiding
- Classes and subclasses
- Inheritance (overriding, dynamic dispatch)
- Polymorphism (subtype polymorphism vs. inheritance)
- Class hierarchies
- Collection classes and iteration protocols

The learning objectives given for this knowledge unit in Appendix A of CC2001 are shown in the left column of Table 5-8. Learning objective 1 will be not be included because object-oriented design is not included in our intersection.

However, the topics of classes and subclasses, polymorphism, and collection classes do not seem to be covered in any of the learning objectives. Classes and subclasses are part of the discussion of inheritance, but do not specifically appear in any learning objective discussing inheritance. Therefore, these terms will be added to learning objective 4. Also, it is more common to see that relationships occur between classes, not objects. In fact, the specification for UML, the design tool used most commonly by object-oriented developers illustrates this fact (Rumbaugh, Jacobson, and Booch 1999). Therefore, the learning objective will be changed to reflect this usage.

Polymorphism is a topic that is alluded to in learning objective 6, but is not explicitly stated. Also, the difference between subtype polymorphism and inheritance is an idea presented in the topic list that is not given in any learning objective. Two new learning objectives will be created to incorporate these ideas.

Collection classes are alluded to in learning objective 7 but, once again, not explicitly stated. A new learning objective will be inserted to ensure that students are able to create and use collections.

Table 5-8 shows the original learning objectives for this knowledge unit and the newly revised learning objectives for this knowledge unit; changes are indicated in boldface.

Original Learning Objectives	Revised Learning Objectives
1. Justify the philosophy of object-oriented design and the concepts of encapsulation, abstraction, inheritance, and polymorphism. 2. Design, implement, test, and debug simple programs in an object-oriented programming language. 3. Describe how the class mechanism supports encapsulation and information-hiding. 4. Design, implement, and test the implementation of the "is-a" relationship among objects using a class hierarchy and inheritance. 5. Compare and contrast the notions of overloading and overriding methods in an object-oriented language. 6. Explain the relationship between the static structure of the class and the dynamic structure of the instances of the class. 7. Describe how iterators access the elements of a container.	1. Design, implement, test, and debug simple programs in an object-oriented programming language. 2. Describe how the class mechanism supports encapsulation and information hiding. 3. Design, implement, and test the implementation of "is-a" relationships among classes using a class hierarchy and inheritance. **Distinguish between the superclass and the subclasses in these relationships.** 4. Compare and contrast the notions of overloading and overriding methods in an object-oriented language. 5. Explain the relationship between the static structure of the class and dynamic structure of the instances of the class, **especially in the context of how dynamic dispatch is involved in subtype polymorphism.** 6. Describe how iterators access the elements of the collection. **7. Describe the difference between subtype polymorphism and inheritance.** 8. **Create a collection, insert elements into a collection, and iterate over elements in a collection.**

Table 5-8: Comparison of old and revised learning objectives for PL6. Object-oriented Programming

5.5 Conclusion

After compiling the list of topics for the intersection of programming-first CS1-CS2 and looking at the learning objectives that are given in Appendix A of CC2001 for each of the knowledge units described in CC2001, a final list of learning objectives for the topics in the intersection has been created. Learning objectives that do not deal with topics in the intersection have been eliminated, and the search has discovered that some learning objectives needed to be re-worded or added to fit all of the topics in the intersection. Table 5-9 gives the final list of all learning objectives for the topics in the intersection of programming-first CS1-CS2.

PF1. Fundamental Programming Constructs Learning Objectives

1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs covered by this unit.
2. Modify and expand short programs that use standard conditional and iterative control structures and functions.
3. Design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions.
4. Choose appropriate conditional and iteration constructs for a given programming task.
5. Describe the mechanics of parameter passing.

PF3. Fundamental Data Structures Learning Objectives

1. Discuss the representation and use of primitive data types and built-in data structures.
2. Describe common applications for each data structure in the topic list.
3. Implement the user-defined data structures in a high-level language.
4. Compare alternative implementations of data structures with respect to performance.
5. Write programs that use each of the following data structures: arrays, strings, linked lists, stacks, queues, hash tables, trees, and graphs.
6. Compare and contrast the cost and benefits of dynamic and static data structure implementations.
7. Choose the appropriate data structure for modeling a given problem.
8. Demonstrate ability to parse, concatenate, and compare strings, use substrings, and describe the various types of operations that are built into a high-level programming language for use with strings.

PF4. Recursion Learning Objectives

1. Describe the concept of recursion and give examples of its use.
2. Identify the base case and the general case of a recursively defined problem.
3. Compare iterative and recursive solutions for elementary problems such as factorial.

4. Describe the divide-and-conquer approach.
5. Implement, test, and debug simple recursive functions and procedures.
6. Describe how recursion can be implemented using a stack.
7. Determine when a recursive solution is appropriate for a problem.
8. Express a recursive mathematical function in terms of a base case and a recursive case.

AL1. Basic Algorithm Analysis Learning Objectives

1. Explain the use of Big O notation to describe the amount of work done by an algorithm.
2. Use Big O notation to give asymptotic upper bounds on time and space complexity of algorithms.
3. Determine the time and space complexity of simple algorithms.
4. Identify the standard complexity classes and arrange them in order of growth rate.

AL3. Fundamental Computing Algorithms Learning Objectives

1. Implement the most common searching algorithms as well as the most common quadratic and O(N log N) sorting algorithms.
2. Discuss the computational efficiency of the principle algorithms for sorting, searching (including binary search trees), and hashing.
3. Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data.
4. Demonstrate the following capabilities: to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in programming context.
5. Implement simple numerical algorithms, such simple arithmetic (addition, subtraction, multiplication, division, modulus), as well as known mathematical formulae (geometric area, discriminant, linear regression, Simpson’s rule, etc.) in programs, using both user-defined and any language-provided mathematical functions needed.

PL4. Declarations and Types Learning Objectives

1. Identify and describe the properties of a variable such as associated address, value, scope, persistence, and size.
2. Discuss type incompatibility.
3. Demonstrate different forms of binding, visibility, scoping, and lifetime management.

PL5. Abstraction Mechanisms Learning Objectives

1. Demonstrate the difference between call-by-value and call-by-reference parameter passing.
2. Demonstrate the ability to use parameterized types in programs.

PL6. Object-oriented Programming Learning Objectives

1. Design, implement, test, and debug simple programs in an object-oriented programming language.
2. Describe how the class mechanism supports encapsulation and information hiding.
3. Design, implement, and test the implementation of "is-a" relationships among classes using a class hierarchy and inheritance. Distinguish between the superclass and the subclasses in these relationships.
4. Compare and contrast the notions of overloading and overriding methods in an object-oriented language.
5. Explain the relationship between the static structure of the class and dynamic structure of the instances of the class, especially in the context of how dynamic dispatch is involved in subtype polymorphism.
6. Describe how iterators access the elements of the collection.
7. Describe the difference between subtype polymorphism and inheritance.
8. Create a collection, insert elements into a collection, and iterate over elements in a collection.

Table 5-9: Final List of Learning Objectives

Chapter 6

Creation and Critique of Exam

6.1 Introduction

The questions on our exam are designed to reflect the refined list of topics and learning objectives. The initial drafts of the test were given a fine-grained critique by three instructors of introductory courses at two different institutions. Both were 4-year undergraduate and graduate universities, one public, one private. Other faculty members at those same institutions also commented on and critiqued the exam. The final version incorporates these comments and criticisms; it is presented in full as Appendix A of this dissertation. The construction of the exam is discussed in this chapter. Questions presented are shown with point weightings. These weightings and discussion of grading the exam are given in Chapter 7. The results of administrating and grading it are discussed in Chapter 8.

6.2 Creating Questions

To paraphrase Lewis Carroll, “begin at the beginning.” This is much easier said than done with a list of about 50 topics to be covered. The starting point in the process of creating an exam of this type was not immediately obvious. Therefore, the exam was essentially created in pieces, each roughly corresponding to the knowledge units to be included in the exam. The topics motivated the various questions, and the learning objectives provided their foundation.

One problem was that many topics needed a code-based question to really assess the student's mastery of it. One of the original goals of the assessment was language-independence. However, this conflicts with the decision to make the assessment based on programming-first approaches. A good deal of the time in a programming-first sequence is spent on the language and on programming.

In an effort to keep this exam from being an exam about language X, an effort was made to make every question that had a coding component focus on topics that are independent of language-specifics. The lack of emphasis on an actual language also makes its way into the grading guideline for the exam. Chapter 7 discusses how the exam should be graded; it will be seen that syntactic goals are secondary to other ideas presented in the questions.

However, a language needed to be chosen for the creation of the exam. The language decision was made in deference to the test subjects for the exam: students in CSE 116 (CS2) at the University at Buffalo, SUNY. The CS1-CS2 introductory sequence at UB (CSE115-CSE116) is taught in Java, so Java was chosen for the language of implementation for all code-based questions. Instructors at other institutions can easily substitute other programming languages for Java.

The questions will be discussed in thematic units (primarily corresponding to knowledge unit, and grouped together for reference by number, as indicated in Table 6-1, 6-2, and 6-3 showing which topics are covered in which group). At the end of this chapter, each question on the exam is given a group coding that refers back to the groups discussed. In the exam itself, questions mostly appear grouped together in this way. The only notable exception is in a set of true-false questions that were grouped together by type of question rather than coverage (questions 46 – 50).

The groupings that are presented here are based primarily on the way the questions were developed. Topics from knowledge units that naturally went together had questions developed together. The numbering of the groups has no significance other than roughly corresponding to the order in which the first topic in that group appears in the overall list of topics in the intersection created for this exam.

KU[16]	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7
PF1	* Basic syntax and semantics of a higher-level language	* Variables, types, expressions, and assignment * Simple I/O * Conditional and iterative control structures * Functions and parameter passing					
PF3		* Strings and string processing	* Primitive types * Pointers and references	* Arrays * Linked structures * Stacks, queues, and hash maps * Graphs and trees * Strategies for choosing the right data structure			
PF4					* The concept of recursion * Simple recursive procedures * Divide-and-conquer strategies * Implementation of recursion		
AL1						* Asymptotic analysis of upper and average complexity bounds * Big O notation * Standard complexity classes	

[16] KU is knowledge unit. The knowledge unit names have been eliminated from this chart because of space constraints.

KU	Group 1	Group 2	Group 3	Group 4	Group 5	Group 6	Group 7
AL3		* Simple numerical algorithms		* Hash tables * Binary search trees		* Sequential and binary search algorithms * Quadratic sorting algorithms (selection, insertion) * O(N log N) sorting algorithms (Quicksort, heapsort, mergesort)	
PL4			* Overview of type checking * The conception of types as a set of values together with a set of operations * Declaration models (binding, visibility, scope, and lifetime)				
PL5			* Parameterization mechanisms (reference vs. value) * Type parameters and parameterized types				
PL6				* Collection classes and iteration protocols			* Encapsulation and information-hiding * Classes and subclasses * Inheritance (overidding, dynamic dispatch) * Polymorphism (subtype polymorphism vs. inheritance) * Class hierarchies

Table 6-1: Topics from knowledge units included in each group

6.3 Structure of Exam Question Groups

The following sections discuss the questions that were created for each group. Originally, the questions on the exam were in the basic order of these groups. However, the grouping of questions within the group was essentially random, in that they were organized in the order that they were written, which was determined essentially by chance and the particular inspiration I had on a particular day. The ordering was changed during the critique process. Therefore, no particular emphasis should be given to the ordering of the questions, or the order in which they appear in the exam in relation to other questions. There is no particularly well-defined structure for the placement of the questions. The question numbers referred to in each section refer to the number(s) of the questions on the exam in its completed form.

6.3.1 Basic Syntax Questions (Group 1)

This group is made up of questions that cover only one learning objective from PF1 Fundamental Programming Constructs: basic syntax and semantics of a higher-level language. While this objective is tested in every question that involves code, there is a facet of this topic that involves the vocabulary of programming. For example, understanding syntax assumes not only the creation of a declaration for a variable but knowledge of what a variable is.

Therefore, the questions for this section ask the students to look at a piece of code and identify various programming elements in it. Questions 60 – 78 ask the students to identify 19 different parts of code from a code example provided.

Given the following list of 19 parts of code, you should identify one example of each of the items in the code provided for this section (in the answer sheet) by precisely circling and clearly identifying by number the element in the code segment. Make sure that your circles are clearly identified with numbers that are clearly written. If the markings are not clear, the question will simply be marked incorrect and given no credit. If there is no example of the item in the code, you should write the words "Does not exist" on the line next to the element in the answer sheet.

60) Class name [1 point]
61) Constructor definition [1 point]
62) Assignment statement [1 point]
63) Comment [1 point]
64) Instance variable declaration [1 point]
65) Actual parameter (argument) [1 point]
66) Formal parameter [1 point]
67) Statement that displays information [1 point]
68) Access (Visibility) control modifier [1 point]
69) Accessor method definition [1 point]
70) Mutator method definition [1 point]
71) Creation/instantiation of an object [1 point]
72) Method call/invocation [1 point]
73) Method return type specification [1 point]
74) Superclass name [1 point]
75) Subclass name [1 point]
76) Interface name [1 point]
77) Name of a class that implements an interface [1 point]
78) Method overloading (identify one of the methods that is overloaded) [1 point]

```
/* The classes given below were written for the purposes of
 * this exam. In reality, they would each be in their own
 * separate file, but are reprinted here as one long file
 * for ease of reading. This "print-out" spans two pages,
 * so please look at both pages while answering the
 * following questions.
 */

public class App {
    private Puppy _puppy;
    private ID _id;
    public App (){
      System.out.println("App constructor called.");
      _puppy = new Puppy(new Toy());
      this.setID(new ID(this, _puppy));
      System.out.println("App constructor end.");
    }
    public void setID(ID id) {
      _id = id;
    }
    public static void main (String[] args) {
      App app = new App();
    } // end of main ()
}// App

public interface Colorable {
   java.awt.Color getColor();
   void setColor(java.awt.Color color);
}// Colorable

public class ID implements Colorable{
   private Animal _animal;
   private java.awt.Color _color;
   public ID (App app, Animal animal){
      _animal = animal;
      _color = java.awt.Color.BLACK;
   }
   public java.awt.Color getColor() {
      return _color;
   }
   public void setColor(java.awt.Color color) {
      _color = color;
   }
}// ID

public class Animal {
   private Toy _toy;
   public Animal (){ _toy = new Toy(); }
   public Animal (Toy toy) { _toy = toy; }
   protected Toy getToy() { return _toy; }
   public void somethingShouldHappen() {
      _toy.doSomething();
   }
}// Animal

public class Puppy extends Animal{
   public Puppy() {}
   public Puppy (Toy toy){
      super(toy);
      this.doSomethingWithThisColor(
                              this.getToy().getColor());
   }
   public void doSomethingWithThisColor
                                    (java.awt.Color color){
      this.getToy().setColor(color.darker());
   }
   public void somethingShouldHappen() {
      super.somethingShouldHappen();
      this.getToy().doNothing();
   }
}// Puppy

public class Toy {
   private java.awt.Color _color;
   private String[] _sounds;
   public Toy (){ _color = java.awt.Color.RED; }
   public void setColor(java.awt.Color color) {
      _color = color;
   }
   public java.awt.Color getColor() { return _color; }
   public void doSomething() {
      _sounds = new String[20];
      for (int count = 0; count < _sounds.length; count++){
       _sounds[count] = "Squeak";
      } // end of for ()
      System.out.println(_sounds);
   }
   public void doNothing() {
      //This method really does nothing.
   }
}// Toy
```

To adapt these questions for different languages, not only must the code be changed, but the vocabulary terms should also be reviewed. For example, if an introductory sequence uses a language that does not support method overloading, that question should be removed from the assessment, because students would most likely not have ever used the term, and it would simply cause confusion while taking the exam. It is possible that another programming construct implemented in the language could be substituted for the removed term.

6.3.2 Fundamentals and API Programming (Group 2)

The questions in this group cover the rest of the PF1 knowledge unit, Strings from PF3, and simple numerical algorithms from AL3. The topic of API[17] programming is included in this group because the question that asks students to use an API also involves the use of strings.

6.3.2.1 Functions and parameter passing

Questions 79 – 82 of the exam test the basic concept of functions and parameter passing. They ask students to look at a definition of a class with several methods defined inside it, each taking different parameters. The questions ask the students what values will be returned or output when calling the methods with values for the parameters. Note that even though the question contains a class definition, the focus of this question is the methods and the values returned or output. Rewriting these questions in most languages would simply involve syntactic manipulations and possible removal of the outer class (if the language does not support classes).

Use the class `SimpleParams` and `SimpleParamsApp` defined below to answer questions 79–82.

```
public class SimpleParams () {
   private double _data;
   public SimpleParams() {
      _data = 5.75;
```

[17] API stands for Application Programmers Interface. An entire knowledge unit devoted to this topic was eliminated from the intersection of topics for this exam (see Chapter 4). As noted in Chapter 4, sometimes it becomes necessary to include an API when asking students to write code so as not to test their ability to memorize library functions, but rather to use them effectively to perform some other task. In this case, the students are asked to perform some string processing and are given a set of library functions to assist in that task.

```
    }
    public String method1(String s) {
      return s + “ additional stuff”;
    }
    public void method2(int input) {
      int temp = input + 1;
      System.out.println(“Input was: “ + input
                         + “ and temp is: “ + temp);
    }
    public void method3(double input) {
      _data = input;
    }
    public double getData() {
      return _data;
    }
}//SimpleParams

public class SimpleParamsApp {
      public SimpleParamsApp() {
            SimpleParams sp = new SimpleParams();
            double answer79 = sp.getData();
            String answer80 = sp.method1(“Simple stuff.”);
            sp.method2(6);  //Needed for question 81
            sp.method3(3.8);
            double answer82 = sp.getData();
      }
      public static void main(String[] args) {
            SimpleParamsApp spa = new SimpleParamsApp();
      }
}//SimpleParamsApp
```

79) When the code for `SimpleParamsApp` is executed, what value will answer79 be assigned? [1 point]

80) When the code for `SimpleParamsApp` is executed, what value will answer80 be assigned? [1 point]

81) When the code for `SimpleParamsApp` is executed, and `method2` is called with the value 6, as indicated in the code with a comment, what text will be outputted? [1 point]

82) When the code for `SimpleParamsApp` is executed, what value will answer82 be assigned? [1 point]

6.3.2.2 Arithmetic and logical expressions

Questions 83 – 87 test basic expression evaluation of some, but not all, arithmetic and logical expressions. They give the students a set of numerical and Boolean variables that have been assigned values. Students are then presented with a number of arithmetic and logic expressions and asked to evaluate them. These expressions do not cover all of the arithmetic operators in Java, only the common ones (including modulus) for which there are analogous operations in most languages. Some of the unary operators (like plus), the bitwise operators on integers, the bit-shift operators, and the logical operators that do not support short-circuit Boolean evaluation were not included in these questions.

Use the following variables and their values to evaluate the expressions given in questions 83 - 91. Suppose each expression is executed independently (ie – no later expression depends on a result of a previous expression).

```
int a = 4;                    double d = 4.5;
int b = 6;                    double e = 3.3;
int c = -3;                   double f = 0.5;

boolean g = true;
boolean h = false;
boolean i = true;
```

83) `(a + b) * (c – c)` [1 point]

84) `(d / f) + (a % b)` [1 point]

85) `b < c` [1 point]

86) `d != f` [1 point]

87) `(g && h) || (!i && h)` [1 point]

6.3.2.3 Expressions and Assignment

Questions 88 – 90 use the idea of evaluating expressions, but assign the values of the expressions back to a variable and ask the students for the value that the variable will be assigned in the expression. These three questions differ from the previous questions only in this way. However, the concept of assignment to a variable is just as important as evaluation of expressions, and that idea is tested with these questions.

Now suppose the following lines of code have been executed. The variables `a` and `c` refer back to the previous page.

```
int x = a;
int y = c++;
```

88) What is the value of `x`? [1 point]

89) What is the value of `y`? [1 point]

90) What is the value of `c`? [1 point]

Questions 92 – 93[18] test the ability to analyze the results of the execution of multiple expressions that make up a numerical algorithm. Students are given a method that computes the distance between two points and are asked to evaluate variables and determine what values are returned. This tests their ability to look at a numerical algorithm in code and analyze its results. Even though this function is familiar to most students, the way it is expressed inside the code is not in the form that would be presented in a mathematics text, so a certain amount of analysis is needed to discern what is being computed by the function.

Use the code for the method `exp1` given below to answer questions 92 - 93.

[18] Question 91 is a question in this section that actually falls into Group 3 (§6.3.3).

```
public double exp1 (int x1, int x2, int y1, int y2) {
      int tempX = (x2 – x1) * (x2 – x1);
      int tempY = (y2 – y1) * (y2 – y1);

      return Math.sqrt(tempX + tempY);
}
```

Suppose that the `exp1` method is called in the following way:

```
      exp1(12, 16, 24, 27);
```

92) What is the value that will be computed for `tempX` while the method is running? [1 point]

93) What value is returned from the method call? [1 point]

Questions 94 - 96 test the ability to understand conditional statements. Students are presented with a single method whose body is a multiple-branch conditional statement. The students are given method calls with values for the parameters and asked to give the return value, which tests their ability to understand conditional statements. In these questions, the students are given a nested if-then-else because the code involves ranges of numbers, so a case statement is not appropriate.

Use the code for the class `Conditional` given below to answer questions 94 – 96. For questions 94 – 96, you are presented with a method call. In the space provided, you should give the value that is returned from the method call.

```
public class Conditional {

  public String cond2 (double input) {
     if (input <= 5.0 && input >= 0.0) {
           return “First Branch”;
     }
     else if (input > 5.0 || input <= -2.0) {
           return “Second Branch”;
     }
     else {
           return “Third Branch”;
     }
  }
}
```

94) `cond2(3.5);` [1 point]
95) `cond2(7.345);` [1 point]
96) `cond2(-1.9);` [1 point]

Questions 97 – 100 test the ability to understand looping constructs. The students are presented with a class that has three methods, where each method's body is a loop. The students are given method calls and values for parameters and asked to state what values are returned from the method. One of the methods prints out information in addition to returning a value. Asking for the return value also tests the student's ability to understand that printing information is not the same as returning a value. This could be viewed by some as a "trick" question. It is not designed to be. It is designed to test the ability of the student to understand that printing out information is not the same as returning a value from a method.

Use the code for the class `Looper` given below to answer questions 97 – 100. For questions 97 – 100, you are presented with a method call. In the space provided, you should give the value that is returned from the method call.

```
public class Looper {

   public int loop1(int input) {
      for (int i = 1; i <= 20; i++) {
         input++;
      }
      return input;
   }

   public int loop2() {
      for (int counter=10; counter>0; counter=counter–2) {
         System.out.println("counter = " + counter);
      }
      return 0;
   }

   public int loop3(int input) {
```

```
            while (input < 10) {
               input = input * 2;
            }
            return input;
         }
      }
```

97) `loop1(20);` [1 point]

98) `loop2();` [1 point]

99) `loop3(3);` [1 point]

100) `loop3(32);` [1 point]

Questions 101 – 103 test string processing and reading from a file. Since many string manipulation functions are built into Java, this question also overlaps API programming, because the API for the class to help with the reading of files and the `String` class are given as reference.

For questions 101 – 103, you will be filling in the methods for the class `StringFun` as described in each question. The empty skeleton for this class is given below for reference. You will fill in the areas with the ellipses (…). Please also note the abbreviated API given for both the `java.io.BufferedReader` class as well as the `String` class as these could be of help to you while answering these questions.

```
import java.io.*;

public class StringFun {
   private java.util.ArrayList<String> _strings;
   public StringFun() {
      _strings = new java.util.ArrayList<String>();
    }
    //Loads the strings from the file specified into the
    //ArrayList
    public void loadFile(String filename) {[
      ...
   }
   //Indicates the number of Strings in the ArrayList that
   //are the right size
   public int rightSize() {
      ...
   }
```

```
      //Counts the total number of letter Ps in all the
      //strings in the ArrayList
      public int countPs() {
         ...
      }
   }
```

Abbreviated API for java.io.BufferedReader (from Sun's Java API docs)

Constructor Summary
`BufferedReader``(Reader in)` Create a buffering character-input stream that uses a default-sized input buffer.

Method Summary	
`void`	**`close`**`()` Close the stream.
`int`	**`read`**`()` Read a single character.
`int`	**`read`**`(char[] cbuf, int off, int len)` Read characters into a portion of an array.
`String`	**`readLine`**`()` Read a line of text.

Abbreviated API for java.lang.String (from Sun's Java API docs)

Method Summary	
`char`	**`charAt`**`(int index)` Returns the `char` value at the specified index.
`int`	**`compareTo`**`(String anotherString)` Compares two strings lexicographically.
`int`	**`compareToIgnoreCase`**`(String str)` Compares two strings lexicographically, ignoring case differences.
`boolean`	**`endsWith`**`(String suffix)` Tests if this string ends with the specified suffix.
`boolean`	**`equals`**`(Object anObject)` Compares this string to the specified object.
`boolean`	**`equalsIgnoreCase`**`(String anotherString)` Compares this `String` to another `String`, ignoring case considerations.
`int`	**`length`**`()` Returns the length of this string.

String	**replace**(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
boolean	**startsWith**(String prefix) Tests if this string starts with the specified prefix.
String	**substring**(int beginIndex) Returns a new string that is a substring of this string.
String	**substring**(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
String	**toLowerCase**() Converts all of the characters in this `String` to lower case using the rules of the default locale.
String	**toUpperCase**() Converts all of the characters in this `String` to upper case using the rules of the default locale.
String	**toUpperCase**(Locale locale) Converts all of the characters in this `String` to upper case using the rules of the given `Locale`.
String	**trim**() Returns a copy of the string, with leading and trailing whitespace omitted.

101) In your answer booklet, you will finish writing the code for the method `loadFile`. Note that some of the code is already written for you. The file is already loaded into the `BufferedReader`. Your task is to read each line of the file and input each one into the `ArrayList`. Please note that we are also assuming that some other object will handle the exceptions that might be thrown. [8 points]

```
public void loadFile(String filename) throws
                   FileNotFoundException, IOException{

 BufferedReader in = new BufferedReader(new
                                FileReader(filename));

 //Your code begins here.
 //Write your code in the answer booklet.
}
```

102) Write the code for the method `rightSize` so that it returns the number of strings in `_strings` whose length is between 3 and 10 characters inclusive. [8 points]

```
public int rightSize() {
//Write the code for this method in your answer
//booklet
```

```
        }
```

103) Write the code for the method `countPs` so that it returns the total number of occurences of the letter P in all of the strings in `_strings`. Your method should count both lower case (p) and upper case (P) letters. [8 points]

```
        public int countPs() {
        //Write the code for this method in your answer
        //booklet
        }
```

6.3.3 Types, Declaration Models, and Parameter Passing (Group 3)

The questions in this group cover the topics of types, scoping, lifetime and parameter passing mechanisms, from PF3, PL4 and PL5. These questions sometimes stand alone and sometimes are inter-mixed inside of other groupings of questions where the opportunity presented itself to test topics from these knowledge units.

Question 4 of the exam tests student understanding of the parameterized type (generics) mechanism. The question presents students with the creation of two data structures, one using Java generics (parameterized types), the other without generics. Students are asked to recognize that when an element is removed from a structure that does not use generics, the type of the object returned is not the type of the object that was inserted into the collection.

For question 4, consider the following code segment:

```
java.util.HashMap<Integer, String> mapOne =
                  new java.util.HashMap<Integer, String>();

java.util.HashMap mapTwo = new java.util.HashMap();

mapOne.put(1, “First name”);
mapTwo.put(1, “First name”);

String s1 = mapOne.get(1);
String s2 = mapTwo.get(1);
```

4) Which of the two assignments of “First name” to a String variable does not work correctly and why? (Circle only one answer). [1 point]

a. Assignment to `s1` does not work because `get()` returns an `Object`, not a `String`.
b. Assignment to `s1` does not work because `s1` is not a `String`.
c. Assignment to `s1` does not work because `HashMap`s cannot use `Integers` as keys.
d. Assignment to `s2` does not work because `get()` returns an `Object`, not a `String`.
e. Assignment to `s2` does not work because `s2` is not a `String`.
f. Assignment to `s2` does not work because `HashMaps` cannot use `Integers` as keys.
g. Neither assignment works because `get()` returns an `Object`, not a `String`.
h. Neither assignment works because neither `s1` nor `s2` is a `String`.
i. Neither assignment works because `HashMaps` cannot use `Integers` as keys.

Questions 47 and 48 of the exam are true-false questions that probe the understanding of the difference between primitive types and object types. While the language might be skewed towards Java terminology, the idea of the difference between a built-in type and user-defined type is probed, as well as the difference between a primitive type and a reference type.

47) When we declare a variable whose type is a primitive data type, we are actually creating a reference to a space of allocated memory. [1 point]
a. TRUE
b. FALSE

48) Primitive types are not objects and therefore do not have methods defined on them. [1 point]
a. TRUE
b. FALSE

Question 91 tests to see if students recognize type mismatch in an expression. For example, in Java (and many other languages), integer numbers and whole numbers are not considered the same type. Often, one cannot assign the result of arithmetic with floating point numbers to an integer.

> Use the following variables and their values to evaluate the expressions given in questions 83 - 91. Suppose each expression is executed independently (ie – no later expression depends on a result of a previous expression).

```
int a = 4;                    double d = 4.5;
int b = 6;                    double e = 3.3;
int c = -3;                   double f = 0.5;
```

> 91) The following line of code does not compile (`e` & `b` refer back to the previous page). What do you need to do to get the line of code to work? [4 points]

```
    int z = e * b;
```

> (Circle all answers from the choices below that would make the code compile.)
> a. You need to cast `b` to be a double.
> b. You need to cast `b` to be an integer.
> c. You need to cast `e` to be an integer.
> d. You need to cast `e` to be a double.
> e. You need to cast the result of `e * b` to be an integer.
> f. You need to cast the result of `e * b` to be a double.
> g. You need to make `z` a double.
> h. You need to make `z` an object.

Questions 104 – 112 have the students look at a group of two classes, an interface, and a class with a main method in it. One of the classes implements the interface and the other does not. Each class has several methods in it. The class that implements the interface has an instance variable of type java.awt.Color (a reference type). The other class has two instance variables, one of the type that implements the interface (a reference type) and one of type int (a primitive type). Various methods are defined that take either a reference type or both as parameters in this class. The Driver class creates some instances and calls some methods on them.

Use the following code segment for the classes named `Types`, `Thing`, and `Driver`, the interface named `Colorable`, and your knowledge of Java to answer the questions 104 – 112. If the question has multiple choices, you should circle the letter of the best answer for each question, unless instructed otherwise.

```
public interface Colorable {
      public void setColor (java.awt.Color color);
      public java.awt.Color getColor();
}//Colorable

public class Thing implements Colorable{
      private java.awt.Color _color;
      public Thing() {
            _color = java.awt.Color.WHITE;
      }
      public void setColor (java.awt.Color color) {
            _color = color;
      }
      public java.awt.Color getColor() {
            return _color;
      }
}//Thing

public class Types {
      private Thing _thing;
      private int _number;

      public Types() {
            _thing = new Thing();
            _number = 0;
      }
      public void incrementNumber (int increment) {
            _number += increment;
    }
}//Types

public class Driver {
      public Driver() {
            int i = 5;
            Colorable t = new Thing();
            //Line for question 109 inserted here
            this.changeParams(i, t);
      }
      public void changeParams (int input,Colorable thing){
            input = input * 2;
            thing.setColor(java.awt.Color.RED);
      }
      public static void main (String[] args) {
            Driver d = new Driver();
      }
}//Driver
```

Questions 104 and 105 test the students understanding of a null reference and then what happens after the reference is initialized to a non-null value.

104) What is the value of `_thing` **before** the constructor is run for the class `Types`? [1 point]

a. A null reference.
b. A random value assigned value assigned by the compiler.
c. An object of type `Thing` whose instance variables are set to null.
d. `_thing` does not exist before the constructor is run.

105) What is the value of `_number` **after** the constructor is run for the class `Types`? [1 point]

a. null
b. 0
c. -1
d. undefined

Question 106 asks the student to identify in the code the difference between variables of reference type and primitive type.

106) Which of the variables presented in this code segment are object references? Circle the letters of all that apply. [5 points]

a. `_color`
b. `_thing`
c. `_number`
d. `increment`
e. `i`
f. `t`
g. `input`
h. `thing`
i. `d`
j. None of these variables are references.
k. All of these variables are references.

Questions 107 and 108 test the student's knowledge of visibility and scope of methods and variables inside the code segment.

107) Which of the members (variables or methods) from the class `Types` are accessible from outside the class? Circle the letters of all that apply. [6 points]

a. `_thing`
b. `_number`
c. `Types()` constructor
d. `incrementNumber(int increment)` method
e. None of the members are accessible outside of the class.
f. All of the members are accessible outside of the class.

108) Which of the members from the class `Driver` are not local and only accessible from inside the class? Circle the letters of all that apply. [6 points]

a. `i`
b. t
c. `Driver()` constructor
d. `changeParams(int input, Colorable thing)` method
e. `main(String[] args)` method
f. None of the members are only accessible from inside the class.

Question 109 presents a scenario in which one variable is to be assigned the value of another variable, but the two variables are not of compatible types. The question requires the student to notice that the variables are of different types and know that this type of assignment would therefore not be allowed.

109) Suppose we add the following line to the constructor in the space indicated by the comments in `Driver`: [1 point]

```
t = i;
```

Is this valid? What would happen?

a. It is perfectly valid. The code would run.
b. It is valid. The type of i is a primitive and t is an object type and you can always assign a primitive type to any object type because primitives are subclasses of objects.
c. This is not valid. The code would compile, but would cause a run-time error.
d. This is not valid. The code would not compile because t and i are not of compatible types.

Question 110 relies on this code segment, but actually tests knowledge of inheritance, which will be discussed in §6.3.7 of this chapter.

110) Under what circumstances would you be allowed to add the following line of code to the end of the class `Driver`'s constructor: [1 point]

```
t = new OtherThing();
```

a. No special circumstances, this line of code would always work.
b. Only when `OtherThing` is a subclass of `Thing`.
c. Only when `OtherThing` is a superclass of `Thing`.
d. This line of code would never work because the declared type of `t` is `Thing`, so you must assign a `Thing` object to `t`.

Question 111 tests knowledge of what happens to the value of a variable of primitive type that is passed into a method and then changed in the method. In Java, all parameters are passed by value, so no change is caused by the change of the value within the method.

111) Looking at the code for `Driver`, what is the value of `i` after the method `changeParams` has been called? [1 point]
a. The value is unchanged, 5.
b. The value is 2 times the value, 10.
c. The value is 0 because `i` was never initialized.
d. The value will be null because you can not change the value of `i` from within a method.

Question 112 asks this same question, but with a variable whose type is a reference type. The same action would cause a change in the value of the variable because it is a reference type.

112) Again looking at the code for `Driver`, after the method `changeParams` has been called in the constructor, suppose we add the following line of code:

```
      java.awt.Color color = t.getColor();
```

What would be the value of color? [1 point]
a. java.awt.Color.WHITE
b. java.awt.Color.RED
c. java.awt.Color.PINK
d. null

Questions 113 – 115 also test the student's knowledge of reference types. For these questions, another code example is given. These questions can be characterized as the typical “pointers” question common when teaching C, C++, or any other language with pointers. Two variables of a particular type are created and originally point to different values. The student is asked about their values when they both point to the same value. Then the value is changed using one of the pointers, and the student is asked what value the other points to. Last, the pointers are redirected to point to different values, and the student is queried again about the values they point to. This series of questions is implemented in Java using references, and therefore classes, but could be easily modified for pointers (if the language supports them).

For questions 113 - 115, use the following code to help you answer the questions.

```
public class Ball {
      private java.awt.Color _color;
      public Ball() {
            _color = java.awt.Color.GREEN;
      }
      public java.awt.Color getColor() {
            return _color;
      }
      public void setColor (java.awt.Color color) {
            _color = color;
      }
}

public class Driver {
      public Driver() {
            Ball ball = new Ball();
            ball.setColor(java.awt.Color.RED);

            Ball ball2 = new Ball();
            ball2 = ball;   //Question 113 refers up to
                            //this point

            ball.setColor(java.awt.Color.BLUE);
                  //Question 114 code
            java.awt.Color question114 = ball2.getColor();
                 //Question114 code

            ball2 = new Ball();
                //Question 115 code
            ball2.setColor(java.awt.Color.BLACK);
```

```
                //Question 115 code
            java.awt.Color question115 = ball.getColor();
               //Question115 code
      }
      public static void main(String[] args) {
            Driver d = new Driver();
      }
}
```

113) After the line of code in `Driver` that reads

```
            ball2 = ball;
```

is executed, which reference refers to a green ball? [1 point]

a. `ball`
b. `ball2`
c. both `ball` and `ball2`
d. neither `ball` or `ball2`

114) Focus your attention on the lines of code that is the code for Question 114 as indicated by comments. What will the value of the variable `question114` be? [1 point]

a. `java.awt.Color.RED`
b. `java.awt.Color.GREEN`
c. `java.awt.Color.BLUE`
d. no color – it will be an error

115) Focus your attention on the lines of code that is the code for Question 115 as indicated by comments. What will the value of the variable `question115` be? [1 point]

a. `java.awt.Color.RED`
b. `java.awt.Color.GREEN`
c. `java.awt.Color.BLUE`
d. `java.awt.Color.BLACK`

6.3.4 Data Structures (Group 4)

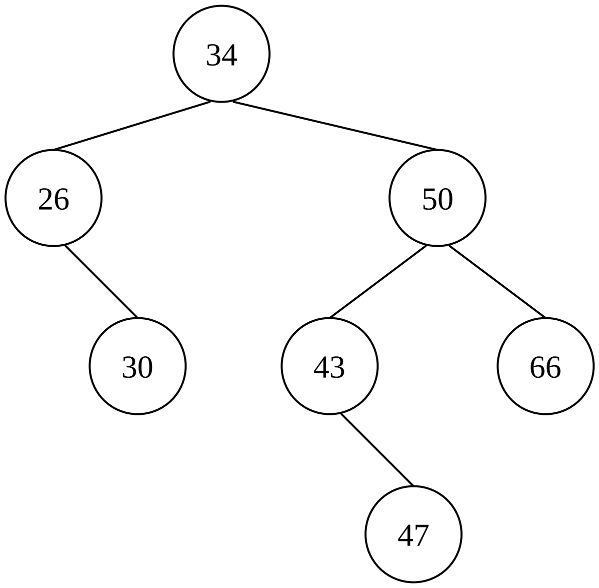
This group consists of questions dealing with the various types of data structures assessed by this instrument. It finishes up the topics in PF3: strings, linked structures, stacks, queues, hash maps, graphs, trees, and strategies for choosing the right data structure. It also incorporates some questions on asymptotic analysis and Big O notation with questions concerning data structures. The topics of hash tables (more accurately, hashing) and binary search trees from AL3 are part of this group. Questions about the topic collection classes and iteration protocols from PL6 are also included in this section.

Question 1 asks the students to construct a binary search tree given the elements to be inserted. Question 2 asks the students to trace through the search algorithm for binary search trees. Question 3 asks the students to construct a valid binary search tree once the root of the given binary search tree has been removed.

1) Draw the binary search tree which results when the following items are inserted, in the order given into an initially empty BST. [8 points]

Elements: 62, 55, 37, 106, 202

Given the following BST, answer questions 2 – 3.



2) You call search (find) and are looking for the number 32. List of nodes that are visited while you are determining that 32 is not in the BST. [3 points]

3) You want to delete 34 (the root) from this tree. Show one possible valid binary search trees that could result from deleting the root. [8 points]

Question 5 asks students to write code to iterate over a collection of objects using an iterator and call a method on each object in a collection.

> 5) Write the body of the following method named `changeColors`. The method takes as a parameter, a `java.util.Collection` of `java.awt.Colors`. The `changeColors` method should call the method `setColor(java.awt.Color)`, which is inherited from `javax.swing.JPanel`, for each color in the `Collection` so that the user sees a changing background color for the panel on their program. You can assume that this method appears in a class that extends `JPanel` so you can simply call the `setColor` method from within this method. You must use an iterator/for-each loop in your solution to this question to receive full credit. [8 points]

```
void changeColors(java.util.Collection<java.awt.Color>
                                    colorsForBackground) {
}
```

Questions 6 – 8 test the students' knowledge of how indexing of arrays works (i.e., in Java, that arrays are indexed beginning at 0 and ending at size – 1). Question 9 asks the students to write code to re-size an array retaining the original elements in the array but giving space to add new elements into the array. Question 10 asks students to write code that creates an array whose elements correspond to the square of the index that the element is stored at. Question 11 asks the students to write a find method on a two-dimensional array.

> Assume you have created the following array in a program:
```
            int[] holder = new int[50];
```

> Use this information to answer questions 6 – 9.
>
> 6) What is the maximum number of elements that can be stored by `holder`? [1 point]
>
> 7) At which index would the first integer in `holder` be stored? [1 point]
>
> 8) At which index would the last integer in `holder` be stored? [1 point]

9) As you are using the array in your program, you find out that you need to store more than the maximum number of elements you listed in question 8. You do not know how many more elements you will be storing, just that you need more space in your array. You are asked to write a method, `needMoreSpace` that takes in an array and performs the necessary operations to return a larger array with the same elements as the original, but with space to store additional elements. Since you don't know how many elements you will eventually need to store, you should write the method body so that it could be called at a later time if the array needs to get bigger again. [8 points]

```
public int[] needMoreSpace(int [] originalArray) {

}
```

10) Fill in the method below so that it creates and returns an array of size `size` and populates the array with elements each of whose values is the square of the index at which the element is stored. For example, at array index 3, the value 9 should be stored. [8 points]

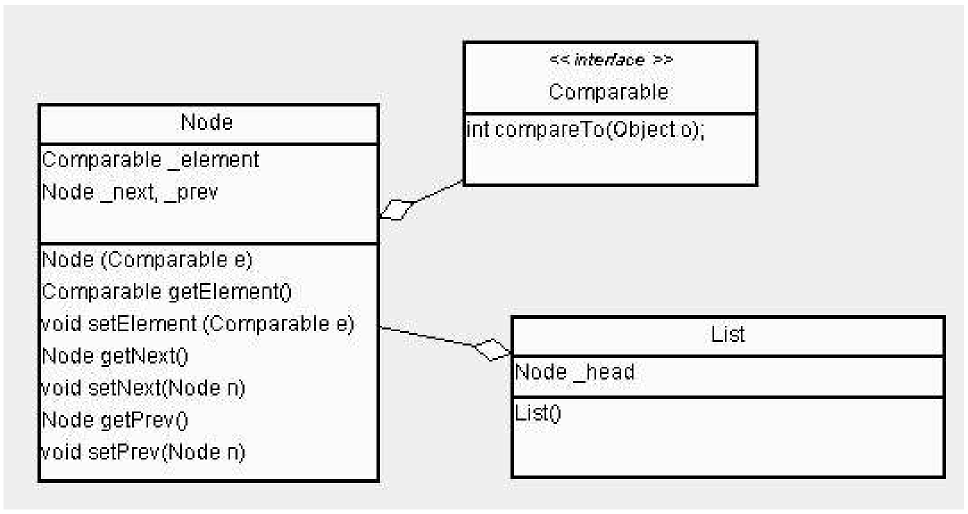
```
public int[] arrayOfSquares (int size) {

}
```

11) Fill in the method below so that it returns `true` if the value passed in as a parameter is contained inside the matrix and returns false otherwise. [8 points]

```
public boolean contains(double[][] matrix, double value) {

}
```

Question 12 gives the design (in UML) for a doubly-linked list and asks the student to write the code for the delete method of the list. The UML given in this question is not intended to be the testable material. Rather, this question is given in terms of the design of the code, not necessarily the implementation. This is not a requirement of the question, and another expression of design can be substituted for UML, or even the partially implemented code.

12) Given the following UML diagram for a doubly linked list, fill in the method `delete` below, which is a method in the List class and takes an element to be deleted and returns the deleted element when finished. [8 points]



Notes about the classes in the diagram:

- `Node`'s constructor sets the value of `_element` to the value passed in and sets the value of `_next` and `_prev` to `null`. The other elements are simple accessors and mutators for `_element,` `_node,` and `_prev`.

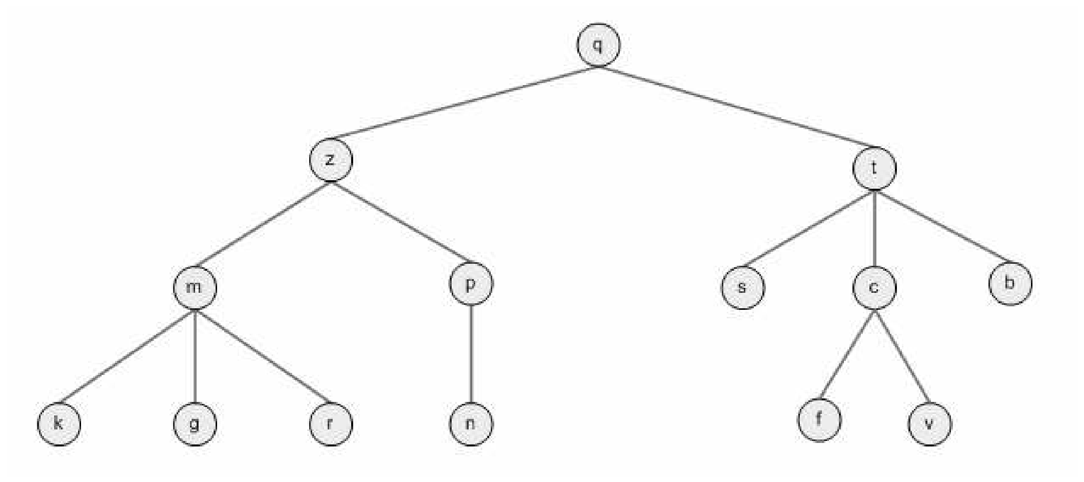
- `Node` holds an element that implements the interface `Comparable`. Recall that a class that implements this interface has a method named `compareTo` that takes in an `Object obj`, and returns a positive number if `this > obj`, the value 0 (zero) if the two are the same, or a negative value if `this < obj`.

- `List`'s constructor simply sets the value of `_head` to `null`.

```
public Comparable delete(Comparable element) {

}
```

Questions 13 – 18 test students on their knowledge of basic tree vocabulary: root, leaf, parent, child, and height of a tree.

Use the following representation of a tree data structure to answer questions 13 - 18.



13) What is the value stored in the node that is the root? [1 point]

14) Give the value stored in one of the leaves of this structure. [1 point]

15) What is the height of a tree that just contains a root and no other nodes? [1 point]

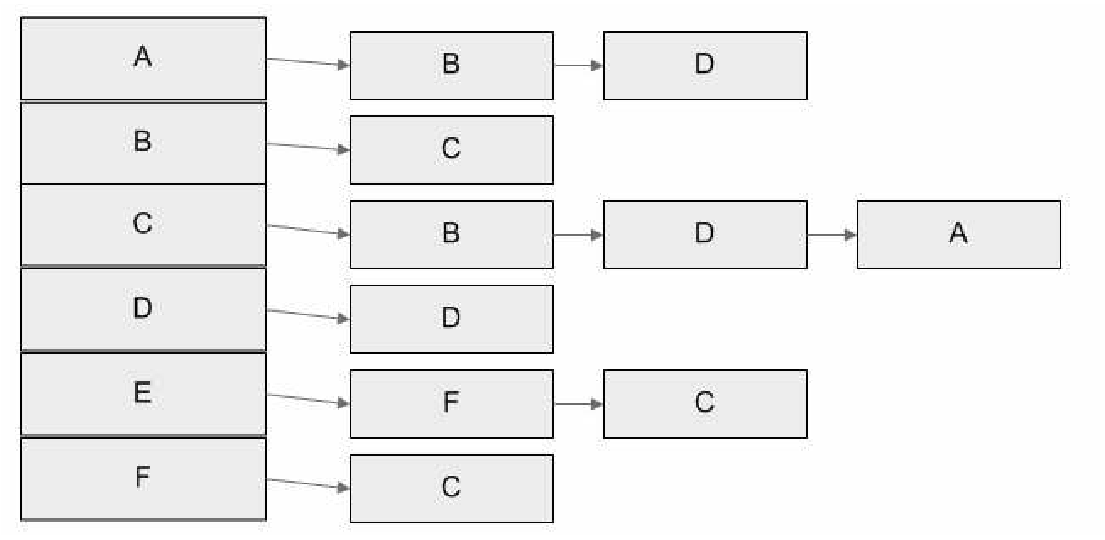
16) What is the height of this structure? [1 point]

17) Give the value stored in the node that is the parent of n. [1 point]

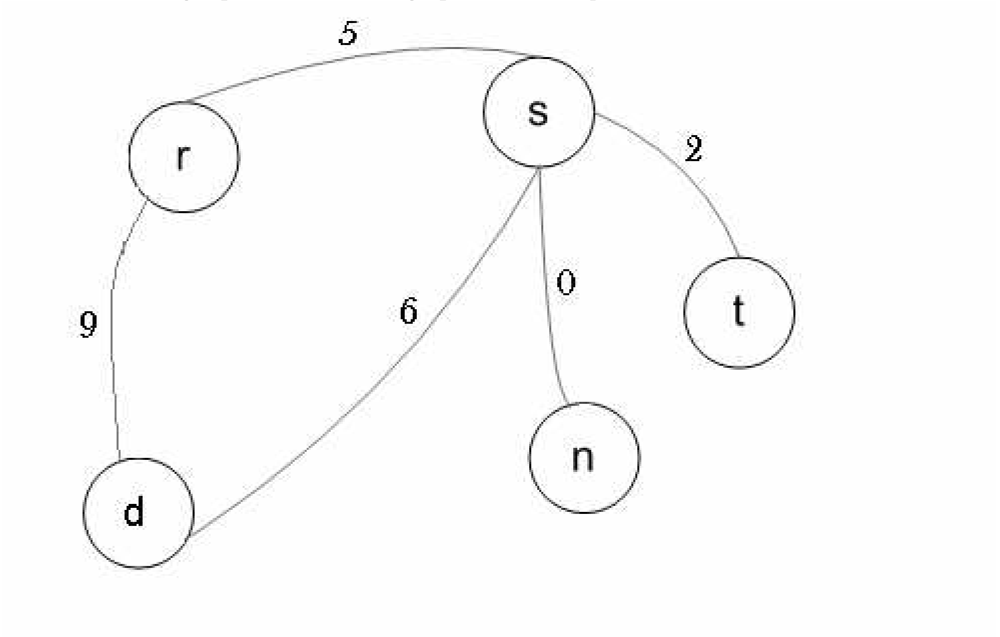
18) Give the values stored in all the children of m. [3 points]

Question 19 gives students an adjacency list for a graph and asks them to draw the graph that the adjacency list represents. Questions 20 and 21 test basic graph vocabulary including: directed, undirected, weighted, unweighted, simple, complete, acyclic, isomorphic, and adjacent nodes.

19) Given the following adjacency list for a directed graph, draw the graph structure it represents. [8 points]



Use the following representation of a graph to answer questions 20 and 21.



20) Circle the letters of all of the words that accurately describe the graph above. [4.5 points]
a. directed
b. undirected
c. weighted
d. unweighted
e. simple
f. complete
g. acyclic
h. isomorphic

i. rooted

21) Circle the letters corresponding to all the pairs of nodes given that are adjacent in the above graph. [4 points]
a. r and s
b. t and n
c. d and s
d. n and d

Question 22 asks the students to assess which type of implementation (array-based or link-based) would be a more efficient implementation for a linked structure when using linear search.

22) If a data structure is linear in nature (list, vector, etc), which implementation would perform better asymptotically in a linear search. Circle one of the implementations listed: [1 point]
a. array-based
b. linked list-based
c. neither – they would both perform the same on the linear search.

Question 23 combines knowledge of both inheritance and data structures. Students are asked why it would be inappropriate for a stack to be a subclass of vector. The data structures portion of this question is that students must know that a stack is a limited-access structure and that invariant (property) should be preserved when implementing a stack. Students must also then be able to identify why using inheritance has the potential for breaking this invariant.

23) Referring to your knowledge of data structures and inheritance, why is it inappropriate for a java.util.Stack to be a subclass of java.util.Vector? [8 points]

Questions 24 – 31 present the student with one of: a definition of a data structure, a fact about a data structure, or a scenario for using data structures, and asks the students to select, from a list of data structures, which structure or structures would be the most appropriate answers for each question.

From the list of data structures given, choose the best answer or answers for questions 24 – 31. If there is no appropriate answer, write “None”. If you feel that more than one answer is appropriate, list all appropriate answers. It is possible that some answers from the box will not be used.

Linked List	Array
Graph	Stack
Tree	Queue
Hash Map	

24) Structure that associates a key with a value. [6 points]

25) Structure whose insertion/removal strategy can be defined as LIFO. [6 points]

26) Structure whose insertion/removal strategy can be defined as FIFO. [6 points]

27) Structure that is non-linear. [6 points]

28) Structure whose elements are always stored in a contiguous block of memory. [6 points]

29) You are creating software for a call center that does technical support. Technicians are supposed to answer calls in the order they are received. What structure would be best for keeping track of which call should be answered next? [6 points]

30) Your company has decided to create a program to help cell-phone customers everywhere. It is an on-line program that allows the user to type a person’s name and will return a list of all cell phone numbers registered to them. You are asked to recommend a structure to hold onto the information. Which structure would you recommend? [6 points]

31) You are working for a brand new on-line mapping company. This company needs to maintain information about locations and roads that connect them so that it can tell customers about various routes between locations. What type of structure would be best for them to use to store their information? [6 points]

Question 40 asks students to identify the running time of a hashing function. Question 41 and 42 presents students with code for two different types of insertion into a linked list and asks them to identify the running time of each insertion.

40) If your hashing function worked every time with no collisions, what would be the running time of a method to find an element in a hash table of size n? [1 point]

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

Use the code for a node and linked list given below to answer questions 41 and 42. Please note that some methods from both classes may have been removed if they do not pertain to the questions.

```
public class Node<E> {
   private E data;
   private Node<E> next;

   public Node<E> (E element, Node nextNode) {
     data = element;
     next = nextNode;
   }
   public void setNext(Node nextNode) { next = nextNode; }
}

public class LinkedList<E> {
   private Node<E> head = null;
   private Node<E> tail = null;

   public LinkedList() {}
   public void insert (E element) {
     Node<E> newNode = new Node(element, null);
     tail.setNext(newNode);
     tail = newNode;
   }
   public void insertAtFront(E element) {
     Node<E> newHead = new Node(element, head);
     head = newHead;
   }
}
```

41) What is the big-oh running time of the LinkedList's method insert in the worst case? [1 point]

a. O(1)
b. O(log n)
c. O(n)
d. O(n^2)

42) What is the big-oh running time of the LinkedList's method insertAtFront in the worst case? [1 point]

a. O(1)
b. O(log n)
c. O(n)
d. O(n^2)

Question 50 tests to see if students understand that an array is simply a container and can hold any type of data (not just primitive typed data). It is a true-false question.

> 50) We can create an array to hold elements of primitive types (int, char, double, etc), but to hold elements of object type, we must use another type of data structure. [1 point]
>
> a. TRUE
> b. FALSE

6.3.5 Recursion (Group 5)

This group of questions assesses the entirety of the topics in PF4 Recursion.

Question 39 asks students to identify which of the six algorithms listed use a divide-and-conquer strategy in implementations.

> 39) Circle any and all of the following algorithms that use a divide and conquer strategy to perform their specific task. If none of the listed algorithms use a divide and conquer strategy, circle choice F. [6 points]
>
> a. Linear Search
> b. Quicksort
> c. Mergesort
> d. Insertion Sort
> e. Selection Sort
> f. None of the above.

Question 51 asks students to identify, out of a series of examples of processes, which are recursive. Questions 52 – 56 present two methods to the students, one recursive and one not. The students are then asked to evaluate the two methods on various inputs. On some inputs the methods behave the same, but on others they do not. Question 56 asks students to select under which conditions the methods execute differently. These questions involve the students tracing through a recursive procedure and understanding how it functions.

51) Parts a – d describe four procedures in code and through words. Circle the letter of each procedure that can be categorized as recursive. [4 points]

a.
```
public int partA(Object[] items,Comparable x,int y,int z){
      if ( y > z) {
            return -1;
      }
      else {
            int a = ( y + z )/2;
            int b = x.compareTo(items[a]);
            if (b == 0) {
                  return a;
            }
            else if (b < 0) {
                  return partA(items, x, y, a – 1);
            }
            else {
                  return partA(items, x, a + 1, z);
            }
      }
}
```

b.
```
public int partB(int x) {
      int r = x;
      r = r / 30;
      Math.power(x, 2);
      return x;
}
```

c.
```
public int partC (int x) {
      int y = 0;
      for (int i = 0; i < x; i++) {
            y = y + i
      }
      return y;
}
```

```
d.
Procedure for Writing Down Names of People Waiting in line
for Movie Tickets:

 1) If line is empty go back to office.
 2) If line is not empty:
     a.     Walk up to first person in line and ask for
     their name.
     b.     Write name on official sheet and give
     participant free popcorn coupon.
     c.     Move person to “fast pass” line for tickets.
     d.     Begin Procedure for Writing Down Names of
     People Waiting in line for Movie Tickets again.
```

Use the following code segment to answer questions 52 – 56. Some of the questions ask about the output of a method on a particular input. If the method goes into an infinite loop or infinite recursion on an input, write “infinite loop” as your answer.

```
public int method1 (int x, int y) {
      if (y == 0) {
            return 1;
      }
      else {
            return x * method1(x, y – 1);
      }
}

public int method2 (int x, int y) {
      int result = 1;
      for (int i = 0; i < y; i++) {
            result = result * x;
      }
      return result;
}
```

52) What is the value returned from the following method call: [1 point]

```
method1(2,1);
```

53) What is the value returned from the following method call: [1 point]

```
method2(2,2);
```

54) What is the value returned from the following method call: [1 point]

```
method2(2,-5);
```

55) What is the value returned from the following method call: [1 point]

```
method1(2,-3);
```

56) These methods function differently on different inputs. On which class of inputs do these methods behave differently (circle all that apply)? [5 points]

a. When both x and y are positive numbers.
b. When both x and y are the number 0 (zero).
c. When both x and y are negative numbers.
d. When x is positive and y is negative.
e. When x is negative and y is positive.
f. When x is zero and y is positive.
g. When x is zero and y is negative.
h. When x is positive and y is zero.
i. When x is negative and y is zero.
j. The methods never function differently.

Questions 57 – 58 present the mathematical definition of a recursive sequence (the Lucas sequence). Students are not expected to have had previous experience with the Lucas sequence. The recursive definition of the sequence is given to the students to aid them in answering this question. Students are asked to identify from the definition which are base cases and which are recursive cases. Question 59 then asks the students to create a recursive method that gives the nth element of the Lucas sequence.

Given the following definition of the Lucas sequence, answer questions 57 – 59.

L(1) = 1;
L(2) = 3;
L(n) = L(n – 1) + L(n – 2) for n > 2

57) State what the base case(s) is/are for the Lucas sequence. [4 points]

58) State what the recursive case is for the Lucas sequence. [4 points]

59) Write the Java code for a recursive method that takes as a parameter an integer n and returns the nth element of the Lucas sequence. You can assume that n will always be a number greater than zero. [8 points]

6.3.6 Searching and Sorting Algorithms, and Algorithm Analysis (Group 6)

This group covers all of the topics in AL1 Basic Algorithmic Analysis as well as the rest of the topics in AL3 concerning searching and sorting algorithms. Any question that concerns Big O[19] notation is included in this group as well as questions that ask about searching and sorting algorithms.

Questions 32 – 37 give a list of common searching and sorting algorithms and ask students to identify all valid Big O bounds on the worst case running time of each of the algorithms. Question 38 asks the students to identify which of the sorting and searching algorithms function correctly only on sorted inputs (the only one is binary search).

In questions 32–37 you are given an algorithm for sorting or searching. You are to circle any and all valid big-oh bounds on the worst-case performance of each of the algorithms listed.

32) Binary Search [6 points]
a. O(1) b. O(log n) c. O(n) d. O(n log n) e. O(n^2) f. O(2^n)

33) Linear Search [6 points]
a. O(1) b. O(log n) c. O(n) d. O(n log n) e. O(n^2) f. O(2^n)

34) Selection Sort [6 points]
a. O(1) b. O(log n) c. O(n) d. O(n log n) e. O(n^2) f. O(2^n)

35) Insertion Sort [6 points]
a. O(1) b. O(log n) c. O(n) d. O(n log n) e. O(n^2) f. O(2^n)

36) Quicksort [6 points]
a. O(1) b. O(log n) c. O(n) d. O(n log n) e. O(n^2) f. O(2^n)

37) Mergesort [6 points]

[19] In the text of this dissertation, I use Big O to describe the asymptotic running time. In the exam, it is referred to as big-oh running time. They should be taken to mean the same thing. Since the exam has already been written and administered, it is left in that form for the dissertation, even though it is inconsistent with the dissertation wording.

a. O(1) b. O(log n) c. O(n) d. O(n log n) e. $O(n^2)$ f. $O(2^n)$

38) Circle any and all of the following algorithms that only function correctly on sorted inputs. If none of the algorithms require sorted inputs to function correctly, circle choice F. [6 points]

a. Binary Search
b. Linear Search
c. Selection Sort
d. Quicksort
e. Mergesort
f. None of the above.

Questions 40 – 42 ask about the running time of methods on data structures, which fit both into the section on data structures §6.3.4 as well as this section because they deal with Big O notation. See §6.3.4 for these questions.

Questions 43 and 44 are true-false questions about Big O. Students are given a mathematical function and its Big O bound and asked to indicate whether the statement is true or false (i.e., if the Big O bound is correct). If the statement is false, the students are asked to correct the Big O of the statement so that it would be correct. Question 46 is another true-false question that requires that the student know that upper bounds on the growth of a function are transitive.

For questions 43 - 44, decide whether the statement is true or false and circle the appropriate word true or false. If the statement is false, rewrite the big-oh notation so that it would be true in the space provided.

43) $n^3 + 2n + 25 = O(n)$ [3 points]
true false
Rewritten statement (if false):

44) $n^2 + 30n + 4362 = O(n^2)$ [3 points]
true false
Rewritten statement (if false):

46) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$. [1 point]
a. TRUE

b. FALSE

Question 45 asks the students to arrange a set of functions in order from slowest growing to fastest growing. The functions represent the basic levels of complexity of algorithms: 1, log n, n, n^2, 2^n, n!, n^n.

> 45) Arrange the following functions in order from slowest growing to fastest growing. [7 points]
> n, $n!$, n^2, log n, 1, 2^n, n^n

6.3.7 Object-Oriented Programming (Group 7)

The topics in this group are those covered from PL6. There are no specific questions about encapsulation or information hiding. However, every class that uses private instance variables and public methods, especially those in the questions about data structures, illustrates this concept. The students are not directly tested on their ability to re-create encapsulation, but they need to understand the concept in order to answer any question where the code uses it.

Question 23 straddles two groups. Its first group was data structures, and the other is object-oriented programming. Students must be aware of the implications of inheritance to correctly answer the question about why Stack should not inherit from Vector.

> 23) Referring to your knowledge of data structures and inheritance, why is it inappropriate for a java.util.Stack to be a subclass of java.util.Vector? [8 points]

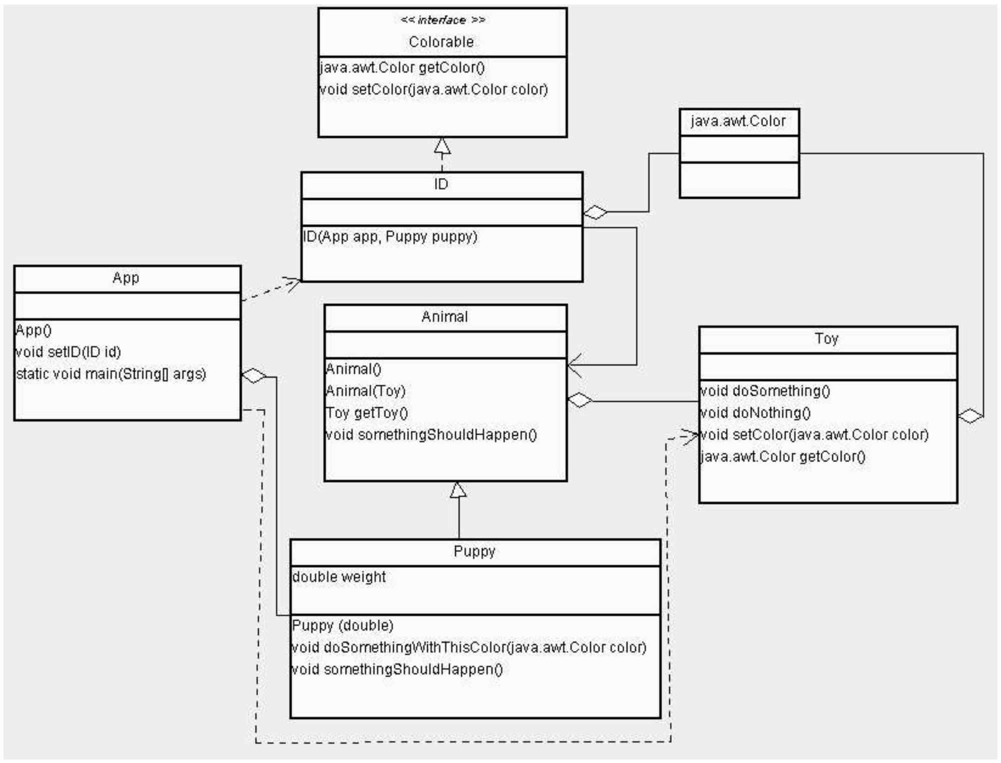
Question 49 is a true-false question testing student’s knowledge of inheritance and polymorphism. Question 110, which is sandwiched inside a set of questions that are from

Group 3, tests students knowledge of the difference between the declared type and the actual type of a variable, which is a component of understanding how inheritance and polymorphism work together. This question fits in with the code example used in that part of the exam, so was included in that section, but really tests ideas from this group.

> 49) Suppose Triangle, Circle, and Square are all subclasses of Shape. In our program, we create an array that stores objects of type Triangle. That array can hold any number of Circles, Squares, and Triangles because they are all subclasses of Shape. [1 point]
>
> a. TRUE
> b. FALSE
>
> 110) Under what circumstances would you be allowed to add the following line of code to the end of the class `Driver`'s constructor: [1 point]
>
> `t = new OtherThing();`
>
> a. No special circumstances, this line of code would always work.
> b. Only when `OtherThing` is a subclass of `Thing`.
> c. Only when `OtherThing` is a superclass of `Thing`.
> d. This line of code would never work because the declared type of `t` is `Thing`, so you must assign a `Thing` object to `t`.

Questions 116 – 127 are based on a system of classes and interfaces that is first illustrated using a UML diagram and then given in code. Students need to analyze these classes to answer the questions. Questions 116 – 125 give the students a set of variable declarations. Some of the variables have a different declared type and actual type, once again the setup for polymorphism. These questions ask students to identify which methods are allowed to be called on a variable and which methods will be executed if the call is legal. Question 126 asks students to identify methods that are inherited within the class structure, and Question 127 asks students to identify if a method is partially overridden or totally overridden in the code example.

Use the UML diagram given below as well as the code segment given after the diagram to answer questions 116 – 127.



<< interface >>
Colorable
java.awt.Color getColor()
void setColor(java.awt.Color color)
java.awt.Color
ID
ID(App app, Puppy puppy)
App
App()
void setID(ID id)
static void main(String[] args)
Animal
Animal()
Animal(Toy)
Toy getToy()
void somethingShouldHappen()
Toy
void doSomething()
void doNothing()
void setColor(java.awt.Color color)
java.awt.Color getColor()
Puppy
double weight
Puppy (double)
void doSomethingWithThisColor(java.awt.Color color)
void somethingShouldHappen()

```
/* The classes given below were written for the purposes of
 * this exam. In reality, they would each be in their own
 * separate file, but are reprinted here as one long file
 * for ease of reading. This “print-out” spans two pages,
 * so please look at both pages while answering the
 * following questions.
 */

public class App {
    private Puppy _puppy;
    private ID _id;
    public App (){
      System.out.println("App constructor called.");
      _puppy = new Puppy(new Toy());
      this.setID(new ID(this, _puppy));
      System.out.println("App constructor end.");
    }
    public void setID(ID id) {
      _id = id;
    }
    public static void main (String[] args) {
      App app = new App();
    } // end of main ()
}// App
```

```
public interface Colorable {
   java.awt.Color getColor();
   void setColor(java.awt.Color color);
}// Colorable

public class ID implements Colorable{
   private Animal _animal;
   private java.awt.Color _color;
   public ID (App app, Animal animal){
      _animal = animal;
      _color = java.awt.Color.BLACK;
   }
   public java.awt.Color getColor() {
      return _color;
   }
   public void setColor(java.awt.Color color) {
      _color = color;
   }
}// ID

public class Animal {
   private Toy _toy;
   public Animal (){ _toy = new Toy(); }
   public Animal (Toy toy) { _toy = toy; }
   protected Toy getToy() { return _toy; }
   public void somethingShouldHappen() {
      _toy.doSomething();
   }
}// Animal

public class Puppy extends Animal{
   public Puppy() {}
   public Puppy (Toy toy){
      super(toy);
      this.doSomethingWithThisColor(this.getToy()
                                              .getColor());
   }
   public void doSomethingWithThisColor
                                 (java.awt.Color color){
      this.getToy().setColor(color.darker());
   }
   public void somethingShouldHappen() {
      super.somethingShouldHappen();
      this.getToy().doNothing();
   }
}// Puppy

public class Toy {
   private java.awt.Color _color;
   private String[] _sounds;
   public Toy (){ _color = java.awt.Color.RED; }
   public void setColor(java.awt.Color color) {
      _color = color;
   }
   public java.awt.Color getColor() { return _color; }
   public void doSomething() {
      _sounds = new String[20];
      for ( int count = 0; count <_sounds.length; count++){
          _sounds[count] = "Squeak";
      } // end of for ()
      System.out.println(_sounds);
   }
   public void doNothing() {
      //This method really does nothing.
   }
}// Toy
```

For questions 116 – 125, assume the following variable declarations. Note that any ellipses (…) indicates material that will not affect your answer to the question and can be safely ignored. For each of the method calls in questions 116 - 125, you should circle the name of the class/interface that defines the method that will be executed for the method call. If the call is Illegal, circle the choice that corresponds to "Illegal".

```
Colorable c = new ID(…);
Animal animal = new Puppy();
Puppy puppy = new Puppy();
```

116) `c.getColor();` [1 point]

a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

117) `c.setColor(…);` [1 point]

a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

118) `c.setID();` [1 point]

a. App
b. Animal
c. Puppy
d. Colorable
e. ID

f. Toy
g. Illegal

119) `animal.getToy();` [1 point]
a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

120) `animal.somethingShouldHappen();` [1 point]
a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

121) `animal.doSomethingWithThisColor(…);` [1 point]
a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

122) `puppy.somethingShouldHappen();` [1 point]
a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

123) `puppy.doSomethingWithThisColor(…);` [1 point]
a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

124) `puppy.getToy();` [1 point]
a. App
b. Animal

c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

125) `puppy.getColor();`[1 point]
a. App
b. Animal
c. Puppy
d. Colorable
e. ID
f. Toy
g. Illegal

Recall that questions 126 – 127 still refer to the UML diagram and code used for questions 116-125.

126) Circle the names of all methods that are simply inherited (not overridden) by some other class. If no methods are inherited, circle the choice that corresponds to "None". [6 points]
a. void main (String[] args) //in class App
b. void setColor(java.awt.Color color) //in class ID
c. Animal () //in class Animal
d. Animal (Toy toy) //in class Animal
e. Toy getToy() //in class Animal
f. void somethingShouldHappen() //in class Animal
g. Puppy () //in class Puppy
h. Puppy (Toy toy) //in class Puppy
i. void somethingShouldHappen() //in class Puppy
j. void doSomethingWithThisColor(java.awt.Color color) //in class Puppy
k. void setColor(java.awt.Color color) //in class Toy
l. `None`

127) Is the method `somethingShouldHappen` in the class `Puppy` partially overridden or totally overridden? [1 point]
a. Partially overridden
b. Totally overridden

6.4 Critique of the Exam

The exam underwent three distinct rounds of critique by a total of five distinct reviewers at two different institutions (one public, one private). Some of the reviewers reviewed the exam multiple times. All critiques were completed before the data collection began for the study described in Chapters 7 and 8 of this dissertation.

In the first critique, the initial question pool was narrowed to a more reasonable number of questions for the three-hour time limit by eliminating duplicate questions. After this round of critique, I solicited the help of three of my current teaching assistants to simulate an exam administration. While the teaching assistants are not introductory level students, they were each able to complete the exam in under two hours, which gave me initial confidence that students would be able to complete the exam in the time allotted.

In the second critique, instructors for the CS2 course in which the exam was to be offered as the final exam gave their commentary on its contents. Some additional questions were removed at this stage, others were reformatted, and grammatical and spelling mistakes were corrected.

In the third and final critique, instructors from outside the course were asked for an independent analysis of the exam. Three instructors, who had all been involved in teaching CS2 at some point in the recent past but were not teaching the course when the study was administered, were asked to give general commentary on the exam and answer the questions on the reviewer questionnaire (see Appendix C of this dissertation). Comments were considered and questions changed appropriately to clarify directions and any additional spelling and grammatical errors.

6.5 Conclusion

This chapter discussed the creation of the exam and how the exam covers all of the topics left in our topic list after the elimination of topics discussed in Chapter 4. This chapter also discussed the process by which the exam was reviewed for content by domain experts. The comments and critiques of these experts were taken under consideration when creating the final version of the exam and many suggestions were implemented based on reviewer feedback.

Aside from issues with question particulars, which were either addressed, or discussion given as to why the suggestions were ignored, the reviewers felt that the exam had the correct level of difficulty for the students and that content was comprehensive enough for a CS1-CS2 assessment.

The only major issue that has come up repeatedly from most of the reviewers has been length. The issue of length is addressed in the analysis chapter (Chapter 8) as part of the administration process for this exam was a recording of the time to completion for each student.

6.6 Coding of Questions on Exam

The following table gives an alternate view of the categorization of the questions on the exam in the order that the questions appear on the exam and which section of this chapter discusses these questions.

Question Number	Brief Description	Section
1	BST Insert	6.3.4
2	BST Find	6.3.4
3	BST Delete Root	6.3.4
4	Collection with generic type	6.3.3
5	Iterating over a collection	6.3.4
6	Array indexing	6.3.4
7	Array indexing	6.3.4
8	Array indexing	6.3.4
9	Array re-sizing	6.3.4
10	Creating and populating an array	6.3.4
11	Searching a two-dimensional array	6.3.4
12	Deletion from a doubly-linked list	6.3.4
13	Tree vocabulary	6.3.4
14	Tree vocabulary	6.3.4
15	Tree vocabulary	6.3.4
16	Tree vocabulary	6.3.4
17	Tree vocabulary	6.3.4
18	Tree vocabulary	6.3.4
19	Graph representations	6.3.4
20	Graph vocabulary	6.3.4
21	Graph vocabulary	6.3.4
22	Running time based on implementation of data structure	6.3.4, 6.3.6
23	Why should a stack not inherit from a vector?	6.3.4, 6.3.7
24	Choose the most appropriate data structure	6.3.4
25	Choose the most appropriate data structure	6.3.4
26	Choose the most appropriate data structure	6.3.4
27	Choose the most appropriate data structure	6.3.4
28	Choose the most appropriate data structure	6.3.4
29	Choose the most appropriate data structure	6.3.4
30	Choose the most appropriate data structure	6.3.4
31	Choose the most appropriate data structure	6.3.4
32	Algorithmic running time	6.3.6
33	Algorithmic running time	6.3.6
34	Algorithmic running time	6.3.6
35	Algorithmic running time	6.3.6
36	Algorithmic running time	6.3.6
37	Algorithmic running time	6.3.6
38	Conditions for correct algorithm function	6.3.6
39	Divide and conquer strategy	6.3.5
40	Hashing function running time	6.3.4, 6.3.6
41	Analysis of linked list function running time	6.3.4, 6.3.6
42	Analysis of linked list function running time	6.3.4, 6.3.6
43	Big O notation	6.3.6

44	Big O notation	6.3.6
45	Common complexity classes	6.3.6
46	Big O notation	6.3.6
47	Primitive types	6.3.3
48	Primitive types	6.3.3
49	Inheritance	6.3.7
50	Arrays	6.3.4
51	Identifying recursion	6.3.5
52	Tracing recursive methods	6.3.5
53	Tracing recursive methods	6.3.5
54	Tracing recursive methods	6.3.5
55	Tracing recursive methods	6.3.5
56	Tracing recursive methods	6.3.5
57	Recognizing base case of recursion	6.3.5
58	Recognizing recursive case of recursion	6.3.5
59	Writing a recursive function	6.3.5
60	Programming vocabulary	6.3.1
61	Programming vocabulary	6.3.1
62	Programming vocabulary	6.3.1
63	Programming vocabulary	6.3.1
64	Programming vocabulary	6.3.1
65	Programming vocabulary	6.3.1
66	Programming vocabulary	6.3.1
67	Programming vocabulary	6.3.1
68	Programming vocabulary	6.3.1
69	Programming vocabulary	6.3.1
70	Programming vocabulary	6.3.1
71	Programming vocabulary	6.3.1
72	Programming vocabulary	6.3.1
73	Programming vocabulary	6.3.1
74	Programming vocabulary	6.3.1
75	Programming vocabulary	6.3.1
76	Programming vocabulary	6.3.1
77	Programming vocabulary	6.3.1
78	Programming vocabulary	6.3.1
79	Parameter passing mechanisms	6.3.2
80	Parameter passing mechanisms	6.3.2
81	Parameter passing mechanisms	6.3.2
82	Parameter passing mechanisms	6.3.2
83	Arithmetic/logical expression evaluation	6.3.2
84	Arithmetic/logical expression evaluation	6.3.2
85	Arithmetic/logical expression evaluation	6.3.2
86	Arithmetic/logical expression evaluation	6.3.2
87	Arithmetic/logical expression evaluation	6.3.2

88	Arithmetic/logical expression evaluation	6.3.2
89	Arithmetic/logical expression evaluation	6.3.2
90	Arithmetic/logical expression evaluation	6.3.2
91	Typecasting/type checking	6.3.3
92	Simple numeric algorithms	6.3.2
93	Simple numeric algorithms	6.3.2
94	Selection using conditionals	6.3.2
95	Selection using conditionals	6.3.2
96	Selection using conditionals	6.3.2
97	Iteration using loops	6.3.2
98	Iteration using loops	6.3.2
99	Iteration using loops	6.3.2
100	Iteration using loops	6.3.2
101	String processing, loops, conditionals	6.3.2
102	String processing, loops, conditionals	6.3.2
103	String processing, loops, conditionals	6.3.2
104	Reference types	6.3.3
105	Reference types	6.3.3
106	Reference types	6.3.3
107	Visibility/scope	6.3.3
108	Visibility/scope	6.3.3
109	Type incompatibility	6.3.3
110	Inheritance	6.3.7
111	Parameter passing mechanisms	6.3.3
112	Parameter passing mechanisms	6.3.3
113	Pointers and references	6.3.3
114	Pointers and references	6.3.3
115	Pointers and references	6.3.3
116	Inheritance/Polymorphism	6.3.7
117	Inheritance/Polymorphism	6.3.7
118	Inheritance/Polymorphism	6.3.7
119	Inheritance/Polymorphism	6.3.7
120	Inheritance/Polymorphism	6.3.7
121	Inheritance/Polymorphism	6.3.7
122	Inheritance/Polymorphism	6.3.7
123	Inheritance/Polymorphism	6.3.7
124	Inheritance/Polymorphism	6.3.7
125	Inheritance/Polymorphism	6.3.7
126	Inheritance, overriding	6.3.7
127	Inheritance, overriding	6.3.7

Table 6-2: Categorization of Questions on Exam

Chapter 7

Exam Administration and Grading

This chapter will discuss the administration of the exam, the procedures followed during the data collection process and the grading process for the exam.

7.1 General Exam Administration Guidelines

The exam is a closed-book, closed-notes, closed-neighbor (i.e., not collaborative) exam designed to be administered at the end of the CS2 semester. No electronic devices should be used while completing this exam. It is assumed that most institutions have some mechanism in place for end-of-the-semester final exams. The exam is designed to be given in a three-hour time block, but designed for students to be able to finish in two hours allowing an extra hour for students to have extra time to think about the problems and not feel rushed.

Students are given an exam booklet and an answer booklet; all answers should be written only in the answer booklet. Students should be instructed that answers written in the test booklet that are not also in the answer booklet will not be graded.

As with any exam, an appropriate number of exam administrators should monitor the exam. This number should be dictated by the common practices of the institution and the number of students that will be taking the exam at one time. Exam administrators in the exam room should be familiar with the exam itself so as to be able to answer questions that may be asked by the students. Under no circumstances should answers to questions be provided to the students by exam administrators. However, student questions about where to write answers to questions and what type of answer (code, prose, etc.) should be answered, provided that the answers given do not provide the student with the answer to a particular question.

It is common practice at our institution to allow students to leave an exam room as soon as they have completed the exam. This is not a necessary component of the exam administration process but is allowed if it is common at the institution of administration.

When a student has completed the exam, it is recommended that the exam administrators collect both the exam booklet and the answer booklet from the student and not allow them to remove any exam materials from the exam room. This is to help permit the reuse of the exam in subsequent semesters, because there will be no copies of the questions available outside of the exam room. It is not recommended that the exam be reused in its current form across multiple semesters, but rather versioned to change the questions slightly in each semester. For example, in the computation questions, change values of variables, switch the order of answers for multiple choice questions, and flip some of the values of the true-false questions.

After all students have completed the exam, the exams should be graded using the grading procedures outlined in this chapter. The exam has 127 questions and is graded out of 354 points. A student's percentage scores are achieved by dividing their total number of points earned by 354 and multiplying by 100. In a particular version of the CS1-CS2 sequence at a particular institution, certain topics that are covered by this exam may not have been covered in the CS1-CS2 sequence. Exam administrators can choose not to consider certain questions from the exam in final grading for that particular exam administration. However, if the exam is to be used as a benchmarking or comparison tool, the same questions must be considered for subsequent administrations of the exam.

The administrations of this exam so far have been for data gathering and statistical-analysis purposes, so students were instructed to answer all questions to the best of their ability. After the exam was finished, the instructors for the individual CS2 courses decided which questions would and would not be counted toward the student's final exam grade. However, it is also reasonable to allow an instructor to tell the students ahead of time to skip certain questions on the exam. Still another option is to reprint the exam with those questions removed.

In any case, which questions are analyzed for a particular administration of the exam is up to the individual instructors. However, the need to remove questions should indicate to instructors that there is material missing from their CS1-CS2 sequence in relation to the guidelines given in CC2001. There are very few redundant questions on this exam because many of the repeated questions were removed during critique to help make the exam shorter.

7.2 Grading Procedure Development

There is little dispute that a rubric for grading is essential for ensuring consistency of test scores for questions that are subjective. Such a rubric should specify how the questions should be graded and what weighting (if any) certain questions should be given over others. In any question that is not multiple-choice or true-false, where the student has the opportunity to express the answer in his or her own words, the rating of the response must be interpreted by the rater.

In an article about grading essay assignments in a computer ethics course, Moskal, Miller, and King (2002) give examples of a rubric for grading student essays and give their recommendation that a rubric helps to better define the way an answer should be graded. McCauley (2003) also praises the use of a rubric that gives a clear description of what a particular grading criterion is as well as the level to which the student should demonstrate proficiency with that criterion (i.e., should partial credit be awarded, and, if so, how should that credit be determined?). Walker (2000) weighs in on the grading debate, with a discussion of how important grading is, but also how time consuming grading computer science questions can be as evidenced by published results of how many AP Computer Science exams are graded in a specific time period as opposed to other AP exams.

Therefore, it is evident that the creation of the grading rubric and the grading of the exam itself are just as important as the creation of the questions for the exam. The grading guideline is reprinted as Appendix B of this dissertation. In this chapter, a discussion of the grading specifics begins in §7.2.3.

7.2.1 Multiple Choice Questions

7.2.1.1 Questions with Only One Answer

Questions that were given multiple choices in any way (i.e., traditional multiple choice questions or true-false questions) where only one choice was required to be circled were treated as the easiest category to grade. Since there is only one correct answer out of a given number of choices (and the students would know this from reading the question), the answers that a student gives for that question were either correct or incorrect. These questions are categorized in the grading guideline as MC1A: multiple choice questions that have only one correct answer. This distinction is given to the following 30 questions (24% of the exam). The questions are: 4, 22, 40-42, 46-50, 104-105, 109-125, and 127.

7.2.1.2 Questions with More than One Answer

There are questions on the exam where multiple choices are given, but the student is expected to indicate all that are correct from the list of choices given for that question. In these questions, all correct answers must be indicated for the entire answer to be correct.

These questions are categorized in the grading guideline as MCMA: multiple choice questions that can have multiple correct answers. This distinction is given to the following 17 questions (13% of the exam): 20 - 21, 32 - 39, 51, 56, 91, 106 - 108, and 126.

7.2.2 Non-Multiple Choice Questions

7.2.2.1 Objective Free-Response Questions – One Answer

A number of questions on the exam do not give the students choices for their answers, yet have clearly correct answers. For example, questions that ask the students to state what is printed to the screen when an expression is evaluated generally have one answer. The questions like this on the exam are constructed in such a way so that there is a definite correct answer. The answers to these questions are not based on student impression or the methodology that the student uses to solve the question, as can be the case with essay questions or questions that ask the student to write source code to perform a specific function.

These questions are categorized in the grading guideline as FR1A: free response (i.e., no choices given) with only one correct answer. This distinction is given to the following 32 questions (25% of the exam): 6 - 8, 13 - 16, 52 - 55, 79 - 90, and 92 - 100.

7.2.2.2 Objective Free-Response Questions – Complex Answer

There is also a set of questions on the exam where choices are not given, but the answers are not simply one-word or one-statement answers. These answers at times require several items in a particular order, or the identification of multiple items. The other type of question that is lumped in this category are those that ask students to identify the parts of code from a given code segment. These types of questions have answers that are not as simple as one-word answers, but still are not as subjective as the questions discussed in the next section.

These questions are categorized in the grading guideline as FRCA: free response (i.e., no choices given) with possibly more than one correct answer. This distinction is given to the following 33 questions (26% of the exam): 1-3, 18-19, 24-31, 45, and 60-78.

7.2.2.3 Subjective Free-Response Questions

The last set of questions is arguably the most difficult to grade. These questions require the students to either write an explanation of an answer to a question (a short-answer-style essay) or to write source code to solve a particular problem. These questions will most likely be the questions that take the students the longest to finish and take the most time in the grading process and also subject to the most variability on the part of the raters of the exam.

These questions are categorized in the grading guideline as SG: subjective grading. There are not necessarily definitively correct answers for the questions, and they require that the raters read a more complex grading rubric for how to assign credit for each question. This distinction is given to the following 14 questions (11% of the exam): 5, 9 - 12, 23, 43 - 44, 57 - 59, and 101 - 103.

7.2.3 Weighting of Questions

The point weight for each of the questions is greatly determined by which type of question category they fall into. Questions that require the student to pick one answer out of a list of choices are considered for the purposes of this assessment to be worth less than questions that require the student to produce code as an answer to a question. I have always viewed the act of writing code to solve a problem to be more difficult than analyzing a pre-existing piece of code. I would relate it to the fact that we can find many people who are good readers of written work but far fewer who are good writers. Forming a solution in a programming language requires a synthesis of the information known about the language itself as well as the problem.

However, this high-level distinction of more points for code-writing questions and fewer points for multiple-choice questions is too coarse for this exam. A finer-grained distinction can be made by looking at the types of questions grouped as described in §7.2.1 and §7.2.2. Even within the multiple-choice questions, there are questions that require the students to pick out one answer (MC1A) or decide for each choice if the choice is a correct answer for that question (MCMA). Therefore, it was decided that questions that MC1A questions would be weighted as 1 point on the exam. MCMA questions would be weighted so that each answer choice was worth 1 point on the exam, in effect, treating each answer choice as its own mini-question. For example, if there are five choices for a question where only one is correct, and the student selects the wrong one, the student has effectively gotten two answers incorrect and three answers correct.

Free response questions that only have one answer were considered to be in the same category as single answer multiple choice questions and were then weighted as 1 point per question.

Free response questions that have a more complex answer (FRCA) were considered more difficult than the previous types of questions and were given a weighting to reflect this level of difficulty. The default weighting for these questions was 8 points each. The decision for 8 points was partially arbitrary, but partially motivated by the fact that points on a question could be broken into groups each worth either 2 points or 4 points. Also, it stresses the fact that these questions are considered of greater difficulty than other questions on the exam[20]. However, certain questions necessitated a deviation from this point system. The special cases will be explained in more detail in the next section.

Questions for which students needed to write code were considered to be as difficult as FRCA questions and were given the same weighting, 8 points each.

[20] Arguably, this method of assigning harder questions to be worth more points could be considered unfair to the students. Since the question is harder, it is more likely that the students will get the question wrong and therefore lose a larger number of points. If the harder questions were weighted the same as the easier questions, the students would lose the same number of points no matter which questions were answered incorrectly. This is perhaps a more fair approach to point assignment, but it is not the decision that was made for this assessment.

However, this method of assigning weights fails for multiple choice questions that have multiple answers with greater than 8 choices. Therefore, for questions with 8 or more answer choices, the answer choices would be worth only ½ point to keep their point value lower than the FRCA and coding questions.

7.2.3.1 Special Cases

There are a few special cases in this question-weighting scheme, included in the questions that are categorized as FRCA or SG. The questions that have weightings that deviate from the standards discussed previously are questions 2, 18, 24-31, 43-44, 45, 57-58, and 60-78[21].

Question 2 asks the student to produce the list of nodes visited in a binary search tree search. The answer to this question could be partially correct and partially incorrect, but there are only three elements in this particular search, so it was decided that each element of the search would be worth 1 point. The grading guideline explains how to assign partial credit for this question.

Question 18 asks the student to list all of the children of a particular node of a tree. The question is really a free-response question that has a definite answer. However, since three answers are expected, it is possible that a student would forget one. Doing that should not cause the student to lose all credit for the question. So, the three answers were weighted 1 point each, making the question worth 3 points. This was done to keep each

[21] Refer to Appendix A of this dissertation for the exact wording of the questions referred to in this section.

question worth a whole number of points wherever possible and to avoid fractional point values other than ½.

Questions 24 - 31 ask the students to choose, out of a list of given data structures, the ones that are most accurately described by the problems given in each question. Some of these could have multiple answers and require the students to apply what they know about data structures to novel problems. However, they are not as complex as the coding questions, because the students are given an answer-bank of choices. Also, some of the questions had three answers, which was hard to divide into 8 points. Therefore, a compromise of 6 points per question was assigned to them.

Questions 43 and 44 are true-false questions that ask the student to re-write a statement to be true if they believe it is false. Both of these questions involve Big O notation, so the students have to give the correct Big O bounds for the question. This is more complex than a regular true-false question. The true-false part of the question was weighted the same as the other true-false questions on the exam (1 point). The re-writing was given a weighting of 2 points, because it required a little more effort than merely stating the truth value of a statement. Therefore, each of these two questions is worth 3 points.

Question 45 asks the student to organize functions by growth rate. Originally the question was to be weighted 8 points, but since there are only 7 functions to be ranked, it made point breakdown easier to assign the question 7 points.

Questions 57 and 58 ask the student to provide the base case and recursive case of a recursive formula. These two questions were given half the weight of a code question, because they are not as complex as some of the coding questions, yet require a bit more than FR1A question.

Questions 60 – 78 ask the students to identify the parts of code associated with a given programming vocabulary term. The answers to these questions are free response and are categorized “complex” because the answer that needs to be provided is not as simple as some of the other free-response questions, in turn because students have to look at a piece of code and identify the correct part. Also, the grading is not as straightforward, because there may actually be multiple correct answers for one of the vocabulary terms. In the grading guide, the possible answers are elaborated. These questions fit better into the category of a simple free-response question or even a multiple-choice question, because the students have to pick out from a given code segment where the vocabulary terms are. Therefore, the questions were given the weighting of 1 point each.

7.2.4 Partial Credit (The Triage Theory of Grading)

The grading guideline gives a detailed description of how the coding questions are to be graded. However, the justification of this grading system must be given. When working on the grading system for the exam, several issues were considered. Among them were the relative weightings of the individual types of questions. The most difficult consideration was how to handle questions that had the potential for partially correct answers. These questions are primarily the coding questions on the exam, although a few of the complex-answer, free-response questions use a system for partial credit as well.

The system for partial credit is based on the Triage Theory of Grading (Rapaport, 2006). In this system, a totally correct answer is given full credit and an answer that is clearly wrong is given minimal credit. In this system, zero points is reserved for not putting any answer for a question. Any answer that falls in between is given half credit. In this way, the points for a question do not have to be broken down across syntactically specific constructs of the particular question, but rather across the general themes of the question.

This theory concisely explains how to grade coding questions in a way that could be easily communicated to the raters in the grading guideline. Also, it allows for the language of implementation to be changed without necessitating an entire re-write of the grading guideline. The theory was adopted for the coding questions and some of the free response questions.

The student’s final score is the total number of points earned on the exam. This number is then divided by the total number of possible points to get a percentage score. Instructors can use the percentage score however they see fit within their own classrooms. This leaves the correlation of a particular percentage score to a letter grade to the discretion of the course instructor.

7.3 Study Design

In order to determine the reliability and provide data to determine the validity of the assessment instrument, the exam was administered to students so that their scores could be analyzed. Since the exam was designed to be an assessment of the CS1-CS2 sequence, the instructors for the CS2 course (CSE 116) at UB were approached about the possibility of giving this exam as their final exam for the course in both the Fall 2005 and Spring 2006 semesters. As discussed in Chapter 6, the instructors for both courses agreed to administer the exam and offered suggestions for the improvement of the instrument as well.

7.3.1 Research Questions

For validity, students who participated in the study consented to have their final grades for CSE 115 and CSE 116 analyzed. In an attempt to show criterion validity, the students' results on the exams were compared to both their CSE 115 and CSE 116 grades to look for any correlations between the two scores. It was hypothesized that the exam score would be correlated with their performance in both of these courses. However, it is known that the exam itself is factored into the student's final grades for CSE 116. Therefore, student's scores in CSE 116 are compared both to this final grade as well as a recomputed grade with the final exam score removed.

Other questions that the study attempted to answer were:

1) How long on average does it take students to finish the exam?
2) Would students of different genders perform differently on the exam?
3) Would students of different ages or level in school perform differently on the exam?
4) Would computer science and engineering majors or minors perform differently from non-majors on the exam?
5) Would students who did not take CS1 at the University at Buffalo perform differently on the exam?
6) Would students who repeated either CS1 or CS2 perform differently on the exam?
7) Would students with prior programming experience perform differently on the exam?

7.3.2 Subjects

The subjects of this study were students enrolled in CSE 116, Introduction to Computer Science for Majors II, at the University at Buffalo in the Fall 2005 and Spring 2006 semesters. This course at the University at Buffalo is equivalent to a CS2 as described in CC2001. Institutional Review Board approval was obtained before data collection began for this study. The instructors for these courses agreed to give this exam as a final exam for CSE 116 in those two semesters. Students were required by the syllabus to take the final exam for the course. However, the students were not required to participate in the study that analyzed the results of their exam scores.

Students were informed of the study and their ability to participate in it prior to final exam day. On the day of the exam, students were presented with the consent form to sign if they were interested in participating in the study. One hundred students agreed to participate in this study[22].

7.3.3 Study Protocol

In order to correlate student performance on the exam with their performance overall in both CSE 115 and CSE 116, and to help the instructors of CSE 116 use the exam data as final exam grades for their students, student names needed to be associated with their exam papers in some way. However, since the exam needed to be graded, having student names on the exam could introduce rater-bias effects if the student was known to the rater.

To eliminate this, the students were assigned an exam number for the study. This exam number appeared with their name on only the first page of the exam booklet. Student names did not appear on the answer booklet at all, and students were instructed not to put their name or any other identifying information on the answer booklet.

To gather information to answer the additional questions given in §7.3.1, a demographic questionnaire was created. This questionnaire is given as Appendix D of

[22] See §9.2.1 for a comparison of students who did participate in the study to those students who did not.

this dissertation and collects information about gender, age, major, and prior programming experience in an attempt to provide answers to the additional research questions. Only the student's exam number appeared on the demographic questionnaire.

University at Buffalo final exams are scheduled by a university-wide scheduling system. The date and time of the exam is publicly announced, and instructors use this schedule to inform students when the final exam for a course will be held. Before the exam began, exam packets (study consent form, demographic questionnaire, exam booklet, and answer booklet) were distributed in the exam room by the exam administrators. Students were spaced appropriately in the room for a final exam so that they could not directly see any other students' papers.

When students began arriving at the exam room, they were instructed to take a seat where there was an exam, but not to open the exam booklet until we officially began the exam. At the exam time, I informed the students about the study, the consent form, the demographic questionnaire, their exam numbers, and the answer booklet. Students were then given a few minutes to complete the demographic questionnaire and read the instructions on the front page of the exam. They also had the opportunity to ask any questions about the study at this time.

All of the students began the exam together and were given three hours to complete it. Exam administrators were in the room for the entire exam, and students were free to leave the exam room as soon as they were finished with the exam. When a student finished the exam, it was brought to the front of the room, and the time of completion was noted on the front page of the answer sheet.

Students were not allowed to leave the room unsupervised while they were taking the exam. If a student requested to leave the room for any reason other than a visit to the lavatory, the request was denied. If a student requested to use the lavatory, one of the exam administrators escorted them to the lavatory and waited to escort them back to the exam room.

Students could ask questions during the exam. These questions mainly consisted of confusion about where to write answers. Some students did not realize at first that there was an answer booklet. Questions 60 – 78 were particularly problematic, because students did not seem to notice that the directions said that the code for those questions was in the answer booklet only.

After three hours, all exams were collected. Exams were kept in storage by me until the grading process could begin. Interesting conflicts uncovered while grading the exams are described in §7.4. However, to ensure the integrity of the original answers, copies of the answer sheets were made, and it was these copies that were actually graded during the grading stage of the process.

7.3.4 Exam Grading for Study Participants

As described above, many of the questions on the exam are objective, having one and only one correct answer. These questions are easy to grade, because they do not suffer from the problem of personal judgment invading the grading process. They suffer from grading errors, however, due to simple human error. This type of error is easily eliminated if the exam is machine graded. Machine grading of this exam was not attempted for the purposes of this study or this dissertation.

Other questions, mainly the coding questions, require that the raters use their judgment to assign a grade to the student. This type of subjective grading must be carefully monitored to ensure consistency among the grades assigned to each question. To prevent rater inconsistency, a grading rubric was provided. In order to test that the rubric was clear and that the question grading would be consistent, two raters were assigned to grade each question that had the possibility for partial credit. After both raters graded a question, their grades were compared. The results were surprising and are discussed in §7.4.

When all of the exam grades were computed, the scores were given to the CSE 116 instructors for use as they saw fit in their respective courses. The data collected from the exam scores and the demographic questionnaires were then analyzed; the results are discussed in Chapter 8.

7.4 Rating the Raters

For questions that had the potential to be awarded partial credit, two raters were assigned to each student’s paper. They were each given a copy of the student answer sheets and the grading rubric. Grading of the questions was done independently and then the scores were compared. Even though inconsistent scores were expected, the raters personally expressed surprise at the number of inconsistent scores uncovered in the grading of the exam. In an attempt to resolve the conflicts and to try to determine the cause of the discrepancies, the raters discussed the justification for their ratings. During this process, some clarifications were made to the grading guideline. If clarifications were made to the grading guideline, the questions were re-graded using the new guideline. These re-graded scores are the ones that have been analyzed for this study.

However, the most interesting result of the discussion were the number of conflicts that were made simply by mistake; after looking at the student's answer a second time, some raters realized that they had in fact given the student an incorrect score the first time.

This leads to a recommendation in the grading guideline that those questions (subjective free response) be graded by two raters if possible, to trap for such inconsistencies. However, this is not always feasible and is not a necessary part of the grading process for this exam.

7.4.1 Questions Double Graded to Ensure Rater Consistency

After the exam was graded by both raters, it was extremely disheartening to discover that only 14 of the 100 exams for study participants did not have some sort of grading conflict in the subjective grading questions. This meant that 86% of the exams had at least one grading conflict that needed to be resolved. Tables 7.1 and 7.2 show the statistical breakdown of how many errors were present in each exam and for each question.

Appendix E gives a full discussion of the errors noticed within the grading discrepancies. This discussion takes place in the context of a table that shows the exams that have grading discrepancies as well as the grades given by each of the two raters. The discussion included in the tables in the appendix will elaborate on what each discrepancy was between the raters and how it was resolved.

Question Number	Number of Exams that had Discrepancies[23]
Question 5	12
Question 11	12
Question 12	13
Question 57	14
Question 58	15
Question 59	18
Question 10	19
Question 102	23
Question 101	25
Question 23	27
Question 103	28
Question 9	29

Table 7-1: Discrepancies by Question

[23] Note that this number also corresponds to the percentage of exams with a conflict because the total number of exams studied was 100.

Number of Discrepancies in Exam	Number of Exams with that Number of Discrepancies[24]
7	1
8	1
6	4
5	6
4	9
0	14
1	19
3	20
2	26

Table 7-2: Number of Discrepancies per Exam

7.4.2 Discussion of Rating the Raters

Appendix E gives a detailed breakdown of the discrepancies uncovered from the double-grading of the subjective questions on the exam discussed in §7.4.1. Also discussed in Appendix E are the resolutions of the discrepancies and which rater was correct in each case. In summary, rater 1 was correct approximately 44.5% of the time, while rater 2 was correct 47% of the time and neither rater was correct 8.5% of the time when just considering the discrepancies. It is interesting to note that every question that was double graded had at least one exam where the raters gave two different grades, and, upon the two raters coming together to discuss the discrepancies in grading, the decision was reached that neither grade originally given was actually correct.

To resolve the conflicts, the raters were brought together in the same room with all the exam papers that were in conflict. The conflicts were handled on a question-by-

[24] Note once again that since the number of exams is 100, this number is a raw value as well as the percentage of the total.

question basis, meaning that all conflicts for question 5 were resolved first, followed by question 9 and so on. Before looking at any student answer papers, the question and grading guideline were reviewed so that each rater could remember what the question was and how it was to be graded.

The raters were each given the answer book that they graded for a particular question in order to see their own notes (if any) about the thought process they used while grading the particular answer. For each student, the raters read the answer for the student in their answer book and then discussed what rating the student should have. Any conflicts were talked over and resolved through discussion at this point, and a single score was decided for each answer.

Through these discussions, it was discovered that both raters made errors in grading that were simply human errors. The errors were not the fault of a poor grading guideline or even a poor understanding of the guideline. When discussing the inconsistencies between the two raters, there were times that one rater looked up at the other and said things like "I'm sorry, this should be given no credit – I don't know what I was thinking." This exchange was repeated numerous times throughout the sessions.

Overall, the conflicts were resolved by developing a few minor modifications to the wording of the grading guideline for that question. These changes are now printed in the grading guideline to be used to grade the exams. These modifications were clarifications, rather than actual re-writes of the guideline. However, the exams were looked at once more with the newer, refined guidelines.

The raters recommended re-writing questions 57 and 58 as multiple-choice questions that ask the students to identify the base case(s) and recursive case(s) of a recursive definition. The raters noted that students copied down various configurations of answers for the base case and recursive case, making accurate grading difficult. This way, the student's knowledge of base cases and recursive cases is tested, not their ability to copy notation from the question in a form the rater will believe mimics understanding. This change has not been implemented in the version of this exam reprinted in Appendix A, but is left for future revisions of the exam.

7.5 Recommendations for Grading

7.5.1 Two Raters for Subjective Questions

It is highly recommended that questions that could be graded subjectively be graded by more than one rater. These questions are indicated as type SG in the grading guideline. Of course, this could be difficult, if not impossible. However, the benefit of two raters and the comparison of their ratings is significant. It can point to a failure in understanding of the guideline and therefore a skew in the scoring of the exam. It is also important that, if two or more raters' grades conflict, the conflicts be discussed and resolved between the raters, so that one score is agreed upon for that question. Due to the triage style of grading the questions, the discrepancies will not involve minor syntactic minutiae, but rather the larger issues of the question.

If it is not possible to ensure double rating for these questions, the next-best solution is to have one person rate the entirety of the questions for a single administration of an exam. In this way, decisions about how questions are graded are resolved with the single rater and the rater will know how previous answers were graded in an effort to minimize errors. Many instructors would assume that a single rater will be consistent, but as discovered by the exercise in having two people grade questions on the exams for the study, even one person can be inconsistent with themselves. A single rater should be encouraged to grade all questions twice being blind to the previous rating. It is also encouraged that the second grading be in a different order from the first grading.

Overall though, the checks and balances of two raters is preferred.

7.5.2 Grading Simultaneously

If two raters will be grading the same question on the same exam, or if it becomes necessary to break the grading of a single question across two or more raters (not to be checked for discrepancies), it is recommended that the grading take place simultaneously. In fact, it is recommended that the grading be accomplished at the same time in the same room, with discussion encouraged between the raters.

The discussion between the raters ensures that everyone grading a particular question has understanding of the guideline as well as what constitutes partial credit for a question. Also, it allows intermittent discussion while grading, if concerns arise over a particular student's answer. This will allow for greater consistency between the raters, if both are rating the same question for the same student. However, it can also establish rules for all the raters, if the grading is split among them. Standards that can be adhered to can be established during this process.

If grading must be split among different raters with each rater grading a disjoint subset of the total number of exams, it is beneficial to have the raters sit down with at least two different students' answers and grade them together, to further facilitate communication among the raters and to provide a quick way to check that all the raters understand the grading for a particular question.

7.5.3 Grading Anonymous Tests

It was found surprisingly refreshing by the raters for there not to be any student names on the answer booklets. While this is by no means required of the raters of this exam, it is noted that there seems to be a greater focus on the grading of an exam answer when a student's name is not present on the paper, and therefore no pre-conceived notions of the student are available to potentially cloud the judgment of the rater (for either good or bad).

Also, when looking at statistical information (mean, median, high/low score), having anonymous data was considered a benefit. The instructors were able to be more analytical about decisions about the appropriateness of the scores. The instructors commented that they were not influenced by the notion that student X, who is a good student, did not do well on this particular question or the exam as a whole, which might cause them to reconsider the weighting of the particular question or exam as it pertains to overall grades. Since it was not known at first which scores belonged to which students, the aggregate data could be analyzed to deem the results of a question acceptable or not.

Once again, these are observations that were made throughout the grading process of the exams that will be used as part of the study. I encourage faculty members to try anonymous grading for a particular exam to see if they too notice this difference. The only administrative overhead for this style of grading is to number the exams and to maintain an external list of correlations between exam numbers and students.

7.6 Conclusion

This chapter discussed the way the exam should be administered as well as the creation of the grading rubric for the exam. A study was undertaken to collect data to analyze the reliability and validity of the exam, and the design of the study was described in this chapter as well.

The chapter ended with a discussion of the process and discrepancies that were found when rating the student exams. All discrepancies were resolved before analysis of data began, but the discrepancies shed interesting light on the grading process and allowed for further refinement of the grading rubric for the exam.

Chapter 8

Experimental Results and Analysis

This chapter presents a statistical analysis of the collected data from two administrations of the exam as a final exam for CSE 116 at the University at Buffalo.

8.1 Overall Exam Statistics

Recall that the exam has a total of 127 questions and is scored out of 354 points. Reporting of the statistics will give scores out of the total points possible for the exam as well as that raw score as a percentage. One hundred exam scores were analyzed during the course of the study; therefore, the *n* for all statistical tests should be assumed to be 100, unless otherwise stated. All statistics are reported in aggregate, that is, the two administrations of the exam are treated as one for purposes of statistical analysis. An alpha level of 0.05 was used for all statistical analyses in this chapter[25].

- The minimum score on the exam was a 138 (38.9%).
- The median score on the exam was a 254 (71.7%).

[25] Alpha level indicates the confidence level for statistical analysis. An alpha level of .05 (or a confidence level of 95%) indicates that *p* values for all statistical tests run will need to be less than .05 to be considered statistically significant. Therefore, any *p* values less than .05 are considered significant results for a particular statistical test. The *p* value is the name given to the value analyzed for statistical significance.

- The maximum score on the exam was a 334 (94.3%). No one earned a perfect score on the exam.
- The mean score on the exam was a 243.13 (68.6%).

8.2 Time

8.2.1 Time to Complete[26]

As students completed the exam, their time to completion was noted on the top of their answer paper. From these times, we have been able to determine the following information:

- The minimum time that any student spent on the exam was 1 hour 20 minutes.
- The median time spent on the exam was 2 hours 36 minutes.[27]
- The maximum time that any student spent on the exam was 3 hours 00 minutes. All students were stopped at this time regardless of whether or not they had completed all the questions on the exam.
- The mean time for completion of the exam was 2 hours 31 minutes.[3]

[26] The *n* for these time statistics is 98 because there were two students whose time was not reported on their answer sheets.

[27] Both the median and the average times is longer than a 2-hour exam. One of the goals of the exam was to create a 2-hour exam that could be administered in a 3-hour time period. Since the median time is longer than 2 hours, a possible direction for future work is to look to making the exam shorter so that the median time fits into the original 2-hour window.

Table 8-1 shows the times to complete and the number of students who completed in that time. Figure 8-1 shows a graph of these times.

Time to Complete	Number of Students Completed in that Time	Time to Complete	Number of Students Completed in that Time
1:20	1	2:27	2
1:23	1	2:28	2
1:26	1	2:29	2
1:45	1	2:30	2
1:47	1	2:31	3
1:49	2	2:36	3
1:50	2	2:37	4
1:51	1	2:39	1
1:52	1	2:40	1
1:58	2	2:41	1
1:59	1	2:43	1
2:04	1	2:44	1
2:06	2	2:45	1
2:07	1	2:46	2
2:08	2	2:47	1
2:09	1	2:48	1
2:11	2	2:50	1
2:12	1	2:53	2
2:13	1	2:54	1
2:15	4	2:55	1
2:16	1	2:56	1
2:19	2	2:57	2
2:20	3	2:58	1
2:21	1	2:59	2
2:24	1	3:00	22[28]

Table 8-1: Time to Complete Exam

[28] See §8.2.3 for discussion of this group of students.

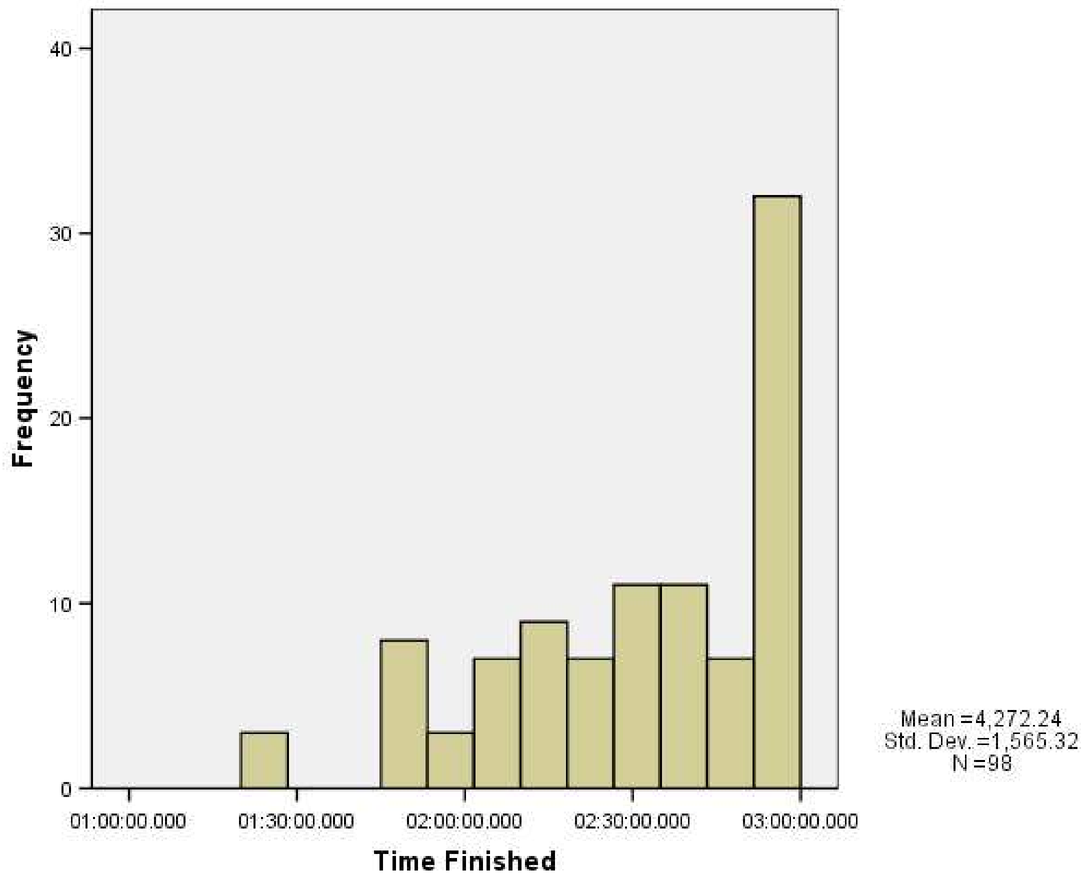


Figure 8-1: Histogram for Time to Complete Exam

8.2.2 Correlation with Exam Score

An investigation was undertaken to determine if there was a correlation between student’s time to complete the exam and their score on the exam. Figure 8-2 shows the scatterplot of time to complete the exam versus exam score. This plot does not show evidence of a linear relationship.

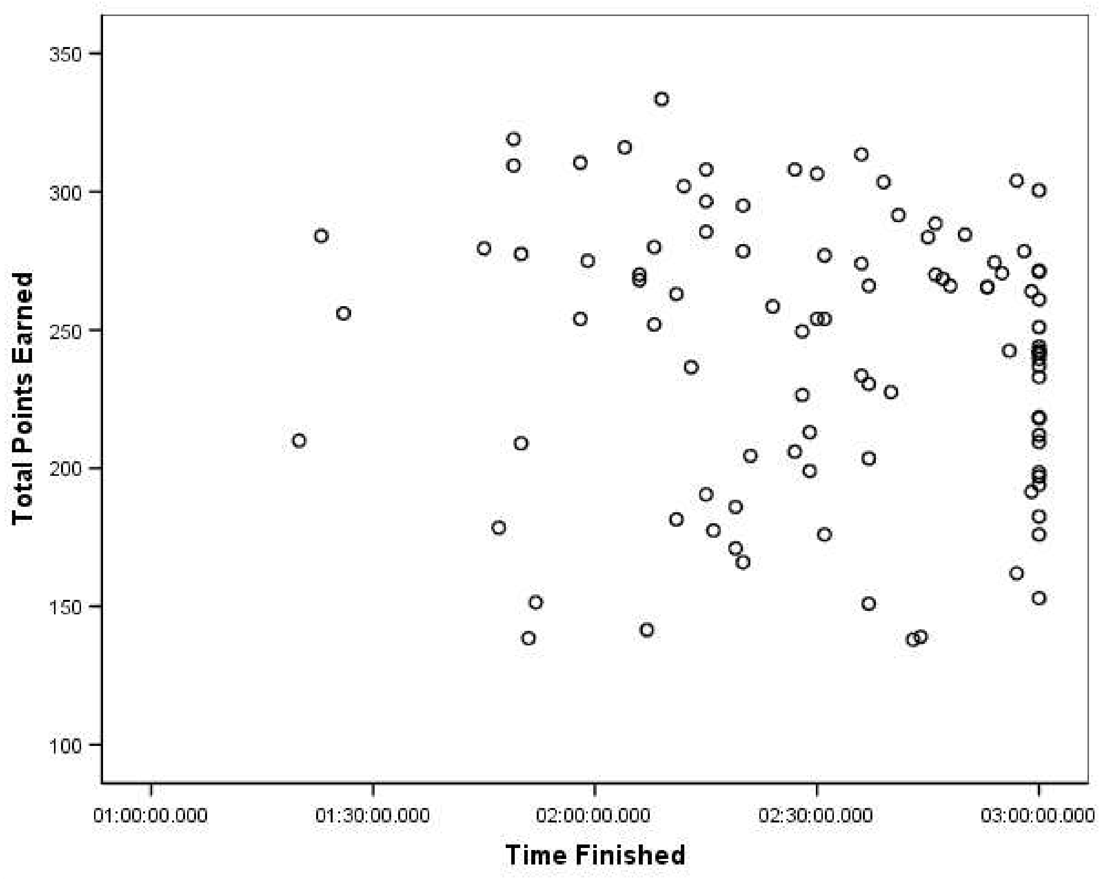


Figure 8-2: Plot of Total Points Earned versus Time Finished

The results of the correlation show that the time to complete does not correlate with student performance on the exam in the positive or negative direction.

8.2.3 Analysis of students who took the full three hours to complete exam

Even though 22 student papers were collected at the end of the 3 hour time limit, it does not appear that these students were unable to complete the exam rather that they were simply continuing to refine answers and go back to skipped questions on the exam. Looking at the individual question scores for those 22 students, none of the students in that group appeared to leave a significant portion of the exam blank. For example, all students answered questions and received credit up to and including the last question on the exam. Analyzing the two groups statistically, 76 students finished the exam before the 3 hour time expired (*M* = 247.48, *SD* = 52.107) and 22 student (*M* = 227.95, *SD* = 35.075) papers were collected at the 3-hour limit[29].

8.2.3.1 Statistical Results

Table 8-2 shows the results of the independent samples t-test for scores on the exam[30]. However, the independent samples t-test can only be used accurately if the variances between the two groups are equal. To ensure this, Levene's Test for Equality of Variances is performed as a precursor to the t-test. For these two groups, Levene's Test for Equality of Variances showed a significant p-value, which means that the two groups do not have equal variances and the traditional t-test does not apply in this case. Table 8-3 gives the results of Levene's Test.

Therefore, a t-test that does not assume equal variances must be used. Table 8-4 gives the results of such a t-test. Another alternative test when the groups studied do not have equal variances is the Mann-Whitney U. Table 8-5 gives the results of using the

[29] *M* is the mean score on the exam for the group. *SD* is the standard deviation on the exam for the group. Recall that for this particular analysis, time data was not recorded for 2 students in the study, so the *n* is 98.

[30] An independent samples t-test measures the difference in variance between two groups to determine if the groups are actually from two different populations. The variance of a group of scores tells you how spread out the scores are around the mean (Aron 2002: 28).

Mann-Whitney U to compare these two groups. In both cases, there is a statistically significant difference detected between these two groups.

	t-test for Equality of Means[31]						
	t	*Df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-1.650	96	.102	-19.526	11.837	-43.021	3.970

Table 8-2: t-test for time to complete exam

	Levene's Test for Equality of Variances	
	F	*p*
Exam Scores	6.095	.015

Table 8-3: Levene's Test for Equality of Means for time to complete exam

	t-test for Equality of Means (Equal Variances not assumed)						
	t	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-2.040	50.619	.047	-19.526	9.573	-38.748	-.303

Table 8-4: t-test (unequal variances) for time to complete

[31] In this and all tables presenting results of a t-test, the t column gives the t-value, the name given to the value that the t-test actually computes. The df column gives the degrees of freedom, or the number of scores in a sample that are free to vary. The p column is the p-value indicating significance of the result. The Mean Difference column gives the difference of the means between the two groups. The Std. Error Difference gives the difference between the standard errors of the two groups. The 95% Confidence Interval of the Difference shows that the difference between the means of these two groups falls between these two values. These values are the standard reported values for the t-test. It is only when the p-value is significant that the values have meaning. For purposes of my analysis, when a p-value is significant, we can conclude a difference between the two groups being studied.

	Mann-Whitney Test for Comparison of Means			
	Mann-Whitney U	*Wilcoxon W*	*p*	*Z*
Exam Scores	585.000	838.000	.033	-2.137

Table 8-5: Mann-Whitney test for time to complete

8.2.3.2 Analysis of Results

The results of the t-test assuming non-equal variances and the Mann-Whitney U test are statistically significant. However, exactly what meaning can be prescribed to this difference is unclear. Since there is no evidence that the students were unable to complete the exam (due to the lack of large blocks of skipped questions), it could signify that the students who took longer were not as adept with the material as those who finished earlier. However, it could also mean that the students are slower workers and need more time to complete tasks of significant size. Further exploration of this issue is needed and §8.2.3.3 discusses these two groups of students further.

8.2.3.3 Additional Statistical Results & Analysis

The students who took the full three hours to complete the exam performed differently on the exam when compared to students who completed the exam before the three hour time period had elapsed. Looking at the means for the two groups, it would appear that the students who took three hours performed worse on the exam than the others. In an effort to see if these two groups of students also performed differently in their computer science courses so far, additional t-tests were performed looking at overall course grades for the two groups in both CSE 115 and CSE 116.

Looking at course grades in CSE 115 there were 73 students who completed the test in under the three hour time limit also completed CSE 115 at UB and have recorded course grades (*M* = 3.2653, *SD* = 0.75740) and 22 students who used the entire three hours had course grades recorded for CSE 115 (*M* = 3.2273, *SD* = 0.64617). Table 8-6 shows how letter grades were converted to a 4.0 scale for statistical analysis.

Letter Grade	Conversion to 4.0 scale
A	4.0
A-	3.67
B+	3.33
B	3.0
B-	2.67
C+	2.33
C	2.0
C-	1.67
D+	1.33
D	1.0
F	0.0

Table 8-6: Conversion of Letter Grades to 4.0 Scale

Table 8-7 shows the results of the independent samples t-test for grades in CSE 115 for the two groups. As can be seen from the table, there were no significant differences between the two groups in their grades in CSE 115.

	t-test for Equality of Means						
	t	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	.213	93	.832	-.03807	.17846	-.39246	.31632

Table 8-7: t-test for CSE 115 overall course grades

Looking to the students performance in CSE 116, there were 76 students who completed the test in under the three hour time limit also completed CSE 116 (*M* = 3.0611, SD = 0.95044) and 22 students who used the entire three hours had course grades recorded for CSE 116 (M = 2.8641, SD = 0.85834). Conversion from letter grade to 4.0 scale once again uses the conversions in Table 8-6.

Table 8-8 shows the results of the independent samples t-test for grades in CSE 116 for the two groups. As can be seen from the table, there were no significant differences between the two groups in their grades in CSE 116. This is an interesting result given that the score on the exam is a contributing factor to the overall CSE 116 grade. Therefore, an additional analysis was performed with recomputed CSE 116 grades not including the final exam score.

	t-test for Equality of Means						
	T	*Df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-.874	96	.384	-.19696	.22541	-.64440	.25048

Table 8-8: t-test for CSE 116 overall course grade

Looking to the students recalculated performance in CSE 116, there were 76 students who completed the test in under the three hour time limit also completed CSE 116 (*M* = 3.1053, SD = 0.99396) and 22 students who used the entire three hours had course grades recorded for CSE 116 (M = 2.8336, SD = 1.08238). Conversion from letter grade to 4.0 scale once again uses the conversions in Table 8-6.

Table 8-9 shows the results of the independent samples t-test for recalculated grades in CSE 116 for the two groups. As can be seen from the table, there were no significant differences between the two groups in their grades in CSE 116.

	t-test for Equality of Means						
	T	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-1.107	96	.271	-.27163	.24548	-.75890	.21565

Table 8-9: t-test for recalculated CSE 116 overall course grades

Lastly, average student performance across CSE 115 and CSE 116 were considered. There were 73 of the 76 students who completed the test in under the three hour time limit also completed CSE 115 and CSE 116 at UB (M = 3.19, SD = 0.787) and 22 students who used the entire three hours had course grades recorded for CSE 115 and CSE 116 (M = 3.03, SD = 0.751). Conversion from letter grade to 4.0 scale once again uses the conversions in Table 8-6.

Table 8-10 shows the results of the independent samples t-test for average grades in CSE 115 and CSE 116 for the two groups. As can be seen from the table, there were no significant differences between the two groups in their average grades in CSE 115 and CSE 116.

	t-test for Equality of Means						
	t	*df*	*P*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-.835	93	.406	-.158	.190	-.535	.21

Table 8-10: t-test for averaged CSE 115 and CSE 116 overall course grades

8.2.3.4 Conclusions about Students who Took Three Hours to Complete

The exam scores obtained from these two different groups of students point to a difference in these two groups. Analysis of the two groups in performance overall in the courses they have completed (CSE 115 and CSE 116) yielded no significant differences between the groups in performance in the courses. In any case, the students who worked until the very end of the exam appeared to have “finished” the exam, even if not to the same level of performance as the other students who decided for themselves that they had completed the exam. Further analysis of this phenomenon is perhaps necessary to draw any additional conclusions either about the students themselves or about the exam.

8.3 Reliability

Recall that a necessary condition for determining the validity of an instrument is to first determine the instrument’s reliability. For the purposes of this dissertation, a measure of internal consistency reliability was chosen for the advantages of ease of data collection and exam administration. Test-retest reliability was not used because of the inherent difficulty in getting the same group of students to take the exam twice. Also, given that this is a test in knowledge in a particular area, it is possible that a student's knowledge could improve in this area over even a short time (especially because, for many students, this is their major). Assessing reliability through multiple forms was not attempted for this study to alleviate any further complications in the grading process.

Due to the choice of internal consistency, Cronbach's alpha was chosen as the method to assess internal consistency reliability.[32] An alpha greater than 0.7 is considered minimally acceptable for an instrument. The closer the alpha number is to 1 (meaning the instrument is perfectly internally consistent) the more internally consistent the instrument is. Cronbach's alpha was 0.903, with Cronbach's Alpha Based on Standardized Items being 0.940. This alpha number is considered to be very good.

8.4 Demographic Information

Demographic information on all of the students was collected during administration of the exam. Students were not required to answer the demographic questionnaire and were instructed to not answer any questions that they did not feel comfortable answering.

[32] Cronbach's alpha is a test used to measure internal consistency. One way to determine internal consistency is to split the test in half and compare the variance in the scores of one half of the test to the other half. Cronbach's alpha computes all possible combinations of this type of splitting of the exam.

Therefore, the *n* for some of the statistical tests will vary slightly from 100 depending on how many students elected not to answer a particular question.

8.4.1 Gender

Gathering information on gender allows us to assess whether there are gender differences in the level of performance on the exam. Ninety men (*M* = 243.67, *SD* = 49.069) and 10 women (*M* = 238.25, *SD* = 54.201) took the exam.

8.4.1.1 Statistical Results

Table 8-11 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*Df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	.328	98	.743	5.422	16.521	-27.363	38.208

Table 8-11: t-test for Gender

8.4.1.2 Analysis of Results

The results of the t-test are encouraging first steps into asserting that the exam has no gender bias. However, the number of females that took the exam is small (due to the sheer disproportionate nature of the computer science and engineering discipline).

Therefore, more subject data will need to be assessed for me to be confident in claiming that the test is free of gender bias. However, these results are encouraging in that even in this preliminary stage, they do not point towards a gender bias (in either direction).

8.4.2 Age

The age variable was gathered by asking students to choose from the following choices: 18, 19, 20, 21, 22, 23, 24, 25 – 29, 30 – 34, 35 – 39, 40 – 44, 45 – 49, 50 and over. The first several choices represent the typical age range of undergraduate students. As we move away from the typical age range, a range of age choices is presented. It is most often the case that students who are enrolled in CS1 and CS2 are freshmen in school. It is not unreasonable to assume that these students came to college right after high school. Therefore, their age is typically 18 or 19. For purposes of this analysis, we will consider that to be the typical age of a CS1-CS2 student. A breakdown of how many students fall into each age category is given in Table 8-12. We will compare the results of the typically aged CS1-CS2 student (*n* = 63, *M* = 242.74, *SD* = 51.107) and the non-typically aged CS1-CS2 student (*n* = 37, *M* = 243.80, *SD* = 46.854).

Age (Age Range)	Number of Students in Age Range
18	30
19	33
20	9
21	8
22	1
23	3
24	4
25 – 29	6
30 – 34	2
35 – 39	3
40 – 44	1
45 – 49	0
50 and over	0

Table 8-12: Age Ranges of Participants

8.4.2.1 Statistical Results

Table 8-13 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	T	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-.103	98	.918	-1.059	10.271	-21.441	19.323

Table 8-13: t-test for Age

8.4.2.2 Analysis of Results

The results of this t-test are further encouragement for a lack of bias in the exam. Since the two groups did not perform differently on the exam, the results point towards an exam that is not age-biased.

8.4.3 Year in School

Year in school was collected to be either freshman, sophomore, junior, or senior. This data was collected from the demographic questionnaire, so it captures the year in school the participants consider themselves. This can be different than the year in school the university considers the student for a variety of reasons (AP credit, transfer credits, etc).

8.4.3.1 Statistical Results

In the analysis, year in school was treated as a dichotomous variable[33]. This was done by classifying students into groups of freshmen (*n* = 52, *M* = 243.67, *SD* = 51.092), and non-freshmen (*n* = 47, *M* = 241.65, *SD* = 48.013). I have decided to treat this variable this way because I am most interested in looking at the "typical" CS1-CS2 student versus a "non-typical" student to see if academic maturity has any effect on performance on this exam. Table 8-14 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

[33] Dichotomous means that the variable only takes on two values.

	t-test for Equality of Means						
	T	*Df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	.203	97	.840	2.024	9.994	-17.811	21.859

Table 8-14: t-test for Year in School

8.4.3.2 Analysis of Results

Since age is not always indicative of year in school, these results begin to show that the exam is not biased in any direction to year in school. This is important because it could be the case that an exam like this has hidden biases for students who have had many years of experience with college courses and course final exams and would therefore cause a difference in performance for those students. These results point towards the fact that this is not the case.

8.4.4 Major

Student major was analyzed as being either computer science, computer engineering, or "other". If the student chose "other", they were asked to specify their intended major. For "other", the answers included: Bioinformatics, Business, Engineering specialties other than computer engineering, English, GIS/Cartography, Mathematics, Media Study, and Undeclared. For purposes of this analysis, all students who indicated that their major was one other than computer science or computer engineering were classified in one group.

8.4.4.1 Statistical Results

For this variable, testing was undertaken in two ways. First, computer science and computer engineering majors (*n* = 76, *M* = 242.18, *SD* = 50.795) were compared to non-majors (*n* = 23, *M* = 244.48, *SD* = 45.562). Table 8-15 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	T	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-.195	97	.846	-2.301	11.817	-25.755	21.154

Table 8-15: t-test for Major (Computer Science or Computer Engineering vs. Other Majors)

Second, computer science majors were considered independent of computer engineers and the non-majors were not considered. Since the CS1-CS2 sequence, especially with a programming-first curricular influence, could be viewed by some as inherently computer science and not computer engineering, this second test was undertaken to see if differences existed between those two groups on the exam. Once again, major can be viewed as a dichotomous variable, computer science majors (*n* = 50, *M* = 247.64, *SD* = 48.761) and computer engineering majors (*n* = 26, *M* = 231.67, *SD* = 53.905). Table 8-16 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	T	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	1.306	74	.196	15.967	12.224	-8.390	40.324

Table 8-16: t-test for Major (Computer Science Majors vs. Computer Engineering Majors)

8.4.4.2 Analysis of Results

Even though the exam is designed to assess the results of CS1 and CS2, it should not be biased towards majors, because the CS1 and CS2 course could be taken by non-majors, and they should have the same opportunity to succeed as the majors. The results of this analysis seem to indicate that non-majors have the same opportunity to succeed on this assessment.

The second analysis shows that the test is not biased between computer science or computer engineering majors. At our institution, the computer science and computer engineering majors take many of the same courses in first two years of their respective programs. However, since this test was designed for the computer science curriculum only, it was a concern that it would be biased towards computer science majors. However, this does not seem to be the case.

8.4.5 How Courses Were Taken

This section describes analysis undertaken with information provided by students on the demographic questionnaire about where they took CS1 and CS2 and also when they took CS1 and CS2.

8.4.5.1 Students Who Took Courses at Other Institutions

The first analysis that was attempted was to compare students who took any of the CS1-CS2 sequence at an institution other than the University at Buffalo. However, when the data from the demographic questionnaire were compiled, there were only 2 students who completed CS1 at another institution. This group size was not large enough to show a meaningful result, so analysis did not proceed further.

8.4.5.2 Statistical Results

Two separate analyses were performed on this data. The first compared students who took CS1 and CS2 in consecutive semesters (*n* = 85, *M* = 246.56, *SD* = 46.950) versus those who did not take CS1 and CS2 in consecutive semesters (*n* = 6, *M* = 211.17, *SD* = 46.071). For purposes of this analysis, the summer semester counted for consecutive semesters, so there were four possible ways a student could take CS1-CS2 in consecutive semesters (Fall-Spring, Spring-Fall, Spring-Summer, or Summer-Fall). Table 8-17 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	1.786	89	.077	35.392	19.812	-3.973	74.757

Table 8-17: t-test for Taking CS1-CS2 in consecutive semesters

The second analysis compared students who took CS1 and CS2 in the traditional academic year, i.e. CS1 in fall semester and CS2 in spring semester, (*n* =71, *M* = 244.80, *SD* = 49.609) versus those who did not (*n* =23, *M* = 245.30, *SD* = 40.889). Table 8-18 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-.044	92	.965	-.509	11.437	-23.223	22.206

Table 8-18: t-test for Taking CS1-CS2 in traditional academic year

8.4.5.3 Analysis of Results

The results of this analysis point towards no bias as to the taking of the introductory sequence. The exam is designed to assess knowledge of the introductory sequence. It would not be a desirable result that the group who took CS1-CS2 in a traditional academic year performed better than a group that did not, or vice versa. Likewise, a requirement of the exam should not be completion of the CS1-CS2 sequence in consecutive semesters. The results of the tests show that neither group performed differently from the other on this exam.

8.4.6 Repeaters

The next group of students that was analyzed was students who repeated CS1 or CS2 or both. Considering students who failed either, or both of the courses, was the performance on the exam of these groups different?

8.4.6.1 Statistical Results

The analysis on this variable was conducted in four ways. The first was comparing students who failed CS1 at least one time previously (n = 6, M = 236.33, SD = 22.631) with those students who had never failed CS1 (n = 94, M = 243.56, SD = 50.603). Students were asked to report whether or not they had ever failed CS1, not whether they had ever taken CS1 before. Therefore, this analysis only looks at students who definitely failed the course before and had to repeat it. Table 8-19 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*df*	*P*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-.681	8.653	.513	-7.230	10.611	-31.383	16.922

Table 8-19: t-test for Repeaters (Students who failed CS1 vs. those who did not)

The second analysis compared students who failed CS2 at least one time previously (*n* = 3, *M* = 271.17, *SD* = 31.086) with those students who had never failed CS2 (*n* = 97, *M* = 242.26, *SD* = 49.648). Table 8-20 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*df*	*P*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	.999	98	.320	28.904	28.923	-28.493	86.301

Table 8-20: t-test for Repeaters (Students who failed CS2 vs. those who did not)

The third analysis compared students who had failed either CS1 or CS2 (inclusive) at least once previously (*n* = 8, *M* = 248.75, *SD* = 31.336) with those students who had never failed either of CS1 or CS2 (*n* = 92, *M* = 242.64, *SD* = 50.693). Table 8-21 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there were no significant differences between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*df*	*P*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	.498	10.507	.629	6.109	12.275	-21.064	33.281

Table 8-21: t-test for Repeaters (Students who failed CS1 and/or CS2 vs. those who did not)

The final analysis compared students who had failed both CS1 and CS2 at least one time previously with those students who had not failed both courses before. After compiling the demographic information, it was discovered that only 1 student failed both courses before. This group was too small to analyze and no further analysis was performed.

8.4.6.2 Analysis of Results

This analysis looks in some ways for a practice effect that makes results on the exam different for the different groups. The students who failed at least one of the courses before would have had more time with the material and therefore might display different results on the exam, in the positive direction. Another possibility is that the repeating students are actually the weakest students and would therefore perform worse than the other students. The fact that there is no difference between the groups of repeaters and non-repeaters points to neither of these things.

8.4.7 Previous Programming Experience

In gathering information about students' programming experience prior to taking CS1 and CS2, students were asked to identify how many years experience they had with various programming or programming-like languages, including: C, C++, Java, Perl, JavaScript, VB, VBScript, Fortran, BASIC, Assembly, and HTML. HTML is included to trap for students who claim to have programmed before, but only have experience using HTML[34]. Students also had the opportunity to fill in other languages that they have used and their level of experience with those languages. The other languages indicated by the students were: ActionScript, ASP, Basic 8, C#, Commodore Basic, CSS, Expect/TCL, Foxpro, Karel, Pascal, PHP, Python, Ruby, QBasic, Scheme, Smalltalk, SQL, Visual Foxpro, and XML. Note that of these languages, CSS, SQL, and XML are not considered programming languages.

8.4.7.1 Statistical Results

The analysis of this variable was conducted in three ways. The first was comparing students who had programmed before taking CS1 (n = 79, M = 245.53, SD = 49.699) and those who had not (n = 21, M = 234.12, SD = 48.052). Table 8-22 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there was no significant difference between the two groups in their scores on the exam.

[34] HTML is a markup language, not a full-fledged programming language.

	t-test for Equality of Means						
	t	*df*	*P*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	.941	98	.349	11.406	12.120	-12.646	35.459

Table 8-22: t-test for Prior Programming Experience

The second analysis compared students who had programmed in Java before taking CS1 (*n* = 39, *M* = 253.51, *SD* = 50.913) and those who had not (*n* = 61, *M* = 236.49, *SD* = 47.541). Table 8-23 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there was no significant difference between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*Df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	1.699	98	.093	17.021	10.021	-2.865	36.907

Table 8-23: t-test for Prior Programming (Prior Java programming)

The third analysis compared students who had previous experience in any of the C-derived languages (C, C++, C#, Java) before taking CS1 in Java (n = 65, M = 250.20, SD = 48.337) and those who did not (n = 35, M = 230.00, SD = 49.164). The C-derived languages can be described as those whose syntax was primarily derived from C. Table 8-24 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there was no significant difference between the two groups in their scores on the exam based on the decision rule of p-values less than 0.05.

	t-test for Equality of Means						
	t	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	1.981	98	.050	20.200	10.195	-6.581	46.981

Table 8-24: t-test for Prior Programming (C-derived languages)

8.4.7.2 Analysis of Results

The results all point towards no advantage or disadvantage on this exam if a student has programming prior to taking CS1 and CS2. The further breakdown of looking at prior Java experience not affecting scores is encouraging and supports the assertion that the test is not a test of language, but rather of concepts. Looking at all C-derived languages and seeing no difference in performance indicates that experience with languages with similar syntax does not impact performance on the exam if the decision rule is interpreted strictly. However, because the p-value is right on the border of significance, it is actually difficult to make a claim either way about this result.

8.4.8 First Programming Language

In gathering information about the first language that students ever programmed in, the students were asked to pick from the languages, C, C++, Java, VB, Basic, or, if the student selected “other”, to specify what language was their first. The other languages that students indicated were: Logo, Karel, HTML, Foxpro, ActionScript, FORTRAN, and Pascal.

8.4.8.1 Statistical Results

The analysis of first programming language compared students who indicated that their first programming language was Java (*n* = 20, *M* = 234.85, *SD* = 48.148) versus those who indicated another language (*n* = 57, *M* = 242.82, *SD* = 55.234). Table 8-25 shows the results of the independent samples t-test for scores on the exam. As can be seen from the table, there was no significant difference between the two groups in their scores on the exam.

	t-test for Equality of Means						
	t	*df*	*p*	Mean Difference	Std. Error Difference	95% Confidence Interval of the Difference	
						Lower	Upper
Exam Scores	-.573	75	.568	-7.975	13.911	-35.688	19.739

Table 8-25: t-test for First Language (Java vs. not Java)

8.4.8.2 Analysis of Results

Significant differences in Java first programmers and non-Java first programmers would point towards a tendency in the exam to rely too heavily on the language of implementation of the coding examples and not on the larger CS1-CS2 concepts. Because this test did not show a significant difference in the performance of the two groups, it helps to support the premise that the exam is not testing programming language skills.

8.5 Grades in CS1 and CS2 (including Exam Score)

Data was also collected on each of the student's performance in CSE 115 and CSE 116 through the collection of their recorded letter grade for each course. These letter grades were then converted to a 4.0 scale using the weightings given earlier in Table 8-6. Once these grades were converted to the 4.0 scale, the 115 and 116 grades were averaged together to produce an average grade across CS1 and CS2.

8.5.1.1 Statistical Results

The analysis of letter grades for CS1 and CS2 proceeded in three ways. The first analysis looked for a correlation between scores on the exam and letter grades in CS1. Figure 8-3 shows the scatter plot of CSE 115 grades and total points on the exam. We can see from this graph evidence of a relationship between the two variables.

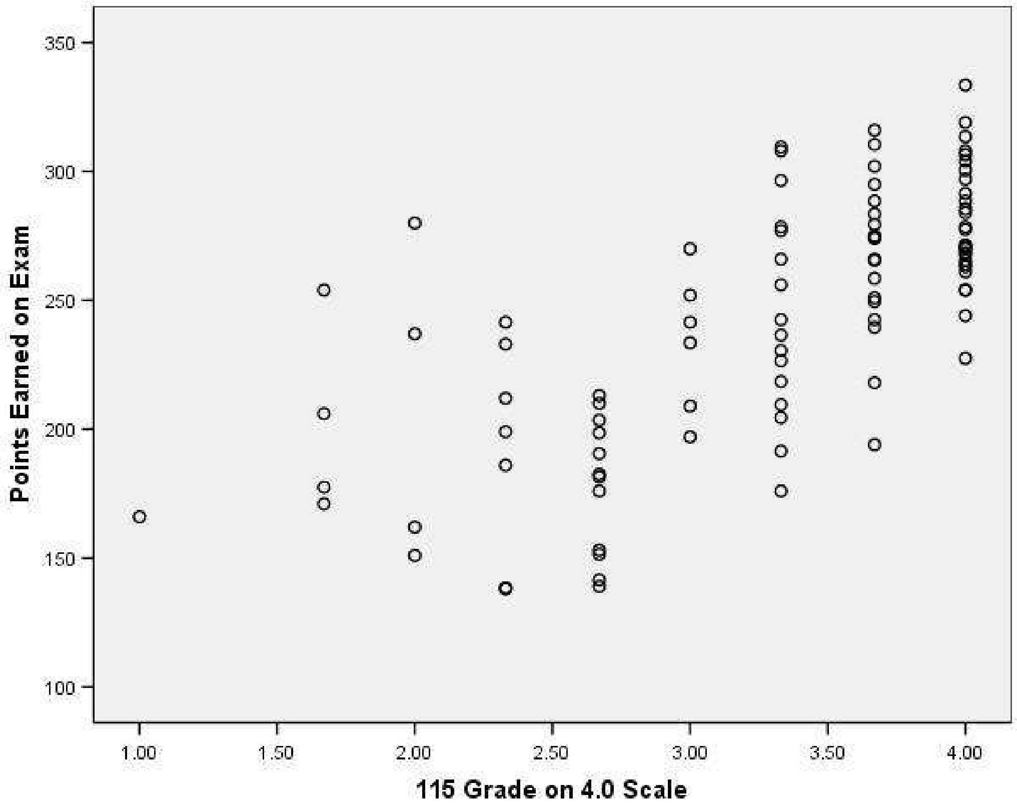


Figure 8-3: Plot of Points Earned on Exam vs. CSE 115 Overall Course Grade

To analyze whether or not there was a correlation between these two variables, Pearson's correlation coefficient (one-tailed) was computed[35]. These two measures were positively correlated $r(96) = 0.692$, $p < 0.01$.

[35] Pearson's correlation coefficient is a descriptive statistic used to describe the degree and direction of linear correlation within the particular group studied (Aron 2002).

The second analysis looked for a correlation between scores on the exam and letter grades in CS2. Figure 8-4 shows the scatter plot of CSE 116 grades and total points on the exam. We can see from this graph evidence of a relationship between the two variables.

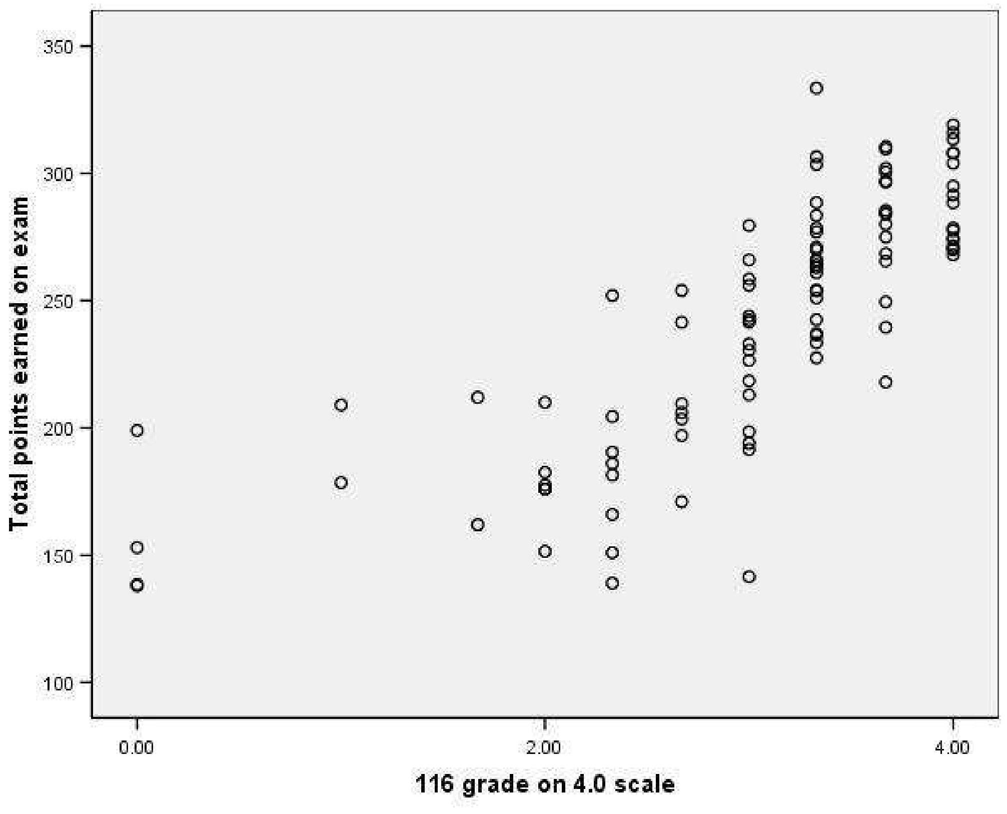


Figure 8-4: Plot of Points Earned on Exam vs. CSE 116 Overall Course Grade

To analyze whether or not there was a correlation between these two variables, Pearson's correlation coefficient (one-tailed) was computed. These two measures were positively correlated *r*(99) = 0.777, *p* < 0.01.

The third analysis looked for a correlation between scores on the exam and averaged letter grade for CS1 and CS2. Figure 8-5 shows the scatter plot of the average of CSE 115 and CSE 116 grades and total points on the exam. We can see from this graph evidence of a relationship between the two variables.

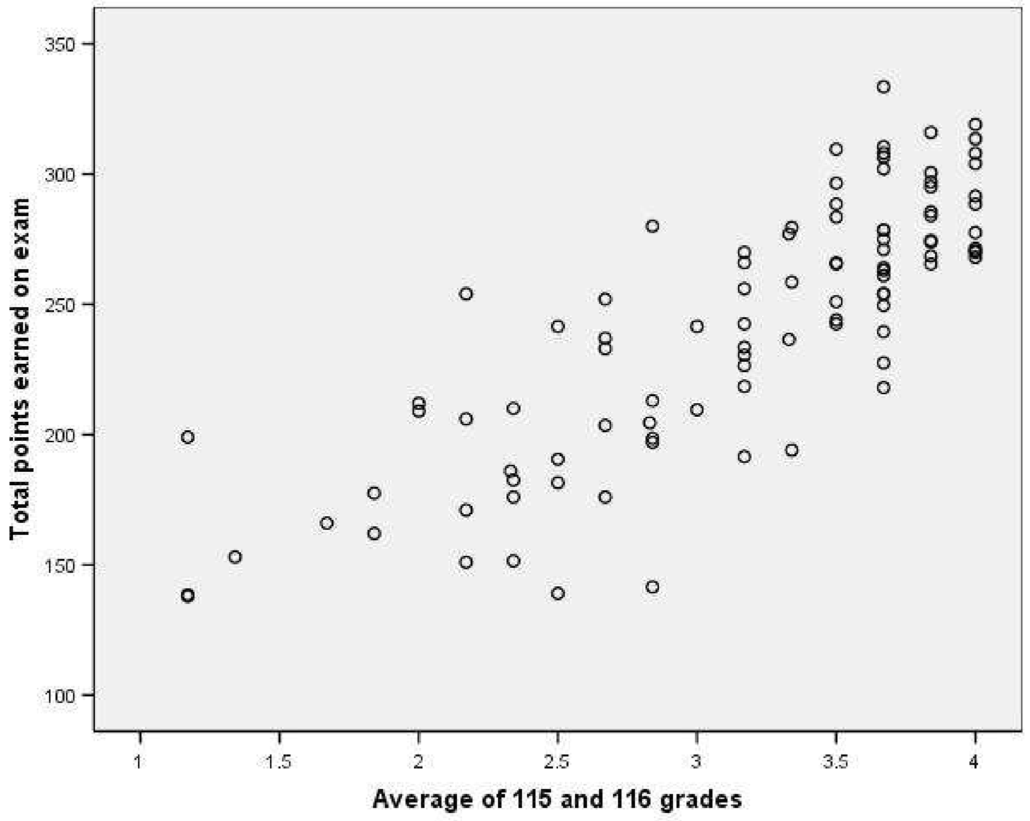


Figure 8-5: Plot of Points Earned on Exam vs. Averaged CSE 115 & CSE 116 Overall Course Grade

To analyze whether or not there was a correlation between these two variables, Pearson's correlation coefficient (one-tailed) was computed. These two measures were positively correlated $r(96) = 0.816$, $p < 0.01$.

8.5.1.2 Analysis of Results

The results of the analysis of the correlation were as hoped for. If students performed well in CSE 115 or CSE 116 or both, the test should reflect that. It is desired that students who do well in those courses overall would do well on this exam. That is what the statistical evidence shows.

However, it can be argued and should be argued that the analysis of CS2 (CSE 116) is skewed because the grade that the students received on the exam was used in computing their CS2 grades. Therefore, additional analysis was performed on their CS2 grades with the exam score removed.

8.6 Grades in CS1 and CS2 (Exam Score Removed)

To complete the analysis of the CS2 grade with the exam score removed, the instructors for the courses were asked to provide the way the final course grades were computed for each student. Then, the exam was removed and the final course grade recomputed to get a new letter grade. These letter grades were then converted using the same 4.0 scale given previously in Table 8-6. Once again, the grade for 115 and the new 116 grade were average together to produce an average grade across CS1 and CS2.

8.6.1.1 Statistical Results

The analysis of CS1 scores was not repeated at this time because the grades for CS1 did not change.

A new analysis performed looked for a correlation between scores on the exam and letter grades in CS2 computed without the exam score factored in. Figure 8-6 shows the scatter plot of the new CSE 116 grades and total points on the exam. We can see from this graph evidence of a relationship between the two variables.

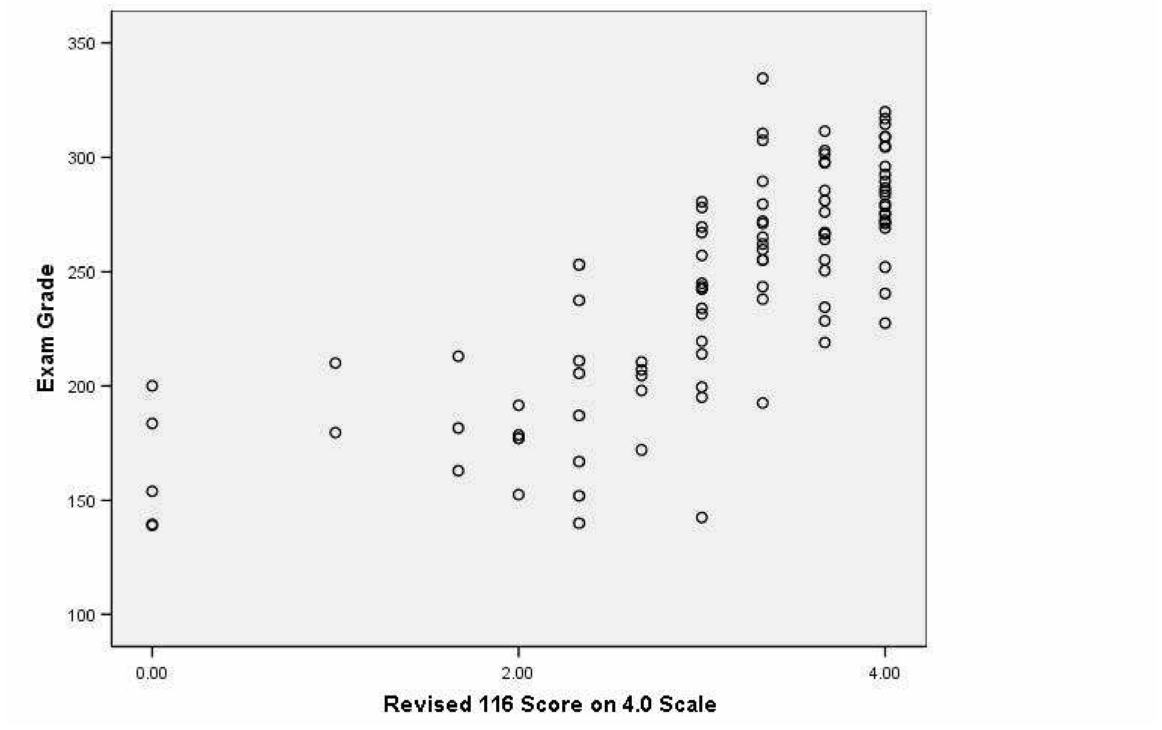


Figure 8-6: Plot of Points Earned on Exam vs. Revised CSE 116 Overall Course Grade

To analyze whether or not there was a correlation between these two variables, Pearson's correlation coefficient (one-tailed) was computed. These two measures were positively correlated $r(99) = 0.757$, $p < 0.01$.

The second new analysis looked for a correlation between scores on the exam and averaged letter grade for CS1 and CS2. Figure 8-7 shows the scatter plot of the average of CSE 115 and revised CSE 116 grades and total points on the exam. We can see from this graph evidence of a relationship between the two variables.

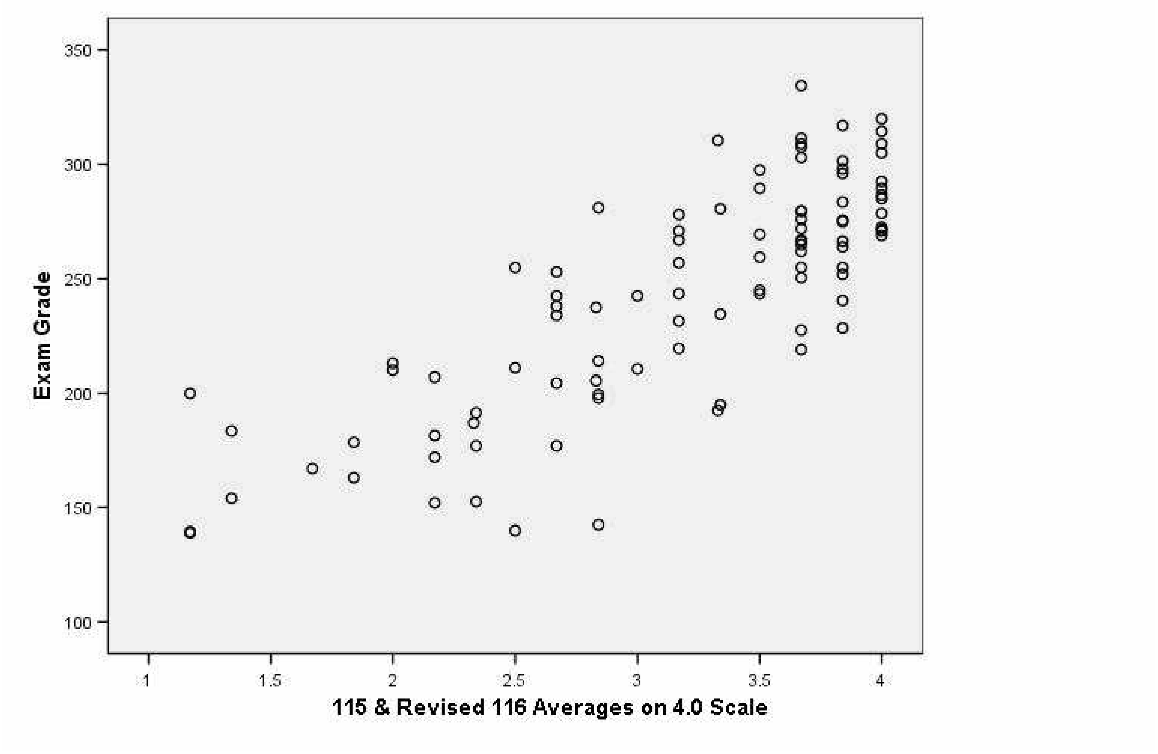


Figure 8-7: Plot of Points Earned on Exam vs. Averaged CSE 115 & CSE 116 Overall Course Grade

To analyze whether or not there was a correlation between these two variables, Pearson's correlation coefficient (one-tailed) was computed. These two measures were positively correlated $r(96) = 0.806$, $p < 0.01$.

8.6.1.2 Analysis of Results

The results of the analysis of the correlation were as hoped for. If students performed well in CSE 115 or CSE 116 or both, the test should reflect that. It is desired that students who do well in those courses overall would do well on this exam. That is what the statistical evidence shows.

Chapter 9

Discussion

The work of this dissertation sought to create a language-independent assessment for the programming-first introductory computer science courses based on the recommendations of the CC2001 curriculum document.

9.1 Discussion of Exam Creation Process

During the development of the instrument, the CC2001 recommendations were analyzed. Through this analysis, decisions were made to focus the assessment on the programming-first approaches to the curriculum (imperative-first, objects-first, and functional-first). These decisions prompted the decision that the exam needed to choose a language of implementation for code examples and student answers. However, the inclusion of a language in the exam made it imperative to create questions that did not rely on specific syntactically-oriented features of the particular language of implementation, but rather general introductory computing concepts that were simply illustrated by code examples.

A core group of topics was identified from the CC2001 recommendations as common to all the programming-first approaches. This group was only large enough with the inclusion of both the CS1 and CS2 course, making the exam an assessment for the entire first year of instruction in computer science.

The exam was created using the topic list identified as well as the learning objectives given in CC2001. Also, a grading guideline was created for the exam to be used for consistent scoring of the exam. Once the test was created, the reliability and validity of the instrument needed to be evaluated by administering it to a sample student population.

After the administration, the exam was scored using the grading rubric created and the results of using the grading rubric were studied. For multiple graders, some inconsistencies in grading were uncovered. Some of these inconsistencies were related to simple human error, while others necessitated changes or clarifications in the grading rubric and re-grading of some questions. Recommendations for the grading of this exam based on the process used in this study include anonymous grading and the use of multiple graders for both the subjective and non-subjective questions on the exam to maximize consistency with the established grading guidelines.

9.2 Discussion of Analysis of Exam

Following the administration and grading, the scores of the students were statistically analyzed to gather information about the validity and reliability of the instrument as well as to look for potential exam biases.

Face and content validity information was gathered by asking a panel of five experts in the field to analyze the appropriateness of this exam as an assessment of introductory computer science. These experts gave numerous suggestions for improvements of the exam as well as ways to decrease the number of questions by eliminating duplication. It is important to note that even though the information gathered from these experts is used for establishing face and content validity of the exam, this analysis was completed and changes to the exam implemented before administration to the sample population.

Despite initial fears expressed by the reviewers of the exam being too long, the students finished, on average, in an acceptable time frame. Furthermore, statistical analysis showed that the time students used on the exam did not have a statistically significant correlation with their performance.

After the exams were scored, the scores of the students underwent various statistical tests. The first was to look for reliability of the instrument. Cronbach's alpha was computed for the instrument, revealing an acceptable reliability coefficient of .94. These results indicate that the exam is internally consistent.

Using the demographic data that was collected from the students while administering the exam, preliminary investigations were undertaken to look for exam bias based on gender, age, major, and previous programming experience. The results of these analyses were promising, because no biases were found in the data gathered and analyzed so far. There was no statistically significant difference in scores between the two genders, between freshmen and non-freshmen students, or between intended majors and non-majors.

While previous programming experience before taking the CS1-CS2 sequence did not make a difference in the scores, a borderline statistically significant result (p=0.05, but not less than 0.05) was found for students who had experience with C-derived languages before taking the CS1-CS2 course and this exam. Because of the borderline nature of the result, further research should involve the collection of more data for analysis to see if there definitely is a difference that can be detected between those with previous experience with C-derived languages and those who have no previous experience.

An attempt was made to show the criterion validity of the exam by using overall course grades in the CS1 and CS2 courses. The results of this analysis revealed that the exam score and course grades in CS1, CS2, and the average of the CS1-CS2 sequence positively correlated with one another. The original CS2 grades would have included the exam score within their computation. CS2 scores and the CS1-CS2 average were recomputed with the exam score removed, and a positive correlation was still found. This result strengthens the validity of the exam for use as an assessment of the CS1-CS2 sequence.

Overall, the work of this dissertation achieved its goals of creating an assessment for the programming-first approaches to the introductory curriculum that has been shown reliable. Also, work on the face validity, content validity, and criterion validity has been undertaken, with all results pointing to an overall validity of the instrument for its task.

9.2.1 Students who chose not to participate in study

Since participation in the study was voluntary, there were students enrolled in the CS2 course (CSE 116) when the exam was administered that chose not to be included in the study. It is important to look at this group of students in general to see if the sample used in the study was indicative of the general population of students enrolled in the CSE 116 course.

The information for the courses is available to instructors through their university class lists. From the data available to the instructors, I was able to obtain anonymous information about total enrollments, gender, year in school, declared major, and final student letter grades assigned in the course for the entire CSE 116 population across the two semesters the exam was administered. This information was obtained by the instructors from the information provided to them on their official university class lists. Therefore, some of the demographic information obtained in the study was not available for students that did not participate in the study. However, with the information that is available about the enrollment overall, we can see a picture of any potential differences between those students who participated in the study and those students who did not.

9.2.1.1 Overall enrollment

The total enrollment of students for the two semesters of the study was 135 students. Of this original 135, 14 students elected to resign the course before the end of the semester and receive a grade of “R” on their transcript. This left a potential candidate pool of 121 total students for participation in the study. Of these 121, 100 students elected to participate, leaving only 21 students not participating in the study. Therefore, 83% of the available student population was analyzed by the study.

9.2.1.2 Gender

Of the 121 students, 110 (91%) were men and 11 (9%) were women. Ninety (90%) men and 10 (10%) women chose to participate in the study. Therefore, the percentage of men and women in the study population and the regular population were similar.

9.2.1.3 Year in School

Of the 121 students, 45 (37%) were categorized by the university as freshmen, and 76 (63%) were categorized as non-freshmen. Fifty-two (52.5%) freshmen and 47 (47.5%) non-freshmen agreed to participate in the study. These numbers seem to have an obvious conflict, because more freshmen are enrolled in my study than are reported by the university records. However, this discrepancy can be explained by the fact that the university considers class year by credits earned, not by how many years a student has been enrolled at the university. Therefore, many “freshmen” entering the university actually have accumulated university credit before they even take one day of classes at the university. In my study, the year in school was a self-reported variable. Therefore, students in their first year of study at university commonly identify themselves as freshmen, even if the university records indicate otherwise.

9.2.1.4 Declared Major

Of the 121 students, 37 (31%) were declared computer engineering majors, 46 (38%) were declared computer science majors, and 38 (31%) were declared to be some other major (including undecided). Twenty-six (26%) computer engineering majors, 50 (50.5%) computer science majors, and 23 (23.5%) other majors agreed to participate in the study.

These numbers are also subject to the same self-report-versus-university-records problems as the year in school. At the University at Buffalo, students do not have to be formally accepted to a major until the end of their second year of study. Therefore, many students intend to pursue a particular major, but, since they have not been formally accepted to that major yet, the university does not recognize them as majoring in that subject. Also, a student may have decided to pursue a major and not yet informed the University of their intent at the time of the creation of the class lists. Nonetheless, the numbers for students overall in the class and the students enrolled in the study are roughly the same.

9.2.1.5 Grade in course

Table 9-1 gives a breakdown of course grades earned by the 121 students enrolled in CSE 116 with a percentage breakdown of the grade within the class, as well as the number of those students who participated in the study and a percentage of how many students who earned each letter grade participated in the study.

Recorded Course Grade	Number of students	Percentage of students overall earning grade	Number of study participants	Percentage of students who participated in study
A	18	15%	17	94%
A-	18	15%	16	89%
B+	25	21%	22	88%
B	19	16%	16	84%
B-	8	7%	7	88%
C+	10	8%	8	80%
C	7	6%	6	86%
C-	3	2%	2	67%
D	3	2%	2	67%
F	10	8%	4	40%

Table 9-1: Course grade breakdown for all CSE 116 students

Table 9-2 gives a breakdown of course grades earned by the 21 students who elected not to participate in the study.

Recorded Course Grade	Number of students who earned that grade
A	1
A-	2
B+	3
B	3
B-	1
C+	2
C	1
C-	1
D	1
F	6

Table 9-2: Course grade breakdown for CSE 116 students who elected not to particpate in the study

Those students who elected not to participate in the study were fairly spread throughout the grade spectrum. The only group of students who did not seem to particpate at the same rate as the other students were those students who received an overall grade of F in the course. Only 40% of the F-students participated in the study, while other groups of students had participation rates that ranged from 67% to 94%.

Upon further investigation, it turns out that only 116 students total took the exam across the two administrations, which means that 5 students did not attend the final exam at all and therefore could not elect to participate in the study. It is hypothesized that the students who did not take the final exam were among the students who earned an overall grade of F in the course, because, with the weighting that the final exam was given for the course, it would be almost impossible for a student to pass the course without taking the final exam.

Therefore, between the 4 students who did particpate and the 5 who did not take the exam, it appears that only 1 student who earned an F in the course did not participate in the study, keeping the participation rate of students who earned an F similar to those students who earned other letter grades.

9.2.1.6 Conclusions about Participants

Comparing the data gathered from university records to the data collected about the participants in the study, I feel confident in saying that the sample represented in the study is not significantly different from the overall possible pool of students enrolled in CSE 116. There were relatively very few students who did not participate. There was virtually no difference in gender, year in school, major, or overall course performance as indicated by overall course grade. Therefore, I feel confident to conclude that the results of the study were not skewed in any way by the self-selection of participants in the study.

9.3 Future Work

This section presents directions for future work on this assessment instrument.

9.3.1 Additional Student Data

Although the data collected for this dissertation were adequate for analysis of the exam's reliability and helped to make preliminary assertions about the exam's validity, a future goal is to collect even more data about the exam by administering the test to more students. With additional data, more detailed item analysis can be performed. With this type of analysis, trends might be found in the exam to suggest that certain questions could be eliminated without affecting overall student outcomes.

These types of analyses could also identify a group of questions that are the predictors for a student's score on the exam. If these questions were identified, it might be possible to edit the exam so it contains just those questions, or equivalent derivatives of those questions, which would make the exam shorter. Alternatively, some questions might be replaced by items that would improve the reliability and validity of the exam.

Additionally, more demographic data could be collected to continue the investigation into test bias. Most notably, the number of females who took the test was large enough to perform statistical analysis; it is not enough to satisfy the question of gender bias on the exam. Other areas of the demographics can also continue to be explored, including age, major, and additional programming language experience.

9.3.2 Continuation of Predictors Research

This work was inspired by work in the area of predictors of success in CS1 (Ventura, 2003). Ventura gathered various pieces of demographic information as well as administered a test of critical thinking ability to find the best predictors for success in an objects-first CS1. His measure for success included course grades on various components and was never validated. Our assessment instrument has been shown reliable and information has been presented for validity. This instrument could be used as the measure for success for the predictors research. However, this assessment is a measure of success for CS1 and CS2. The initial work on predictors would need to be revisited to include predictors for CS2 as well as CS1. Only then could the assessment that I created be used for further examination of this work.

9.3.3 Testing of curricular changes

One of the initial goals in creating this assessment was to have the ability to use it to measure the effectiveness of curricular changes. Since the exam is based on the content for the introductory sequence as outlined by CC2001, it can provide a baseline indicator of what information the students should know after completing the CS1 and CS2 sequence.

Therefore, if this test was administered at the end of a year and then curricular changes were made for the next year, one would expect that the scores on the exams would not be statistically significantly different, or if they were different, that an improvement of overall the scores would be viewed as an affirmation of the value of the curricular changes.

A decrease in the overall scores could point to a failure of the new curricular direction. However, that cannot be the immediate conclusion based on this type of result. There could be many other factors impacting performance of a group of students on any exam, and they should be explored. It would be best if scores on this exam could be observed both before the curricular change and after the change to give an adequate picture of the student performance as it relates to the curricular change.

9.3.4 Trends and Longitudinal Research

Another way that this assessment could be used is simply to track general trends of student performance irrespective of curricular innovation. General performance of students across years, semesters, and instructors would be possible if each group of students were given this assessment at the end of CS2.

Additionally, longitudinal tracking would be possible of students who took the assessment. Students who took the assessment could be tracked throughout their careers to look for any predictive value of this assessment in their future success in their studies of computing.

9.3.5 Multiple Languages and Multiple Forms

Initially, this assessment was conceived as being language-independent. However, early in the work on the instrument and because of the instrument's focus on programming-first approaches to the curriculum, it was decided that a language needed to be used for code examples and for student answers. Therefore, a natural extension of this work is to modify the exam for use in CS1-CS2 sequences that do not use Java as their main language of instruction. A change to another language would involve the modification of questions and code examples used within the test as well as analysis of the grading rubric to ensure that none of the answers to questions would be affected by the change in language.

To further reinforce the reliability data collected, equivalent forms of the exam should also be created and administered to students. Ideally, these parallel forms should be created for the Java version of the exam as well as the other language versions. In the effort to create parallel forms, it would be useful to create an exam template that could be easily changed to create new forms of the exam quickly and for the same administration of the exam.

Another change to the exam that could be considered an alternate form is to create a better alternative for the multiple choice questions on the exam, either with Scantron (or similar) technology, or a way to put questions on a computer-delivered testing system to help automate grading of those questions.

9.3.6 Multi-Institutional Analysis

To further reinforce the reliability and validity data collected, the exam needs to move outside of the walls of the University at Buffalo. This would allow the creation of tests in different languages as talked about previously, but also to allow for introductory sequences taught in the different approaches to use the exam.

Testing students who have been taught with an imperative-first or functional-first approach would provide further evidence of the validity of the instrument across the different programming-first approaches.

Furthermore, having other instructors administer and grade the exam will allow for further testing of the administration processes and grading rubric for the exam.

Another advantage of porting the instrument to other institutions comes simply from the organization of the institution. Our department is a computer science and engineering department, servicing both computer science and computer engineering majors. Other departments might simply only serve computer science majors, or a mix of computer science and information technology majors in their CS1-CS2 sequence. Analysis of data from these types of departments would allow further conclusions to be drawn about exam bias for different majors.

9.3.7 Updates for Future Curricula

As I tell my students, computing and computer science is a field that is constantly changing. There is no way to tell where the field will be or what will be taught in CS1-CS2 five or ten years from now. Simply looking at the curriculum reports of the past can show us how the field has changed. Therefore, this exam can never be static in the face of curricular change. When new curriculum documents are published and schools begin to adopt the new recommendations, this assessment for the CS1-CS2 sequence needs to adapt and change accordingly.

In the face of change, it should be noted that the process used to create this assessment can be repeated when new curriculum documents are produced. Changes to the curriculum would most certainly necessitate updates for the exam and questions contained within it, but the practice of finding a common intersection could easily be repeated.

References

1. ACM/IEEE-CS Joint Curriculum Task Force Curricula. Computing Curricula 1991, [1991 cited 2003]. Available from http://www.acm.org/education/curr91/homepage.html.

2. ACM/IEEE-CS Task Force on Computing Curricula 2001. Final Report of the Joint ACM/IEEE-CS Task Force on Computing Curricula 2001 for Computer Science [2001 cited 2007]. Available from http://acm.org/education/curric_vols/cc2001.pdf.

3. Alphonce, Carl, and Philip R Ventura. "Object-Orientation in CS1-CS2 by Design." In ITiCSE 2002. Aarhus, Denmark, 2002.

4. AP. AP Course Descriptions, 2003 [cited November 13, 2003]. Available from http://apcentral.collegeboard.com/repository/ap03_cd_computer_scie_4315.pdf

5. AP CS A Reliability. Advanced Placements CS A Exam Reliability, 2004 [cited October 15 2004]. Available from http://apcentral.collegeboard.com/article/0,3045,152-167-0-2021,00.htm#reliability.

6. AP CS A Test Description. 2004 [cited November 7 2004]. Available from http://apcentral.collegeboard.com/members/article/1,3046,152-171-0-22913,00.html.

7. AP CS AB Reliability. Advanced Placements CS AB Exam Reliability, 2004 [cited October 15 2004]. Available from http://apcentral.collegeboard.com/article/0,3045,152-167-0-2021,00.htm#reliability.

8. AP CS AB Test Description. 2004 [cited November 7 2004]. Available from http://apcentral.collegeboard.com/members/article/1,3046,152-171-0-22912,00.html.

9. AP CS Development Committee. 2004 [cited November 7 2004]. Available from http://apcentral.collegeboard.com/members/article/1,3046,152-167-0-2030,00.html.

10. AP Exam Grading. 2004 [cited November 7 2004]. Available from http://www.apcentral.collegeboard.com/article/0,3045,152-167-0-1994,00.html.

11. AP Validity. Exam Validation, 2004 [cited October 21 2004]. Available from http://apcentral.collegeboard.com/article/0,345,152-167-0-2052-00.html.

12. Aron, Arthur, and Elaine N. Aron. Statistics for the Behavioral and Social Sciences: A Brief Course. Second ed. Upper Saddle River, NJ: Prentice Hall, 2002.

13. Astrachan, Owen, and David Reed. "AAA and Cs1." In SIGCSE 1995. Nashville, TN, 1995.

14. Bond, Ian. Ian Bond - Assignment 2, 2004 [cited January 27, 2005]. Available from http://www.massey.ac.nz.~iabond/159234/assignment2.pdf.

15. Cantwell Wilson, Brenda, and Sharon Shrock. "Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors." In SIGCSE 2001. Charlotte, NC, 2001.

16. Committee on Computer Science Curriculum. "Curriculum 68: Recommendations for the Undergraduate Program in Computer Science." Communications of the ACM 11, no. 3 (1968): 151-97.

17. Committee on Computer Science Curriculum. "Curriculum 78: Recommendations for the Undergraduate Program in Computer Science." Communications of the ACM 22, no. 3 (1978): 147-66.

18. Cooper, Steven, Wanda Dann, and Randy Pausch. "Teaching Objects-First in Introductory Computer Science." In 34th SIGCSE technical symposium on Computer Science Education. Reno, Nevada, 2003.

19. Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. Introduction to Algorithms. Cambridge, Massachusetts: The MIT Press, 2000.

20. CS Content Rep Study. Content Representativeness Study GRE, 2002 [cited December 3 2004]. Available from http://ftp.ets.org/pub/gre/conrepresults.pdf.

21. Culwin, Fintan. "Object Imperatives!" In SIGCSE 1999. New Orleans, LA, 1999.

22. Dade Computer Programming I Description. Computer Programming I. Pdf, 2001 [cited March 16, 2007 2007]. Available from http://portal.dadeschools.net/cbc/Volume%20II/Instructional%20Technology/Senior%20High/Grade%2010/Computer%20Programming%20I.pdf.

23. Daly, Charlie, and John Waldron. "Assessing the Assessment of Programming Ability." In SIGCSE 2004. Norfolk, VA, 2004.

24. Dann, Wanda, Steven Cooper, and Randy Pausch. Learning to Program with Alice. Upper Saddle River, NJ: Prentice Hall, 2006.

25. Decker, Adrienne. "A Tale of Two Paradigms." Journal of Computing Sciences in Colleges 19, no. 2 (2003): 238-46.

26. Dietel, H. M., and P.J. Dietel. C: How to Program. Fourth Edition ed. Upper Saddle River, New Jersey: Prentice Hall Inc., 2004.

27. Dietel, H. M., and P.J. Dietel. C. C++: How to Program. Fifth Edition ed. Upper Saddle River, New Jersey: Prentice Hall Inc., 2005.

28. Dietel, H. M., and P.J. Dietel. C. Java: How to Program. Sixth Edition ed. Upper Saddle River, New Jersey: Prentice Hall Inc., 2005.

29. ETS. ETS Major Field Test, 2003 [cited 2003]. Available from http://ftp.ets.org/pub/corp/hea/ContComSci2.pdf

30. ETS Major Field Test Description. 2004 [cited November 7 2004]. Available from http://ftp.ets.org/pub/corp/hea/ContComSci2.pdf.

31. ETS Reliability. ETS Major Field Test Computer Science Reliability, 2004 [cited October 15 2004]. Available from http://ftp.ets.org/pub/corp/hea/reliability03.pdf.

32. Evans, Gerald E., and Mark G. Simkin. "What Best Predicts Computer Proficiency." Communications of the ACM 32, no. 11 (1989): 1322-27.

33. Evans, M.D. "A New Emphasis & Pedagogy for a CS1 Course." inroads - The SIGCSE Bulletin 28, no. 3 (1996): 12 - 16.

34. Fincher, Sally. "What Are We Doing When We Teach Programming?" In 29th ASEE/IEEE Frontiers in Education Conference. San Juan, Puerto Rico, 1999.

35. GRE Score Use. Guide to the Use of Scores, 2003 [cited 2003]. Available from http://www.ets.org/Media/Tests/GRE/pdf/994994.pdf

36. GRE Subject Test Computer Science Description. 2004 [cited November 8 2004]. Available from http://www.gre.org/subdesc.html#compsci.

37. GRE Subject Test General Description. 2004 [cited November 8 2004]. Available from http://www.gre.org/pbstest.html.

38. Guzdial, Mark, and Barbara Ericson. Introduction to Computing and Programming with Java: A Multimedia Approach. Upper Saddle River, NJ: Prentice Hall, 2006.

39. Guzdial, Mark, and Elliot Soloway. "Teaching the Nintendo Generation to Program." Communications of the ACM 45, no. 4 (2002): 17-21.

40. Hagan, Dianne, and Selby Markham. "Does It Help To Have Some Programming Experience before Beginning a Computing Degree Program?" In ITiCSE 2000. Helsinki, Finland, 2000.

41. Hanly, Jeri R., and Elliot B. Koffman. Problem Solving and Program Design in C. Fourth Edition ed. Boston, Massachusetts: Addison-Wesley, 2003.

42. Harvey, Brian, and Matthew Wright. Simply Scheme: Introducing Computer Science. Second Edition ed. Cambridge, Massachusetts: The MIT Press, 1999.

43. Hayes, Brian. "The Semicolon Wars." American Scientist 94, no. July-August (2006): 299-303.

44. Horstmann, Cay. Big Java. Second ed. Hoboken, New Jersey: John Wiley & Sons, Inc., 2006.

45. Joint Task Force on Computing Curricula. Curricula Recommendations, 2001 [cited 2007]. Available from http://www.acm.org/education/curricula.html.

46. Kaplan, R M, and D P Saccuzzo. Psychological Testing: Principles, Applications and Issues. Belmont, California: Wadsworth/Thomson Learning, 2001.

47. Kolling, Michael, and David J. Barnes. "Enhancing Apprentice-Based Learning of Java." In SIGCSE 2004. Norfolk, VA, 2004.

48. Kuncel, N. R., S. A. Hezlett, and D. S. Ones. "A Comprehensive Meta-Analysis of the Predictive Validity of the Graduate Record Examinations: Implications for Graduate Student Selection and Performance." Psychological Bulletin 127, no. 1 (2001): 162-81.

49. Kurtz, Barry L. "Investigating the Relationship between the Development of Abstract Reasoning and Performance in an Introductory Programming Class." In SIGCSE 1980. Kansas City, MO, 1980.

50. Leeper, R. R., and J. L. Silver. "Predicting Success in a First Programming Course." In SIGCSE 1982. Indianapolis, IN, 1982.

51. Lewis, John, and William Loftus. Java Software Solutions: Foundations of Program Design. Fifth Edition ed. Boston, Massachusetts: Addison-Wesley, 2007.

52. Lister, Raymond, and John Leaney. "Introductory Programming, Criterion-Referencing, and Bloom." In SIGCSE 2003. Reno, NV, 2003.

53. Major Field Test. Major Field Tests: Description of Test Reports, 2003 [cited 2003]. Available from http://ftp.ets.org/pub/corp/hea/ContComSci2.pdf

54. Major Field Test. Reliability Coefficients and Standard Error of Measurements, 2002 [cited 2003]. Available from http://ftp.ets.org/pub/corp/hea/reliability03.pdf.

55. Major Field Test Content. Major Field Tests Computer Science Description, 2003 [cited 2003]. Available from http://ftp.ets.org/pub/corp/hea/ContComSci2.pdf

56. Marion, William. "CS1: What Should We Be Teaching?" inroads - The SIGCSE Bulletin 31, no. 4 (1999): 35-38.

57. Marshall, J C, and L W Hales. Essentials of Testing. Reading, Massachusetts: Addison-Wesley Publishing Company, 1972.

58. Mazlack, Lawrence J. "Identifying Potential to Acquire Programming Skill." Communications of the ACM 23, no. 1 (1980): 14-17.

59. McCauley, Renee. "Rubrics as Assessment Guides." inroads - The SIGCSE Bulletin 35, no. 4 (2003): 17 - 18.

60. McCracken, Michael, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. "A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year Cs Students: Report by the ITiCSE 2001 Working Group on Assessment of Programming Skills of First-Year CS Students." inroads - The SIGCSE Bulletin 33, no. 4 (2002): 1-16.

61. Morgan, R., and L. Ramist. Advanced Placement Students in College: An Investigation of Course Grades at 21 Colleges ETS, 1998 [cited November 8 2004]. Available from http://apcentral.collegeboard.com/repository/ap01.pdf.in_7926.pdf.

62. Moskal, Barbara, Keith Miller, and L. A. Smith King. "Grading Essays in Computer Ethics: Rubrics Considered Helpful." In SIGCSE 2002. Covington, KY, 2002.

63. Nagappan, Nachiappan, Laurie Williams, Miriam Ferzli, Eric Wiebe, Kai Yang, Carol Miller, and Suzanne Balik. "Improving the CS1 Experience with Pair Programming." In SIGCSE 2003. Reno, Nevada, 2003.

64. Neebel, Danial J., and Brenda Tuomi Litka. "Objective Based Assessment in a First Programming Course." In 32nd ASEE/IEEE Frontiers in Education Conference. Boston, MA, 2002.

65. Owens, Barbara Boucher, Robert D. Cupper, Stuart Hirshfield, Walter Potter, and Richard Salter. "New Models for the CS1 Course: What Are They and Are They Leading to the Same Place?" In SIGCSE 1994. Phoenix, AZ, 1994.

66. Parker, Peter E., Paul D. Fleming, Steve Beyerlein, Dan Apple, and Karl Krumsieg. "Differentiating Assessment from Evaluation as Continuous Improvement Tools." In 31st ASEE/IEEE Frontiers in Education Conference. Reno, NV, 2001.

67. Pattis, Richard. "The "Procedures Early" Approach in CS1: A Heresy." In SIGCSE 1993. Indianapolis, IN, 1993.

68. Proulx, Viera, Richard Rasala, and Harriet Fell. "Foundations of Computer Science: What Are They and How Do We Teach Them?" In 1st conference on integrating technology into computer science education. Barcelona, Spain, 1996.

69. Ramalingam, Vennila, and Susan Wiedenbeck. "Development and Validation of Scores on a Computer Programming Self-Efficacy Scale and Analyses of Novice Programmer Self-Efficacy." Journal of Educational Computing Research 19, no. 4 (1998): 367-81.

70. Rapaport, William J. How I Grade (the Triage Theory of Grading), 2006 [cited July 11 2006]. Available from http://www.cse.buffalo.edu/faculty/rapaport/howigrade.html.

71. Ravid, R. Practical Statistics for Educators. Lanham: University Press of America, 1994.

72. Reges, Stuart. "Back to Basics in CS1 and CS2." In SIGCSE 2006. Houston, TX, 2006.

73. Reges, Stuart. "Conservatively Radical Java in CS1." In SIGCSE 2000. Austin, TX, 2000.

74. Reges, Stuart. "Resolved: Objects Early Has Failed." In SIGCSE 2005. St. Louis, MO, 2005.

75. Rumbaugh, James, Ivar Jacobson, and Grady Booch. The Unified Modeling Language Reference Manual. Boston: Addison-Wesley, 1999.

76. SAT Program Handbook. 2006 [cited 2007]. Available from http://www.collegeboard.com/prod_downloads/highered/ra/sat/2006-07-SAT-Program-Handbook.pdf

77. Savitch, Walter. Absolute Java. Second ed. Boston, Massachusetts: Addison-Wesley, 2006.

78. Sethi, Ravi. Programming Languages: Concepts and Constructs. Second ed. Reading, Massachusetts: Addison-Wesley, 1996.

79. Stein, Lynn Andrea. "Interactive Programming: Revolutionizing Introductory Computer Science." ACM Computing Surveys 28, no. 4es (1996).

80. Ventura, Philip R. "On the Origins of Programmers: Identifying Predictors of Success for an Objects-First CS1." University at Buffalo, SUNY, 2003.

81. Walker, Henry M. "Notes on Grading." inroads - The SIGCSE Bulletin 32, no. 2 (2000): 18-19.

82. Whitfield, Deborah. "From University Wide Outcomes to Course Embedded Assessment of CS1." Journal of Computing Sciences in Colleges 18, no. 5 (2003): 210-20.

Appendix A
Exam Questions

The actual exam begins on the next page to preserve the formatting of the original.

Course Number **Course Title** **Semester Year**

Final Exam

Name (**Print**): ______________________________

Signature: ______________________________

Person#: ____________________ Exam Number: ____________________

<u>Exam Instructions:</u>

You have been assigned an exam number for this exam. The only place you should put your name is on this first page of the test. You should not put your name, UBIT name, person number, or social security number on any other page of this exam, answer sheet, or demographic questionnaire.

Feel free to write any scratch work in the exam booklet. However, the only answers that will be scored are those that you write in pen on the ***<u>answer sheet</u>***. If at any time, you need extra paper for your work, please ask the exam administrators.

You are not allowed to refer to any outside materials (notes, books, reference materials, your neighbor) while completing this exam.

The exam was designed to test information that students should know after the first year of computer science/programming instruction, which corresponds to CSE 115 and 116 at UB. Because of the general nature of the exam, there may be questions that you are unable to answer. You should leave those answers blank.

Some of the questions on this test present multiple choices for answers. Some of those questions instruct you to pick as many answers as are appropriate. Be sure to read the question to determine if you should indicate more than one answer for a particular question. If the question requires only one answer and you have narrowed the choices down to two, you should make an educated guess about the answer for the question.

If you have a question during the exam, please raise your hand and one of the exam administrators will come to you and answer your question. Please do not walk to an exam administrator with a question. The only time you should leave your seat is when you have completed the exam and are ready to hand it in.

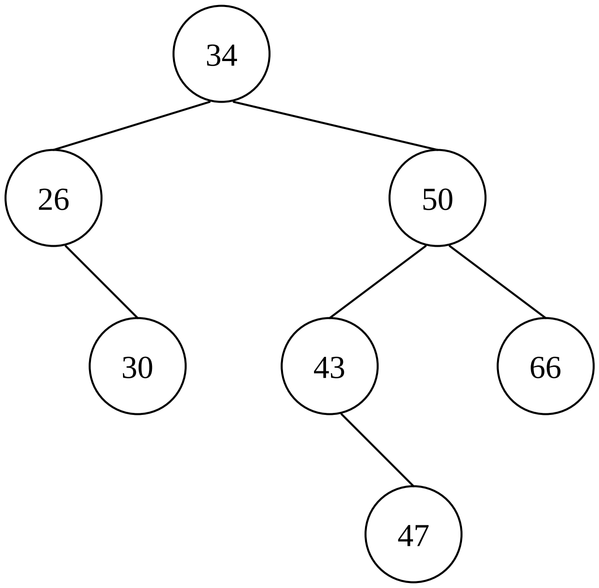
You will have three (3) hours to complete this exam. At the end of the three hours, you are required to hand in your paper.

When you have completed the exam, you will take your demographic questionnaire, exam booklet, and answer sheet to an exam administrator.

1) Draw the binary search tree which results when the following items are inserted, in the order given into an initially empty BST. [8 points]

Elements: 62, 55, 37, 106, 202

Given the following BST, answer questions 2 – 3.



2) You call search (find) and are looking for the number 32. List of nodes that are visited while you are determining that 32 is not in the BST. [3 points]

3) You want to delete 34 (the root) from this tree. Show one possible valid binary search trees that could result from deleting the root. [8 points]

For question 4, consider the following code segment:

```
java.util.HashMap<Integer, String> mapOne =
                                 new java.util.HashMap<Integer,
                                 String>();

java.util.HashMap mapTwo = new java.util.HashMap();

mapOne.put(1, “First name”);
mapTwo.put(1, “First name”);

String s1 = mapOne.get(1);
String s2 = mapTwo.get(1);
```

4) Which of the two assignments of “First name” to a String variable does not work correctly and why? (Circle only one answer). [1 point]

a. Assignment to `s1` does not work because `get()` returns an `Object`, not a `String`.
b. Assignment to `s1` does not work because `s1` is not a `String`.
c. Assignment to `s1` does not work because `HashMap`s cannot use `Integers` as keys.
d. Assignment to `s2` does not work because `get()` returns an `Object`, not a `String`.
e. Assignment to `s2` does not work because `s2` is not a `String`.
f. Assignment to `s2` does not work because `HashMaps` cannot use `Integers` as keys.
g. Neither assignment works because `get()` returns an `Object`, not a `String`.
h. Neither assignment works because neither `s1` nor `s2` is a `String`.
i. Neither assignment works because `HashMaps` cannot use `Integers` as keys.

6) Write the body of the following method named `changeColors`. The method takes as a parameter, a `java.util.Collection` of `java.awt.Color`s. The `changeColors` method should call the method `setColor(java.awt.Color),` which is inherited from `javax.swing.JPanel`, for each color in the `Collection` so that the user sees a changing background color for the panel on their program. You can assume that this method appears in a class that extends `JPanel` so you can simply call the `setColor` method from within this method. You must use an iterator/for-each loop in your solution to this question to receive full credit. [8 points]

```
void changeColors(java.util.Collection<java.awt.Color>
colorsForBackground) {





}
```

Assume you have created the following array in a program:

```
            int[] holder = new int[50];
```

Use this information to answer questions 6 – 9.

6) What is the maximum number of elements that can be stored by `holder`? [1 point]

7) At which index would the first integer in `holder` be stored? [1 point]

8) At which index would the last integer in `holder` be stored? [1 point]

9) As you are using the array in your program, you find out that you need to store more than the maximum number of elements you listed in question 8. You do not know how many more elements you will be storing, just that you need more space in your array. You are asked to write a method, `needMoreSpace` that takes in an array and performs the necessary operations to return a larger array with the same elements as the original, but with space to store additional elements. Since you don't know how many elements you will eventually need to store, you should write the method body so that it could be called at a later time if the array needs to get bigger again. [8 points]

```
public int[] needMoreSpace(int [] originalArray) {









}
```

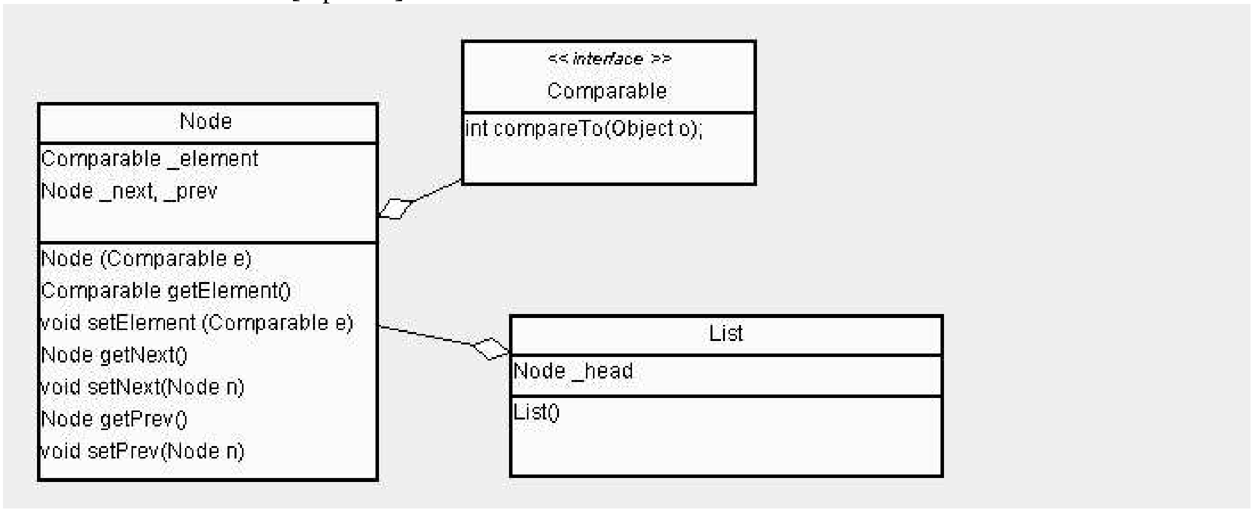
10) Fill in the method below so that it creates and returns an array of size `size` and populates the array with elements each of whose values is the square of the index at which the element is stored. For example, at array index 3, the value 9 should be stored. [8 points]

```
public int[] arrayOfSquares (int size) {









}
```

11) Fill in the method below so that it returns `true` if the value passed in as a parameter is contained inside the matrix and returns false otherwise. [8 points]

```
public boolean contains(double[][] matrix, double value) {









}
```

12) Given the following UML diagram for a doubly linked list, fill in the method `delete` below, which is a method in the List class and takes an element to be deleted and returns the deleted element when finished. [8 points]



Notes about the classes in the diagram:

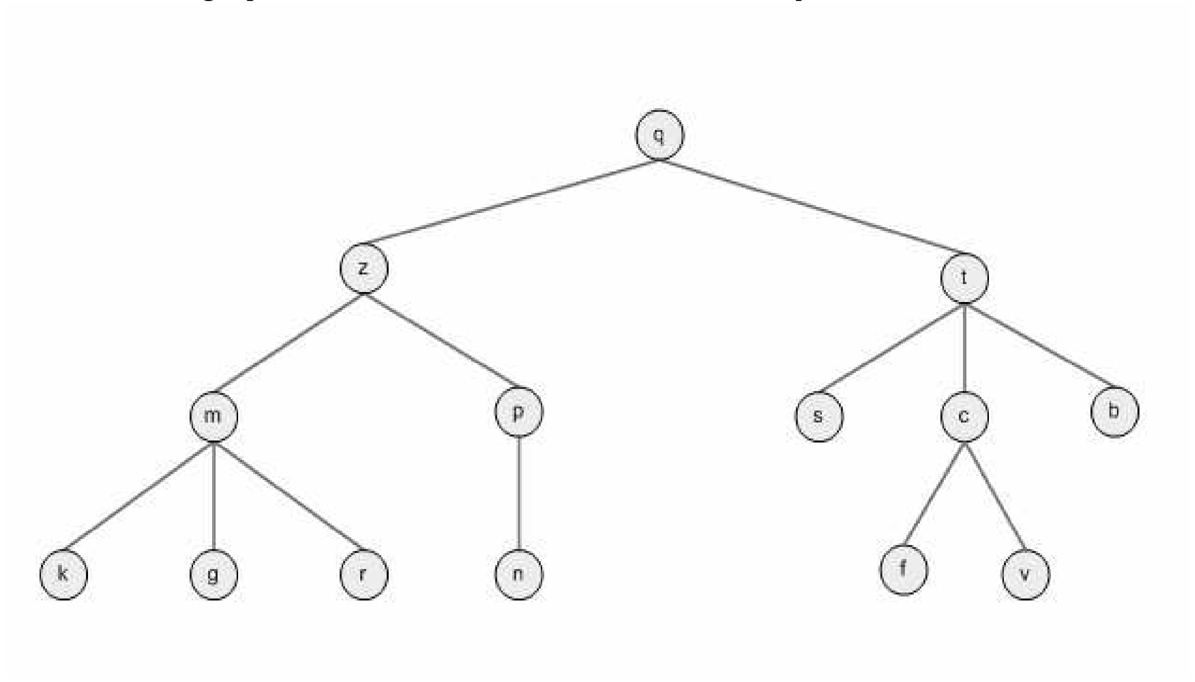
- `Node`'s constructor sets the value of `_element` to the value passed in and sets the value of `_next` and `_prev` to `null`. The other elements are simple accessors and mutators for `_element`, `_node`, and `_prev`.

- `Node` holds an element that implements the interface `Comparable`. Recall that a class that implements this interface has a method named `compareTo` that takes in an `Object obj`, and returns a positive number if `this > obj`, the value 0 (zero) if the two are the same, or a negative value if `this < obj`.

- `List`'s constructor simply sets the value of `_head` to `null`.

```
public Comparable delete(Comparable element) {







}
```

Use the following representation of a tree data structure to answer questions 13 - 18.



13) What is the value stored in the node that is the root? [1 point]

14) Give the value stored in one of the leaves of this structure. [1 point]

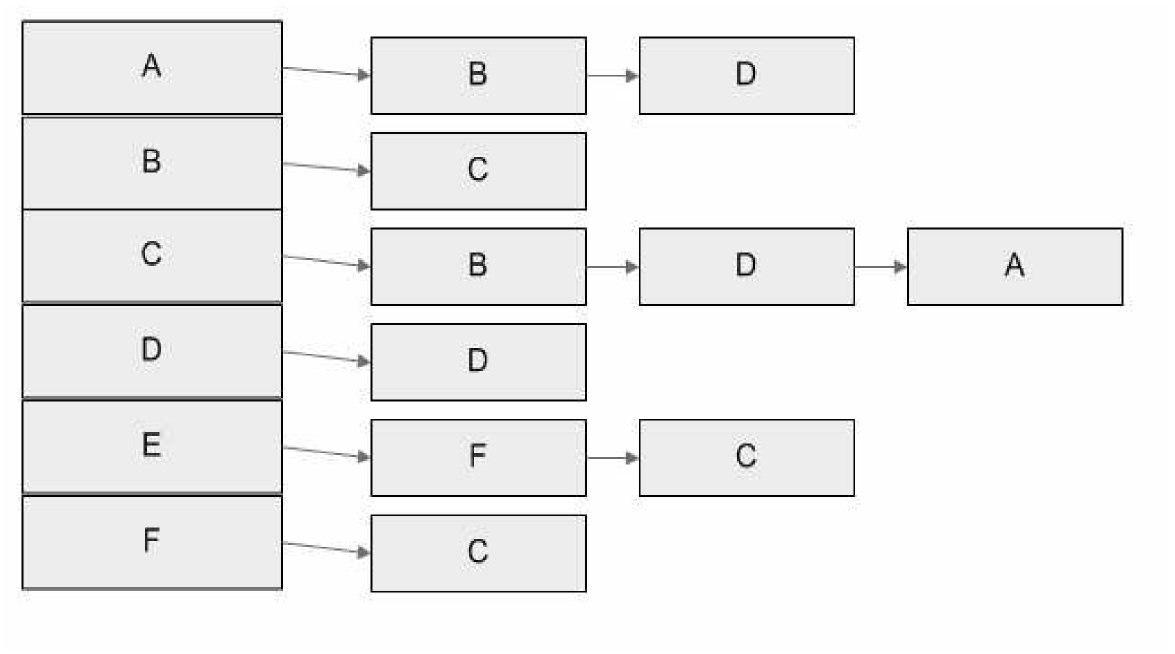
15) What is the height of a tree that just contains a root and no other nodes? [1 point]

16) What is the height of this structure? [1 point]

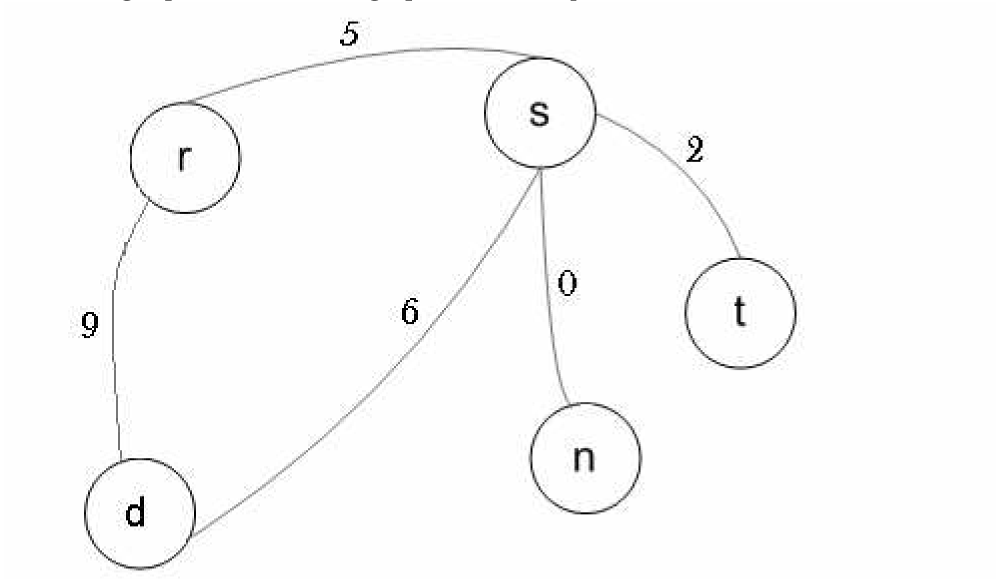
17) Give the value stored in the node that is the parent of n. [1 point]

18) Give the values stored in all the children of m. [3 points]

19) Given the following adjacency list for a directed graph, draw the graph structure it represents. [8 points]



Use the following representation of a graph to answer questions 20 and 21.



20) Circle the letters of all of the words that accurately describe the graph above. [4.5 points]
   a. directed
   b. undirected
   c. weighted
   d. unweighted
   e. simple
   f. complete
   g. acyclic
   h. isomorphic
   i. rooted

21) Circle the letters corresponding to all the pairs of nodes given that are adjacent in the above graph. [4 points]
   a. r and s
   b. t and n
   c. d and s
   d. n and d

22) If a data structure is linear in nature (list, vector, etc), which implementation would perform better asymptotically in a linear search. Circle one of the implementations listed: [1 point]
   a. array-based
   b. linked list-based
   c. neither – they would both perform the same on the linear search.

23) Referring to your knowledge of data structures and inheritance, why is it inappropriate for a java.util.Stack to be a subclass of java.util.Vector? [8 points]

From the list of data structures given, choose the best answer or answers for questions 24 – 31. If there is no appropriate answer, write “None”. If you feel that more than one answer is appropriate, list all appropriate answers. It is possible that some answers from the box will not be used.

Linked List	Array
Graph	Stack
Tree	Queue
Hash Map	

24) Structure that associates a key with a value. [6 points]

25) Structure whose insertion/removal strategy can be defined as LIFO. [6 points]

26) Structure whose insertion/removal strategy can be defined as FIFO. [6 points]

27) Structure that is non-linear. [6 points]

28) Structure whose elements are always stored in a contiguous block of memory. [6 points]

29) You are creating software for a call center that does technical support. Technicians are supposed to answer calls in the order they are received. What structure would be best for keeping track of which call should be answered next? [6 points]

30) Your company has decided to create a program to help cell-phone customers everywhere. It is an on-line program that allows the user to type a person's name and will return a list of all cell phone numbers registered to them. You are asked to recommend a structure to hold onto the information. Which structure would you recommend? [6 points]

31) You are working for a brand new on-line mapping company. This company needs to maintain information about locations and roads that connect them so that it can tell customers about various routes between locations. What type of structure would be best for them to use to store their information? [6 points]

In questions 32–37 you are given an algorithm for sorting or searching. You are to circle any and all valid big-oh bounds on the worst-case performance of each of the algorithms listed.

32) Binary Search [6 points]

a. $O(1)$ b. $O(\log n)$ c. $O(n)$ d. $O(n \log n)$ e. $O(n^2)$ f. $O(2^n)$

33) Linear Search [6 points]

a. $O(1)$ b. $O(\log n)$ c. $O(n)$ d. $O(n \log n)$ e. $O(n^2)$ f. $O(2^n)$

34) Selection Sort [6 points]

a. $O(1)$ b. $O(\log n)$ c. $O(n)$ d. $O(n \log n)$ e. $O(n^2)$ f. $O(2^n)$

35) Insertion Sort [6 points]

a. $O(1)$ b. $O(\log n)$ c. $O(n)$ d. $O(n \log n)$ e. $O(n^2)$ f. $O(2^n)$

36) Quicksort [6 points]

a. $O(1)$ b. $O(\log n)$ c. $O(n)$ d. $O(n \log n)$ e. $O(n^2)$ f. $O(2^n)$

37) Mergesort [6 points]

a. $O(1)$ b. $O(\log n)$ c. $O(n)$ d. $O(n \log n)$ e. $O(n^2)$ f. $O(2^n)$

38) Circle any and all of the following algorithms that only function correctly on sorted inputs. If none of the algorithms require sorted inputs to function correctly, circle choice F. [6 points]

a. Binary Search
b. Linear Search
c. Selection Sort
d. Quicksort
e. Mergesort
f. None of the above.

39) Circle any and all of the following algorithms that use a divide and conquer strategy to perform their specific task. If none of the listed algorithms use a divide and conquer strategy, circle choice F. [6 points]

a. Linear Search
b. Quicksort
c. Mergesort
d. Insertion Sort
e. Selection Sort
f. None of the above.

40) If your hashing function worked every time with no collisions, what would be the running time of a method to find an element in a hash table of size n? [1 point]

a. $O(1)$
b. $O(\log n)$
c. $O(n)$
d. $O(n^2)$

Use the code for a node and linked list given below to answer questions 41 and 42. Please note that some methods from both classes may have been removed if they do not pertain to the questions.

```
public class Node<E> {
   private E data;
   private Node<E> next;

   public Node<E> (E element, Node nextNode) {
      data = element;
      next = nextNode;
   }

   public void setNext(Node nextNode) { next = nextNode; }
}

public class LinkedList<E> {
   private Node<E> head = null;
   private Node<E> tail = null;

   public LinkedList() {}

   public void insert (E element) {
      Node<E> newNode = new Node(element, null);
      tail.setNext(newNode);
      tail = newNode;
   }

   public void insertAtFront(E element) {
      Node<E> newHead = new Node(element, head);
      head = newHead;
   }
}
```

41) What is the big-oh running time of the LinkedList's method insert in the worst case? [1 point]

a. O(1)
b. O(log n)
c. O(n)
d. O(n^2)

42) What is the big-oh running time of the LinkedList's method insertAtFront in the worst case? [1 point]

a. O(1)
b. O(log n)
c. O(n)
d. O(n^2)

For questions 43 - 44, decide whether the statement is true or false and circle the appropriate word true or false. If the statement is false, rewrite the big-oh notation so that it would be true in the space provided.

43) $n^3 + 2n + 25 = O(n)$ [3 points]

true   false

Rewritten statement (if false):

44) $n^2 + 30n + 4362 = O(n^2)$ [3 points]

true false

Rewritten statement (if false):

45) Arrange the following functions in order from slowest growing to fastest growing. [7 points]

n, $n!$, n^2, $\log n$, 1, 2^n, n^n

For questions 46-50, you are given a statement that is either true or false. Circle the letter of the choice TRUE or FALSE for each statement given.

46) If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$. [1 point]

a. TRUE
b. FALSE

47) When we declare a variable whose type is a primitive data type, we are actually creating a reference to a space of allocated memory. [1 point]

a. TRUE
b. FALSE

48) Primitive types are not objects and therefore do not have methods defined on them. [1 point]

a. TRUE
b. FALSE

49) Suppose Triangle, Circle, and Square are all subclasses of Shape. In our program, we create an array that stores objects of type Triangle. That array can hold any number of Circles, Squares, and Triangles because they are all subclasses of Shape. [1 point]

a. TRUE
b. FALSE

50) We can create an array to hold elements of primitive types (int, char, double, etc), but to hold elements of object type, we must use another type of data structure. [1 point]

a. TRUE
b. FALSE

51) Parts a – d describe four procedures in code and through words. Circle the letter of each procedure that can be categorized as recursive. [4 points]

a.
```
public int partA(Object[] items, Comparable x, int y, int z) {
       if ( y > z) {
              return -1;
       }
       else {
              int a = ( y + z )/2;
              int b = x.compareTo(items[a]);
              if (b == 0) {
                     return a;
              }
              else if (b < 0) {
                     return partA(items, x, y, a – 1);
              }
              else {
                     return partA(items, x, a + 1, z);
              }
       }
}
```

b.
```
public int partB(int x) {
       int r = x;
       r = r / 30;
       Math.power(x, 2);
       return x;
}
```

c.
```
public int partC (int x) {
       int y = 0;
       for (int i = 0; i < x; i++) {
              y = y + i
       }
       return y;
}
```

d.
```
Procedure for Writing Down Names of People Waiting in line for Movie Tickets:

   3) If line is empty go back to office.
   4) If line is not empty:
          a. Walk up to first person in line and ask for their name.
          b. Write name on official sheet and give participant free popcorn
             coupon.
          c. Move person to “fast pass” line for tickets.
          d. Begin Procedure for Writing Down Names of People Waiting in line
             for Movie Tickets again.
```

Use the following code segment to answer questions 52 – 56. Some of the questions ask about the output of a method on a particular input. If the method goes into an infinite loop or infinite recursion on an input, write "infinite loop" as your answer.

```
public int method1 (int x, int y) {
      if (y == 0) {
            return 1;
      }
      else {
            return x * method1(x, y – 1);
      }
}

public int method2 (int x, int y) {
      int result = 1;
      for (int i = 0; i < y; i++) {
            result = result * x;
      }
      return result;
}
```

52) What is the value returned from the following method call: [1 point]

```
  method1(2,1);
```

53) What is the value returned from the following method call: [1 point]

```
  method2(2,2);
```

54) What is the value returned from the following method call: [1 point]

```
  method2(2,-5);
```

55) What is the value returned from the following method call: [1 point]

```
  method1(2,-3);
```

56) These methods function differently on different inputs. On which class of inputs do these methods behave differently (circle all that apply)? [5 points]

a. When both x and y are positive numbers.
b. When both x and y are the number 0 (zero).
c. When both x and y are negative numbers.
d. When x is positive and y is negative.
e. When x is negative and y is positive.
f. When x is zero and y is positive.
g. When x is zero and y is negative.
h. When x is positive and y is zero.
i. When x is negative and y is zero.
j. The methods never function differently.

Given the following definition of the Lucas sequence, answer questions 57 – 59.

L(1) = 1;
L(2) = 3;
L(n) = L(n – 1) + L(n – 2) for n > 2

57) State what the base case(s) is/are for the Lucas sequence. [4 points]

58) State what the recursive case is for the Lucas sequence. [4 points]

59) Write the Java code for a recursive method that takes as a parameter an integer n and returns the n^{th} element of the Lucas sequence. You can assume that n will always be a number greater than zero. [8 points]

Given the following list of 19 parts of code, you should identify one example of each of the items in the code provided for this section (in the answer sheet) by precisely circling and clearly identifying by number the element in the code segment. Make sure that your circles are clearly identified with numbers that are clearly written. If the markings are not clear, the question will simply be marked incorrect and given no credit. If there is no example of the item in the code, you should write the words “Does not exist” on the line next to the element in the answer sheet.

60) Class name [1 point]

61) Constructor definition [1 point]

62) Assignment statement [1 point]

63) Comment [1 point]

64) Instance variable declaration [1 point]

65) Actual parameter (argument) [1 point]

66) Formal parameter [1 point]

67) Statement that displays information [1 point]

68) Access (Visibility) control modifier [1 point]

69) Accessor method definition [1 point]

70) Mutator method definition [1 point]

71) Creation/instantiation of an object [1 point]

72) Method call/invocation [1 point]

73) Method return type specification [1 point]

74) Superclass name [1 point]

75) Subclass name [1 point]

76) Interface name [1 point]

77) Name of a class that implements an interface [1 point]

78) Method overloading (identify one of the methods that is overloaded) [1 point]

Use the class `SimpleParams` and `SimpleParamsApp` defined below to answer questions 79–82.

```
public class SimpleParams () {
   private double _data;

   public SimpleParams() {
      _data = 5.75;
   }

   public String method1(String s) {
      return s + “ additional stuff”;
   }

   public void method2(int input) {
      int temp = input + 1;
      System.out.println(“Input was: “ + input
                             + “and temp is: “ + temp);
   }

   public void method3(double input) {
      _data = input;
   }

   public double getData() {
          return _data;
     }
}//SimpleParams

public class SimpleParamsApp {

     public SimpleParamsApp() {

          SimpleParams sp = new SimpleParams();
          double answer79 = sp.getData();

          String answer80 = sp.method1(“Simple stuff.”);

          sp.method2(6);  //Needed for question 81

          sp.method3(3.8);
          double answer82 = sp.getData();
     }

   public static void main(String[] args) {
          SimpleParamsApp spa = new SimpleParamsApp();
     }
}//SimpleParamsApp
```

79) When the code for `SimpleParamsApp` is executed, what value will answer79 be assigned? [1 point]

80) When the code for `SimpleParamsApp` is executed, what value will answer80 be assigned? [1 point]

81) When the code for `SimpleParamsApp` is executed, and `method2` is called with the value 6, as indicated in the code with a comment, what text will be outputted? [1 point]

82) When the code for `SimpleParamsApp` is executed, what value will answer82 be assigned? [1 point]

Use the following variables and their values to evaluate the expressions given in questions 83 - 91. Suppose each expression is executed independently (ie – no later expression depends on a result of a previous expression).

```
int a = 4;                   double d = 4.5;
int b = 6;                   double e = 3.3;
int c = -3;                       double f = 0.5;

boolean g = true;
boolean h = false;
boolean i = true;
```

83) `(a + b) * (c – c)` [1 point]

84) `(d / f) + (a % b)` [1 point]

85) `b < c` [1 point]

86) `d != f` [1 point]

87) `(g && h) || (!i && h)` [1 point]

Now suppose the following lines of code have been executed. The variables `a` and `c` refer back to the previous page.

```
int x = a;
int y = c++;
```

88) What is the value of `x`? [1 point]

89) What is the value of `y`? [1 point]

90) What is the value of `c`? [1 point]

91) The following line of code does not compile (`e` & `b` refer back to the previous page). What do you need to do to get the line of code to work? [4 points]

```
int z = e * b;
```

(Circle all answers from the choices below that would make the code compile.)

- i. You need to cast `b` to be a double.
- j. You need to cast `b` to be an integer.
- k. You need to cast `e` to be an integer.
- l. You need to cast `e` to be a double.
- m. You need to cast the result of `e * b` to be an integer.
- n. You need to cast the result of `e * b` to be a double.
- o. You need to make `z` a double.
- p. You need to make `z` an object.

Use the code for the method `exp1` given below to answer questions 92 - 93.

```
public double exp1 (int x1, int x2, int y1, int y2) {
     int tempX = (x2 – x1) * (x2 – x1);
     int tempY = (y2 – y1) * (y2 – y1);

     return Math.sqrt(tempX + tempY);
}
```

Suppose that the `exp1` method is called in the following way:

```
                exp1(12, 16, 24, 27);
```

92) What is the value that will be computed for `tempX` while the method is running? [1 point]

93) What value is returned from the method call? [1 point]

Use the code for the class `Conditional` given below to answer questions 94 – 96. For questions 94 – 96, you are presented with a method call. In the space provided, you should give the value that is returned from the method call.

```
public class Conditional {

  public String cond2 (double input) {
     if (input <= 5.0 && input >= 0.0) {
          return “First Branch”;
     }
     else if (input > 5.0 || input <= -2.0) {
      return “Second Branch”;
     }
     else {
      return “Third Branch”;
     }
  }
}
```

94) `cond2(3.5);` [1 point]
95) `cond2(7.345);` [1 point]
96) `cond2(-1.9);` [1 point]

Use the code for the class `Looper` given below to answer questions 97 – 100. For questions 97 – 100, you are presented with a method call. In the space provided, you should give the value that is returned from the method call.

```
public class Looper {

     public int loop1(int input) {
    for (int i = 1; i <= 20; i++) {
      input++;
    }
    return input;
  }

     public int loop2() {
          for (int counter = 10; counter > 0; counter = counter
- 2) {
               System.out.println("counter = " + counter);
          }
          return 0;
     }

     public int loop3(int input) {
          while (input < 10) {
               input = input * 2;
          }
          return input;
     }
}
```

97) `loop1(20);` [1 point]

98) `loop2();` [1 point]

99) `loop3(3);` [1 point]

100) `loop3(32);` [1 point]

For questions 101 – 103, you will be filling in the methods for the class `StringFun` as described in each question. The empty skeleton for this class is given below for reference. You will fill in the areas with the ellipses (…). Please also note the abbreviated API given for both the `java.io.BufferedReader` class as well as the `String` class as these could be of help to you while answering these questions.

```
import java.io.*;

public class StringFun {
      private java.util.ArrayList<String> _strings;

      public StringFun() {
            _strings = new java.util.ArrayList<String>();
   }

      //Loads the strings from the file specified into the
ArrayList
      public void loadFile(String filename) {[
            ...
      }

      //Indicates the number of Strings in the ArrayList that are
the right size
      public int rightSize() {
            ...
      }

      //Counts the total number of letter Ps in all the strings
in the ArrayList
      public int countPs() {
            ...
      }
}
```

Abbreviated API for java.io.BufferedReader (from Sun's Java API docs)

Constructor Summary
`BufferedReader``(Reader in)` Create a buffering character-input stream that uses a default-sized input buffer.

Method Summary	
`void`	**`close`**`()` Close the stream.
`int`	**`read`**`()` Read a single character.
`int`	**`read`**`(char[] cbuf, int off, int len)` Read characters into a portion of an array.
`String`	**`readLine`**`()` Read a line of text.

Abbreviated API for java.lang.String (from Sun's Java API docs)

Method Summary	
`char`	**`charAt`**`(int index)` Returns the `char` value at the specified index.
`int`	**`compareTo`**`(String anotherString)` Compares two strings lexicographically.
`int`	**`compareToIgnoreCase`**`(String str)` Compares two strings lexicographically, ignoring case differences.
`boolean`	**`endsWith`**`(String suffix)` Tests if this string ends with the specified suffix.
`boolean`	**`equals`**`(Object anObject)` Compares this string to the specified object.
`boolean`	**`equalsIgnoreCase`**`(String anotherString)` Compares this `String` to another `String`, ignoring case considerations.
`int`	**`length`**`()` Returns the length of this string.

`String`	**`replace`**`(char oldChar, char newChar)` Returns a new string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
`boolean`	**`startsWith`**`(String prefix)` Tests if this string starts with the specified prefix.
`String`	**`substring`**`(int beginIndex)` Returns a new string that is a substring of this string.
`String`	**`substring`**`(int beginIndex, int endIndex)` Returns a new string that is a substring of this string.
`String`	**`toLowerCase`**`()` Converts all of the characters in this `String` to lower case using the rules of the default locale.
`String`	**`toUpperCase`**`()` Converts all of the characters in this `String` to upper case using the rules of the default locale.
`String`	**`toUpperCase`**`(Locale locale)` Converts all of the characters in this `String` to upper case using the rules of the given `Locale`.
`String`	**`trim`**`()` Returns a copy of the string, with leading and trailing whitespace omitted.

101) In your answer booklet, you will finish writing the code for the method `loadFile`. Note that some of the code is already written for you. The file is already loaded into the `BufferedReader`. Your task is to read each line of the file and input each one into the `ArrayList`. Please note that we are also assuming that some other object will handle the exceptions that might be thrown. [8 points]

```
    public void loadFile(String filename) throws
FileNotFoundException,
                                                      IOException {

        BufferedReader in = new BufferedReader(new
FileReader(filename));
             //Your code begins here.
             //Write your code in the answer booklet.
```

```
      }
```

102) Write the code for the method `rightSize` so that it returns the number of strings in `_strings` whose length is between 3 and 10 characters inclusive. [8 points]

```
      public int rightSize() {
            //Write the code for this method in your answer
booklet









      }
```

103) Write the code for the method `countPs` so that it returns the total number of occurences of the letter P in all of the strings in `_strings`. Your method should count both lower case (p) and upper case (P) letters. [8 points]

```
      public int countPs() {
            //Write the code for this method in your answer
booklet









      }
```

Use the following code segment for the classes named `Types`, `Thing`, and `Driver`, the interface named `Colorable`, and your knowledge of Java to answer the questions 104 – 113. If the question has multiple choices, you should circle the letter of the best answer for each question, unless instructed otherwise.

```
public interface Colorable {
     public void setColor (java.awt.Color color);
     public java.awt.Color getColor();
}//Colorable

public class Thing implements Colorable{
     private java.awt.Color _color;

     public Thing() {
          _color = java.awt.Color.WHITE;
     }

     public void setColor (java.awt.Color color) {
          _color = color;
     }

     public java.awt.Color getColor() {
          return _color;
  }
}//Thing



public class Types {
     private Thing _thing;
     private int _number;

     public Types() {
          _thing = new Thing();
          _number = 0;
     }

     public void incrementNumber (int increment) {
          _number += increment;
   }
}//Types
```

```
public class Driver {
     public Driver() {
          int i = 5;
      Colorable t = new Thing();
      //Line for question 109 inserted here
      this.changeParams(i, t);
     }

     public void changeParams (int input, Colorable thing) {
          input = input * 2;
          thing.setColor(java.awt.Color.RED);
     }

     public static void main (String[] args) {
          Driver d = new Driver();
     }
}//Driver
```

104) What is the value of `_thing` **before** the constructor is run for the class `Types`? [1 point]

a. A null reference.
b. A random value assigned value assigned by the compiler.
c. An object of type `Thing` whose instance variables are set to null.
d. `_thing` does not exist before the constructor is run.

105) What is the value of `_number` **after** the constructor is run for the class `Types`? [1 point]

a. null
b. 0
c. -1
d. undefined

106) Which of the variables presented in this code segment are object references? Circle the letters of all that apply. [5 points]

a. `_color`
b. `_thing`
c. `_number`
d. `increment`
e. `i`
f. `t`
g. `input`
h. `thing`
i. `d`
j. None of these variables are references.
k. All of these variables are references.

107) Which of the members (variables or methods) from the class `Types` are accessible from outside the class? Circle the letters of all that apply. [6 points]

a. `_thing`
b. `_number`
c. `Types()` constructor
d. `incrementNumber(int increment)` method
e. None of the members are accessible outside of the class.
f. All of the members are accessible outside of the class.

108) Which of the members from the class `Driver` are not local and only accessible from inside the class? Circle the letters of all that apply. [6 points]

a. `i`
b. t
c. `Driver()` constructor
d. `changeParams(int input, Colorable thing)` method
e. `main(String[] args)` method
f. None of the members are only accessible from inside the class.

109) Suppose we add the following line to the constructor in the space indicated by the comments in `Driver`: [1 point]

```
t = i;
```

Is this valid? What would happen?

a. It is perfectly valid. The code would run.
b. It is valid. The type of i is a primitive and t is an object type and you can always assign a primitive type to any object type because primitives are subclasses of objects.
c. This is not valid. The code would compile, but would cause a run-time error.
d. This is not valid. The code would not compile because t and i are not of compatible types.

110) Under what circumstances would you be allowed to add the following line of code to the end of the class `Driver`'s constructor: [1 point]

```
t = new OtherThing();
```

e. No special circumstances, this line of code would always work.
f. Only when `OtherThing` is a subclass of `Thing`.
g. Only when `OtherThing` is a superclass of `Thing`.
h. This line of code would never work because the declared type of `t` is `Thing`, so you must assign a `Thing` object to `t`.

111) Looking at the code for `Driver`, what is the value of `i` after the method `changeParams` has been called? [1 point]

e. The value is unchanged, 5.
f. The value is 2 times the value, 10.
g. The value is 0 because `i` was never initialized.
h. The value will be null because you can not change the value of `i` from within a method.

112) Again looking at the code for `Driver`, after the method `changeParams` has been called in the constructor, suppose we add the following line of code:

```
      java.awt.Color color = t.getColor();
```

What would be the value of color? [1 point]

e. java.awt.Color.WHITE
f. java.awt.Color.RED
g. java.awt.Color.PINK
h. null

For questions 113 - 115, use the following code to help you answer the questions.

```
public class Ball {

      private java.awt.Color _color;

      public Ball() {
            _color = java.awt.Color.GREEN;
      }

      public java.awt.Color getColor() {
            return _color;
      }

      public void setColor (java.awt.Color color) { _color = color; }
}

public class Driver {

      public Driver() {
            Ball ball = new Ball();
            ball.setColor(java.awt.Color.RED);

            Ball ball2 = new Ball();
            ball2 = ball;              //Question 113 refers up to this
point

            ball.setColor(java.awt.Color.BLUE);          //Question 114
code
            java.awt.Color question114 = ball2.getColor();
//Question114 code

            ball2 = new Ball();                            //Question
115 code
            ball2.setColor(java.awt.Color.BLACK);         //Question
115 code
            java.awt.Color question115 = ball.getColor();
//Question115 code
   }
```

```
      public static void main(String[] args) {
            Driver d = new Driver();
      }
}
```

113) After the line of code in `Driver` that reads

```
            ball2 = ball;
```

is executed, which reference refers to a green ball? [1 point]

e. `ball`
f. `ball2`
g. both `ball` and `ball2`
h. neither `ball` or `ball2`

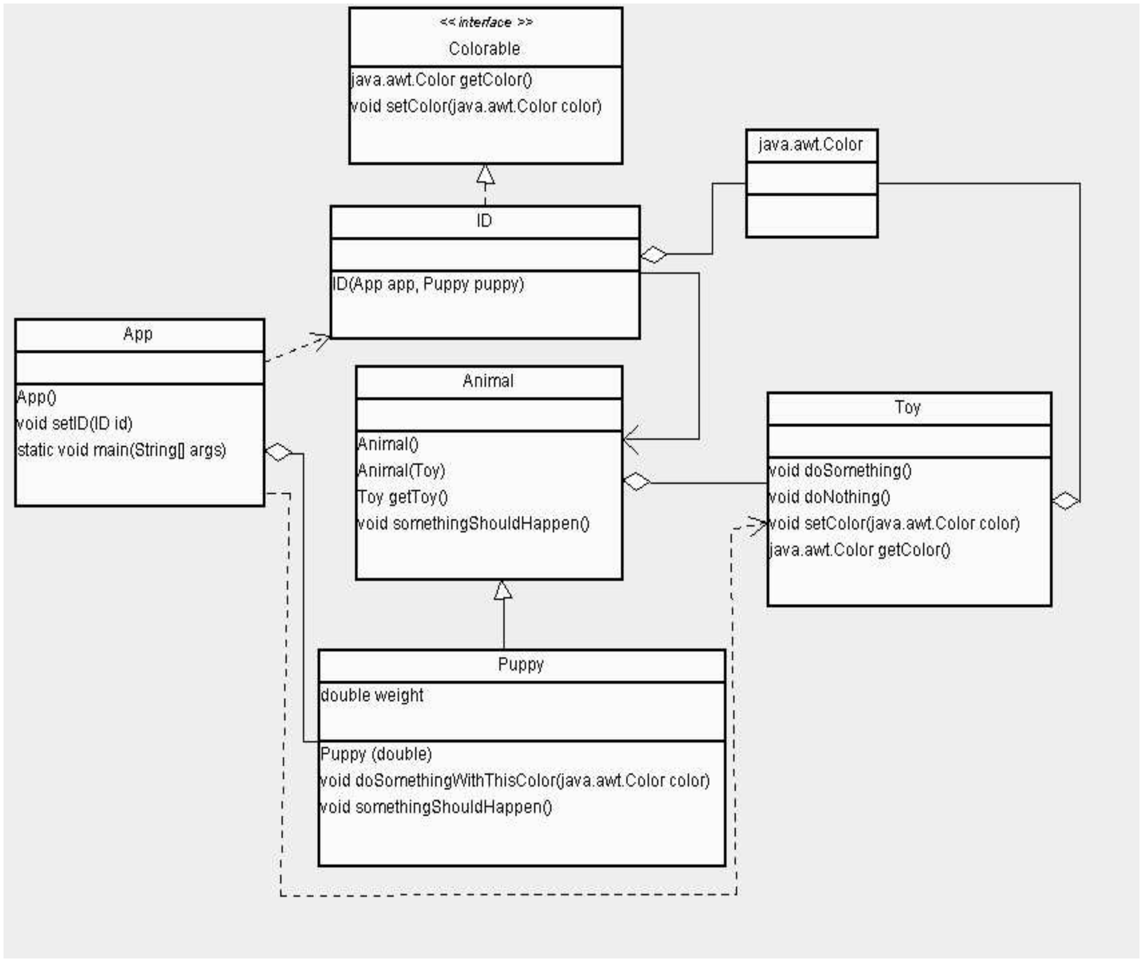
114) Focus your attention on the lines of code that is the code for Question 114 as indicated by comments. What will the value of the variable `question114` be? [1 point]

e. `java.awt.Color.RED`
f. `java.awt.Color.GREEN`
g. `java.awt.Color.BLUE`
h. no color – it will be an error

115) Focus your attention on the lines of code that is the code for Question 115 as indicated by comments. What will the value of the variable `question115` be? [1 point]

e. `java.awt.Color.RED`
f. `java.awt.Color.GREEN`
g. `java.awt.Color.BLUE`
h. `java.awt.Color.BLACK`

Use the UML diagram given below as well as the code segment given after the diagram to answer questions 116 – 127.



<< interface >>
Colorable
java.awt.Color getColor()
void setColor(java.awt.Color color)
java.awt.Color
ID
ID(App app, Puppy puppy)
App
App()
void setID(ID id)
static void main(String[] args)
Animal
Animal()
Animal(Toy)
Toy getToy()
void somethingShouldHappen()
Toy
void doSomething()
void doNothing()
void setColor(java.awt.Color color)
java.awt.Color getColor()
Puppy
double weight
Puppy (double)
void doSomethingWithThisColor(java.awt.Color color)
void somethingShouldHappen()

```
/* The classes given below were written for the purposes of this exam.
In
 * reality, they would each be in their own separate file, but are
reprinted
 * here as one long file for ease of reading. This “print-out” spans
two
 * pages, so please look at both pages while answering the following
 * questions.
*/

public class App {

    private Puppy _puppy;
    private ID _id;

    public App (){
      System.out.println("App constructor called.");
      _puppy = new Puppy(new Toy());
      this.setID(new ID(this, _puppy));
      System.out.println("App constructor end.");
    }

    public void setID(ID id) {
      _id = id;
    }

    public static void main (String[] args) {
      App app = new App();
    } // end of main ()
}// App

public interface Colorable {
   java.awt.Color getColor();
   void setColor(java.awt.Color color);
}// Colorable

public class ID implements Colorable{

   private Animal _animal;
   private java.awt.Color _color;

   public ID (App app, Animal animal){
      _animal = animal;
      _color = java.awt.Color.BLACK;
   }

   public java.awt.Color getColor() {
      return _color;
   }

   public void setColor(java.awt.Color color) {
      _color = color;
   }
}// ID

public class Animal {
   private Toy _toy;

   public Animal (){ _toy = new Toy(); }

   public Animal (Toy toy) { _toy = toy; }

   protected Toy getToy() { return _toy; }

   public void somethingShouldHappen() { _toy.doSomething(); }
}// Animal

public class Puppy extends Animal{
   public Puppy() {}

   public Puppy (Toy toy){
      super(toy);
      this.doSomethingWithThisColor(this.getToy().getColor());
   }

   public void doSomethingWithThisColor(java.awt.Color color) {
      this.getToy().setColor(color.darker());
   }

   public void somethingShouldHappen() {
      super.somethingShouldHappen();
      this.getToy().doNothing();
   }
}// Puppy

public class Toy {
   private java.awt.Color _color;
   private String[] _sounds;

   public Toy (){ _color = java.awt.Color.RED; }
   public void setColor(java.awt.Color color) { _color = color; }
   public java.awt.Color getColor() { return _color; }

   public void doSomething() {
      _sounds = new String[20];
      for ( int count = 0; count < _sounds.length; count++) {
       _sounds[count] = "Squeak";
      } // end of for ()
      System.out.println(_sounds);
   }
   public void doNothing() {
      //This method really does nothing.
   }
}// Toy
```

For questions 116 – 125, assume the following variable declarations. Note that any ellipses (…) indicates material that will not affect your answer to the question and can be safely ignored. For each of the method calls in questions 116 - 125, you should circle the name of the class/interface that defines the method that will be executed for the method call. If the call is Illegal, circle the choice that corresponds to "Illegal".

Colorable c = new ID(…);
Animal animal = new Puppy();
Puppy puppy = new Puppy();

116) `c.getColor();` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

117) `c.setColor(…);` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

118) `c.setID();` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

119) `animal.getToy();` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable

l. ID
m. Toy
n. Illegal

120) `animal.somethingShouldHappen();` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

121) `animal.doSomethingWithThisColor(…);` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

122) `puppy.somethingShouldHappen()`; [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

123) `puppy.doSomethingWithThisColor(…);` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

124) `puppy.getToy();` [1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

125) `puppy.getColor();`[1 point]

h. App
i. Animal
j. Puppy
k. Colorable
l. ID
m. Toy
n. Illegal

Recall that questions 126 – 127 still refer to the UML diagram and code used for questions 116-125.

126) Circle the names of all methods that are simply inherited (not overridden) by some other class. If no methods are inherited, circle the choice that corresponds to “None”. [6 points]

m. void main (String[] args) //in class App
n. void setColor(java.awt.Color color) //in class ID
o. Animal () //in class Animal
p. Animal (Toy toy) //in class Animal
q. Toy getToy() //in class Animal
r. void somethingShouldHappen() //in class Animal
s. Puppy () //in class Puppy
t. Puppy (Toy toy) //in class Puppy
u. void somethingShouldHappen() //in class Puppy
v. void doSomethingWithThisColor(java.awt.Color color) //in class Puppy
w. void setColor(java.awt.Color color) //in class Toy
x. None

127) Is the method `somethingShouldHappen` in the class `Puppy` partially overridden or totally overridden? [1 point]

c. Partially overridden
d. Totally overridden

Appendix B

Grading Guideline for Exam

The table below gives the categorization of each of the questions on the exam as well as the point value for each question. The categories are:

- MC1A: Multiple choice question where students are asked to provide one answer
- MCMA: Multiple choice question where students are asked to provide all possible answers from a list of choices
- FR1A: Free response question (no choices given) where there is a clearly correct and usually short answer
- FRCA: Free response question (no choices given) where the answer may be slightly more complex than a short answer, or the grading might allow for partial credit
- SG: Question that has been designated as subjective. The content of the answer is sufficiently complex that there is much room for variance among answers. It is recommended that whenever possible, these questions be

graded by more than one rater and the raters' ratings are compared for consistency. If it is not possible, it would be best if one rater graded one question for the entire group of exams. If neither of the above are possible and the grading must be split amongst multiple graders, it is recommended that the graders grade the same questions at the same time and engage in discourse about how they have interpreted the guideline so as to ensure consistency of scoring. This is probably best achieved if the grading takes place in the same location.

Question Number	Question Category	Number of Choices	Total Points Possible for Question	Individual Choice Weights[36]
1	FRCA	N/A[37]	8	N/A
2	FRCA	N/A	3	N/A
3	FRCA	N/A	8	N/A
4	MC1A	9	1	N/A
5	SG	N/A	8	N/A
6	FR1A	N/A	1	N/A
7	FR1A	N/A	1	N/A
8	FR1A	N/A	1	N/A
9	SG	N/A	8	N/A
10	SG	N/A	8	N/A
11	SG	N/A	8	N/A
12	SG	N/A	8	N/A
13	FR1A	N/A	1	N/A
14	FR1A	N/A	1	N/A
15	FR1A	N/A	1	N/A
16	FR1A	N/A	1	N/A
17	FR1A	N/A	1	N/A
18	FRCA	N/A	3	N/A
19	FRCA	N/A	8	N/A

[36] See Grading Non-Multiple Choice Questions section for more information about partial credit for multiple choice questions where students may need to circle more than one answer.
[37] N/A means the column category is not applicable to that particular question.

20	MCMA	9	4.5	0.5
21	MCMA	4	4	1
22	MC1A	3	1	N/A
23	SG	N/A	8	N/A
24	FRCA	N/A	6	N/A
Question Number	Question Category	Number of Choices	Total Points Possible for Question	Individual Choice Weights
25	FRCA	N/A	6	N/A
26	FRCA	N/A	6	N/A
27	FRCA	N/A	6	N/A
28	FRCA	N/A	6	N/A
29	FRCA	N/A	6	N/A
30	FRCA	N/A	6	N/A
31	FRCA	N/A	6	N/A
32	MCMA	6	6	1
33	MCMA	6	6	1
34	MCMA	6	6	1
35	MCMA	6	6	1
36	MCMA	6	6	1
37	MCMA	6	6	1
38	MCMA	6	6	1
39	MCMA	6	6	1
40	MC1A	4	1	N/A
41	MC1A	4	1	N/A
42	MC1A	4	1	N/A
43	SG	N/A	3	N/A
44	SG	N/A	3	N/A
45	FRCA	N/A	7	N/A
46	MC1A	2	1	N/A
47	MC1A	2	1	N/A
48	MC1A	2	1	N/A
49	MC1A	2	1	N/A
50	MC1A	2	1	N/A
51	MCMA	4	4	1
52	FR1A	N/A	1	N/A
53	FR1A	N/A	1	N/A
54	FR1A	N/A	1	N/A
55	FR1A	N/A	1	N/A
56	MCMA	10	5	0.5
57	SG	N/A	4	N/A

58	SG	N/A	4	N/A
59	SG	N/A	8	N/A
60	FRCA	N/A	1	N/A
61	FRCA	N/A	1	N/A
62	FRCA	N/A	1	N/A
Question Number	Question Category	Number of Choices	Total Points Possible for Question	Individual Choice Weights
63	FRCA	N/A	1	N/A
64	FRCA	N/A	1	N/A
65	FRCA	N/A	1	N/A
66	FRCA	N/A	1	N/A
67	FRCA	N/A	1	N/A
68	FRCA	N/A	1	N/A
69	FRCA	N/A	1	N/A
70	FRCA	N/A	1	N/A
71	FRCA	N/A	1	N/A
72	FRCA	N/A	1	N/A
73	FRCA	N/A	1	N/A
74	FRCA	N/A	1	N/A
75	FRCA	N/A	1	N/A
76	FRCA	N/A	1	N/A
77	FRCA	N/A	1	N/A
78	FRCA	N/A	1	N/A
79	FR1A	N/A	1	N/A
80	FR1A	N/A	1	N/A
81	FR1A	N/A	1	N/A
82	FR1A	N/A	1	N/A
83	FR1A	N/A	1	N/A
84	FR1A	N/A	1	N/A
85	FR1A	N/A	1	N/A
86	FR1A	N/A	1	N/A
87	FR1A	N/A	1	N/A
88	FR1A	N/A	1	N/A
89	FR1A	N/A	1	N/A
90	FR1A	N/A	1	N/A
91	MCMA	8	4	0.5
92	FR1A	N/A	1	N/A
93	FR1A	N/A	1	N/A
94	FR1A	N/A	1	N/A
95	FR1A	N/A	1	N/A

96	FR1A	N/A	1	N/A
97	FR1A	N/A	1	N/A
98	FR1A	N/A	1	N/A
99	FR1A	N/A	1	N/A
100	FR1A	N/A	1	N/A

Question Number	Question Category	Number of Choices	Total Points Possible for Question	Individual Choice Weights
101	SG	N/A	8	N/A
102	SG	N/A	8	N/A
103	SG	N/A	8	N/A
104	MC1A	4	1	N/A
105	MC1A	4	1	N/A
106	MCMA	11	5.5	0.5
107	MCMA	6	6	1
108	MCMA	6	6	1
109	MC1A	4	1	N/A
110	MC1A	4	1	N/A
111	MC1A	4	1	N/A
112	MC1A	4	1	N/A
113	MC1A	4	1	N/A
114	MC1A	4	1	N/A
115	MC1A	4	1	N/A
116	MC1A	7	1	N/A
117	MC1A	7	1	N/A
118	MC1A	7	1	N/A
119	MC1A	7	1	N/A
120	MC1A	7	1	N/A
121	MC1A	7	1	N/A
122	MC1A	7	1	N/A
123	MC1A	7	1	N/A
124	MC1A	7	1	N/A
125	MC1A	7	1	N/A
126	MCMA	12	6	0.5
127	MC1A	7	1	N/A

The table beginning on page 9 gives the answers to the questions that are multiple choice (both MC1A and MCMA) as well as those that are free response, but have one clearly recognizable answer (FR1A).

For multiple choice questions with only one answer (MC1A) or free response one answer (FR1A) questions, the answer is either correct or incorrect, no partial credit is awarded. Each of these questions is worth 1 point and either the student earns the point for the correct answer, or does not earn credit if the answer is incorrect.

For any question that is free response, a student may give extraneous information that is not asked for by the question. If the extraneous information is correct, neither award, nor deduct points. If the information is not correct, a deduction of one point should be made for the question no matter how many pieces of extraneous information were given. If incorrect extraneous information is given on the next question as well, another one point deduction is appropriate. Essentially, each question should be treated individually in this case. If the student gives incorrect extraneous information for every question on the exam, they will be deducted one point for every question on the exam.

For multiple choice questions with more than one answer (MCMA), grading is based on the correct state of each of the choices for the question. That means, if an answer is supposed to be circled and it is, the student earns the appropriate credit, either 1 point or ½ point (refer to first table for correct point value). If an answer is not supposed to be circled and it is not, the student earns the appropriate credit. If the answer should be circled and it is not the student does not earn credit. Likewise, if the answer was not circled and should have been circled the student does not earn credit. Let's look at a few examples of this. The first example is a question where each answer choice is worth one point. The second example is a question where each answer choice is worth ½ point.

Question X (Total points: 4; each choice 1):
Answer Choices: A B C D
Correct Answers: B D

Student Answers: B D
Earns: 4 points
Reason: A & C are correctly not answers and B & D are answers, all choices in correct state.

Student Answers: A B D
Earns: 3 points
Reason: Choices B, C, and D are in the correct state. Choice A is not, so the student loses 1 point.

Student Answers: B
Earns: 3 points
Reason: Choices A, B, and C are in the correct state. Choice D is not, so the student loses 1 point.

Student Answers: B C
Earns: 2 points
Reason: Choices A and B are in the correct state. Choices C and D are not, so the student loses 2 points.

Student Answers: A C
Earns: 0 points
Reason: None of the choices are in the correct state.

Question Y (Total points: 4, each choice ½):
Answer Choices: A B C D E F G H
Correct Answers: B C F G H

Student Answers: B C F G H
Earns: 4 points
Reason: All Choices are in the correct state, answers are circled, non-answered are not.

Student Answers: B C F H
Earns: 3.5 points
Reason: Choice G is missing from the answers, but all other choices in correct state.

Student Answers: A B F G H
Earns: 3 points
Reason: Choices B, D, E, F, G, and H are in the correct state Choices A and C are not.

Student Answers: B D E F G H
Earns: 2.5 points
Reason: Choices A, B, F, G, and H are in the correct state. Choices C, D, and E are not.

Student Answers: A B C E F
Earns: 2 points
Reason: Choices B, D, E, and F are in the correct state. Choices A, C, G, and H are not.

Student Answers: A B C D E
Earns: 1 point
Reason: Choices B and C are not. Choices A, D, E, F, G, and H are not.

Question Number	Question Category	Correct Answers
4	MC1A	D
6	FR1A	50
7	FR1A	0
8	FR1A	49
13	FR1A	q
14	FR1A	b, f, g, k, g, n, r, s, or v (only one answer needed, but any of these answers is correct)
15	FR1A	0 or 1 (depending on what students were taught)
16	FR1A	3 (if answer to 15 was 0) or 4 (if answer to 15 was 1)
17	FR1A	p
20	MCMA	B C E
21	MCMA	A C
22	MC1A	C
32	MCMA	B C D E F
33	MCMA	C D E F
34	MCMA	E F
35	MCMA	E F
36	MCMA	E F
37	MCMA	D E F
38	MCMA	A
39	MCMA	B C
40	MC1A	A
41	MC1A	A
42	MC1A	A
46	MC1A	A
47	MC1A	B
48	MC1A	A
49	MC1A	B
50	MC1A	B
51	MCMA	A D
52	FR1A	2
53	FR1A	4
54	FR1A	1
55	FR1A	Infinite loop
56	MCMA	C D G
79	FR1A	5.75
80	FR1A	Simple stuff. additional stuff
81	FR1A	Input was 6 and tamp is: 7
82	FR1A	3.8

83	FR1A	0
84	FR1A	13
85	FR1A	false[38]
86	FR1A	true
87	FR1A	false
88	FR1A	4
89	FR1A	-3
90	FR1A	2
91	MCMA	C E G
92	FR1A	16
93	FR1A	5
94	FR1A	“First branch”
95	FR1A	“Second branch”
96	FR1A	“Third branch”
97	FR1A	40
98	FR1A	0
99	FR1A	12
100	FR1A	32
104	MC1A	A
105	MC1A	B
106	MCMA	A B F H I
107	MCMA	C D
108	MCMA	F
109	MC1A	D
110	MC1A	B
111	MC1A	A
112	MC1A	B
113	MC1A	D
114	MC1A	C
115	MC1A	C
116	MC1A	E
117	MC1A	E
118	MC1A	G
119	MC1A	B
120	MC1A	C
121	MC1A	G
122	MC1A	C
123	MC1A	C

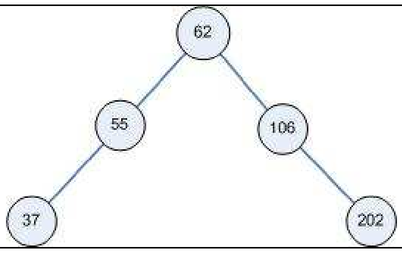
[38] For questions 85, 86, and 87, students are asked to evaluate an expression in Java that uses booleans. No credit should be given if the words true and/or false are not given. The letters T or F, the values 0 or 1, or the words TRUE or FALSE are not the boolean constants in Java and should receive no credit.

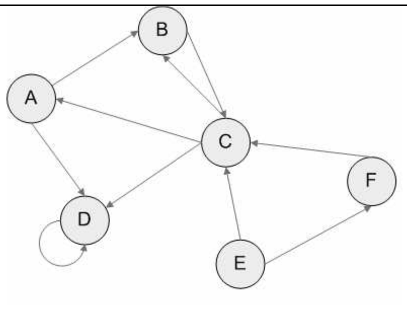
124	MC1A	B
125	MC1A	G
126	MCMA	E
127	MC1A	A

Grading Non-Multiple Choice Questions

Free response complex answer questions have slightly more complex answers or slightly more complex grading rules than the free response one answer questions. For each of the questions that fall into this category, the answers to the questions are given as well as any necessary explanation of the point breakdown.

Recall the discussion in the previous section about incorrect extraneous information. The same rules apply for these questions.

Question Number (FRCA)	Total Points Possible for Question	Answer	Notes about Grading
1	8		Eight points awarded if tree is correct. If one or two nodes are misplaced in the tree, student earns 4 points. If more than two nodes are misplaced, student earns 0 points.
2	3	34, 26, 30	There are three "places" for answers for this question. Each place should be treated as a separate answer. The first place should contain the number 34. If it does, award one point. The second place should contain the number 26. If it does, award one point. The third place should contain the number 30. If it does, award one point.
3	8	Will vary depending on what deletion strategy was taught at a particular institution.	To earn eight points (full credit), the tree given as an answer must be a valid binary search tree (BST) containing all of the nodes of the original except 34. If the student produces a valid BST that is missing one node in addition to 34, award only 4 points. If the student produces a BST that has greater than one node missing in addition to 34 (ie – an entire branch of the original tree has been deleted with the root), award zero points. If the tree given was not a valid BST, award zero points.
18	3	k and g and r	One point for each correct answer given. All three answers need to be given for full credit. Additional answers that are not correct do not earn any credit, but

			they do not lose any credit either.
19	8		If all nodes present and all arcs correct connecting nodes, award eight points. If the student has arcs B to D (perhaps twice), D to A, and F to C, they have just tried to copy the adjacency list into a graph and do not understand how to translate between the two and should be awarded zero points.
24	6	Hash Map	No other answers correct for this question. If other answers listed along with this answer, award only 3 points of the 6 possible.
25	6	Stack	No other answers correct for this question. If other answers listed along with this answer, award only 3 points of the 6 possible.
26	6	Queue	No other answers correct for this question. If other answers listed along with this answer, award only 3 points of the 6 possible.
27	6	Graph & Tree	The answers graph and tree are required and earn 3 points each for a total of 6 points. If the student wrote Hash Map, the student should not earn additional credit, or lose any points. If other answers are listed beside the three mentioned, subtract 3 points from the total earned, but the point value should not go below zero.
28	6	Array	No other answers correct for this question. If other answers listed along with this answer, award only 3 points of the 6 possible.
29	6	Queue	No other answers correct for

			this question. If other answers listed along with this answer, award only 3 points of the 6 possible.
30	6	Hash Map	No other answers correct for this question. If other answers listed along with this answer, award only 3 points of the 6 possible.
31	6	Graph	No other answers correct for this question. If other answers listed along with this answer, award only 3 points of the 6 possible.
45	7	1, log n, n, n^2, 2^n, n!, n^n	There are seven "places" for answers for this question. Each place should be treated as a separate answer. The first place should contain 1. If it does, award one point. The second place should contain log n. If it does, award one point. The third place should contain n. If it does, award one point. The fourth place should contain n^2, and so on.
60	1	Any one of: App, ID, Animal, Puppy, Toy	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
61	1	Student should circle the entirety of the constructor definition from the word public to the }. There are constructors for App, ID, Animal (2 constructors present), Puppy, and Toy.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or

			markings are illegible).
62	1	Students should circle any statement that performs assignment (using =). The entire statement including the ; should be circled. Examples are: `_puppy =new Puppy(new Toy());` `_animal = animal;` `_color =java.awt.Color.BLACK;` `_color = color;` `_toy = toy;` `_color = java.awt.Color.RED;` `int count = 0;`	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
63	1	Students can identify either a multi-line comment (indicated by the /* and ended with */) or an in-line comment (indicated by //). The entire comment should be circled for full credit.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
64	1	The declaration of an instance variable is not the assignment of that variable to a value. Examples of this include: `private Puppy _puppy;` `private ID _id;` `private Animal _animal;` `private Toy _toy;` `private String[] _sounds;` `private java.awt.Color _color;`	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
65	1	The actual parameter is the value passed into a method call. Therefore, this answer must be in a method call to be correct. Some examples include: `“App constructor called.”` `new Toy()` `this` `puppy` `“App constructor end”` `this.getToy().getColor()` `color.darker()`	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).

		`_sounds`	

66	1	The formal parameter is the parameter declared in the method signature. This declaration includes both the type and name of the parameter and both must be circled for full credit. Some examples include:
		`ID id`
		`String[] args`
		`java.awt.Color color`
		`App app`
		`Animal animal`
		`Toy toy`
			One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).

67	1	Any statement that is of the form `System.out.println` displays information to wherever out points to. The entire System.out statement should be circled for full credit.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
68	1	The access control modifiers for Java are `public`, `private`, and `protected`. Circling any one of these will earn full credit.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
69	1	Accessors get information from the instance variables. They commonly have the name getX. The only three accessors are `getColor` defined in ID, `getToy` defined in Animal, and `getColor` defined in Toy. The entire method definition from public to the } needs to be circled for full credit.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
70	1	Mutators set the information stored in the instance variables. They commonly have the name setX. The	One point if code segment identified correctly, zero points if not identified

		only three mutators are `setID` defined in App, `setColor` defined in ID, and `setColor` defined in Toy. The entire method definition from public to the } needs to be circled for full credit.	correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).

71	1	Objects are created by using the keyword new in Java. Anytime new is used in code an object is created. The examples in this code are:
		`new Toy()`
		`new ID(this, puppy)`
		`new App()`
			One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).

72	1	Examples of method calls in the code are:
		`System.out.println(“App constructor called.”)`
		
		`this.setID(new ID(this, puppy))`
		
		`System.out.println(“App constructor end.”)`
		
		`_toy.doSomething()`
		
		`super(toy)`
		
		`this.doSomethingWithThisColor(this.getToy().getColor());`
		
		`this.getToy().getColor()`
		
		`This.getToy()`
		
		`this.getToy().setColor(color.darker())`
		
		`Color.darker()`
		
		`super.somethingShouldHappen()`
			One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).

		`this.getToy().doNothing()` `System.out.println(_sounds )` Any call to the constructor will also be accepted for full credit for this question.	
73	1	The method return type is specified in the method signature before the name of the method. If the method does not return anything the keyword void is used. In this code, return types are `void`, `java.awt.Color`, and `Toy`. These words must be circled in the method signature to be given full credit.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
74	1	The name of the superclass of a class follows the keyword extends. In this code, the only superclass is `Animal`, which is the superclass of Puppy.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
75	1	In this code, `Puppy` is the subclass of Animal. This is the only example of inheritance in the code given.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
76	1	The only interface declared in this code example is `Colorable`.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is

			being identified for which question (ie – the numbers or markings are illegible).
77	1	The only class that implements an interface in this code example is `ID`.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).
78	1	In the class Animal, the constructor is overloaded, so either one can be identified for full credit. In Puppy, the constructor is overloaded as well, so either one can be identified for full credit.	One point if code segment identified correctly, zero points if not identified correctly. Zero points if more than one segment identified for each question. Zero points if it is not clear what is being identified for which question (ie – the numbers or markings are illegible).

Subjective questions will be the most time consuming of the questions on the exam to grade. All but four of these questions involve grading student code segments. When grading a segment of student code, the syntax of the code is not what is graded. However, the syntax will be present the answer and will play some role in the grading by way of helping the rater to see if the student understands how to solving the problem.

Grading will be using a triage system of grading. Eight points is awarded for an answer that has all sufficiently used the main themes of the answer. Each question's themes are given in the chart following this section. If some, but not all of the themes are present or adequately addressed, four points is awarded. If none of the themes are present or adequately addressed, zero points are awarded.

However, the fundamental pieces of each question are the main themes addressed by each answer, not the intricate syntactic minutia of the particular language of implementation. This means, that if the correct name for a method is `add` and the student writes `insert` instead, points should not be deducted for this mistake, provided that an API was not given to help answer a particular question.

These details are further elaborated in the grading guideline chart for each question given below. Note that question 43, 44, 57, and 58 are not coding questions and have different point breakdowns.

Question Number (SG)	Total Points Possible for Question	Further discussion of grading of question
5	8	Student must demonstrate the ability to use an iterator or a for-each loop, and properly use the color when calling the setColor method. If no iterator is used, zero points awarded. For an iterator, the exact method names (hasNext() and next()) are not necessary for full credit, but a loop stopping when there are no more elements and getting the next element are necessary. (4 points) Inside the loop, student must call the setColor method passing the elements from the collection in as parameters. (4 points) If using a for-each loop, syntax does not have to be perfect, but student must demonstrate knowledge that the structure is for-each element of the collection named colorsForBackground (4 points), call setColor and pass in each element in the body of the for-each loop (4 points).
9	8	Creating a new array that is larger than the array passed in as a parameter (4 points).

		Copying contents of original array into new larger array (4 points). New array should not simply be one bigger than previous array. Preferable is an array that is twice as big as the original, but any significant growth will suffice (the benchmark was a growth of 10 elements). If the new array is not sized big enough, award only 4 points.
10	8	Creating a new array of the appropriate size (4 points). Inserting appropriate elements into array (4 points). Values that are inserted into the array are squares of the array index. The correct syntax for squaring a number should be something like: element * element. However, full credit should be awarded for variations like element^2 or element^2.
11	8	Basic find structure (ie looping through the structure and maintaining some sort of “flag” about the status of the search, and returning that value when the search is over). [4 points] In this case, it is important for the student to realize that this structure does not have an iterator and needs a system of nested loops to search it. [4 points] Of secondary importance is the actual syntax for dealing with a 2D array. Syntax can be incorrect and student should still earn full points if all the above criteria are met. If only basic find structure is there, or if an iterator is used in one loop instead of nested loops, or if there is simply only one loop used, award only four points.
12	8	The programmer must be sure to maintain the linkage of the nodes in the list after the node is removed. There are a few cases that must be given some sort of consideration in the code to receive full credit: - Finding the node that is to be removed (using a loop of some sort) - The case when the node to be removed is the head of the list - The case when the node to be removed is somewhere in the middle of the list or at the end of the list Missing one of the above bullet points constitutes a question only receiving half credit (4 points). Missing more than one will result in receiving 0 points.

		In each of these cases, there is what we describe as a “null-check” that should be performed in order to successfully restore the links of the list. For example, every call to getNext() or getPrev() should be preceded by a check to make sure that null is not returned. These checks of the values in the list indicate a student that really has a mastery of the material, but is not an integral part of the nature of deleting a node from this structure. Therefore, if a student does not provide adequate checking, but satisfies the bullet points above, they should not be deducted points.
23	8	Students should receive full credit only if they express both the ideas that: – when a class inherits from a superclass, it inherits the superclass’ methods – a stack should only be LIFO and have methods push, pop, peek If only one of these ideas is adequately expressed, award only 4 points.
43	3	One point awarded for student indicating that the statement was false. Two points awarded for the correction of the Big-O statement, stating: $n^3 + 2n + 25 = O(n^3)$ or any other proper Big-O bounds. If the student selects true, zero points awarded.
44	3	The answer to this question is true, so three points are awarded if true is answered, zero points if false is chosen.
57	4	Base case of recursion must be given. The base cases are L(1) = 1 and L(2) = 1. If only one is given, award half credit. If only L(1) and L(2) are listed, award full credit. If the answer given is 1, do not award any credit.
58	4	Recursive case of recursion must be given. The recursive case is L(n) = L(n-1) + L(n-2) where n > 2. If all parts are correct and n>2 is the only thing missing, award full credit. If L(n) is given as the recursive case and L(1) and L(2) are given as answers to 57, award full credit.

		If the expression L(n-1) + L(n-2) is given, award zero points.
59	8	Both base cases return the correct value (4 points). Recursive case correct computes the value (4 points). If the method is not recursive, award zero points.

101	8	Loop through the file. End of file not specified in API, so a reasonable guess as to how to know when the file is at end is acceptable for full credit (4 points).
		
		Place each line of the file in the array list (4 points). Recall that exact name of insert method on ArrayList is not necessary for full credit.
		
		Note: If student writes something similar to this:
		While (in.readLine() != xxx) {
		//some code that calls readLine() again
		}
		Only award 4 points

102	8	Loop through the collection of strings and keep count of number of strings that are correct size/length (4 points). The ability to find in the API the method that can be used to find the string's length (4 points). While student were given the parameters for the correct size, the idea of correct size is more important than exactness. For example, incorrect boundary conditions should still receive full credit. Students should not receive full credit for using incorrect method name for length in this case because the API for the String class was given to them. Only award 4 points if student uses incorrect name.
103	8	Loop through the collection of strings and look at each string in that collection to find the number of Ps in that string and then in the overall collection (4 points). The ability to find in the API the methods which would be most useful for finding the Ps in the string (4 points). If student performs operations correctly, but does not check for both upper and lower case p, full credit should still be awarded. Students should not receive full credit for using incorrect method names for length or other string operations they may choose to use in

		this case because the API for the String class was given to them. Only award 4 points if student uses incorrect names.

To compute the student's score on the exam, you should add up the total points earned and divide by the total number of points on the test (354 if using all the questions on the exam), and then multiplying by 100 to get a percentage score.

Appendix C

Reviewer Questionnaire

Thank you for agreeing to review this assessment instrument for CS1-CS2.

As you may know, the exam is designed to be paradigm-independent, so a student who takes an objects-first, imperative-first, or functional-first CS1 should be able to successfully complete this exam.

However, the creation of this exam presented two unique challenges. First, there is a lot of material in the first year courses. Some of the material may not be represented in the exam due to time considerations. This is a three-hour, pencil and paper exam, which limited the scope and amount of questions that could be on the exam.

Second, it is important to remember the nature of programming-first approaches to CS1-CS2. Many of the questions on this test needed code examples or require the student to write code. Therefore, a language needed to be chosen for the exam. This version of the exam is a Java version. It is expected that future instructors who use the exam will be able to change the language to one that is most appropriate for their students.

To complete the review of this exam, I would appreciate your answers to the following questions. You only need to provide me with an electronic version of your responses to the questions, so please feel free to insert them directly into this document after each question.

In addition, if you feel you have other comments to offer about the exam or about particular questions, please feel free to give comments directly on the exam itself (using the Word commenting features or whatever method you'd prefer). If you have commented, please send the commented version back to me. Otherwise, all you need to send is this questionnaire.

Thank you again.

Name of Reviewer:

Institution:

Language used in CS1/CS2:

Approach used in CS1/CS2 (objects-first, etc):

1. What is your overall impression of this exam? Please feel free to speak specifically about the difficulty of questions, the nature of the questions (analysis of code, versus code generation, etc), the type of questions (multiple-choice, short answer, etc) or any other impressions you have about the exam.

2. Aside from any language issue, would you feel comfortable giving this exam at the end of your CS2 course? With the present content, do you believe it adequately covers the material presented during your first year courses? If you have reservations about giving the exam, what are they?

3. Are there any topics that you feel are given too much coverage in the exam?

4. Are there topics that you feel are missing that would dramatically improve the exam without extending it beyond the constraints of a three-hour paper and pencil exam?

5. If you have any comments, criticisms, or suggestions about a particular question or directions for a particular question set, please indicate those here. Please include question numbers.

6. Please add any additional comments here

Appendix D

Demographic Questionnaire

Please take a moment to fill out these demographic questions before beginning the exam. Please do not place your name or person number on this demographic questionnaire. If there is a question you do not feel comfortable answering, please leave the answer blank. There are a total of 15 questions.

1. What is your gender?
    a. Male
    b. Female

2. What is your age?
    a. 18
    b. 19
    c. 20
    d. 21
    e. 22
    f. 23
    g. 24
    h. 25 – 29
    i. 30 – 34
    j. 35 – 39
    k. 40 – 44
    l. 45 – 49
    m. 50 and over

3. What is your year in school?
    a. Freshman
    b. Sophomore
    c. Junior
    d. Senior

4. What is your major?
   a. Computer Science
   b. Computer Engineering
   c. Other (Please indicate)

   ____

5. If you are not a computer science/engineering major, are you planning to pursue a minor?
   a. YES
   b. NO

6. Did you take CSE 115 at UB? (If no, skip to question 9)
   a. YES
   b. NO
7. Which semester did you take CSE 115?

8. Did you ever fail CSE 115?
   a. YES
   b. NO

9. If you did not take CSE 115 at UB, why not?
   a. AP credit
   b. Transfer credit from another school

      Give name of school: ____

   c. Other (please state reason – like “not a major”)

   ____

   ____

10. Did you take CSE 116 at UB? (If no, skip to question 13)
   a. YES
   b. NO

11. Which semester did you take CSE 116?

12. Did you ever fail CSE 116?
    a. YES
    b. NO

13. If you did not take CSE 116 at UB, why not?
    a. AP credit
    b. Transfer credit from another school

       Give name of school: ____

    c. Other (please state reason – like “not a major”)

       ____

       ____

14. Please circle the number of years programming experience you had with the following languages prior to taking CSE 115 and CSE 116 (or equivalent courses). If you did not program in that language prior to CSE 115 & CSE 116, do not circle any answer. There is space at the end to insert other languages not given in the list.

C
a. 1 year
b. 2 years
c. 3 years
d. 4+ years

C++
a. 1 year
b. 2 years
c. 3 years
d. 4+ years

Java
a. 1 year
b. 2 years
c. 3 years
d. 4+ years

HTML

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

Perl

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

JavaScript

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

VB

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

VBScript

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

Fortran

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

BASIC

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

Assembly

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

Other (Give Name)

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

Other (Give Name)

a. 1 year
b. 2 years
c. 3 years
d. 4+ years

15. What was the first programming language you programmed in ever?

a. C
b. C++
c. Java
d. VB
e. Basic
f. Other (Please state):

Appendix E

Analysis of Raters of Exam

Question 5

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2003	1	4	0	2	The student did not demonstrate proper creation of iterator nor use it to cycle through the elements (called collectionName.getNext() instead of iterator.getNext()).
	2	0			
2009	1	4	0	2	Student did not use iterator at all.
	2	0			
2049	1	4	0	2	Student did not use iterator at all.
	2	0			
2050	1	4	0	2	Student did not use iterator at all.
	2	0			
3030	1	8	4	Neither	The mechanism for looping with the iterator is incorrect, but the other aspects are good.
	2	0			
3133	1	4	4	1	Student does not demonstrate knowledge of what for-each loop does by their parameter to setColor, they are trying to access the element in the collection by a color (the type, not a value).
	2	8			
3253	1	8	0	2	Student does not use iterator properly, using a counter for looping and does not call setColor appropriately or even pass in a parameter. Too many errors
	2	0			

3261	1	0	0	1	Student does not use iterator properly and does not access elements from the collection properly in neither case using the name of the collection, but rather the type.
	2	4			
3262	1	4	0	2	Student uses name of collection as name of iterator, not proper looping with iterator, not proper call to setColor.
	2	0			
3276	1	8	8	1	Student uses type instead of name of collection in for-each loop, but still knew the form of the loop.
	2	4			
3352	1	4	0	2	Student not appropriately using iterator and loop combination. Calling setColor and passing in the iterator.
	2	0			
3399	1	0	0	1	Student uses name of collection as name of iterator, not proper looping with iterator, not proper call to setColor.
	2	4			

Question 9

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2002	1	8	8	1	Student penalized too harshly for not returning array.
	2	4			
2003	1	4	4	1	Student creates new array of bigger size appropriately but does not demonstrate knowledge of copying elements from old array to new one.
	2	0			
2004	1	4	8	2	Student penalized too harshly for not sizing array to double in the new array.
	2	8			
2007	1	4	8	2	Student put [] in front of array, one grader took off half credit for this.
	2	8			
2009	1	4	8	Neither	Student penalized for using

	2	0			System.arrayCopy. This is appropriate for this question.
2015	1	4	8	2	Student penalized too harshly for not sizing array to double in the new array. The new size was still large enough.
	2	8			
2022	1	4	4	1	Student does not correctly create the new, bigger array.
	2	8			
2025	1	4	8	2	Student penalized too harshly for not sizing array to double in the new array. The new size was still large enough.
	2	8			
2028	1	4	8	2	Student penalized too harshly for not sizing array to double in the new array. The new size was still large enough.
	2	8			
2030	1	4	8	2	Student penalized too harshly for not sizing array to double in the new array. The new size was still large enough.
	2	8			
2038	1	4	8	2	Student penalized too harshly for not sizing array to double in the new array. The new size was still large enough.
	2	8			
3109	1	8	8	1	Student mixed array notation and method calls. However, they created the new array appropriately and did move the elements, but in a syntactically incorrect way.
	2	4			
3124	1	8	4	2	Student uses clone to copy array, which would be ok, if not perfect syntax except for the fact that they assign the reference from the new array to the clone of the original, thereby erasing the resizing. Student has some understanding, but not full understanding.
	2	4			
3131	1	4	8	2	Student code has all required parts and is actually syntactically perfect as well. Simple grader error.
	2	8			
3135	1	8	8	1	One grader too harsh about correct syntax for System.arrayCopy.
	2	4			

3185	1	4	4	1	Student using for-each inappropriately with array. Believes it loops through indices, but it does not.
	2	8			
3186	1	8	4	2	Student did not appropriately demonstrate knowledge of copying from old array to new array.
	2	4			
3308	1	8	4	2	Student did not appropriately demonstrate knowledge of copying from old array to new array.
	2	4			
3329	1	8	4	2	Student did not appropriately demonstrate knowledge of copying from old array to new array.
	2	4			
3352	1	4	8	2	Student penalized too harshly for not returning array.
	2	8			
3353	1	0	4	2	Grader error – student should have received half.
	2	4			
3372	1	4	4	1	Student has half of the themes – creating an array of bigger size, but did not demonstrate knowledge of copying from old array to new array.
	2	0			
3376	1	8	8	1	Student penalized too harshly for not returning array.
	2	4			
3395	1	8	4	2	Student did not create the array.
	2	4			
3399	1	0	0	1	Student has no clear understanding of syntax of arrays. Brackets everywhere with no rhyme or reason. Too much confusion – zero points.
	2	4			
3423	1	8	8	1	Student penalized too harshly for not returning array.
	2	4			
3438	1	4	0	2	Student clobbers over reference to array by reassigning it to old array – demonstrating lack of fundamental knowledge about arrays.
	2	0			
3460	1	8	4	2	Student did not appropriately demonstrate knowledge of copying from old array to new array.
	2	4			
3475	1	4	4	1	Grader error – student should have received half.
	2	0			

Question 10

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2007	1	4	8	2	Student put [] in front of array name. One grader took off half credit for this.
	2	8			
3052	1	4	0	2	Declares private variables in method, does not create array of appropriate size, does not use indices of array properly.
	2	0			
3069	1	8	4	Neither	Creating an iterator that is not used in the code (not needed and not appropriate). However, student created array of appropriate size and attempted to put the correct values in it, but using a method instead of the index, so half credit is most appropriate.
	2	0			
3177	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3186	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3203	1	8	4	2	Student inserted incorrect value into array.
	2	4			
3220	1	8	4	2	Student inserted incorrect value into array.
	2	4			
3225	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3262	1	4	4	1	Creates array incorrectly, but then inserts properly.
	2	0			
3276	1	8	4	2	Student did not do assignment into array, but knew that cubing was important. Too much confusion about issue for full credit.
	2	4			
3302	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3329	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3372	1	8	4	2	Not appropriately using loop variable as

	2	4			index and calling a weird getIndex() method.
3374	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3382	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3423	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3438	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3447	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			
3460	1	8	8	1	Grader too harsh about syntactic issue with inserting into array.
	2	4			

Question 11

Exam Number	Rater Number	Original Grade	Correct ed Grade	Correct Rater	Discussion of Conflict Resolution
2025	1	4	8	Neither	Basic find structure present using two loops even if syntax not perfect. Graders too harsh about syntactic issues.
	2	0			
2034	1	8	8	1	One grader believed that the code would not work – in fact it is syntactically perfect.
	2	4			
2036	1	8	8	1	One grader believed that the code would not work – in fact it is syntactically perfect.
	2	4			
2038	1	4	4	1	Student shows a nested looping structure, but believes that exceptions will be thrown if and if-statement does not evaluate to true. Fundamental issues about code that deserved only half credit.
	2	0			
3076	1	4	8	2	Basic find structure present using two loops even if syntax not perfect. One grader was too harsh about syntactic issues.
	2	8			

3171	1	4	8	2	One grader too harsh about syntactic issues with looping structure.
	2	8			
3217	1	0	4	2	Student believed that for-each loop would work and did not have two loops, but basic find structure is there.
	2	4			
3261	1	0	4	2	Student believed that one loop would work if based on size of the structure, but basic find structure is there.
	2	4			
3374	1	8	4	2	Student believed that one loop would work if based on size of the structure, but basic find structure is there – should have been a deduction for not using two loops.
	2	4			
3395	1	4	0	2	Student using double as name of array, not array name.
	2	0			
3438	1	0	4	2	Student believed that one loop would work if based on size of the structure, but basic find structure is there.
	2	4			
3447	1	0	4	2	Student believed that one loop would work if based on size of the structure, but basic find structure is there.
	2	4			

Question 12

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2007	1	0	0	1	Student did not handle head case. Loop is present, but the reference does not seem to advance. Does not link up around deleted node properly.
	2	4			
2009	1	0	4	2	Student has loop for finding node in place, but other issues not properly addressed.
	2	4			
3005	1	8	4	2	Not covering head case (missed by one grader)
	2	4			

3066	1	0	4	Neither	Received zero points initially because the grader thought that the structure did not contain a loop to go through the list. However, it does, but the fundamental structure of the code is an if-else, where the else case basically says loop if _head is null. There is also a problem with the advancement of the reference throughout the list. However, there is enough internal to the code to award half credit, but not full credit because of the two above mentioned errors. .
	2	8			
3076	1	0	0	1	Student calling remove method from their code – this is the method they were supposed to write.
	2	4			
3120	1	0	0	1	Student resetting this to go through the list, which would not be appropriate as the original this in the method points to a list and they are then assigning that reference to a node. They are also not handling the head case, or checking for nulls.
	2	8			
3135	1	0	4	2	Student used a construct described in class to solve this problem (a Visitor). This list does not support a visitor, which the student pointed out, but proceeded to use anyway. However, if the list accepted a visitor, the code is almost perfect, warranting half credit. This is a case where domain knowledge of way the courses are taught comes in handy.
	2	4			
3164	1	4	4	1	Head case missing..
	2	8			
3253	1	4	4	1	Head case missing. However, loops through and would work for other cases.
	2	0			
3312	1	4	8	2	Has head case and looping structure. No null cases checked, but deserving of full credit.
	2	8			
3321	1	8	4	2	Head case missing.

	2	4			
3359	1	4	0	2	No loop in code.
	2	0			
3409	1	8	4	2	Head case missing.
	2	4			

Question 23

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2007	1	8	8	1	Student demonstrates why the inheritance would break the stack invariant.
	2	4			
2011	1	8	8	1	Student demonstrates why the inheritance would break the stack invariant.
	2	4			
2014	1	8	8	1	Student demonstrates why the inheritance would break the stack invariant.
	2	4			
2025	1	8	4	Neither	Student expresses what inheritance will do, but not what problems it would cause.
	2	0			
2028	1	4	4	1	Student does not express the nature of the inheritance relationship accurately in the answer.
	2	8			
2038	1	8	8	1	Student demonstrates why the inheritance would break the stack invariant.
	2	4			
2046	1	4	0	2	Student does not express either idea appropriately.
	2	0			
2049	1	4	4	1	Student expresses stack invariant property, but not enough about what would happen in the inheritance.
	2	8			
3081	1	0	0	1	Student asserts that stacks are ordered and can be popped at the top or pushed at the bottom, demonstrating a fundamental misunderstanding of the concept of a stack.
	2	4			
3095	1	4	4	1	Student did not articulate clearly what the invariant of a stack should be or why the inheritance gives the “user too much power”.

	2	8			However, there is some level of understanding that this would be inappropriate, so half is more appropriate.
3120	1	4	4	1	Student expresses difference between stack and vector and that inheritance would not be appropriate because of inheriting methods. Ideas not articulated clearly enough to illustrate full understanding, but enough to allow for half credit.
	2	0			
3146	1	0	0	1	Student does not show understanding of issue with why this is a problem.
	2	8			
3164	1	4	4	1	Student demonstrates belief that this should not be allowed, but does not articulate why it would be inappropriate.
	2	8			
3171	1	4	4	1	Student shows that you should not be allowed to access the stack internally, but does not clearly state why you would be allowed to do so using inheritance.
	2	8			
3174	1	0	4	Neither	Student demonstrates that stack should not be accessed in the middle,but not articulated enough of the main ideas for full credit.
	2	8			
3177	1	4	8	2	Student does demonstrate understanding of issues.
	2	8			
3187	1	8	4	2	Student does not indicate why you would not want the stack to inherit the methods from vector.
	2	4			
3193	1	8	0	2	Upon a second reading of the answer, the grader who gave full credit realized that student does not express either of fundamental ideas needed for full credit.
	2	0			
3220	1	4	4	1	Student discusses how inheritance works, but not about the appropriate invariant for a stack.
	2	8			
3262	1	8	4	2	Student demonstrates understanding of what inheritance will allow, but not why it is bad.
	2	4			
3382	1	0	4	2	Student demonstrates understanding of what inheritance will allow, but not why it is bad.
	2	4			

3389	1	0	0	1	Student demonstrates facts about a stack, but cannot put them together to answer question. Zero appropriate.
	2	4			
3399	1	0	0	1	Student demonstrates facts about a stack, but cannot put them together to answer question.
	2	4			
3401	1	4	4	1	Student demonstrates understanding of what inheritance will allow, but not why it is bad.
	2	8			
3404	1	0	0	1	Student demonstrates facts about a stack, but cannot put them together to answer question.
	2	4			
3409	1	8	8	1	Student does express both ideas needed for full credit.
	2	4			
3447	1	4	4	1	Student demonstrates knowledge of stack invariant, but not why inheritance will break that.
	2	8			

Questions 57 and 58

Exam Number	Rater Number	Original Grade (57)	Original Grade (58)	Correct ed Grade	Correct Rater	Discussion of Conflict Resolution
2004	1	4		4	1	One of the graders did not give credit when student wrote n=1, n=2, they wanted the entire statement. However, if students followed through on 58 with n > 2, clearly indicating the difference between the base case and recursive cases.
	2	0				
2007	1	4	4	4 (57) 4 (58)	1 (57) 1 (58)	(57) One of the graders did not give credit when student wrote n=1, n=2, they wanted the entire statement. However, if students followed through on 58 with n > 2, clearly indicating the

	2	0	0			difference between the base case and recursive cases. (58) Follow through due to answer on 57.
2009	1	4		4	1	Student uses the values of n as the answer, but shows the recursive case for 58, putting the three pieces in the correct positions.
	2	0				

2011	1	4	4	4(57)
				4(58)
					1 (57)
					1 (58)
						(57) One of the graders did not give credit when student wrote n=1, n=2, they wanted the entire statement. However, if students followed through on 58 with n > 2, clearly indicating the difference between the base case and recursive cases.
						
						(58) Follow through due to answer on 57.

	2	0	0			
2014	1	4	4	4 (57) 4 (58)	1 (57) 1 (58)	(57) One of the graders did not give credit when student wrote n=1, n=2, they wanted the entire statement. However, if students followed through on 58 with n > 2, clearly indicating the difference between the base case and recursive cases. (58) Follow through due to answer on 57.
	2	0	0			
2015	1	4	4	4 (57) 4 (58)	1 (57) 1 (58)	(57) Student uses the values of n as the answer and expresses n > 2 as recursive case, which puts the pieces in the correct places. (58) Follow through due to answer on 57.
	2	0	0			

2025	1	4	4	4 (57) 4 (58)	1 (57) 1 (58)	(57) One of the graders did not give credit when student wrote n=1, n=2, they wanted the entire statement. However, if students followed through on 58 with n > 2, clearly indicating the difference between the base case and recursive cases. (58) Follow through due to answer on 57.
	2	0	0			
2034	1	4	4	4 (57) 4 (58)	1 (57) 1 (58)	(57) Student uses L(1) and L(2) for 57 and L(n) for n > 2 for 58, showing the pieces in the correct places. (58) Follow through due to answer on 57.
	2	0	0			
2036	1	4		4	1	Student uses L(1) and L(2) for 57 and entire recursive case for 58, showing the pieces in the correct places.
	2	0				
2050	1	4	4	4 (57) 4 (58)	1 (57) 1 (58)	(57) Student uses L(1) and L(2) for 57 and n > 2 for 58, again showing pieces in correct places. (58) Follow through due to answer on 57.
	2	0	0			
3033	1		0	4	2	Grader error.
	2		4			
3186	1	4	0	4 (57) 0 (58)	1 (57) 1 (58)	(57) Student's notation for base case weird, but idea expressed. Full credit appropriate. (58) Student did not express n > 2, but rather that n != 1 and n!= 2, which demonstrates a fundamental mis-understanding of sequences.
	2	2	2			

3187	1	0	0	2 (57) 2 (58)	2 (57) Neither (58)	(57) Student gives one of the correct base cases, but not both. (58) Student not using correct notation for recursive case, simply writes L(n), but since did not get 57 correct, not sure if student has complete understanding of when the recursive case is appropriate.
	2	2	4			
3189	1		2	0	2	Student lists all cases as recursive case. No understanding of issue. Zero appropriate.
	2		0			
3238	1		4	2	2	Student not using correct notation for recursive case, simply writes L(n), but since did not get 57 correct, not sure if student has complete understanding of when the recursive case is appropriate.
	2		2			
3260	1		4	0	2	Grader error – misread student response
	2		0			
3312	1	2		4	2	Student mis-copied one of the base cases, one grader took off.
	2	4				
3423	1	2		4	2	Student mis-copied one of the base cases, one grader took off.
	2	4				
3447	1		4	4	1	Grader error – misread student response.
	2		0			
3460	1		0	0	1	Student did not express n > 2, but rather that n != 1 and n!= 2, which demonstrates a fundamental mis-understanding of sequences.
	2		4			

Question 59

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2001	1	4	0	2	Student confuses recursive definition and the name of the method. Asks if

	2	0			L(1) = 1 and then does not call method again in recursive case, but calls L, zero points appropriate.
2007	1	4	8	2	Student has valid recursive method, but a helper method that calls it. Helper method would not return the appropriate value. However, the student demonstrated the ability to turn the recursive formulas into a method.
	2	8			
2011	1	4	0	2	Method is actually not recursive, even though base cases are there.
	2	0			
2014	1	4	4	1	Uses the new switch-else construct. Even though it is a syntax issue, the students should know that those two do not mix.
	2	8			
2028	1	4	4	1	Student mixes switch and if together, but also does not call method in recursive call, but rather copies formula.
	2	8			
2046	1	4	0	2	Method does not return anything. Switch/otherwise construct used – does not call method, but copies the formula. Too many errors – zero points.
	2	0			
3052	1	4	0	2	Upon another look at code, there is no structure, no method header or body evident. Base cases not used, and not really recursive.
	2	0			
3065	1	0	0	1	Base cases not handled appropriately and method is not recursive.
	2	4			
3074	1	0	0	1	The method is not recursive as both graders noted, but student not appropriately penalized.
	2	4			
3076	1	0	0	1	Student not appropriately handling base cases, which points to a fundamental lack of understanding of how recursion works.
	2	4			
3133	1	0	4	2	Method is recursive and does have a test for a base case, but it is not the correct base case. However, it is what

	2	4			the student believes is the base case as evidenced by their answer to question 57.
3171	1	0	0	1	Method not recursive – grader noted, but did not deduct properly.
	2	4			
3220	1	0	0	1	Student not appropriately handling base cases, which points to a fundamental lack of understanding of how recursion works.
	2	4			
3225	1	4	4	1	Base case and recursive case switched in the code, demonstrating that student knew they were important, but has them in incorrect order.
	2	8			
3261	1	4	4	1	There is a base case and recursive case present, but will not execute correctly, method has void return type.
	2	8			
3262	1	0	0	1	The method is not recursive as both graders noted, but student not appropriately penalized.
	2	4			
3352	1	8	0	Neither	Looking at the code again showed that the method was not recursive at all meaning that zero points is more appropriate.
	2	4			
3475	1	0	0	1	The method is not recursive as both graders noted, but student not appropriately penalized.
	2	4			

Question 101

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2007	1	4	4	1	ReadLine called twice.
	2	0			
2009	1	8	4	2	ReadLine called twice.
	2	4			
2011	1	4	0	2	Student not inserting into array list – throwing and catching exceptions – calling readLine twice and calling output.
	2	0			
2030	1	4	4	1	Student had improperly inserted into array

	2	0			list and was penalized too harshly for it.
2036	1	8	4	2	ReadLine called twice.
	2	4			
2038	1	4	8	Neither	Student believed that readLine() will throw an exception at end of file, which is doesn't, but if it did, code would work.
	2	0			
2049	1	8	4	2	ReadLine called twice.
	2	4			
3033	1	8	4	2	ReadLine called twice.
	2	4			
3052	1	4	0	1	Creating private, static and final variables inside a method. Arbitrarily deciding that a file has a max size of 50. Calling readLine() on the filename (a String), not the BufferedReader
	2	0			
3065	1	8	4	2	ReadLine called twice.
	2	4			
3069	1	4	0	2	Not appropriately looping through file. Not reading in appropriately (using charAt, which is not defined on a BufferedReader).
	2	0			
3074	1	8	4	2	Looping on length of the string that represents the filename, not on the length of the file. If student had run loop on in.length, full credit would have been appropriate, but in this case, half is appropriate.
	2	4			
3095	1	4	0	2	No loop in code, a fact missed by one of the graders.
	2	0			
3185	1	4	8	Neither	Student uses loop to go through file and uses readLine to read. Inserts into array list with incorrect syntax. Main points covered.
	2	0			
3225	1	0	0	1	Student looping through the strings in the filename (a string), not the BufferedReader and not calling read method.
	2	4			
3257	1	4	4	1	ReadLine called twice.
	2	8			
3260	1	8	4	2	ReadLine called twice.
	2	4			
3262	1	8	4	2	ReadLine called twice.
	2	4			

3312	1	4	8	2	Student believed that readLine() will throw an exception at end of file, which is doesn't, but if it did, code would work.
	2	8			
3323	1	4	0	2	Student used for each loop (array : in.readLine()). Fundamental structure not correct.
	2	0			
3399	1	8	0	2	Student trying to read from filename, which is a string.
	2	0			
3401	1	4	4	1	Student has loop for reading (although really far from correct) and call to readLine.
	2	0			
3423	1	8	4	2	Student runs loop on length of the filename (a string) instead of the file.
	2	4			
3432	1	8	4	2	ReadLine called twice.
	2	4			
3447	1	4	4	1	Student using eof (which is not Java) which would be fine for full credit if the condition was to keep reading until !eof, but the student keeps going on eof.
	2	8			

Question 102

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2011	1	4	0	2	Student using readLine and not appropriately looking in collection.
	2	0			
2014	1	4	8	Neither	Both graders putting too much emphasis on syntax. Basic themes present.
	2	0			
2020	1	4	8	2	Code correct – grader error.
	2	8			
2028	1	4	8	Neither	Both graders putting too much emphasis on syntax. Basic themes present.
	2	0			
2036	1	4	8	2	Code correct – grader error.
	2	8			
2050	1	4	8	Neither	Both graders putting too much emphasis on syntax. Basic themes present.
	2	0			
3029	1	4	8	2	Boundary conditions were given too much weight in question.
	2	8			

3069	1	4	8	2	Student used array index notation and one grader deducted for this mistake.
	2	8			
3124	1	8	8	1	Grader too harsh about returning.
	2	4			
3171	1	4	8	2	Grader too harsh about boundary conditions.
	2	8			
3186	1	0	4	2	Weird indicator of the correct size of the string, but basic structure present.
	2	4			
3189	1	4	8	2	Grader too harsh about syntax for inserting into arraylist.
	2	8			
3217	1	4	8	2	Grader too harsh about boundary conditions.
	2	8			
3253	1	4	4	1	Goes through array list to find length of each string. Syntax errors too great for full credit.
	2	0			
3262	1	8	4	2	Knows to check length of string for bounds and increments count correctly, but does not have syntax even close for looping through the array list.
	2	4			
3321	1	4	8	2	Grader too harsh about boundary conditions.
	2	8			
3323	1	8	8	1	Grader too harsh about correct access to array list.
	2	4			
3359	1	4	8	2	Student used or instead of and, and was deducted.
	2	8			
3389	1	0	0	1	Student not looping through collection properly or using length method – adding collection to another collection.
	2	4			
3401	1	4	8	2	Grader too harsh about returning.
	2	8			
3432	1	8	8	1	Grader too harsh about returning.
	2	4			
3438	1	8	4	2	Uses collection name as the name of an iterator.
	2	4			
3475	1	4	4	1	Student checking length on collection, not strings inside, so not looping properly.
	2	8			

Question 103

Exam Number	Rater Number	Original Grade	Corrected Grade	Correct Rater	Discussion of Conflict Resolution
2009	1	4	8	2	One grader took off for incorrect call to size.
	2	8			
2014	1	4	8	Neither	Graders too harsh about syntax – basic themes present.
	2	0			
2020	1	4	8	2	Code correct – grader error.
	2	8			
2028	1	4	0	2	Question not finished. Not enough info to award credit.
	2	0			
2036	1	8	8	1	Code correct – grader error.
	2	4			
2038	1	4	8	2	Code correct – grader error.
	2	8			
2049	1	4	8	Neither	Graders too harsh about syntax – basic themes present.
	2	0			
2050	1	4	4	1	Student demonstrates knowledge that looping through collection is important and then looking for Ps important, but not using correct methods from API.
	2	0			
3029	1	4	4	1	Loop counters not correctly incremented/reset. Does not show proper use of loops.
	2	8			
3030	1	4	4	1	API not used correctly, calling a method on a char that is really for a String.
	2	8			
3069	1	4	4	1	API not used correctly because student believes that charAt returns a String, and begins looping through string indices at 1, which is not correct.
	2	8			
3091	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			
3109	1	8	8	1	One grader felt that the student using a for-each loop for a string was inappropriate. After considering the main themes of the question, it was felt is was reasonable for the student to make this error that would be

	2	4			caught by compiler because student knew that looping through each character of string was important step in looking for the letter p.
3120	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			
3135	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			
3164	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			
3185	1	0	4	2	Student only loops through the string, not each string in the collection.
	2	4			
3186	1	4	8	2	Thought student was not looking for both cases of p. Upon re-examination of the code, it was found that the student did not understand the question and was actually looking for the number of occurrences of the string “Ps” in the strings. The student did this correctly and should receive full credit for the question.
	2	8			
3189	1	4	8	2	Grader too harsh about syntactic issues with accessing elements in array list.
	2	8			
3217	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			
3253	1	4	0	2	Student writes a loop with a comment inside to “count P’s”, which is what they needed to demonstrate in the question.
	2	0			
3260	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			
3302	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			
3321	1	4	8	2	Code correct – grader error.
	2	8			
3329	1	0	4	2	Only inner loop present to look for Ps in

	2	4			string, not to loop through all strings in collection.
3352	1	0	4	2	Only inner loop present to look for Ps in string, not to loop through all strings in collection.
	2	4			
3368	1	4	8	2	Grader did not notice that student was handling both cases of p.
	2	8			
3401	1	4	8	2	One grader being too harsh about the checking for both upper case P and lower case p.
	2	8			