

PROGRAMMING WITH CONCURRENCY:
BARRIERS TO LEARNING AND EXPLORATIONS IN TEACHING

by

ZHEN LI

(Under the Direction of Eileen Kraemer)

ABSTRACT

Programming activities are not trivial tasks. Rather, to carry out such complex problem solving tasks, programming expertise developed through long-term experience is required. In contrast, the current landscape of computing demands more reliable and efficient implementations of concurrent software programs. In this thesis work, we research topics in psychology of programming and computer science education with an emphasis on programming with concurrency to inform the fields of psychology of programming, computer science education, empirical software engineering and a broader scope of human factors related fields.

We identify the barriers to learning programming with concurrency through review and empirical work. We synthesize the previous research findings with regard to programming expertise, generalize a conceptual framework of the development and application of programming expertise and indicate the importance of the knowledge repository component. We reveal the structure of concurrency-related concepts, and provide insight into the acquisition procedures for such knowledge with our description of a “misconception hierarchy” grounded from qualitative analysis of empirical data. Comprehensive arguments generated through a case study are further provided to describe non-concurrency-related barriers that are critical for students to learn and appreciate programming with concurrency.

We conduct explorations on the impact of existing and innovative techniques and course designs used in teaching programming with concurrency topics. We review the pedagogical impact of pair programming and indicate its protective effect on retaining female and less-experienced students through our quasi-experiments. We also reveal the engineering and cognitive effect of pair programming in that pair programming helps students to write code using better style and promotes more comprehensive and critical thinking in earlier phases of software development. We survey curriculum guides on topics regarding programming with concurrency and identify two concurrency models (shared memory versus message passing), three implementation approaches (Threads, Actors, and Coroutines), and several classic scenarios (Bounded Buffer, Dining Philosopher, Sleeping Barber, etc.) to teach in an upper-level undergraduate computer science course. We provide feedback on benefits and drawbacks of this series of pedagogical innovations including flipped classroom design, using a language-independent pseudocode system, and introducing repeated practice with different implementation approaches on a single problem.

INDEX WORDS: Psychology of Programming, Computer Science Education, Cognition, Expertise, Concurrent System, Empirical Software Engineering

PROGRAMMING WITH CONCURRENCY:
BARRIERS TO LEARNING AND EXPLORATIONS IN TEACHING

by

ZHEN LI

B.E. Fudan University, Shanghai, China, 2008

B.S. University College Dublin, Dublin, Ireland, 2008

A Dissertation Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment
of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

ATHENS, GEORGIA

2013

©2013

Zhen Li

All Rights Reserved

PROGRAMMING WITH CONCURRENCY:
BARRIERS TO LEARNING AND EXPLORATIONS IN TEACHING

by

ZHEN LI

Major Professor: Eileen Kraemer

Committee: John Miller
Tianming Liu

Electronic Version Approaved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2013

DEDICATION

This work is dedicated to all programmers for their creativity in problem solving, all students for their hard work in learning programming and all teachers for their patient guidance along the demanding journey of developing expertise.

TABLE OF CONTENTS

LIST OF TABLES	VII
LIST OF FIGURES.....	X
ACKNOWLEDGEMENT	XIII
CHAPTER 1. INTRODUCTION.....	1
1.1 THESIS STATEMENTS	2
1.2 RESEARCH CONTRIBUTIONS.....	3
1.3 OUTLINE OF THESIS.....	5
CHAPTER 2. OVERVIEW OF EMPIRICAL WORK.....	7
2.1 EMPIRICAL WORK 1: SPRING 2010	7
2.2 EMPIRICAL WORK 2: SPRING 2012	10
2.3 EMPIRICAL WORK 3: SPRING 2013	13
CHAPTER 3. BARRIERS TO LEARNING	20
3.1 PROGRAMMING EXPERTISE	20
3.2 CONCURRENCY-RELATED BARRIERS	48
3.3 OTHER BARRIERS.....	77
3.4 SUMMARY AND FUTURE WORK	85
CHAPTER 4. EXPLORATIONS IN TEACHING.....	89
4.1 PAIR PROGRAMMING AS A PEDAGOGICAL TECHNIQUE	90
4.2 IMPACTS OF PAIR PROGRAMMING.....	93
4.3 CURRICULUM GUIDES AND ELEMENTS OF TEACHING CONCURRENCY.....	113
4.4 TEACHING PROGRAMMING WITH CONCURRENCY.....	128

4.5	SUMMARY AND FUTURE WORKS	140
CHAPTER 5. CONCLUSIONS AND CONTRIBUTIONS.....		142
REFERENCES		147
CHAPTER 6. APPENDIX		157
6.1	DATA AND ANALYSIS PROCESSES	157
6.2	PROGRAMS USED IN THESIS WORK	166
6.3	MATERIALS FROM SPRING 2010 WORK	170
6.4	MATERIALS FROM SPRING 2012 WORK	184
6.5	MATERIALS FROM SPRING 2013 WORK	250

LIST OF TABLES

TABLE 1 OVERVIEW OF WORK CONTRIBUTING TO IDENTIFY BARRIERS TO LEARNING	20
TABLE 2 REINTERPRETATIONS OF STUDY RESULTS ABOUT STRUCTURED VERSUS OPPORTUNISTIC DESIGN BEHAVIORS	25
TABLE 3 REINTERPRETATIONS OF STUDY RESULTS ABOUT PROGRAM PLAN THEORY	28
TABLE 4 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT MEMORY AND RECALL	32
TABLE 5 REINTERPRETATIONS OF EMPIRICAL RESULTS ABOUT OTHER MENTAL MODEL THEORY	35
TABLE 6 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT PRODUCTION VERSUS COMPREHENSION	38
TABLE 7 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT BUG GENERATION.....	39
TABLE 8 REINTERPRETATIONS OF EMPIRICAL RESULTS ABOUT HYPOTHESIS-DRIVEN DEBUGGING BEHAVIORS	41
TABLE 9 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT CHARACTERS OF DEBUGGING BEHAVIORS	41
TABLE 10 REINTERPRETATIONS OF EMPIRICAL RESULTS ABOUT DATA-DRIVEN DEBUGGING BEHAVIORS.....	43
TABLE 11 QUESTIONS WITH SIMILAR METRIC AND DIFFERENT PERFORMANCE.....	56
TABLE 12 INITIAL MISCONCEPTION PYRAMID TABLE	59
TABLE 13 SAMPLE PSEUDOCODE ELEMENTS NOT RELATED TO CONCURRENCY.....	62
TABLE 14 PSEUDOCODE EXTENSIONS ON CONCURRENCY	63
TABLE 15 PSEUDOCODE EXTENSIONS ON SHARED MEMORY MODEL OF CONCURRENCY	64
TABLE 16 PSEUDOCODE EXTENSIONS ON MESSAGE PASSING MODEL OF CONCURRENCY	65
TABLE 17 PSEUDOCODE IMPLEMENTATIONS OF BOUNDED BUFFER.....	66
TABLE 18 THE REFINED MISCONCEPTION HIERARCHY.....	71
TABLE 19 DETAILED MISCONCEPTIONS FOUND IN STUDY.....	71
TABLE 20 SUBJECT PROFILES	73
TABLE 21 CORRELATIONS BETWEEN TOTAL NUMBER OF MISCONCEPTION AND HANDICAP LEVEL.....	76

TABLE 22 CORRELATIONS BETWEEN BREADTH OF MISCONCEPTIONS AND HANDICAP LEVEL	76
TABLE 23 LINEAR REGRESSIONS OF MISCONCEPTIONS TO HANDICAP LEVEL	76
TABLE 24 LINEAR REGRESSIONS OF MISCONCETPION TO HANDICAP LEVEL (SECOND STUDY ONLY)	76
TABLE 25 SAMPLE MODIFICATIONS IN CODE HISTORY	78
TABLE 26 FRAGILE KNOWLEDGE EXHIBITED BY INTERMEDIATE AND NOVICE SUBJECTS	84
TABLE 27 OVERVIEW OF WORK CONTRIBUTING TO CONDUCT EXPLORATIONS IN TEACHING	89
TABLE 28 WITHDRAWAL COUNTS	100
TABLE 29 WITHDRAWAL COUNTS: FEMALE.....	100
TABLE 30 WITHDRAWAL COUNTS: MALE	100
TABLE 31 WITHDRAWAL COUNTS: LESS EXPERIENCED.....	100
TABLE 32 WITHDRAWAL COUNTS: MORE EXPERIENCED	101
TABLE 33 LAB PERFORMANCES OF PAIR AND SOLO PROGRAMMERS	101
TABLE 34 TA OFFICE HOUR USAGES BY PAIR AND SOLO PROGRAMMERS.....	102
TABLE 35 COURSE ORGANIZATION PREFERENCES OF PAIR AND SOLO PROGRAMMERS	102
TABLE 36 RAW LINT LOG FORMAT	107
TABLE 37 CATEGORIES OF LINT STYLE ERRORS	107
TABLE 38 PARSED LINT ERROR DATA	107
TABLE 39 NUMBER OF ERRORS PER FILE.....	107
TABLE 40 OBSERVATION DETAILS.....	111
TABLE 41 QUESTIONS ASKED BY PAIRS AND SOLOS.....	112
TABLE 42 CONCURRENCY ELEMENTS IN ACM/IEEE CURRICULUM GUIDANCE	114
TABLE 43 CONCURRENCY RELATED TOPICS FROM NSF/TCPP CURRICULUM GUIDE COVERED IN OUR TEACHING	116
TABLE 44 CONCURRENCY CONSTRUCTS AND DESIGN PROCEDURES OF THREE APPROACHES	124
TABLE 45 COVERAGE OF CURRICULUM TOPICS BY CONCURRENCY APPROACHES AND SCENARIOS.....	127

TABLE 46 COVERAGE OF CONCURRENCY RELATED CURRICULUM TOPICS	129
TABLE 47 PERFORMANCE ON MIDTERM EXAM	131
TABLE 48 BEHAVIORS, GOALS AND KNOWLEDGE REPOSITORY OF INTERMEDIATE SUBJECT.....	157
TABLE 49 BEHAVIORS, GOALS AND KNOWLEDGE REPOSITORY OF NOVICE SUBJECT	160

LIST OF FIGURES

FIGURE 1 SAMPLE QUESTION OF COMPREHENSION TEST	18
FIGURE 2 FLOWCHART OF ORDER TO COMPLETE TUTORIALS IN SPRING 2010 STUDY	18
FIGURE 3 SAMPLE TUTORIAL SNAPSHOTS IN SPRING 2010 STUDY.....	18
FIGURE 4 A SAMPLE QUIZ SNAPSHOT IN SPRING 2010 STUDY	19
FIGURE 5 CONCEPTUAL FRAMEWORK OF PROGRAMMING	25
FIGURE 6 QUALITATIVE RESEARCH METHOD.....	55
FIGURE 7 SAMPLE QUESTION IN POSTTEST OF SPRING 2010 STUDY	56
FIGURE 8 MISCONCEPTION PYRAMID.....	58
FIGURE 9 A SAMPLE QUESTION IN SHARED MEMORY MODEL PART OF TEST	68
FIGURE 10 A SAMPLE QUESTION IN MESSAGE PASSING MODEL PART OF TEST	68
FIGURE 11 SAMPLE FALL BACK TO LOWER LEVEL MISCONCEPTION	76
FIGURE 12 PAIR PROGRAMMER SPEND LESS TIME ON COMPLETING LABS.....	104
FIGURE 13 PAIR PROGRAMMERS ARE GENERALLY MORE CONFIDENT IN THEIR PERFORMANCE	104
FIGURE 14 PAIR PROGRAMMERS FEEL LESS TEMPORAL PRESSURE FOR COMPLETION OF LAB.....	104
FIGURE 15 PAIR PROGRAMMERS FEEL LESS MENTAL DEMAND THAN SOLO PROGRAMMERS.....	105
FIGURE 16 PAIR PROGRAMMERS COMPLETE LABS WITH LESS EFFORT THAN SOLO PROGRAMMERS	105
FIGURE 17 PAIR PROGRAMMER EXPERIENCE LESS FRUSTRATIONS THAN SOLO PROGRAMMERS	105
FIGURE 18 MONITOR PATTERN OF IMPLEMENTING CONCURRENCY	124
FIGURE 19 REACTOR PATTERN OF IMPLEMENTING CONCURRENCY.....	124
FIGURE 20 GENERATOR PATTERN OF IMPLEMENTING CONCURRENCY.....	124
FIGURE 21 SUBJECTIVE DIFFICULTY LEVEL AND USAGE OF THREE PATTERNS	136

FIGURE 22 PERCEIVED EXPERTISE IN JAVA AND THREAD APPROACH BEFORE AND AFTER COURSE	137
FIGURE 23 PERCEIVED EXPERTISE IN SCALA AND ACTOR APPROACH BEFORE AND AFTER COURSE	138
FIGURE 24 PERCEIVED EXPERTISE IN PYTHON AND COROUTINE APPROACH BEFORE AND AFTER COURSE	139
FIGURE 25 PYTHON HISTORY FILE ORGANIZER FOR STUDYING CODE HISTORY	166
FIGURE 26 PYTHON CODE HISTORY GENERATOR FOR STUDYING CODE HISTORY	167
FIGURE 27 PYTHON LINT OUTPUT PARSER FOR STUDYING CODE HEALTH	168
FIGURE 28 PYTHON LINT QUERY EXECUTOR FOR STUDYING CODE HEALTH	169
FIGURE 29 DEMOGRAPHIC SURVEYS FOR SPRING 2010 STUDY	174
FIGURE 30 MULTI-MEDIA TUTORIALS FOR SPRING 2010 STUDY	175
FIGURE 31 POSTTEST FOR SPRING 2010 STUDY	183
FIGURE 32 LECTURE NOTES FOR SPRING 2012 STUDY	203
FIGURE 33 LAB MATERIALS FOR SPRING 2012 STUDY.....	249
FIGURE 34 SYLLABUS OF CSCI4900 FOR SPRING 2013 STUDY	252
FIGURE 35 PSEUDOCODE GUIDE FOR SPRING 2013 STUDY.....	257
FIGURE 36 PSEUDOCODE IMPLEMENTATIONS OF SHARED MEMORY PROGRAMS FOR SPRING 2013 STUDY	267
FIGURE 37 PSEUDOCODE IMPLEMENTATIONS OF MESSAGE PASSING PROGRAMS FOR SPRING 2013 STUDY	280
FIGURE 38 LAB MATERIALS OF CSCI4900 FOR SPRING 2013 STUDY.....	329
FIGURE 39 MIDTERM EXAM OF CSCI4900 FOR SPRING 2013 STUDY.....	345
FIGURE 40 DEBUGGING CONTEST OF CSCI4900 FOR SPRING 2013 STUDY	350
FIGURE 41 FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY	356
FIGURE 42 JAVA THREADS SOLUTION TO FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY.....	358
FIGURE 43 SCALA ACTORS SOLUTION TO FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY.....	363
FIGURE 44 PYTHON COROUTINES SOLUTION TO FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY	366
FIGURE 45 SURVEY MATERIALS OF CSCI4900 FOR SPRING 2013 STUDY	380

FIGURE 46 COURSE PLAN FOR ONLINE CSCI4900	385
---	-----

ACKNOWLEDGEMENT

I am grateful to all participants in my empirical studies and all my students who provide valuable data towards the findings in this work.

I would like to acknowledge the inspirational instruction of Dr. Eileen Kraemer and the initial impetus to study human factors in computer science domain. I would also like to acknowledge my committee members Dr. John Miller and Dr. Tianming Liu for numerous illuminations that help shape this work. I would like to further acknowledge all collaborators in the Computer Science department at the University of Georgia for their support and assistance.

CHAPTER 1.

INTRODUCTION

Software psychology is a long standing field of study that deals with human factors issues surrounding the use of computer systems. One area of software psychology is the psychology of programming (POP), which focuses on human-related issues in software engineering and programming education. The history of POP dates back to the late 1970s with the realization that cognitive effects should be considered alongside computing power in the design and evaluation of technology innovations. In the first Workshop on Empirical Studies of Programmers (ESP), Shneiderman described the contributions of this area, saying that “measures of human performance in programming have become valued not only for the guidance they provide for professional or novice programmers, but also for the evidence they provide about complex human cognitive processing” (Schneiderman, 1986). Although the realm of programming has expanded greatly in the past decades, the main goals of assisting programmers and informing other human cognition related areas through the research of POP remains unchanged.

Considering the current landscape of evolution in computing hardware and system architectures, which demands more and more implementations of concurrent software, we think it is interesting to flesh out psychology of programming topics within the domain of concurrency computation. Many technological innovations have been proposed to assist programmers engaged in this new computing activity more effectively, including new language constructs such as the concurrency features of C++11 (Meredith, 2009), Java 7 (Oracle, 2009), and Erlang (Larson, 2009), as well as the earlier C++ concurrency API OpenMP (Barney, 2013) and MPI (Barney, 2013), new system and hardware architectures such as transactional memory (Herlihy & Moss, 1993) and advanced message queuing protocol for enterprise

middleware (OHara, 2007), new engineering tools such as MuTMuT (Gligoric, et al., 2010), Kendo (Olszewski, et al., 2009), and Ballerina (Nistor, et al., 2012) and new engineering schemas such as preemption sealing (Ball, et al., 2010), and communicating memory transactions (Lesani & Palsberg, 2011). Yet, less effort has been placed on researching the psychological effects of these computing advances or pedagogical innovations to better train and prepare future software engineers for the new era of concurrency. The curriculum guides on introducing topics of Parallel and Distributed Computing (PDC) into core computer science education is still at an early stage. To achieve the requirements for introducing and synthesizing parallelism and concurrency concepts into undergraduate computer science education, action to explore effective pedagogy of these topics is urgently required. As Shneiderman stated early in the first workshop of ESP, programming is a complex problem solving procedure powered by creativity that “like music, blends esthetics and technology, that high-level plan, the middle-level concepts, and the low-level details must be correct and in harmony with each other”. Thus, we think it is valuable to discover the barriers that hamper this creation and explore effective ways to better prepare our future creators.

1.1 THESIS STATEMENTS

The work of this thesis addresses the development and application of programming expertise. We focus in particular on the current landscape of pervasive and increasing emphasis on concurrent and parallel programming. Based on a review of previous empirical research, we formulate a conceptual framework for the development and application of programming expertise that unifies the results of this prior work and explains apparent contradictions among the results of some of these studies. We apply this framework to discover, in detail, barriers to learning about programming with concurrency. Finally, we develop pedagogical techniques to address these barriers and evaluate both existing approaches and our newly developed techniques.

In support of this thesis work, we conduct the following research activities:

1. Perform a much-needed systematic review of previous empirical research and formulate a conceptual framework for development and application of programming expertise.
2. Discover and describe the content and structure of students' barriers to learn about programming with concurrency through a series of empirical studies.
3. Evaluate the effectiveness of pair programming as a pedagogical technique, through a series of quasi-experiments and provide augmented empirical evidence through empirical studies.
4. Execute innovative pedagogical designs and evaluate the corresponding benefits and caveats of these various teaching innovations.

1.2 RESEARCH CONTRIBUTIONS

This thesis work has the following contributions towards psychology of programming, computer science education and software engineering fields:

1. A conceptual framework for the nature of programming expertise, how such expertise impacts the problem-solving process, and how such expertise is developed. Specifically, we identify the inter-relationship among three entities in the framework: a knowledge base, the mental representation of a problem, and external data. We identify the processes by which 1) both the knowledge base and external data are used to develop and evolve mental representation, 2) the problem-solving process employs the evolving mental representation to guide a search of the knowledge base and for additional data, and 3) the repeated construction and subsequent internalization of mental representations builds the persistent, structured, and connected knowledge that is the basis of expertise.
2. A hierarchical organization of misconceptions exhibited by students engaged in learning about concurrency and how to program with concurrency; this hierarchical structure explains the content of and development mechanism for expertise in programming with concurrency. Specifically, we

identify five levels of knowledge (description, terminology, concurrency, implementation and uncertainty). Our work reveals 1) that a lack of lower level knowledge prevents the acquisition of higher level knowledge, and 2) that a necessary phase exists in the knowledge acquisition process in which an apparent mastery of concepts sacrificed to create a simpler solution space and the resulted incorrect solution helps the reexamination and re-solidification of the correct solution and related concepts.

3. Empirical evidence of student barriers in learning about programming with concurrency. Specifically, we found that non-concurrency related programming knowledge and even natural language related knowledge are critical in the problem solving process of programming concurrent systems.
4. An evaluation of pair programming as a pedagogical technique for the development of programming expertise, and the identification of the implications of these findings. Specifically, we found that pair programming helps retain less-experienced and female students as a pedagogical intervention, encourages all students to write better styled code as an engineering technique, and stimulates students to devote cognitive effort earlier in the software design phase as a problem solving practice.
5. Innovative pedagogical techniques and an evaluation of their benefits and caveats for use in teaching programming with concurrency. Specifically, we found the benefits of repeated programming practice in developing expertise but proposed the use of conceptually-identical-superficially-different problems to remedy the issue of potential discouragement caused by repeatedly practicing with the same problem. We also developed a recommendation for integration of teaching programming of concurrent systems into different computer science courses to better meet the students with different level of expertise.
6. A comparison of three distinct approaches to concurrency, based on empirical studies with popular programming languages. Specifically, we found although the Actors approach is reported easier for

implementation and comprehension tasks, the familiarity of the Java language drives students' choice of using Threads approach. We also found that students exhibit a poor mastery of the Coroutines approach, likely due to the dominance of the subroutine paradigm in their prior experience.

7. An extended pseudocode system and evaluation of its use. Specifically, we extended a pseudocode system to cover concurrency related concepts of both shared memory and message passing models and used it to test students' comprehension of concurrency concepts independently of any programming language. We evaluated and identified caveats of imperfect syntactic design and the lack of a compiler for this pseudocode system during its usage in implementation and comprehension tasks.

1.3 OUTLINE OF THESIS

The rest of the thesis is organized as follows. In CHAPTER 2, we provide a general overview of all the empirical work we carried out that together provides rich data sets for discussions in later sections. This serves as background for the reader to reference the studies chronologically. In the two following sections, we discuss barriers to learning and explorations in teaching concurrency and its programming. These sections each first provide a review of related work, followed by a discussion of work carried out by us as either a supplement to previous work or a new discovery or innovation, and finish with a brief summary.

For the topics in CHAPTER 3, we first survey the previous empirical research results in programming expertise, formulate a general framework for the development and application of programming expertise and point out the implications of the importance of programming knowledge in section 3.1. Then in section 3.1.6, we study and discuss the detailed content and development mechanism of programming knowledge related to concurrency by describing a hierarchical structure of misconceptions. In section 3.3, we present a case study to discuss and provide further empirical

evidence of other required knowledge for solving problems and developing expertise in programming with concurrency. In section 3.4, we provide a brief summary to conclude our discussion of the various barriers to learning concurrency and its programming and a list of future work regarding this topic.

In CHAPTER 4, we first survey research in the impact of pair programming as a pedagogical technique in section 4.1. Then in section 4.2, we discuss pair programming's pedagogical effects for retaining less experienced students and promoting student's performance, engineering effects of encouraging the writing of "healthier" programs and cognitive effects of promoting more detailed design and problem solving strategies. We survey parallel and distributed computing (PDC) topics into computer science core education and propose elements to teach in section 4.3. In section 4.4, we introduce a series of pedagogical explorations, materials designs and course organizations for teaching programming with concurrency as a separate upper-level undergraduate course, with corresponding evaluations and implications. We provide a summary on our explorations in teaching concurrency concepts, its programming and some future work in section 4.5.

Finally, in CHAPTER 5, we summarize our work and findings together and reiterate our research contributions. In appendix, section 6.1 contains two tables of qualitative analysis from the case study discussed in section 3.3. All the auxiliary program codes we used in our research are listed in section 6.2. A list of related materials we created for the empirical work performed from 2010 to 2013 are appended in section 6.3 to section 6.5 as a reference for the reader.

CHAPTER 2.

OVERVIEW OF EMPIRICAL WORK

In this chapter, we provide a brief chronological overview of the empirical work we have carried out for this thesis. We present the basic elements such as time, subjects, materials, procedures and a brief list of related discussions in subsequent sections for each empirical study. The detailed data analysis methods and procedures as well as the actual findings and implications elicited from each work are threaded into later chapters for a more cohesive view. The audience shall notice that each empirical study may map to multiple discussions in sections of later chapters and one finding or result may be derived from the data of multiple empirical studies described in this chapter. Due to the nature of empirical studies that are qualitatively based and aimed to generate grounded theory, in contrast to direct hypothesis-driven experiments, and the limited opportunities to study large groups of subjects, we organize our empirical work as a combination of exploratory and analytical research activities during which rich data sets may be collected for discovery, analysis, and validation both quantitatively and qualitatively.

2.1 EMPIRICAL WORK 1: SPRING 2010

We conducted an initial study in spring 2010. This work was originally designed to evaluate the relative usability of UML 2.0 State Diagrams and UML 2.0 Sequence Diagrams that are intended to address comprehension and implementation challenges of concurrent systems.

Subjects

We sent an email announcement to all graduate and undergraduate students in the Computer Science department at the University of Georgia to recruit volunteer research participants for this study. A flyer was attached to the email message. The inclusion criteria is that all computer science graduate

students who are 18-50 years old are eligible to participate in this study and all computer science undergraduate students who are 18-50 years old and have completed the CSCI 2720 Data Structures course are eligible to participate in this study to guarantee that the recruited undergraduates have sufficient background knowledge to proceed in the study. Finally we recruited 15 Computer Science students drawn from upper-level undergraduate classes and from graduate level classes during the spring semester of 2010. Students were volunteers and were paid \$50 in total for their time.

Materials

The study materials included a demographic survey, six computer-based training modules, five pre-tests (one quiz for each of the first five training modules), and a post-test. Part I of the post-test comprised 24 comprehension questions that involved predicting and reasoning about whether a particular event could happen next in a given execution scenario of a concurrent program. These questions are similar to a combination of the “sequential questions (whether X will happen)” and “circumstantial questions (How will X happen)” as described in (Gilmore & Green, 1984). In part II of the post-test, the questions involved identifying errors, evaluating and creating models and diagrams, and writing code. The concurrent problem used in part I of the posttest is a single-lane bridge scenario. The actual implementation of the problem was not given but we specified the details of the system. It simulated a bridge system that cars from two directions use the single-lane of the bridge alternatively. The main concern of the system was to guarantee that every car had a chance to use the bridge and no car crashed on the bridge. To simplify this problem, we defined and stated in the posttest that the cars moving from left to right were red cars and those moving from right to left were blue cars. To avoid a safety violation, only one kind of car was allowed to be on the bridge at a time. Cars exited the bridge in the order in which they entered and the leading car might exit the bridge at any time. We also structured this system so that each color of car was implemented as a thread, and the shared bridge object was implemented as a monitor with two associated condition variables `okToEnter` and `okToExit`.

The basic functions for entering and exiting the bridge are `redEnter()`, `redExit()`, `blueEnter()` and `blueExit()`. Questions 1-6 are predicting and reasoning questions. Questions 7 and 8 are code debugging and writing questions. The concurrent problem used in part II of the posttest is a readers-writers scenario. Specification of threads and methods used in the system is given and we provide state diagrams of a bounded buffer scenario for reference. The actual posttest can be seen in **Figure 31**. Figure 1 shows a sample question from part I of the posttest.

Procedures

The study procedure was as follow. First, subjects were asked to provide demographic information about themselves including gender and previous academic experience. Then, subjects finished a series of computer-based tutorials that provide background training on concurrency and modeling and implementation of concurrent software. In total, 6 modules were provided and the average time to complete each module is 45-60 minutes. After each training module, subjects completed a quiz on the materials in 10-15 minutes. We maintained records of the scores of these quizzes to assign subjects randomly into three equivalent groups. We defined three subject groups as: 1) TO, Text-only group; 2) TST, Text and UML2.0 State Diagram group; and 3) TSQ, Text and UML2.0 Sequence Diagram group. The study was a 1x3 factorial design. The single factor is the representation of the concurrent system and the three levels are {TO, TSQ, TST}. Subjects were assigned to statistically similar or equivalent groups with very close or equal mean and standard deviation of the pretest (quizzes that accompanied the training modules) scores across the three groups. Figure 2 to Figure 4 illustrate a flowchart of order to complete the different tutorials, a snapshot of a sample tutorial and a snapshot of a sample quiz. After that, subjects were given a post-test consisting of four types of tasks to complete: 1) reasoning about and describing synchronization behaviors of a concurrent program; 2) identifying, reasoning about and fixing errors in concurrent software; 3) generating a code implementation of a concurrent system; 4) generating a UML model based on a textual description. The problems of the first three tasks were the

same for all three groups. However, the TO group only had access to the textual description of the system/program related to the problems while the TST and TSQ groups were also assisted with a collection of state diagrams (TST) or an equivalent set of sequence diagrams (TSQ) that models the system/program. For the last task, the TST group members were required to generate a model using state diagrams. The TSQ group members were required to generate a sufficient number of sequence diagrams to model the system. The TO group members were divided into two subgroups, A and B, to generate models using state or sequence diagrams, respectively.

Discussions of Results

The data analysis methods of this study are discussed in section 3.2.1 and the results are discussed in section 3.2.2.

2.2 EMPIRICAL WORK 2: SPRING 2012

We taught a refined version of the “C++ and Unix Systems Programming” course (CSCI 1730), a required course for CS majors, at the University of Georgia during the spring of 2012. This challenging sophomore-level course has CS2 as a pre-req or co-req and devotes the first half of the semester to C++ for Java programmers and the second half to the application of C/C++ knowledge in the context of UNIX systems programming. The course typically experiences withdrawal rates of 19-25% by the withdrawal deadline (just after the course midpoint).

Subjects

The course consisted of two sections of 30 students each, who were all undergraduate students majoring in Computer Science at the University of Georgia. Some students also had a double major or minor at the time of the study. One section was arbitrarily selected as the pair group and the pair (PP) and solo (SP) groups had similar SAT math and verbal scores and also performed similarly on the language independent pretest of computing concept knowledge.

Materials

The guidelines for pair programming developed by Williams and Kessler (Williams & Kessler, 2002) was distributed and explained to pair programmers. The guide on C++ code style (Weinberger, et al., 2010) was introduced to both sections of students. To assess the relative differences between the two sections, a language-independent pretest of fundamental introductory computing concept knowledge (Tew & Guzdial, 2011) was conducted with both sections of students. Also, students were required to complete an online demographic survey at the beginning of the course. The lab component of the course was re-designed thoroughly. Some of the lab sessions involved ungraded, hands-on exercises. Five graded labs in first half of the semester involved 1) a command line calculator program, 2) a command line matrix multiplication program, 3) a pencil and paper assessment of C++ pointers combined with a string manipulation program, 4) a drawing program involving OO design and implementation of a shape hierarchy, and 5) an exception-handling program. In the second half of the semester, we included two lectures covering concurrency concepts such as threads, shared objects, race conditions, atomic operations and the lock mechanism, synchronization mechanism and conditional waiting. Lecture slides may be found in **Figure 32**. We created four new labs to accompany the teaching of concurrency in this course. These four labs include: 1) Processes and Signals (lab 12), 2) Threads in Java and C++ (lab 13), 3) Conditional Synchronization in Java and C++ (lab 14), and 4) Sockets with C++ (lab 15). Detailed lab materials may be found in **Figure 33**. The NASA Task Load Index (Hart & Staveland, 1988), a series of questions related to the distribution of students' time and effort on the task, and their subjective satisfaction was distributed after the completion of each lab.

Procedures

All the students attended the same two 75-minute lecture meetings per week but each section of students attended a different 50-minute hands-on laboratory meeting. With this procedural setting, we minimize the timetabling issue in that all students participated in same lecture sections. Labs were held

in a 30-person teaching lab. Over the first nine weeks of the semester, students in both sections engaged in and completed five labs, three quizzes, and a midterm exam. In each lab session, after a brief introduction, students worked on the assignment for the class period and then completed the assignment outside of class. Pair-rotation was employed and new pairs were randomly assigned for each lab session. Pair programmers were instructed to work only in pairs and to schedule times to work together. After each lab was submitted, students completed the NASA Task Load Index survey. “In person grading” of laboratory exercises required students to meet with a TA to go over their submission, answer questions about their solutions, and receive feedback with a grade. Both partners of each pair programming team were required to present and answer questions in this grading session. Two of the three quizzes were given during the lab periods and one during the lecture period. A midterm exam was administered at the end of eight weeks of class, just prior to spring break, and scores were posted on the course information system within a few days of the exam. The exam was returned and the solution reviewed on the Tuesday after spring break. After that, the remaining students still attended the same section as in the first half of the semester. Due to this administrative restriction, the study focused mainly on the effect of pair programming in first half of the semester before withdrawal happened since it became hard to continuously compare the two groups after an unequal withdrawal of students from different sections.

The university’s withdrawal policy permits withdrawals with a grade of WP (withdraw passing) up until the Thursday of the week after spring break. A grade of WP does not impact a student’s GPA. Withdrawals after the deadline result in a grade of WF (withdraw failing) which counts as an ‘F’ towards the student’s GPA. In addition, students are limited to a maximum of 4 WP grades over the course of their college careers. The vast majority of entering students at the University of Georgia are on the HOPE Scholarship, which covers roughly 90% of their in-state tuition for up to 120 credit hours but requires that students maintain a GPA of 3.0. Thus, the decision to withdraw from a course is carefully

considered by UGA students due to the potential financial impact: they use up HOPE scholarship credits as well as their withdrawal quota when they register for a course and then withdraw, but they can maintain their HOPE GPA by withdrawing if they expect to receive a grade lower than B.

Discussions of Results

The data analysis methods as well as results of this empirical work regarding the pedagogical impact of pair programming are discussed in section 4.2.1.

2.3 EMPIRICAL WORK 3: SPRING 2013

We proposed and taught a new course that provided a systematic introduction to knowledge and concepts of two forms of concurrent systems (shared memory and message passing) as well as corresponding practical programming language constructs and techniques to program these systems with three approaches (Java Threads, Scala Actors and Python Coroutines). The course was designed to not only emphasize concepts in concurrency and concurrent systems, but also to provide hands-on programming practice and experience. We designed the course so that data collected from integrated course activities provided meaningful insight into students' development of expertise with programming concurrent systems as well as the costs and benefits of different pedagogical meanings and programming approaches involved in the teaching of this course.

Subjects

The subjects of this study were 11 students enrolled in the experimental new course from both upper-division of undergraduates and graduate students in the Computer Science and Math departments.

Materials

In addition to the lecture presentations, assignments and corresponding solutions (details can be found in section 6.5) that were created to teach this course, we also extended the language-independent pseudocode system (Tew & Guzdial, 2011) for both pedagogical and experimental

purposes. The extended pseudocode system included concurrency concepts in both the shared memory and message passing models. Therefore, instead of using programs implemented in actual programming language, we used this language-independent pseudocode system on the comprehension test (midterm exam of the course) to eliminate effects of programming languages.

Procedures

The following paragraphs describe the basic layout of the course. Since our study was integrated with different course elements such as lectures, assignments, quizzes, exams and surveys, this is also the procedure of our semester-long study.

During the first two weeks of the course we briefly introduced students to modern computer architectures, including multi-processor and multi-core architectures. We then provided an overview of parallel and concurrent programming, introducing two basic types of concurrent systems, shared memory systems and distributed memory systems. A primary learning objective of this portion of the course is for students to know the history of parallel and distributed computing and to comprehend the growing importance of parallel and concurrent programming given current trends in hardware development. The lab assignment in this portion of the course involved an observation of the architecture of the student's personal computer, in which students ran two pre-compiled multi-threaded Java programs (a thread pool arithmetic program and a dining philosopher program) and were asked to report on both the nature of the dining philosophers problem and the utilization of CPU, RAM, and other resources during the running of each of these programs.

Next, we spent 1 to 1.5 weeks introducing UML 2.0 state and sequence diagrams and studying how to use these diagrams to model concurrent systems. In particular, we studied the well-defined transformation from state diagrams to threads-based implementations of monitor constructs and condition variables, and a corresponding transformation to a message-passing model. The goal of this module is for students to gain experience in applying abstraction and modeling to the problems of

reasoning about concurrent systems and in mapping from models to code. The lab assignment here was to model a book inventory system using UML state diagrams. Later in the course students implement both shared memory and message passing solutions for this system.

In the next 3-4 weeks of the course we introduced concurrency issues including race conditions, conditional synchronization, deadlock, and fairness with both shared memory and message passing models with an extension of a pseudocode system developed and validated by Allison Elliot Tew (Tew & Guzdial, 2011). Use of this pseudocode allows us to evaluate student comprehension of concurrency concepts in a language-independent manner as we discussed previously. While Tew's pseudocode has been validated for language-independent measurement of CS1 knowledge, our extensions and their use for purposes of evaluating understanding of concurrency concepts is exploratory. The pedagogical objective of this portion of the course is for students to know the two types of concurrency model (shared memory vs. message passing), to comprehend the related concurrency issues (race conditions, conditional synchronization, deadlock and fairness), and to comprehend and apply the corresponding solutions to these issues (lock mechanisms vs. private data, wait and notify vs. message protocol design, and asymmetric design in concurrent systems). Another pedagogical objective is to familiarize students with the pseudocode notation so that they can use this notation to comprehend and reason about various concurrency problems and scenarios. Students completed several in-class quizzes to practice using the pseudocode notation to create or enhance models of different concurrent scenarios such as a sum & workers system (multiple workers increase a shared sum value), a bounded buffer system, a dining philosophers system and a readers-writers system. Students also modeled the book inventory system with pseudocode and used sequence diagrams to depict and reason about some critical scenarios of the system with their model. In a homework assignment, students searched for and studied different concurrent bugs (mainly through the open source MySQL bug report database). The goal of this

assignment is to promote students' understanding of concurrency concepts via these practical examples.

The portion of implementations of concurrent system of the course takes about 8-10 weeks and has three major phases. First, we introduced students to general knowledge about the Java, Scala and Python programming languages. Students in this course are already familiar with Java, but Scala is new to most of them and Python is new to many. We then focused on the threading elements of Java, the Actors elements of Scala, and the Coroutine elements of Python. Finally, we looked at some of the advanced concurrency programming elements in each of these languages. During this portion of the course we employed a "flipped classroom" approach, meaning that students learned about programming in these languages by reading and making use of online resources while at home and then engaged in actual coding in the classroom. Students first completed labs that employed basic Java, Scala and Python programming elements to become familiar with these three languages. Next, students implemented the party-matching and sleeping barber problem with Java threads, Scala Actors and Python Coroutines during in-class lab projects. Finally, students implemented the book inventory system as both a shared memory system and a message passing system. The learning objectives of this portion of the course are for students to know, comprehend, and apply knowledge of these programming languages and their concurrency constructs to implement solutions to concurrent problems.

A research and paper presentation element was conducted in parallel with the implementation of concurrent programming part. The pedagogical objective of this element is to make students aware of the difficulties inherent in programming concurrent software, the historical and practical concerns of designing development environments for these programming activities and the human factors issues involved. Paper presentations and in-class discussions are the means by which the objective is achieved. Students chose a paper that addressed concurrent or parallel software engineering issues or human

factors in programming and presented it to the class. Each student read every paper and participated in the discussion of all presented papers.

Discussions of Results

Part of this work is designed as a repetition of the spring 2010 work and the results are presented in section 3.2.3 and 3.2.4. A case study is also carried out within the framework of this empirical work to discuss other barriers to learning programming with concurrency and is presented in section 3.3. This work itself is designed to explore the benefits and caveats of different pedagogical techniques, materials and class designs to teach concurrency. These corresponding results are reported in section 4.4. An observational study is also carried out within the duration of this empirical work to further provide evidence of the impact of pair programming and the results are presented in section 4.2.3.

4. Suppose that only three threads exist in the system: **redCar1**, **redCar2** and **blueCar1**. Suppose further that **redCar1** is running and has just invoked the *redEnter()* method and the *redEnter()* method has returned. A context switch occurs and the **redCar2** thread begins running and invokes the *redEnter()* method. **redCar2**'s invocation of the *redEnter()* method has not returned.

Which of the following event sequences could happen next? Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

d. A context switch occurs, and the **redCar1** thread begins to run. **redCar1** then invokes *redExit()* and this invocation returns.

YES NO

e. A context switch occurs, the **redCar1** thread begins to run. **redCar1** then invokes the *redExit()* method and blocks on the monitor lock.

YES NO

FIGURE 1 SAMPLE QUESTION OF COMPREHENSION TEST

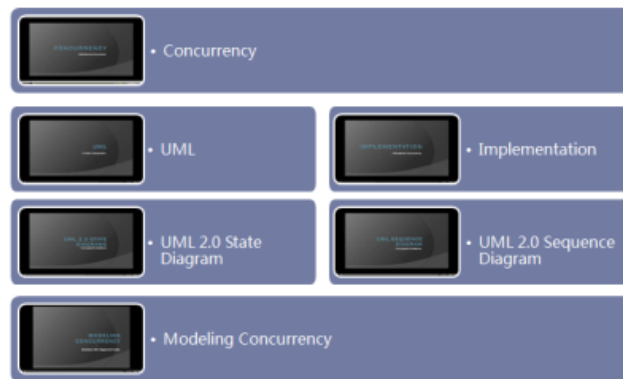


FIGURE 2 FLOWCHART OF ORDER TO COMPLETE TUTORIALS IN SPRING 2010 STUDY

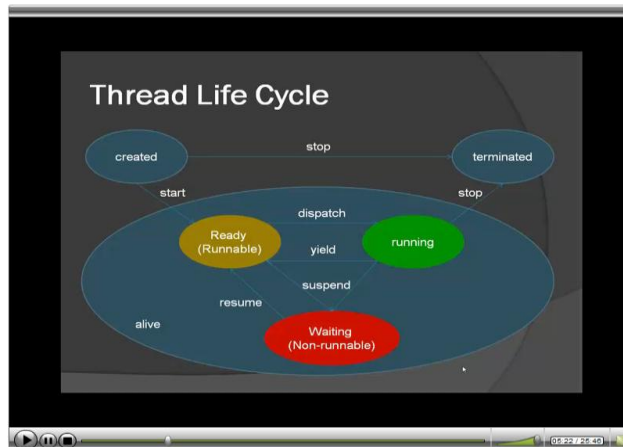


FIGURE 3 SAMPLE TUTORIAL SNAPSHOTS IN SPRING 2010 STUDY

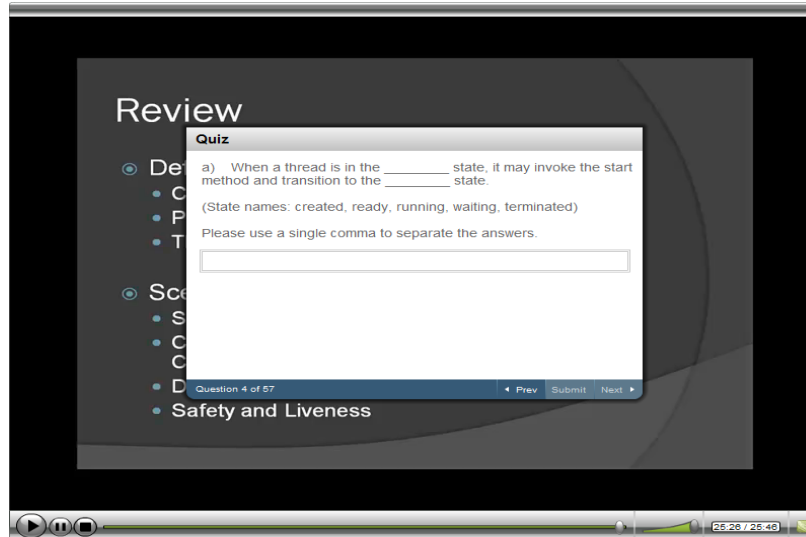


FIGURE 4 A SAMPLE QUIZ SNAPSHOT IN SPRING 2010 STUDY

CHAPTER 3.

BARRIERS TO LEARNING

Understanding the barriers to learning concurrency and to developing related expertise is the first step in our exploration. In this chapter, we review the previous empirical studies and formulate a conceptual framework of the development and use of programming expertise, identify the content and structure of concurrency related misconceptions and report on a case study that explores non-concurrency and even non-programming related knowledge that are critical barriers to learning programming with concurrency.

TABLE 1 OVERVIEW OF WORK CONTRIBUTING TO IDENTIFY BARRIERS TO LEARNING

Work	Data	Results	Section
2011-2013 literature review	previous empirical study on programming expertise	conceptual framework of acquisition and application of programming expertise	3.1
qualitative analysis of Spring 2010 posttest and Spring 2013 midterm exam	answers to sequential and circumstantial questions	misconception hierarchy of content, structure and behaviors associated with knowledge development	3.1.6
case study of Spring 2013 final exam	code history	other barriers to learning programming with concurrency	3.3

3.1 PROGRAMMING EXPERTISE

Computer software is used everywhere by contemporary society. The reliability and performance of many of these software programs are critical to daily life. Although advances in artificial intelligence have been substantial, software is still developed and maintained throughout its life cycle primarily by humans, which suggests that human factors are a significant element of software engineering.

From the initial requirements elicitation to algorithm design to code generation and maintenance, none of these tasks is trivial. Rather, they require programmers to apply complex problem-solving strategies, similar to those seen applied to solve math, physics and chess problems, to successfully achieve goals. Therefore, programming expertise, the human ability to effectively and efficiently

perform programming-related tasks across the software life cycle, becomes an important topic to study for the sake of reliability and performance of software.

Similar to any other expertise, programming expertise is gained through extensive training and long-term experience. Such expertise is applicable across a wide variety of software engineering activities, regardless of the programming language and paradigm being used. Thus, a study of the nature of such programming expertise will provide insightful information on its acquisition. Such insights have the potential to support better training of programmers who may then better create and maintain software.

Several survey papers have reviewed different aspects of the body of research that focuses on providing comprehensive engineering environments (tools, language support, team organizations, schemas and standard operating procedures) to remedy a lack of expertise or to enhance training to develop expertise. These surveys inform various academic fields from different perspectives for different audiences. For example, (Ko, et al., 2011) and (Storey, 2005) summarize studies from an engineering perspective to inform research in the development of programming environments. (Robins, et al., 2003) and (McCauley, et al., 2008) review the literature related to psychology and the pedagogy of programming and provide insights for educational purposes. (Visser, 1994) and (Ball & Ormerod, 1995) focus on a particular programming activity (i.e. debugging) and provide the psychological and cognitive insights behind this task. (Sheil, 1981) and (Moher & Schneider, 1982) critique research methods and experimental designs in an effort to promote improved research practices in empirical studies of programmers and programming activities.

We take the stance that efforts to advance software engineering techniques or to improve pedagogical approaches should be based on a solid understanding of human abilities, the abilities required to perform computer programming tasks, and the nature of these abilities. Considering the many research studies focusing on understanding programming expertise in the past thirty years, we

feel a review with the goal of formulating a general theory is strongly needed to synthesize different research results, inform the research community and make suggestions for future work.

Our selection of research work for this review is restricted by two criteria. First, one or more empirical studies on professional programmers or trainees (students) must be presented in the work. This places few restrictions on the research methods but many on research target. Programmers, who build general purpose programs or are trained to do so, are our major focus, which inherently excludes end-user programmers whose programming product is not for reuse by others. Second, one or more insights into the nature or towards a theory of programming expertise must be discussed in the work. This implies that we do not include usability studies that solely evaluate tools, language paradigms or other support for improved software engineering means. We also eliminated from our discussions in this survey those studies that solely focus on estimating the effectiveness of specific teaching techniques or curricula. Studies that focus on a discussion of team organizations, project productivity and software engineering processes are only included if they provide discussion on individual programmer's thought processes or behaviors. To conclude, we review here empirical studies that provide insights into human abilities related to professional programming tasks.

Based on the above criteria, we select a collection of over seventy papers to review. We cluster the research based on both the context of programming activity and insights into programming expertise provided accordingly. During this meta-analysis, two major threads emerge with regard to the programming activity context, program production and program comprehension, respectively. These two threads broadly cover almost all programming-related activities of the software life cycle. They are the major foci of empirical studies. Insights on programming expertise have been developed independently by researchers in these two threads. Yet, inter-related results impact one another at some focal points. Therefore, we organize our survey largely according to these two major threads and their commonalities, with clusters of research work whose results can be synthesized together into focal

points, to provide a systematic view on our iteratively and gradually formulated conceptual framework of the nature of programming expertise.

3.1.1 BACKGROUND

We notice the existence of many contradictions in the results and implications provided by the various empirical work we reviewed. Therefore, we propose a general conceptual framework to 1) reconcile the contradictory empirical results, 2) reveal the relatedness of different results, 3) re-validate the different results and 4) re-interpret the different results within the framework.

We give the following definitions to clarify the domain of programming expertise. In our context, a **program** is a series of specifications that may be executed on a computational device at some future time with various inputs (Ko, et al., 2011). **Programming** is defined as the process of planning, writing and modifying programs (Ko, et al., 2011). A **programmer** is an individual who carries out programming tasks. We define **behaviors** of a programmer as actions performed by programmers that are directly observable by a third person during programming activities. We define **strategies** of a programmer as plans of actions that are not directly observable by a third person. A **mental model (mental representation)** is the form in which a program exists in a programmer's mind and is associated with a particular program in the context of a particular programming task. A **strong mental model** is suitable for problem solving under particular conditions of program and task. **Knowledge** is a collection of information that exists in long term memory for retrieval in future problem solving. **Data** is a collection of information that in the world exists objectively for access during problem solving.

We propose that **expertise** is the possession of superior knowledge to form strong mental models that 1) better search other knowledge and data for purposes of current problem solving, and 2) better internalize data as well as elements of the mental model and the process used to construct the mental model to form knowledge for future problem solving. Our conceptual framework includes three major parts: 1) a knowledge base, 2) an ever-evolving mental model, and 3) data. The knowledge base is

neither problem nor task specific and is internally possessed by a programmer. The data is a set of external information accessible to the programmer during problem solving. Some of this information is related to the problem and solution, but usually, most of these data are noise and irrelevant. Thus, a search through data to find critical items is important to success. The mental model is problem or task specific, internally possessed by a programmer, but typically exists in temporary (short-term) memories. It includes a re-organized set of information (both from knowledge and data) that is most related to current problem solving and guides the further search of knowledge and data. The content and organization of the mental model change according to the complexity of problem solving task. A frequently constructed mental model or parts of it and the process of constructing the mental model may be internalized into long term memory and become part of knowledge.

The relationship between knowledge, mental model and data in our conceptual framework is illustrated in Figure 5. Knowledge and data together help to form and evolve the mental model (1: retrieve and 2: fetch). The mental model guides the search for knowledge and for data (3: access and 4: search) and the retrieved knowledge and selected data further shapes the mental model. Repeated construction of a mental model causes elements of that model and the process of constructing it to be internalized knowledge (5: internalize). This internalization step takes a long time and many stages of fragile knowledge, in which the mental model is not yet fully internalized or integrated with the prior knowledge, may exist. Based on this conceptual framework of the use and acquisition of expertise, we re-interpret the results of different empirical studies in following sections.

In the following sections, we discuss the details of each empirical result, resolve the contradictions and explore the relatedness of these results through our conceptual framework.

3.1.2 PROGRAM PRODUCTION – FROM REQUIREMENTS TO DESIGNS AND CODES

The program production activities considered in this survey include all activities until an integrated piece of software code is generated. These activities include but are not limited to system or

requirements analysis that happen early in the software life cycle, domain level or algorithm level system designs and also the actual implementation of program codes. Actually, nearly all of these activities have been studied in the following research we review in this section.

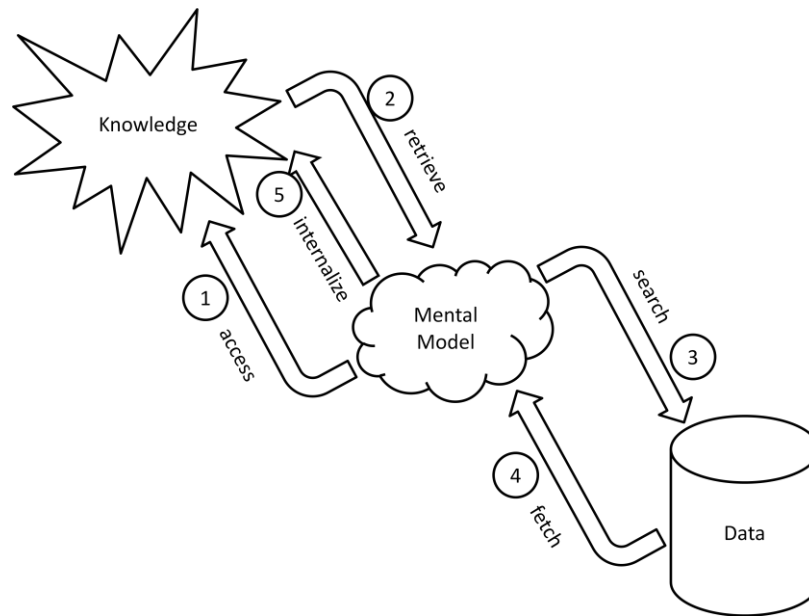


FIGURE 5 CONCEPTUAL FRAMEWORK OF PROGRAMMING

3.1.2.1 STRUCTURED VERSUS OPPORTUNISTIC DESIGN BEHAVIORS

TABLE 2 REINTERPRETATIONS OF STUDY RESULTS ABOUT STRUCTURED VERSUS OPPORTUNISTIC DESIGN BEHAVIORS

Study	Empirical Result	Re-interpretation
Jeffries, et al., 1981	hierarchical goal decomposition	<i>mental model</i> guides data search(3)
Adelson & Soloway, 1985	systematic expansion	
Kant, 1985	elaborating kernel structure	
Guidon, et al., 1987	opportunistic behaviors	fetch(4) data evolves <i>mental model</i>
Guindon, 1990a, b		
Visser, 1987, 1994	data-driven manner	

Jeffries, et al. (Jeffries, et al., 1981) collected think-aloud protocols from experts and novices who engaged in the design of a page-key indexing system for electronic books stored as text files. Through the analysis of protocols, they found that regardless of novice-expert differences, all subjects adopted the same global action control strategy, *hierarchical goal decomposition*. Subjects used this strategy to repeatedly decompose a problem into sub-problems until one sub-problem is considered to be detailed

enough and marked as solved. Not only was this behavior structure observed to be similar across groups of subjects with different levels of experience, the decomposed sub-problems were also found to be similar.

This behavioral pattern of hierarchical decomposition of a problem was also observed in studies carried out in (Adelson & Soloway, 1985) and was termed there as *systematic expansion* behaviors. The authors collected think-aloud protocols from both experts and novices working through three different categories of design problems that cover the conditions of 1) design a familiar object in a familiar domain, 2) design an unfamiliar object in a familiar domain and 3) design an unfamiliar object in an unfamiliar domain (Note that design of a familiar object in an unfamiliar domain does not make sense). Regardless of the category of design problem, both experts and novices in this study demonstrated the common behavioral pattern of forming a general mental model of the problem and then systematically solving the sub-problems to expand the initial model. Such expansion was also observed in the empirical study carried out in (Kant, 1985). Subjects' verbal protocols were collected during algorithm design tasks and then analyzed in this study. A pattern of "planning a solution around a kernel idea and refining or elaborating the kernel structure" was observed to recursively occur among subjects. These studies strongly support the claim of a systematically organized behavior structure within a design episode.

While empirical results as stated above show support for systematically organized behaviors, a cluster of other studies report observations on more or less opportunistic behaviors in design activities. The authors of (Guindon, et al., 1987), (Guindon, 1990), and (Guindon, 1990) studied eight professional programmers as they worked to design a lift control system. Think-aloud protocols were collected and analyzed together with a video recording of subjects' actions and a collection of design artifacts. However, in contrast to previous studies, many opportunistic behaviors were observed during the design process. The author argued that those programmers frequently deviated from a strict top-down decomposition design strategy and instead adopted an opportunistic approach in their work by

demonstrating behaviors such as sudden inferences and immediate development of new requirements, changes of goals, immediate recognition of solutions to sub-problems that are not part of the problem being currently worked on, etc. They suspected that such opportunistic behaviors are advantageous and benefit the design of an ill-structured problem (Simon, 1973), a problem that is ambiguously specified with a large problem space, without stopping rules or an explicit path to solution. Therefore, as stated by the authors, “experts are expected to retrieve knowledge rules and the more complex design schemas in a data-driven manner”.

Such *data-driven* behavior was also observed in a field study reported in (Visser, 1987) and reviewed by (Visser, 1994). In this study, the researcher collected observation notes on a professional programmer working with a real industrial problem in daily settings and found similar opportunistic patterns in the subject’s behaviors. The author argued that the general organization of a design activity is primarily opportunistic with local hierarchical or systematic sub-structures due to the fact that designers make design decisions based on cognitive costs that change from moment to moment. Therefore, even though knowledge of design plans and design schema are successfully and appropriately retrieved during a design process, designers have to make modifications to these plans and therefore their behaviors would deviate from defined schemas.

In an attempt to consolidate the contradictory observations of structural versus opportunistic design behavior, the program plan theory was proposed. In the following sub-section, we discuss the details of this theory.

3.1.2.2 THE PROGRAM PLAN THEORY

The term *program plan* has many different meanings in the literature. Actually, the theory of program plan, like many other theories, has developed and changed with accumulating empirical evidence over a long period of time. In our narrative, we define a program plan as knowledge either about the procedure for achieving a goal (procedural) or about characteristics of a solution to a goal

(declarative). The program plan regarding program production activities is therefore either knowledge about an effective production procedure or knowledge about the content of a final product. Considering a complex design task, knowledge of the final product is unlikely to be already possessed by a programmer. However, knowledge of an effective production procedure and knowledge about solutions of sub-problems may be available. Thus, the theory states that structural behaviors are observed when a program plan is available (plan retrieval) for a sub-problem while opportunistic behaviors are observed when a program-plan is not available (plan creation) (Rist, 1990).

TABLE 3 REINTERPRETATIONS OF STUDY RESULTS ABOUT PROGRAM PLAN THEORY

Study	Empirical Result	Re-interpretation
Soloway & Ehrlich 1984	program discourse is possessed and used by programmers	program discourse is one element of <i>knowledge</i>
Rist, 1986	program plan (basic, object, goal-based) is possessed and used by programmers	content of <i>knowledge</i> distinguishes expertise
Rist, 1989	possession of program plan affect behaviors	content of <i>knowledge</i> affects behavior
Rist, 1990	availability of knowledge cause variability in design decision	retrieved (2) <i>knowledge</i> evolves <i>mental model</i>

The postulation and refinement of this theory occurred over a long time span. The initial empirical clues were provided in (Soloway & Ehrlich, 1984). The idea for this empirical study is derived from research on language discourse in cognitive psychology. The authors created four small pieces of “plan-like” (normal) programs (alpha version) and four corresponding “unplan-like” (valid and runnable but non-standard appearing) programs (beta version) for the study and guaranteed that the beta versions were almost identical to the alpha versions, with only minor syntactic changes that were claimed to violate one or more rules of “program discourse” (established customs for writing programs) such as that a variable name should reflect its function (e.g., a variable named MIN is unplan-like to actually hold a maximum value). Subjects, who were then asked to fill in a blank to best complete a program of either the alpha or beta version, were not only observed to take longer to finish the unplan-like version but were also observed to provide plan-like responses for unplan-like programs. For example, working on a piece of an unplan-like program that intended to find out the maximum of a series but named the

variable storing the final result as “MIN” caused subjects to constantly perform incorrect comparisons. With these findings, the authors of the study claimed that a certain *program plan* that conforms to programming discourse rules is possessed and used by programmers. Although Soloway and Ehrlich proposed the concept of a program plan, an important part of the concept, programming discourse rules, were not well defined. The authors listed five rules they thought of at the time and tested in their empirical studies without giving any definition or descriptions towards the general characteristic of discourse rules.

Another empirical clue regarding plan theory comes from a study carried out by the theory proposer, Rist. In (Rist, 1986), novices and experts were given program codes to group into “lines that did the same thing” without any further specific criteria. Five copies of each program being grouped either from finer to coarser levels or vice versa were collected for analysis of the nature of each group of code lines. Novices were found to largely use syntactic groups while experts were found to use “goal-based” groups, group of codes that achieve a common goal for the program. In this work, plans were described as one of three categories: basic plan, as described in (Soloway & Ehrlich, 1984), object plan, a plan that accesses or modifies an object in the program, and goal-based plan, a plan that achieves a goal or sub-goal of the program.

In (Rist, 1989), the author further defined the term *plan* by a description of a modified category of simple and complex plans. A longitudinal study was also carried out to compare programmers’ behaviors while writing codes when a plan was available to them versus not. Ten relatively novice subjects were asked to finish coding one problem each week over the eight weeks of a programming course. Problems presented each week utilized the declarative programming construct knowledge that had just been learned by the subjects in class. In the first and last week, however, both a simple problem and a hard problem were presented. Think-aloud protocols were collected and videos of the subjects’ generating code on paper were recorded. These two types of data were then transcribed and combined into a

single protocol for analysis. Backward and bottom-up behaviors were observed when subjects were working on problems requiring utilization of freshly learned knowledge while forward and top-down behaviors were observed when subjects were working on simple problems during the first and last sessions or sub-problems that they had worked out before. Therefore, the author proposed a theory of plan creation and retrieval that comprised both structural and opportunistic behaviors reported by different empirical studies. The author stated that if knowledge can be found to guide program design, top-down and forward behavior would be seen, otherwise, bottom-up and backward behavior would be seen. At the same time, the author also claimed that the process of plan generation was actually the process of gaining expertise. Parts of the observations reported in (Jeffries, et al., 1981) and reviewed in section 3.1 support the plan theory in that novices were observed to lack subtle design schema (detailed design plans) or to consider the efficiency or aesthetic aspects of their plan. The novices in Jeffries, et al.'s study were reported to be generally unable to propose alternative designs or to evaluate alternatives. For example, experts were observed to spend much effort on designing the comparison algorithm for the page-key indexing system while novices simply concluded the comparison to be a "trivial" component that did not need any more specific designs.

In (Rist, 1990), the author further extended the plan theory by claiming that the availability of knowledge at different stages of design activity is the cause of variability in final design decisions. An empirical study of programmers coding a relatively well-structured problem (as compared to that used in (Guindon, et al., 1987)) of calculating elapsed seconds between two clock representation, was carried out to empirically validate the claim. The author argued that such a relatively simple question still posed a variety of choices at each design stage, such as: 1) planning stage, which may be conducted in a forward manner, from input to output or backward from output to input, 2) actions in the solution are grouped by shared features and 3) actions are merged in the implemented program, based on these shared features. Within each of these stages, a certain piece of knowledge may be available or not,

which drives the programmer to seek alternatives and finally results in variability in design solutions. In (Rist, 1991), another empirical study using protocol analysis was carried out with novice and intermediate level subjects. The behaviors categorized as plan creation and plan retrieval respectively were correlated with subjects' expertise to validate and finally complete the construction of this theory.

Design is probably one the most cognitively intensive problem solving activities in the production phase of software development. Accordingly, much research effort has been expended to find empirical evidence of the nature of the design process and to form a theory of programming expertise. Empirical studies of design activities have been fruitful with the discovery of the importance of program plan, a kind of knowledge. Yet, research on programming expertise is not complete at this point. With the plan theory in mind, we are interested to ask about the nature of program plans or program knowledge in general. Although the proposal of plan theory addresses the contradictory observations found on the behavioral level, the theory itself provides neither a concise definition of a program plan nor any insight into its content, its organization or the mechanism of creating and retrieving such program plans. Research that explores these questions is discussed in the next section.

3.1.3 FROM PRODUCTION TO COMPREHENSION

To answer questions about the nature and structure of program plans and their creation and retrieval, it is worth noting that empirical studies carried out solely with production activities are not sufficient. To understand the content and organization of programming related knowledge, cognitive psychology experiments may be of help. To understand the mechanism of creating and retrieving such knowledge, empirical studies of activities beyond production are also necessary.

3.1.3.1 MEMORY: MENTAL MODEL AND PROGRAMMING KNOWLEDGE

Research on programmer's memory regarding both long-term possession of knowledge and short-term formation of mental representations has an even longer history than the formation of above described plan theory. Initial studies were inspired by research carried out in the cognitive psychology

field on chess masters' superior recall of real and meaningful chess board layouts (Chase & Simon, 1973). In (Shneiderman, 1976) and (Schneiderman, 1977), the author directly followed the experimental design of studies of chess players' memorization to ask programmers to recall meaningful pieces of codes or scrambled lines of codes. They observed that expert programmers were better at recalling meaningful segments of program codes than were novices. Adelson (Adelson, 1981) further discussed the mechanism behind such superior memorization of semantically rich program codes demonstrated by expert programmers. In her study, sixteen lines of codes were randomly shown to both expert and novice programmers one by one. The subjects, without being told that these sixteen lines of codes could either be organized into three meaningful programs or five syntactically similar groups, were asked to recall these lines in multiple trials. The pauses between subjects' responses (writing out a recalled piece of code on paper) were recorded and items recalled with less than a ten-second pause in between were considered to be members of the same memory *chunk* (groups of items recalled successively as defined in (Chase & Simon, 1973)). The sizes and characters of the chunks were then compared and experts were demonstrated to have both larger memory chunks than novices and to organize chunks semantically rather than syntactically, as did the novices.

TABLE 4 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT MEMORY AND RECALL

Study	Empirical Result	Re-interpretation
Shneiderman, 1976 Shneiderman, 1977	experts recalls meaningful program better	retrieved (2) <i>knowledge evolves mental model</i>
Adelson, 1981	experts memory are semantically organized	
Adelson, 1984	experts form abstract mental model	
Gilmore & Green 1984	mental representation depends on original form of program	fetchd (4) <i>data evolves mental model</i>

Expert's superior recall of meaningful program codes and the insight that experts' chunks of memories are organized semantically, to some extent, support the claim of the plan theory that expert programmers possess a larger repository of program plans, organized as semantic patterns of meaningful program elements that achieve some goal. These program plans exist in experts' long term

memory as a form of knowledge the retrieval of which is precipitated by the semantic elements of the stimuli materials used in recall experiments. Therefore, experts achieve superior recall of semantically meaningful piece of codes and semantically organize their memory of these codes. A caveat of the plan theory is that it can only explain the nature of expertise if program plans are abstracted knowledge formed through experience and independent from any superficial characteristics of program notations. Empirical evidence supporting this claim is provided in (Adelson, 1984). In this study, the subjects were required to answer either a set of “abstract” questions (about the program’s general goal) or a set of “concrete” questions (about the program’s detailed line-by-line functionality regardless of its general goal) for each of the eight experimental programs. Subjects were not informed about what types of questions they would answer for each program while comprehending the programs. Therefore, the authors argued that subjects could either form an abstract mental representation of the given program (an abstracted memory of the program’s goal) or a concrete mental representation (a concrete memory of the program’s line-by-line functions). Since an abstract mental representation is only more helpful in answering abstract questions, interestingly, the experts in this study were observed to perform worse than novices with concrete questions. This showed that experts tend to form an abstract mental representation of programs while novices tend to form a more concrete mental representation. The result of this empirical study supports that a plan (knowledge about a program) formulated in experts’ memory is abstract rather than concrete.

While empirical evidence from the above study demonstrates the abstract character of expert programmer’s mental representations, some contradictory evidence also exists. Gilmore and Green (Gilmore & Green, 1984) showed that mental representation of a program still depended on the original presentation of the program. The subjects in this study were asked to answer either a set of “sequential questions” (questions about whether X will happen next according to the program settings) or “circumstantial questions” (questions about whether under a certain combination of conditions will X

occur according to the program). The authors postulated that if the mental representation of a program were abstract and independent of its original form of presentation, subjects should have relatively equal performance on these two sets of questions with programs written in either a procedural language (e.g. PASCAL) or a declarative language (e.g. PROLOG). Non programmers were recruited for this study. Programs were modified as either procedural or declarative forms of logical statements and questions were modified as logical reasoning questions. The experiment results showed significant differences on the interaction effect between the form of program and type of questions. Subjects were observed to perform better on sequential questions with the procedural form of the program and to perform better on circumstantial questions with the declarative form of the program. In (Gilmore & Green, 1988), the authors further tested the hypothesis that a plan is not an abstract piece of knowledge. They provided “plan cues” (highlighted some part of program code that forms a plan) for both PASCAL and BASIC programmers, but did not observe consistent performance improvements in finding bugs across these two forms of programs. They suspected that either the provided “plan cues” were not correct or that the program plan was not an abstract concept. Since the former possibility was refuted by observed performance improvements while working with PASCAL, the authors postulated that a program plan was just a visible aspect of a program rather than an abstract concept that may explain expertise.

A follow-on theory that modifies the plan theory and extends the understanding of programmers’ mental representations was developed by Pennington in (Pennington, 1987) and (Pennington, 1987). Pennington proposed that five types of knowledge exist in a programmers’ mental representation: 1) operations (low level, detailed step by step function), 2) control flows, 3) data flows, 4) state of a program, and 5) functions (organized operations achieving sub-goals of a program). The relative amount of each type of information in the programmers’ mental representation is decided by the programming tasks that programmers seek to perform. To validate this proposal with empirical evidence, subjects in these studies were asked to perform different tasks before answering questions that were designed to

test the existence of the different types of information described above. Significant shifts in information composition were observed with the shift of tasks. The claim that mental representations change according to task is also confirmed by empirical evidence observed through another contemporary experiment presented in (Holt & Boehm-Davis, 1987) and (Boehm-Davis, et al., 1992). In this study, the subjects were required to finish a two grid free recall of the program they had just worked on (recall as many program components without any grouping criteria first and then recall the relationships among these components). Three forms of programs, 1) in-line procedural program, 2) functional-decomposed procedural program, and 3) object oriented program, combined with two types of modification tasks, simple (require modification at only one place) and complex (require modification at multiple places) were tested with different subject groups, expert and novice. The authors observed that expert programmers' mental representations of a program were greatly affected by the complexity of tasks while novices' mental representations were affected by the form of the programs.

TABLE 5 REINTERPRETATIONS OF EMPIRICAL RESULTS ABOUT OTHER MENTAL MODEL THEORY

Study	Empirical Result	Re-interpretation
Pennington, 1987 Pennington, 1987	mental representation contains 5 types of information content of mental model depends on task	both retrieved (2) knowledge and fetched (4) data evolves <i>mental model</i>
Holt & Boehm-Davis, 1987 Holt & Boehm-Davis, 1992	complexity of tasks affects the content and organization of experts' mental model form of programs affects the content and organization of novices' mental model	dominancy of retrieved (2) knowledge and fetched (4) data affects content and organization of <i>mental model</i>
Ramalingam & Wiedenbeck, 1997 Corritore & Wiedenbeck, 1999	novices' mental models contain largely superficial OO constructs while experts' mental model contains mixed information	

Considering the result of (Holt & Boehm-Davis, 1987) and (Boehm-Davis, et al., 1992) and noticing that (Gilmore & Green, 1984) recruited non-programmers in their study, we suspect that the effect of program presentation on programmer's mental representation is just especially obvious for novice programmers. This was later confirmed by the study of novice and expert's mental representation presented in (Ramalingam & Wiedenbeck, 1997) and (Corritore & Wiedenbeck, 1999). These studies followed the methods used in (Pennington, 1987) and (Pennington, 1987) to test the existence of

different types of information in novice and expert's mental representations after working on either a procedural or object-oriented program. The result showed that while novices' mental representations of procedural and object-oriented program are dominated by control-flow and function related information respectively, experts' mental representations of either type of program achieve a mixed balance across all types of information after working on a challenging debugging task.

Thus, we may conclude that both abstract and concrete knowledge and mental representation exist in programmers' memory and that the variability in abstraction or concreteness according to task demands illustrates programming expertise. It is interesting to notice that the last three works we reviewed above were carried out in the context of comprehension activities (activities that require a solid understanding of the target program, such as debugging, modification and enhancements). Therefore, it is also natural to inquire, how production tasks are similar to or different from comprehension tasks and what these two types of tasks share in term of the nature of programming expertise.

3.1.3.2 PRODUCTION VERSUS COMPREHENSION

Several studies report observations on the relationship between success at production and comprehension tasks. In (Ahmadzadeh, et al., 2005), the authors explored compiler logs and historical versions of novice computer science students' code produced during programming and debugging tasks. The researchers found that the most common error made by novices was that they forgot to define a field before its use. This somehow confirms the plan theory that during novices' plan creation procedures (e.g. programming), calculations are carried out before initialization as novices are working backwards to the goal. The performance results gathered from both programming and debugging tasks in (Ahmadzadeh, et al., 2005) showed that most students who were good at debugging were also good at programming. However, only a portion of students who were good at programming were also good at debugging and the obstacles to comprehending program implementations written by others were

identified as a major reason for this phenomenon. Evidence that “debugging requires more skills” is also confirmed with a multi-institutional study in (Fitzgerald, et al., 2008). The authors reported experiments carried out with novice Java programmers’ writing code for a programming problem and then trying to correct a buggy program that was written by others but that solved the same problem. Objective criteria and measurement methods were developed to grade the quality of code generated by subjects and it was confirmed again that good programmers were not necessarily good at debugging. In (Katz & Anderson, 1987-1988), the authors discussed programmers’ bug location strategies and observed that authorship of program code greatly affected the behaviors programmers used to locate bugs: a forward strategy was used when debugging one’s own codes and a backward strategy used when debugging codes written by others. These empirical studies also demonstrated that comprehension activities are organized in a backward manner while a production activity may combine both forward and backward characteristics.

Although the observed behaviors indicate that comprehension tasks are *different* from production behaviors, empirical evidence also emerged to show their *similarity* in that they share a knowledge base. In (Pennington, et al., 1995), the authors studied the “transfer” of knowledge gained in production to comprehension and vice versa. The experiment design simplified the production and comprehension tasks as evaluation (comprehension) and generation (production) of LISP expressions. The subjects involved in this study were observed to perform better on one type of task after practicing on the other. Based on this observation, the authors hypothesized that the same knowledge base was shared between the superficially different production and comprehension procedures. Empirical clues supporting the claim of shared knowledge are also shown by a study described in (Gray & Anderson, 1987). The authors observed that the goals that require the most effort to design are also most likely to be changed during subsequent programming phases. Later on, this relation between goals and syntactic change is extended to be inclusive of semantic level changes in (Scholtz & Wiedenbeck, 1992) and

(Scholtz & Wiedenbeck, 1993). This study compared expert programmers working with a familiar versus an unfamiliar (new) language. The authors of these studies observed that plans were changed at a high level, algorithm level and low level when knowledge from the familiar language was not sufficient for successful code creation on the first attempt with the unfamiliar language.

TABLE 6 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT PRODUCTION VERSUS COMPREHENSION

Study	Empirical Result	Re-interpretation
Ahmadzadeh, et al., 2005 Fitzgerald, et al., 2008	debugging requires more expertise than programming	programming mostly as a one cycle interaction of retrieve (2) and access (1) <i>knowledge</i> requires less expertise than debugging, a two cycle interaction of retrieve (2) and access (1) <i>knowledge</i> and fetch (4) and search (3) <i>data</i>
Katz & Anderson, 1987-1988	authorship decides debugging strategy	content of <i>mental model</i> guides data search (3) and <i>knowledge retrieval (2)</i>
Pennington, et al., 1995	practice one of the comprehension or production problems helps the performance on the other	<i>mental model</i> is internalized (5) to <i>knowledge</i>
Gray & Anderson, 1987	goals that require the most effort to design are the most likely to change	<i>mental model</i> formed dominantly with fetched (4) <i>data</i> are weaker
Scholtz & Wiedenbeck, 1992 Scholtz & Wiedenbeck, 1993	programmer change plan when knowledge from familiar language was not sufficient for code creation with unfamiliar one	<i>mental model</i> formed dominantly by fetched (4) <i>data</i> when <i>knowledge</i> cannot be accessed (1)

Other empirical studies also provide evidence to support the claim of shared knowledge, from the perspective of exploring the nature of bug generation. In (Bonar & Soloway, 1985), the authors observed novice programmers working on coding tasks while thinking aloud and generalized that most of the novice bugs resulted from a natural strategy adopted to patch the use of programming knowledge with natural language knowledge. This is also confirmed by the study of Pea (Pea, 1986) carried out to analyze interview and think-aloud protocols from multiple institutes that teach senior high school students to program. These novices were observed to use natural language discourse and context to interpret a program's functionality. For example, while a condition stated in an "if statement" in natural language is usually not instantaneously evaluated and assumes the condition to be established within a certain period of time, this is not the case when an "if statement" is written in a program, where the

condition is evaluated at the moment of execution of the statement and the statement then no longer has any effect. Studies with clinical interviews (interviewing and providing help for problem solving) are described in (Perkins & Martin, 1986). The authors observed the same defects in novice's programming knowledge, termed "fragile knowledge", which was either missing (a piece of necessary knowledge that does not exist in programmer's memory), inert (a piece of knowledge that is not retrievable), misplaced (a piece of knowledge that is misconnected to a goal) or conglomerated (a piece of knowledge that is inappropriately combined with other pre-programming knowledge).

TABLE 7 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT BUG GENERATION

Study	Empirical Result	Re-interpretation
Bonar & Soloway, 1985	novice use natural language knowledge to patch programming language knowledge	<i>mental model</i> is formed with retrieved (2) knowledge , either programming related or not
Pea, 1986	novice use natural language discourse to interpret programming language	
Perkins & Martin, 1986	fragile knowledge causes novices' bugs	<i>knowledge</i> is possessed with different levels of accessibility

Through the review of studies in this section, several implications arise. First, programming knowledge seems to be transferrable between production and comprehension activities. Second, incorrect or imperfect programs may result from the lack or misuse of certain programming knowledge. Third, the major challenge imposed by comprehension activities is to build a balanced mental representation of the current working problem (including program and task context), which distinguishes experts and novice in comprehension tasks. However, we still need more detailed empirical evidence about comprehension behaviors to arrive at a generalized theory of programming expertise applicable to all types of tasks. This empirical evidence is provided in the next section.

3.1.4 PROGRAM COMPREHENSION – FROM CODES AND REQUIREMENTS TO DESIGNS

Studies of comprehension activities started early as people noticed the substantial time devoted to these activities. The earliest studies we include here are (Gould & Drongowski, 1974) and (Gould, 1975), in which the authors report observations of "practice effect" in multiple studies in which programmers

performed better and faster with previously debugged programs even though different bugs were seeded inside. Now that we have reviewed the literature about using transferrable programming knowledge to build mental representations of a program under study, the “practice effect” is easy to explain in that after forming a mental representation of a program for debugging, it becomes relatively easy to locate new bugs (and/or make modifications and enhancements). As stated in (Ahmadzadeh, et al., 2005) and (Fitzgerald, et al., 2008), the “understanding of a program written by others” is critical to the success of comprehension activities and “once a bug is located, it is almost always fixed” (Katz & Anderson, 1987-1988). Therefore, our next question is, accordingly, what the details of the comprehension process are.

3.1.4.1 HYPOTHESIS-DRIVEN VERSUS DATA-DRIVEN

In a study presented by (Gugerty & Olson, 1986), the term hypothesis refers to a proposed cause of a bug. In their studies, they observed that expert programmers formed higher quality hypotheses about programs. At that time, the author had not proposed that programmer’s behaviors were driven by the hypotheses they made. Almost at the same time, through analysis of think-aloud protocols produced by programmers during debugging tasks, Letovsky (Letovsky, 1986) formed a grounded theory that debugging was a *hypothesis-driven* activity. He provided a further detailed explanation of this hypothesis-driven theory by enumerating and explaining that subjects took the process of building conjectures to answer the what, why, and how questions (hypotheses). The authors of (LaToza, et al., 2006), (LaToza, et al., 2007) and (LaToza & Myers, 2010) proposed that programmers asked *reachability questions* during debugging through analysis of protocols and surveys. A reachability question can be exemplified as “through what paths does X happen”, which from the theoretical perspective, complements the theory of hypothesis-driven comprehension behavior. Another series of studies, as presented in (Sillito, et al., 2005), (Sillito, et al., 2006) and (Sillito, et al., 2008) provides confirmation and refinement of the hypothesis-driven theory. The authors postulate that forty-four types of questions

were asked by programmers during debugging tasks and provide detailed corresponding behaviors that seek to answer these questions.

TABLE 8 REINTERPRETATIONS OF EMPIRICAL RESULTS ABOUT HYPOTHESIS-DRIVEN DEBUGGING BEHAVIORS

Study	Empirical Result	Re-interpretation
Gould & Drongowski, 1974 Gould, 1975	programmers perform better with programs they have debugged	<i>mental model</i> is program and task specific
Gugerty & Olson, 1986	expert form better hypotheses during debugging	experts have strong mental models to guide search (3) for relevant data
Letovsky, 1986	hypothesis-driven debugging behavior	<i>mental models</i> formulated pre-dominantly with retrieved (2) knowledge guide the search (3) for relevant <i>data</i> in a hypothesis-driven manner
LaToza, et al., 2006, 2007 LaToza & Myers, 2010	reachability questions	<i>mental models</i> guide the search (3) for relevant <i>data</i> and <i>data fetched (4)</i> in this search further evolves the <i>mental model</i>
Sillito, et al., 2005, 2006, 2008	44 types of questions	

TABLE 9 REINTERPRETATIONS OF EMPIRICAL STUDY RESULTS ABOUT CHARACTERS OF DEBUGGING BEHAVIORS

Study	Empirical Result	Re-interpretation
Nanja & Cook, 1987	experts take comprehensive approach while novices take isolated approach	experts' retrieved (2) knowledge forms a stronger <i>mental model</i> to guide the search (3) for relevant <i>data</i>
Littman, et al., 1987	systematic versus as-needed behaviors	
Vessey, 1985 Vessey, 1986	expert use systematic and breadth first strategy	stronger <i>mental models</i> guide more systematic search (3) for relevant <i>data</i>
Ye & Salvendy, 1996	experts use top-down comprehension strategy but novices show opportunism	
Corritore & Wiedenbeck, 2001	different comprehension and debugging strategies used for OO and procedural	fetched (4) data affects the evolution of <i>mental model</i> that guides the search (3) for further <i>data</i>

With regard to the relationship between comprehension strategies and programming expertise, the authors of (Nanja & Cook, 1987) observed that expert programmers took a more comprehensive approach to understanding a program under debugging while novices adopted an isolated approach that would just serve to understand “enough for debugging”. This evidence is further confirmed and developed into the *systematic vs. as-needed* theory in (Littman, et al., 1987). Littman, et al. argued that systematic versus as-needed behaviors, distinguished by the goal, scope and target of comprehension (to understand the program as a whole or to understand just the parts that are possibly related to bugs) decides whether causal knowledge can be formed in a programmers’ mental representation and were

related to their success at debugging. In (Vessey, 1985) and (Vessey, 1986), Vessey analyzed think-aloud protocols and proposed that expert programmers used a systematic and breadth-first strategy to comprehend codes.

Evidence from a quantitative empirical study on the differences in comprehension strategies adopted by programmers with different levels of expertise was presented in (Ye & Salvendy, 1996). Subjects in this study were required to associate code segments with listed goals. The code segments were organized hierarchically so that subject's order of association reflected their direction in comprehension. It was observed that although both novices and experts adopt a top-down comprehension strategy in general, novices showed more opportunism within this top-down structure. This study confirmed that experts used a more systematic strategy in comprehension. The breadth and direction of code comprehension behavior was then quantitatively studied in (Corritore & Wiedenbeck, 2001) with respect to different programming paradigms. Based on the nature of the files being examined (documents are considered the shallowest level, header files a middle level and implementation codes the deepest level) and the number of files being accessed, the authors compared the differences in object-oriented experts' and procedural experts' debugging behaviors in terms of the depth and breadth of comprehension. It was observed that a mixed strategy (top-down in general comprehension, bottom-up in debugging) was used by object-oriented programmers and a consistent bottom-up strategy was used by procedural programmers, while experts working with both forms of program finally achieved a broad view of programs that contain both high-level and deep-level information by examining a considerable number of files in each level.

A similar argument of opportunistic behaviors was proposed for comprehension activities (see section 3.1 for a discussion of opportunistic behaviors with regard to production activities). These studies take the stance that comprehension activities are data-driven rather than being guided by any structure. Although the data-driven nature of comprehension activities was only recently proposed, as

seen in section 3.1, the concept has existed for much longer throughout the discussion of production behaviors. The data-driven theory, termed “information foraging in debugging”, was first postulated in (Ko, et al., 2006). Analyzing the protocol data collected for designing a better integrated development environment from (Ko, et al., 2005), the author proposed a model of how programmers *seek, relate and collate* data during enhancement tasks and claimed that this behavior corresponds to the information foraging theory developed in (Pirolli & Card, 1999).

TABLE 10 REINTERPRETATIONS OF EMPIRICAL RESULTS ABOUT DATA-DRIVEN DEBUGGING BEHAVIORS

Study	Empirical Result	Re-interpretation
Ko, et al., 2006	information foraging in debugging	fetches (4) <i>data</i> affects <i>mental model</i> that supervises the search (3) for further <i>data</i>
Lawrance, et al., 2013	validate information foraging	
Eisenstadt, 1993	data chasing behavior	
McKeithen & Reitman, 1981	experts superior memorization are achieved iteratively	<i>mental model</i> evolves
Wiedenbeck, 1986	experts recognize beacons	retrieves (2) <i>knowledge</i> affects <i>mental model</i> that supervises the search (3) for <i>data</i>
Robillard, et al., 2004	novices have inattention blindness with critical information	

Later on, Lawrance, et al. (Lawrance, et al., 2013) validated the theory of information foraging in debugging through analysis of programmer’s think-aloud protocols and claimed that programmers were actually following a “scent”, the perceived likelihood of a cue such as words, objects, or perceptible runtime behaviors to find a prey (what the programmer seeks to know to fix the bug), rather than formulating and evaluating hypotheses about the program they are debugging. The authors also claimed that debugging behavior explained by information foraging theory was consistent with 1) Activity Theory (Leontjev, 1978), in which plans guide behavior but exact actions or operations are determined by the context in which the action takes place, and 2) the theory of situated cognition (Suchman, 1987), in which plans are inherently vague and the structure of the environment has far more effect on particular actions, as well as 3) the theory of distributed cognition (Hollan, et al., 2000), which argues that a large part of cognition is triggered by interaction with the environment rather than happening predominantly in head. Interestingly, the author of a much earlier work (Eisenstadt, 1993), reported a survey of global

debugging anecdotes and concluded that the major source of difficulty in debugging was the large temporal or spatial chasm between the bug symptom and the bug root cause. Accordingly, the author found that the dominant debugging technique was data gathering, in order to understand the program and the nature of the bug. Noticing the two quantitative studies described before, (Ye & Salvendy, 1996) and (Corritore & Wiedenbeck, 2001), which have different conclusions regarding top-down versus bottom-up comprehension strategies, it seems that we have another clue of the existence of opportunism in program comprehension.

The “program plan” conceptual framework, together with an understanding of the circumstances under which plans are created versus retrieved, helps to explain/reconcile the contradictions seen in the literature regarding opportunistic versus structured behavior related to production activities, but does not explain contradictions in the literature regarding hypothesis-driven versus data-driven behaviors related to comprehension activities. On the other hand, during our review of the previously described empirical studies, we see that (LaToza & Myers, 2010) actually exhibits a dual character. To some extent it supports the theory of hypothesis-driven behaviors by concluding that programmers “ask” reachability questions while on the other hand, it indicates the existence of data-driven behaviors by stating that to answer reachability question, a “search” through feasible paths is necessary. We take the stance that the main contradiction of hypothesis-driven versus data-driven theories hinges on one critical concern about the role of data: whether data simply provides answers to questions or that it actually drives the formation of questions. In reality, data likely serves both roles, as initially available data helps form hypotheses (e.g.: documents, bug reports, enhancement requirements, etc.) that drive the gathering of subsequent data. The successively gathered data (e.g. code, program output) are used to refine mental representations and form new hypotheses. This is a process that starts by using initial data to form an initial mental representation (hypotheses), and then uses the mental representation to guide further

information searching while evolving the mental representation with the gathered information. Some empirical evidence is presented in next sub-section to support this speculation.

3.1.4.2 GUIDED SEARCH AND EVOLVING MENTAL REPRESENTATION

In the previous section, we reviewed the discussion of hypothesis-driven versus data-driven behavior based on empirical results regarding programmers' mental representations and knowledge. However, although many studies provide evidence on the content of such mental representations, few discussions focus on the formation of these representations. One relevant finding is that mental presentations transformed after performing different tasks, as seen in (Pennington, 1987) and (Holt & Boehm-Davis, 1987). While (Adelson, 1981), as reviewed in section 3.1.3.1, stated that expert's chunks of memory were semantically ordered, no discussion was available on how these chunks were formulated into a semantically ordered collection. This concern is addressed to some extent by a peer work (Mckeithen & Reitman, 1981). The subjects in their study, either novice or expert, were required to memorize keywords of a programming language. Subjects were told that re-organizing the keywords would help them in the task but were not given any re-organization criteria. Besides observing that experts' memory chunks are semantically organized, the authors also reported that experts' successful recognition of similar words occurred through multiple trials of recall. Therefore, the authors proposed that programming expertise is different from expertise in other domains (e.g. a chess master recalls a board layout in initial trial) and postulated the reason to be that programming-related information is not available in one generalized pictorial representation as in chess, Go, or electrical diagrams. That expert programmers required multiple trials to create their semantically organized chunks supports the notion that a mental representation is formed with initial data, guides the subsequent data gathering process and also evolves with the newly gathered data.

But then, what role does expertise play in this process? In other words, what makes an expert programmer a better data seeker? We postulate that another factor is also involved in the formation

and evolution of mental representations, knowledge, which plays a similar role to data. Actually, we view knowledge and data just as two forms of information, internal and external. The searching of both forms of information is guided by the mental representation and the searched results of both forms of information evolve the mental representation. Empirical evidence also exists to support this speculation. In (Wiedenbeck, 1986), experts were observed to recall certain key parts of a program, *beacons*, much better than did novices. In (Robillard, et al., 2004), novices were observed to have inattention blindness with information that was critical to their task, information they encountered but did not explicitly search for. We take the stance that a lack of knowledge (internal information) actually causes weaker initial and subsequent mental representations given the same amount and quality of external information (data) in a problem solving context, which consequently results in inferior information searching strategies for both internal information (knowledge) and external information (data).

3.1.5 A GENERAL CONCEPTUAL FRAMEWORK OF PROGRAMMING EXPERTISE

We accept the empirical evidence that illustrates that differences exist between programming expertise and expertise in other domains (Mckeithen & Reitman, 1981) and further postulate that the need for programmers to deal with large volumes of internal and external information in a variety of forms makes the distinction. Therefore, we provide the following description of **the nature of programming expertise** based on its distinction from expertise in other domains: **1)** From a problem-solving perspective, programming expertise is exhibited as a superior repository of internal information (knowledge) that helps form stronger mental representations of the problem, which better guide subsequent searches of both internal information (knowledge) and external information (data); **2)** From an expertise-development perspective, expertise is characterized by the iterative internalization of external information (data) with initially available internal information (knowledge) to form new knowledge.

With the above interpretation of expertise, it is clear and straight forward to interpret existing empirical evidence under this new framework. The structured versus opportunistic behaviors in both production and comprehension activities can be explained by changes in the relative dominance of internal and external information during problem solving. When internal information (knowledge) dominates the evolution of the mental representation, more systematic, forward and top-down guided behaviors are present. When external information (data) dominates the evolution of the mental representation (most probably due to lack of internal information, i.e. knowledge), more opportunistic, backward and bottom-up behaviors are present. This postulate works according to the plan retrieval and creation theory, grounded in empirical studies of production tasks and also unifies the hypothesis-driven and data-driven theory grounded in empirical studies of comprehension tasks. Our postulate also explains the formation of stronger versus weaker mental representations by expert and novices respectively, since experts possess a superior (larger and more varied) repository of internal information (knowledge). With this larger repository of knowledge, the observation of experts' semantically organized memory chunks and mental representations that are less affected by external conditions (program language syntax, availability of a certain type of data) can be explained as experts are less dependent on external information (data) given their superior internal information (knowledge) repository. Actually, in (Romero, et al., 2002), (Romero, et al., 2003), (Fleming, et al., 2008) and (Fleming, et al., 2008), experts were observed to only use additional external information (data) (visualizations provided together with code in those empirical studies) when a task was "sufficiently challenging". With our framework, this again may be explained as that only when internal information (knowledge) is not sufficient (task becomes challenging) does external information (data) begin to dominate the evolution of mental representation. In (Fleming, et al., 2008) and (Fleming, et al., 2008), the authors stated that both static (roles of key data structures and threads, thread's and object's life cycles) and causal (conditions for a certain thread behavior to happen, interactions between static

objects) knowledge are critical to build a strong mental representation with the realm of concurrent programming. We postulate the lack of knowledge, which *is* the essential part of programming expertise that is utilized by human to solving programming problems, causes the unsuccessful searching and formulating of causal information in programmers' mental representations.

Based on our new generalized conceptual framework of the application and development of expertise, we assert several implications for related research fields. For software engineering, with our interpretation that external information (data) patches a lack of internal information and becomes a dominant factor in the evolution of the mental representation, it is definitely promising and surely necessary to develop new engineering interventions that provide information in some form, including tools, schemas, notations, languages and so on. However, we also re-confirm that no silver bullet is available. At the end of the day, the success of a complex problem solving task in which expertise plays a role will require a strong mental representation with a large repository of internal information (knowledge). For programming pedagogy, our interpretation of the importance of internal information (knowledge), suggests that programming education should emphasize helping students to iteratively build a superior repository of knowledge with both declarative (concepts and solutions) and procedural (schemas) information, through *challenging* tasks. This asserts again the importance of practice, and more critically, the importance of practice with challenging tasks (as the acquisition of knowledge is not an easy journey).

3.1.6 A COMPARISON WITH OTHER COGNITIVE MODELS

Other code cognition models have been proposed to abstract how programmers use existing knowledge and external representations to meet the goals of code cognition tasks. Major components of all these cognition models include: 1) a knowledge base that contains both general knowledge and new knowledge related to the software under consideration, 2) an internal working representation of the software under consideration (i.e. a mental model), and 3) a process by which the knowledge base is

used to build the mental model. In this section, we compare and contrast our conceptual framework with previous cognition models.

Generally, we have a slightly different definition of terms in our conceptual framework. The knowledge in our framework only refers to general knowledge that represents expertise and may be retrieved for further problem solving. It may include specific knowledge of a certain piece of software or specific knowledge of the procedure to fulfill a certain task, but does not include particular knowledge of a certain task regarding a certain piece of software. Rather, the knowledge in our framework has general applicability to future problem solving. The particular knowledge of a program under consideration resides in the mental representation defined in our framework. That is, we expand the content of mental representation to include related and re-organized information regarding the current code task, a specific task towards a certain piece of software. Our definition of data is basically equivalent to definitions of external representations in previous cognition models.

Compared to the Letovsky “high-level comprehension model” (Letovsky, 1986), our conceptual framework has the following similarities and differences. First, both his model and our framework argue that knowledge and external representation affect the formation of mental representations. However, while the Letovsky model abstracts the formation of mental representation as the building of annotations (links) between program goals and implementations, our conceptual model does not make this specification. We argue the dynamic aspect of mental representation to not only contain static information such as mapping of goals and implementation, but also procedural information such as guidance of the data searching and knowledge accessing processes. Also, Letovsky’s model states that external representations are “assimilated” to knowledge during problem solving procedure. With our more rigorous definition of knowledge (retrievable expertise generally applicable for future problem solving), our conceptual framework specifically states that only repeatedly formed mental representation may internalize into knowledge. That is, our view of this process is that external

representations are incorporated into the (transitory) mental model, but only through repeated formation of mental models would this external representation be assimilated into generally applicable knowledge.

Shneiderman and Mayer's "chunking model" (Shneiderman & Mayer, 1979) argues that programmers recode a program in short-term memory into an internal semantic representation via a chunking process with the assistance of long-term memory of semantic and syntactic knowledge. Similar to our conceptual framework, their model also indicates the effect of knowledge in building mental representation. However, their model assumes a direct mapping of elements from knowledge to mental representation instead of a "retrieval-accessing" procedure as stated in our conceptual framework. Also, the "chunking model" addresses the difference of design and comprehension activity as two different directions of building mental representation (from problem to program for design and from program to problem for comprehension) while our conceptual framework views the major differences in design and comprehension activity as the volume and availability of knowledge and data, respectively. The "chunking model" also simplifies the external representation as only program code while our framework considers a variety of external information.

Brooks' model (Brooks, 1983) regards program cognition as "bridging" two domains of knowledge, the problem and programming domains, with some intermediate domain knowledge. All three domains of knowledge (problem, programming and intermediate) may be used directly to generate hypotheses based on the current mental representation. Intermediate knowledge may further be used directly to map knowledge between the problem and programming domains. Hypotheses generated through knowledge of each domain are verified through "beacons" against external representations in each domain (e.g. requirements documentation are the external representation in the problem domain; code is the external representation in the programming domain; and detailed design documents are the external representation in the intermediate domain). Brooks' model emphasizes a hypothesis-driven

comprehension process but also indicates that beacons are the main vehicles for verification. In our framework, the hypothesis-driven comprehension process is interpreted as the guidance provided by the formulated mental representation and the driving force of “beacons” is interpreted in the context of retrieved knowledge and fetched data that affect the formation of the mental representation.

The Soloway, Adelson and Ehrlich model (Soloway & Ehrlich, 1984) (Soloway, et al., 1988) describes the program understanding process as the matching external representations to programming plans (knowledge) using rules of discourse (expertise). In their model, knowledge is solely related to the program under consideration, but the rules of discourse are generally applicable expertise, which is very different from our scope and definition of knowledge. Also, their model argues for a top-down constructed mental representation with a hierarchy of goals and plans in which any matched pair of goal and plan becomes new knowledge of the program, which is also different from our view of the mental representation, which contains much more than matches of goals and plans.

The Pennington model (Pennington, 1987) (Pennington, 1987) contains an iterative process of building the mental representation. She argues that a program model (control-flow abstractions) is the first mental representation built bottom-up when code is completely new to programmers. Recognition of code patterns based on knowledge (plan, data structure, etc.) drives the formation of the program model. Then, a situation model is built, also bottom-up with real world domain knowledge. Cross-referencing allows the two models to change according to changes in one another. The Pennington model is most similar to ours in terms of the evolution of mental representation while most other models assume a mental representation that functions only to maintain static information. Also, the Pennington model explicitly indicates the impact of the program model, which is formed first, on building the situation model later in the comprehension process. This corresponds to our description of an evolving mental representation formulated by knowledge and an external representation that guides further search for data and access of knowledge. However, Pennington’s model does not rigorously

delineate problem-specific knowledge from knowledge that is generally applicable to problem solving, as does the internalization process described in our framework.

As a summary, our conceptual framework reveals the evolution of the mental representation and the dual character of this evolution: 1) the mental model evolves through knowledge retrieval and data access, and 2) the evolving mental model guides the accessing of further knowledge and the searching for other data. This is not described or clearly stated in previous models. Also, our framework generalizes different behavioral procedures (opportunistic vs. structured, top-down vs. bottom-up, etc.) through the interpretation of the relative dominance of knowledge versus data in formulating the mental representation, which is not addressed or not clearly stated by previous models. Furthermore, while most previous models only focus on the cognitive process within one certain programming activity, our conceptual framework not only illustrates the cognitive process of problem-solving (for design and comprehension activities), but also captures the expertise development process by illustrating the internalization of generally-applicable knowledge. On the other hand, many previous models provide a more detailed description of the content and structure of knowledge and mental model that our framework does not supply. For example, many previous models indicate that links or mappings or matches of goals (domain elements) and plans (programming elements) are major components of the “knowledge of program under consideration” (defined as part of the mental representation in our framework). To further detail and refine our framework, it is important to understand the structure and content of general knowledge and mental representation (both procedural information and static information) and the interactive mechanism among mental representation, knowledge and external data.

3.2 CONCURRENCY-RELATED BARRIERS

The conceptual framework for the development of expertise in the programming domain that we have developed through our survey and analysis of empirical studies emphasizes the importance of

knowledge in problem-solving activities. We extend this work to explore the content and structure of programming knowledge with regard to expertise for programming with concurrency.

In this section, we describe our “misconception hierarchy” notion. In our work, misconceptions are incorrect ideas that students hold about the state or behavior of a computer system under study. These misconceptions were identified through analysis of student explanations in sequential and circumstantial questions that were posed to them. Using a grounded theory approach, these misconceptions were then grouped into categories (description, terminology, concurrency, implementation, uncertainty). An analysis of student performance indicated that these categories are inter-related. Specifically, misconceptions that exist at lower levels can interfere with comprehension at higher levels.

3.2.1 QUALITATIVE ANALYSIS AND GROUNDED THEORY

The study of human factors in the interaction between human and technology is complex and difficult. Different from other factors, human behaviors are hard to be described and explained through statistics or other quantitative models. Qualitative data are data represented as words and pictures, not numbers (Gilgun, 1992). Qualitative methods are methods used to focus on the study and analysis of qualitative data. These methods allow us to delve into the complexity of problems in studying human factors rather than to abstract the complexity away. Thus, qualitative methods result in richer and more informative findings. Qualitative methods also have drawbacks in that the results are usually harder to summarize or simplify than quantitative ones, but this helps in expressing the complexity inherent in human factors research. As stated in (Taylor & Bogdan, 1984), qualitative research methods were designed, mostly by educational researchers and other social scientists, to study the complexities of human behavior.

Grounded theory is a general methodology for developing theory that is grounded in data systematically gathered and analyzed. The theory evolves during actual research that uses grounded

theory methods through continuous interplay between analysis and data collection. Grounded theory methodology shares the same data source as other qualitative research methods (a combination of qualitative and quantitative data) to study human factors, but it is more directed at developing substantive theories rather than general ones and mandate a verification of resulting hypotheses through iterative collection and analysis of data according to (Strauss & Juliet, 1994). Authors of (Thomas & James, 2006) challenge the grounded theory methods in three aspects: 1) the generated findings are not qualified to be termed as theories; 2) it is not possible to follow the method's requirements of "forming grounded theory without preconceptions"; and 3) the actual process of developing inductive knowledge is not clear.

We address the concerns of legitimacy of grounded theory and qualitative analysis method in our presentation of data analysis and result findings. First, we do not regard our findings as rigid as a theory, but rather a collection of descriptions of the content and structure of knowledge possessed by students with different levels of expertise and their behaviors and procedures of knowledge acquisition. Second, we possess very limited prior assumptions and always formulate exploratory questions towards the emergency of our findings such as "what" and "how" questions. We also report the process of reaching our final results with unsuccessful steps in data analysis to illustrate the inductive course of discovery. As shown in Figure 6, we adopted iterative qualitative analysis of our data collected from the posttest of spring 2010 study and the midterm exam of spring 2013 study. Our major foci are subjects' responses to reasoning questions (whether a certain scenario will happen and why). Subjects' explanations of their answers are the major source of our data. We describe in the following sections our process of analyzing student performance and the path that led to the notion of a "misconception hierarchy".

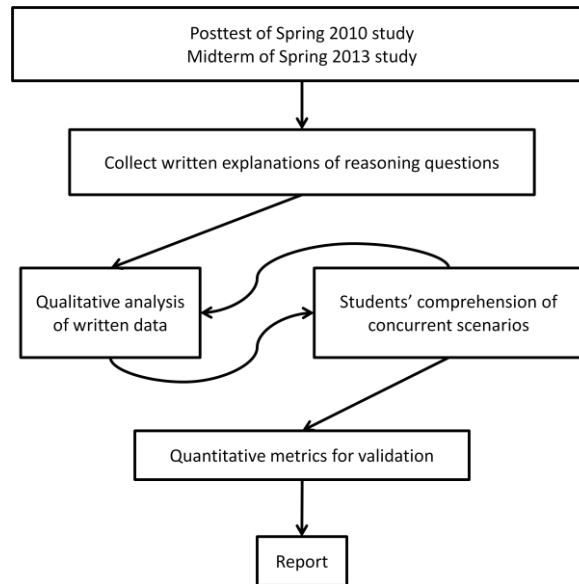


FIGURE 6 QUALITATIVE RESEARCH METHOD

3.2.2 FORMULATING THE MISCONCEPTION HIERARCHY

The formation of our misconception theory was the result of several iterations of data analysis. First, we attempted to correlate students' performance with the metrics designed to describe the complexity of the scenario that students are asked to evaluate. We defined a list of metrics: 1) number of threads, 2) number of lock/unlock operations involved, 3) number of context switches involved, 4) number of waits involved and 5) number of member functions of shared objects involved, to characterize the complexity of each question. However, we soon noticed that some questions with very similar metrics had greatly different subjects' performance. For example, questions 4.d and 4.e both involve 3 threads, 1 lock/unlock operation, 2 context switches, 0 waits and 2 member functions, but the ratios of number of correct answer versus the number of incorrect answer for two questions are 13/1 and 3/12 respectively. Questions 5.d and 5.f demonstrate the same phenomenon, as shown in Table 11.

Therefore, we decided to look more closely into the explanations provided by subjects to figure out what makes them choose correct or incorrect answers in different questions. For example, consider question 1.b Figure 7 describing a scenario from the single-lane bridge problem in which two threads, redCar1 and redCar2, exist in the system (see section 2.1 for a detailed description). Thread redCar1

invokes the `redEnter()` method and has already returned when a context switch occurs and the `redCar2` thread begins to run. One of the sub-questions asks whether it is now possible for the `redCar2` thread to invoke the `redEnter()` method and block on the monitor lock. The answer to this question should be NO. Only two threads exist in the system and `redCar1` should have released the monitor lock before it returned from the `redEnter()` method. Thus, it is not possible for `redCar2` to block on the monitor lock. In answering this question, 9 out of 15 subjects chose the correct answer (NO). However, in looking closely at their explanations, we found that 7 of them thought that the monitor lock would only block blue car threads and regarded the monitor lock in the question as an `okToEnter` condition variable. One of them misunderstood the meaning of the term “block” as “own” or “has” and thought that `redCar1` already owned the monitor lock since it was on the bridge and that `redCar2` could thus not own the same lock. Another student, however, did not understand the question and thought that `redCar2` should not “block” on the monitor lock but lock the monitor lock. Thus, by reading the explanations given by the students we found that actually none of the 9 students who gave the correct answer really understood the monitor lock and its mechanism.

TABLE 11 QUESTIONS WITH SIMILAR METRIC AND DIFFERENT PERFORMANCE

Question	#Threads	#Lock/ Unlock	#Context Switch	#Wait	#Functions	Correct	Neutral	Incorrect
4.d	3	1	2	0	2	13	1	1
4.e	3	1	2	0	2	3	0	12
5.d	3	1	2	1	2	13	0	2
5.f	3	1	2	1	2	5	0	10

1. Suppose that only two threads exist in the system: **redCar1** and **redCar2**. Suppose further that **redCar1** has invoked the `redEnter()` method, and has returned. A context switch occurs and the **redCar2** thread starts to run.

Could the following event sequence happen next? Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

b. **redCar2** invokes `redEnter()`, then blocks on the monitor lock.

YES NO

FIGURE 7 SAMPLE QUESTION IN POSTTEST OF SPRING 2010 STUDY

By performing this type of detailed analysis of student reasoning, we were able to construct an initial list of misconceptions. Based on this list of misconceptions, we looked back into subjects' answers to particular questions, validated and modified the list through grouping, tabulating and correlating the misconceptions on our list to subjects' choice of answers and their explanations. The questions were each designed to test student's knowledge of particular concepts. We found that the occurrence of misconceptions for each subject was not uniformly distributed. Our analysis of the student explanations also revealed that the questions were not always evaluating the intended concept because "lower level" misconceptions interfered with the ability to reach the intended concept. Consider questions 4.d and 4.e mentioned above as another example. These two questions are aimed at testing the subjects' ability to consider multiple possible inter-leavings in an execution. Most of the students were not able to answer both of these questions correctly and the majority failed on question 4.e. However, by looking closely at their explanations, we found the reason for the failure does not truly stem from students' inability to consider the possible interleaving, as expected. Actually, all 9 subjects failed in 4.e because of misconceptions about the monitor lock. Some of them confused it with the `okToExit` or `okToEnter` condition variables. Others were ignorant of the mechanism of the monitor lock so they succeeded in question 4.d, which does not deal with the monitor lock concept but failed in 4.e.

Finally, we converged on and became confident in the idea that student's misconceptions about concurrency and synchronization cannot be fully captured with a simple list of confusions or misunderstandings in concepts, terminologies and mechanisms. Rather, they are correlated with one another, interacting in a hierarchical architecture so that it is not possible to examine higher level misconceptions without first teasing out the impact of lower-level misconceptions, or ensuring that students first have a firm grasp of lower level concepts. In other words, to understand higher level concepts, students must first rid themselves of lower level misunderstandings.

We introduce a misconception pyramid (Figure 8) that captures common misunderstandings that students exhibited when reasoning about a concurrent system. The hierarchical structure of the misconceptions captures the difficulty and dependency relations of understanding the concepts at each level. Understanding concepts at higher levels of the pyramid requires an understanding of the concepts at lower levels first. Descriptions of the types of misconceptions one might find at each level are presented in Table 12 (note the levels in table are in a reverse order to accompany top-down of presentation), which was constructed based on misconceptions identified in the literature and also those that we encountered in our analysis of subjects' explanations of their reasoning in this study.

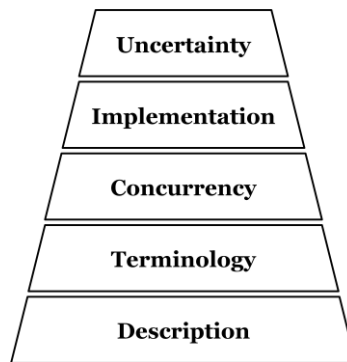


FIGURE 8 MISCONCEPTION PYRAMID

The bottom level of the pyramid is the description level and includes misconceptions about the requirements, constraints and other details of a concurrent system at the level of the “story” about the red cars and blue cars. For example, some subjects wrote explanations such as “redCar2 should wait for redCar1 to invoke the redEnter() method first” or “redCar1 should block the bridge first” demonstrate one common misconception at this level: that the thread labels redCar1 and redCar2 were the actual running order of the threads.

The next level of the pyramid includes misconceptions related to terminology we used in describing concurrent scenarios. A typical example is the misunderstanding of the meaning of “block on” a conditional variable/monitor lock as “hold/own” a conditional variable/monitor lock. This kind of misconception can be seen throughout the explanations given by subjects in our study. Most students

who held this kind of misconception did so consistently, causing them to fail on a particular group of questions. Typical students' explanations that illustrate this level of misconception include but are not limited to "okToEnter is already blocked" or "monitor is already blocked by redCar2".

The third level of the pyramid is the concurrency level, which includes misconceptions about basic thread behaviors such as context switching and the thread life cycle. For example, some students seemed to think that a context switch could not happen while a thread was executing in a critical section and many students thought that a context switch is not allowed during the execution of a method and regarded the whole method body as uninterruptible. Some typical students' explanations are "redCar2 should receive return call then switch out" or "because redCar2 has not done its activity (so it cannot be context switched out)".

TABLE 12 INITIAL MISCONCEPTION PYRAMID TABLE

Description Level	
D1	Misconceptions of system and/or problem descriptions
Terminology Level	
T1	Misconceptions of the meaning of "invoke/call" a method
T2	Misconceptions of the meaning of "return" from a method/invocation
T3	Misconceptions of "block" on a monitor lock as "hold/has" a monitor lock
T4	Misconceptions of "block" on a conditional variable as "hold/has" a conditional variable
Concurrency Level (thread behavior)	
C1	Misconceptions about context switching
C2	Misconceptions about the thread life cycle
Implementation Level	
I1	Misconceptions about conditional variables and the wait/signal mechanism
I2	Misconceptions about monitor lock
I3	Misconceptions about block and unblock mechanism
Uncertainty Level	
U1	Confused about space of executions and thread interleavings

*note the levels in table are in a reverse order to accompany top-down of presentation

The fourth level of the pyramid is the implementation level, which is related to detailed implementation mechanisms such as the monitor lock and condition variables and their functionalities. By investigating the subjects' answers and explanations in our study, we found that few subjects were clear on the basic monitor programming structure. We believe that this is greatly related to students' misunderstandings in the three previous levels. If students do not understand the context switch, they

are not able to appreciate the actual purpose and corresponding mechanism of the monitor lock. Misunderstandings of different terminologies also lead to confusion about the workings of monitor programming structures and functions.

The top level of the pyramid is concerned with failures in dealing with uncertainty; that is, the inability to envision or manage all the possible threads inter-leavings and execution scenarios. While this problem is often cited as the main source of difficulty in the comprehension of concurrent program executions, we found that this level of difficulty was not seen in our study, as students tended to fail much earlier in the pyramid, and thus were not even exposed to these higher-level issues.

3.2.3 EXTENDING THE STUDY

We recognize that our initial spring 2010 study was carried out over a relatively small group of subjects and with a particular implementation of concurrent systems (C++ PThread implementation of a shared memory system). Therefore, the resulting initial misconception hierarchy is incomplete and not applicable to a more abstract level that characterizes the knowledge structure of expertise in programming with concurrency. Also, although we strictly followed the research method of forming grounded theory with iterative paths of qualitative analysis, these analyses were performed by a limited number of researchers, which may cause biases. Realizing these limitations, we carried out an extended second study to re-evaluate and refine our theory. This study is different from the initial one from several different aspects as we illustrate below.

3.2.3.1 INTRODUCING AN EXTENDED LANGUAGE INDEPENDENT PSEUDOCODE SYSTEM

Based on Tew's language independent test on a pseudocode system (Tew & Guzdial, 2011), we created an extension that includes concurrency concepts in both shared memory and message passing models. Therefore, instead of using programs implemented in an actual programming language, we use this language-independent pseudocode system in our comprehension test to eliminate effects based on programming language. By excluding the factor of subject's difference in familiarity with different

programming languages, in the spring 2013 study, we were able to test the comprehension of concurrent systems purely based on an extended pseudocode abstraction that covers both the shared memory and message passing forms of concurrency.

A selected subset of this pseudocode can be seen in tables Table 13 through Table 16. Table 13 shows the pseudocode notation associated with assignment statements and the pseudocode notation associated with conditional statements. In Table 14 we provide an example of the pseudocode we have devised for representing concurrent execution. Pseudocode designed to represent constructs in shared memory model are seen in Table 15 and pseudocode designed to represent constructs in message passing model is seen in Table 16. In Table 17, we demonstrate the use of this pseudocode to specify a shared memory solution to the Bounded Buffer problem.

TABLE 13 SAMPLE PSEUDOCODE ELEMENTS NOT RELATED TO CONCURRENCY

Non-concurrency Related Pseudocode	
Simple Statement <code>variable = expression</code> Simple statements are executed atomically. Assignment is an example of a simple statement	<code>total = 0</code> <code>name = "John Smith"</code> <code>condition = True</code> <code>height = 3.3</code>
If Statement (Conditional) IF <i>condition</i> THEN statement(s) ELSE IF <i>condition</i> THEN statement(s) ELSE statement(s) ENDIF The calculation of <i>condition</i> is not necessarily atomic if it involves function call statements. However, the choice of branch based on a calculated <i>condition</i> value is executed atomically.	IF testScore >= 90 THEN PRINTLN "A" ELSE IF testScore >= 80 THEN PRINTLN "B" ELSE IF testScore >= 70 THEN PRINTLN "C" ELSE PRINTLN "F" ENDIF testScore = 88 Output B

TABLE 14 PSEUDOCODE EXTENSIONS ON CONCURRENCY

Parallel Execution Statements	
<p> PARA <code>statement(s)</code> ENDPARA </p> <p>Statements within the PARA/ENDPARA block are executed concurrently.</p> <p>Atomic statements within PARA/ENDPARA are executed in any order.</p> <p>Statements defined in a function that is called within the PARA/ENDPARA block are executed sequentially.</p> <p>Statements defined in functions that are called within a PARA/ENDPARA block are executed in any order of interleaving with simple statements within the same PARA/ENDPARA block.</p> <p>Statements defined in two functions that are called within the same PARA/ENDPARA block are executed in any order of interleaving while statements from any one of the functions are executed in their order of definition.</p>	<p> PARA PRINT "hello " PRINT "world " ENDPARA </p> <p>Output possibility 1: hello world possibility 2: world hello</p>
	<p> DEFINE print() PRINT "hi" PRINT "there" ENDDEF </p> <p> PARA print() ENDPARA </p> <p>Output hi there</p>
	<p> DEFINE print() PRINT "hi" PRINT "there" ENDDEF </p> <p> PARA print() PRINT "world" ENDPARA </p> <p>Output possibility 1: world hi there possibility 2: hi world there possibility 3: hi there world</p>

TABLE 15 PSEUDOCODE EXTENSIONS ON SHARED MEMORY MODEL OF CONCURRENCY

Shared Memory Concurrency	
<p>Exclusively Accessed Statement</p> <pre> EXC_ACC statement(s) END_EXC_ACC </pre> <p>Only appears within a function definition.</p> <p>When one function call executes statements inside an EXC_ACC/END_EXC_ACC block, other function calls that read or modify the same variables that appear inside the markers may not execute until the first function call completes or executes a WAIT function.</p>	<pre> x = 10 DEFINE changeX(diff) EXC_ACC x = x + diff END_EXC_ACC ENDDDEF PARA changeX(1) changeX(-2) ENDPARA PRINTLN x Output 9 </pre>
<p>Wait and Notify Functions</p> <pre> WAIT() NOTIFY() </pre> <p>Only be called inside a EXC_ACC/END_EXC_ACC block.</p> <p>Once a WAIT() function starts execution, another function call that reads or modifies variables inside the EXC_ACC/END_EXC_ACC block may execute.</p> <p>Once a NOTIFY() function is executed, all WAIT() functions finish their execution.</p> <p>Both WAIT() and NOTIFY() functions are atomic.</p>	<pre> x = 10 DEFINE changeX(diff) EXC_ACC WHILE x + diff < 0 DO WAIT() ENDWHILE x = x + diff NOTIFY() END_EXC_ACC ENDDDEF PARA changeX(-11) changeX(1) ENDPARA PRINTLN x Output 0 </pre>

TABLE 16 PSEUDOCODE EXTENSIONS ON MESSAGE PASSING MODEL OF CONCURRENCY

Message Passing Concurrency	
Message Variable MESSAGE.message-name(value...) <p>A special message variable that carries a collection of values. The <i>message-name</i> is used to distinguish message variables from one another.</p>	m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world")
Send Statement Send(message variable).To(object) <p>Send a message specified by message variable to a receiver object.</p> <p>A send statement is asynchronous, which means that the order in which messages are received may differ from the order in which they were sent.</p>	m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world") Send(m1).To(r1) Send(m2).To(r1)
Receive Statement ON_RECEIVING message statement(s) message statement(s) ... <p>Accept the next message and execute statement(s) according to the type of the message.</p>	CLASS Receiver DEFINE receive ON_RECEIVING MESSAGE.h(var) PRINT var MESSAGE.w(var) PRINTLN var ENDDEF ENDCLASS m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world") r1 = new Receiver() r1.receive() Send(m1).To(r1) Send(m2).To(r1) Output possibility1: hello world possibility2: world hello

TABLE 17 PSEUDOCODE IMPLEMENTATIONS OF BOUNDED BUFFER

Shared Memory Model	Message Passing Model
<pre> CLASS Buffer DEFINE initialize Buffer(capacityVal) items = [] capacity = capacityVal ENDDDEF DEFINE produce(itemVal) EXC_ACC WHILE length(items) > capacity DO WAIT() ENDWHILE items[length(items)] = itemVal NOTIFY() END_EXC_ACC ENDDDEF DEFINE consume() EXC_ACC WHILE length(items) < 1 DO WAIT() ENDWHILE item = items[0] del items[0] NOTIFY() END_EXC_ACC return item ENDDDEF ENDCLASS </pre>	<pre> CLASS Producer DEFINE initialize Producer(bufferVal) buffer = bufferVal ENDDDEF DEFINE run() WHILE True DO buffer.produce(randNum(0,10)) ENDWHILE ENDDDEF ENDCLASS CLASS Consumer DEFINE initialize Consumer(bufferVal) buffer = bufferVal ENDDDEF DEFINE run() WHILE True DO PRINTLN buffer.consume() ENDWHILE ENDDDEF ENDCLASS </pre>

3.2.3.2 INCLUDING COMPREHENSIVE AND INTENSIVE TRAININGS

We realize that the online multimedia tutorial and corresponding quizzes used in spring 2010 study may not impose sufficient cognitive workload on subjects to cause them to understand the materials in depth. Therefore, we integrate the training part of the spring 2013 study into the delivery of an upper-level undergraduate course. Therefore, students received intensive training on the knowledge and concepts of concurrency through lecture talks and graded assignments and quizzes. The usage of the pseudocode system with different concurrency scenarios was illustrated in class in both shared memory and message passing models and was covered in completing graded assignments. The quizzes also required students to use the pseudocode system to implement dining philosopher and readers-writers scenarios in both shared memory and message passing models. Thus, by the time of midterm exam, students should have mastered all concurrency concepts involved in implementing shared memory and message passing concurrent systems and been very familiar with the usage of the pseudocode.

3.2.3.3 UPDATED TEST MATERIAL AND TEST GROUPS

We used same single-lane bridge scenario in the midterm exam of spring 2013 study as in the spring 2010 study. However, we showed the implementation of this program using the extended pseudocode system (see section 6.5). Before the test, we randomly assigned all students into two equivalent groups. The first group of students finished the shared memory model portion of the test first then took the message passing model portion. The second group took the test in reverse order. This design is to eliminate the practice effect. The test questions in the two parts, shared memory and message passing, are worded differently, according to the nature of two different models but were designed to cover equivalent scenarios, as shown in Figure 9 and Figure 10.

PARA

```
redCarA.run()
redCarB.run()
blueCarA.run()
```

END_PARA

Suppose **redCarA** has called the *redEnter()* method on line 9 but has not returned. Then **redCarB** invokes its *run()* method and calls the *redEnter()* method but also has not returned.

Decide if each of the scenarios below (k-t) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(m)**redCarB** returns from the *redEnter()* method, then calls the *redExit()* method on line 19 and blocks on the `EXC_ACC` marker on line 20.

YES NO

Explanation:

FIGURE 9 A SAMPLE QUESTION IN SHARED MEMORY MODEL PART OF TEST

PARA

```
bridge.start()
redCarA.start()
redCarB.start()
blueCarA.start()
```

END_PARA

Suppose **redCarA** has sent the *redEnter* message but has not yet received any messages. Then **redCarB** invokes its *start()* method, and sends the *redEnter* message but has not yet received any messages.

Decide if each of the scenarios below (k-t) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(m)**redCarB** receives a *succeedEnter* message, then sends a *redExit* message and receives `MESSAGE.succeedExit(2)`.

YES NO

Explanation:

FIGURE 10 A SAMPLE QUESTION IN MESSAGE PASSING MODEL PART OF TEST

3.2.4 BUILDING A MISCONCEPTION HIERARCHY OF CONCURRENCY CONCEPTS

We apply the same qualitative analysis method as in the 2010 study to the collected students' explanation in the 2013 study to validate the content of a hierarchical misconception system. We confirm that the content of our previous misconceptions pyramid for a shared memory model still applies in this study. With slight modification, we are also able to extend this hierarchy to cover the message passing model. The refined misconception types are illustrated in Table 18 and the detailed misconceptions found in this repeated study are listed in Table 19.

One major misconception seen with message passing is a misunderstanding of the send function. In [C1]M3, we see that some students interpret a message send as a method call that may not happen unless the condition is satisfied at the receiver. For example, in a scenario in which redCarA successfully entered the bridge, a student indicated that redCarB could enter the bridge but could not exit because "redCarB cannot send the redExit message until redCarA sends redExit". Some students interpret a message send as a synchronous call, writing "redCarA calls redEnter first and the bridge has to process its message first before any other messages."

The next major misconception, seen in [C1]M4, is the assumption that the occurrence of an event (entering/exiting the bridge) implies that an acknowledgement message has been received. For example, one student wrote, "redCarA is not on the bridge since it has not received any message yet".

Students exhibited difficulty in fully managing the asynchronous nature of message passing systems. Table 19 lists four scenarios that may actually happen in asynchronous systems, but due to the nature of single-lane bridge problem, students were only tested on scenario 1 (different senders, same receiver) and scenario 3 (same sender, different receivers). Looking closer into students' explanations, we see that student understanding is quite unreliable – among the six students who displayed the misconception that messages are necessarily received in the order sent, two of the six applied this only to messages from the same color cars but correctly reasoned about messages from different color cars.

The major misconception in reasoning about shared memory was a conflation of the order of method invocation/return with the order of obtaining/releasing the lock (e.g. “redCarA has not returned from the redEnter method so it must still hold the lock”), likely because most students had prior experience in Java, in which entry to a *synchronized* method may be thought to occur simultaneously with obtaining the lock and release of the lock may be thought to occur simultaneously with return from the *synchronized* method.

Also, some students showed misconceptions in differentiating lock mechanisms from wait/notify mechanisms. When the question asked whether a particular thread will be blocked on the acquisition of the lock, the students explained that “the condition is not satisfied yet for the thread to get the lock” or “the first red car has not exited yet, so the second red car cannot get the lock and execute redExit() function”. This misconception is similar to that in which a message send is interpreted as a method call that cannot happen unless the condition is satisfied at the receiver. In both cases, the student’s incorrect reasoning is based on global knowledge not actually available to the current thread or process.

TABLE 18 THE REFINED MISCONCEPTION HIERARCHY

Description Level	
D1	Misconceptions about the system and/or problem description
Terminology Level	
T1	Misinterpretation of a term that describes thread or process behavior
Concurrency Level	
C1	Misconceptions about thread or process behaviors
Implementation Level	
I1	Misconceptions about synchronous mechanisms
I2	Misconceptions about asynchronous mechanisms
Uncertainty Level	
U1	Confusion about the space of executions; include impossible execution sequences or fail to consider possible execution sequences

TABLE 19 DETAILED MISCONCEPTIONS FOUND IN STUDY

Shared Memory	
[D1]S1: Conflate order of cars with their thread's name (#students: 3)	
[T1]S2: Misinterpret "race condition" as "different interleaving" (#students: 1)	
[T1]S3: Misinterpretation on terminology "block on" (#students: 2)	
[C1]S4: Conflate order of method return with order of entering/exiting bridge (#students: 4)	
[C1]S5: Conflate locking with conditional waiting (#students: 9)	
[I1]S6: Misinterpretation of WAIT() function's effect and conflate wait with continuous execution of the enclosing while loop (#students: 1)	
[I1]S7: Conflate order of method invocation/return with get/release lock (#students: 10)	
[U]S8: Uncertainty (#students: 2)	
Increased size of state spaced causes illogical (self-contradictory) reasoning or occurrence of above misconceptions not seen in simpler scenarios	
Message Passing	
[D1]M1: Question setting (#students: 6)	
[T1]M2: Misinterpret "race condition" as "different order of messages" (#students: 1)	
[C1]M3: Send semantics : assume ability to send depends on condition at receiver or interpret send as a synchronous method call (#students: 7)	
[C1]M4: Receive semantics: assume receipt of acknowledgement message is synchronous with the occurrence of the event (bridge entered or exited) (#students: 7)	
[I2]M5: Conflate message sending order with receiving order (#students: 6)	
Four scenarios:	
1) different senders, same receiver (covered by test problem)	
2) different senders, different receivers	
3) same sender, different receivers (covered by test problem)	
4) same sender, same receiver	
[U1]M6: Uncertainty (#students: 7)	
Increased size of state spaced causes illogical (self-contradictory) reasoning or occurrence of above misconceptions not seen in simpler scenarios	

In the initial study, we observed that some “lower-level” misconceptions cause students’ to be unable to understand other “higher-level” concepts. Therefore, the lack of those “higher-level” misconceptions in students reasoning protocols doesn’t necessarily indicate their understanding of those concepts, but just their inability to even begin to understand or misunderstand them. Therefore, we suspect that the lower-level concepts are the basis for understanding the higher-level concepts and a misconception that occurs at the lower-level should be regarded as a more severe misconception than one that occurs at a higher-level.

Using subject profiles as shown in Table 20, in which each row contains the types and frequency of each subject’s misconceptions as well as their performance, we are able to better study and validate our hypothesized hierarchical structure. In the subject profile, the first column of the table indicates the subjects’ ID number. Columns 2-6 correspond to types of the misconception hierarchy. Each cell of (subject, type) contains the number of (answer, explanation) pairs given by the subject that demonstrated the corresponding type of misconception. The column labeled “total” is a count of the total number of misconceptions for each subject. The column labeled “breadth” is a count of the number of types of misconceptions for each subject. The “breadth of misconception” for any individual students can be calculated by Equation 1. The last column is the subjects’ quantitative handicap in the corresponding tests. This value is unified across two studies based on a scale of 0-1 (handicap = 1-average-test-score) since the original total scores of the tests used in the two studies are different.

$$Breadth_{subject} = \frac{count((subject, type) > 0)}{count((subject, type))}$$

EQUATION 1 BREADTH OF MISCONCEPTION

TABLE 20 SUBJECT PROFILES

Subject	Number of Occurrences							Handicaps
	Description	Terminology	Concurrency	Implementatio	Uncertainty	Total	Breadth	
Initial Study								
102	2	6	2	14	0	24	0.8	0.40
108	3	3	4	13	0	23	0.8	0.20
109	3	0	9	11	0	23	0.6	0.17
110	7	13	3	17	0	40	0.8	0.50
113	2	5	7	23	0	37	0.8	0.20
119	1	6	5	19	0	31	0.8	0.37
122	0	4	1	9	0	14	0.6	0.03
126	0	7	3	18	0	28	0.6	0.30
138	1	5	9	11	0	26	0.8	0.03
139	1	14	10	16	0	41	0.8	0.63
141	2	17	5	16	0	40	0.8	0.20
142	0	3	1	13	0	17	0.6	0.10
145	0	0	2	15	0	17	0.4	0.20
Repeated Study								
1	1	1	29	36	25	92	1	0.42
2	0	0	8	18	13	39	0.6	0.23
3	1	0	6	14	12	33	0.8	0.48
5	0	0	5	18	12	35	0.6	0.18
6	0	1	17	17	6	41	0.8	0.18
7	0	0	0	6	5	11	0.4	0.09
8	0	0	30	34	16	80	0.6	0.32
9	0	5	17	28	11	61	0.8	0.28
10	1	0	6	10	8	25	0.8	0.24
11	0	7	17	23	11	58	0.8	0.44
12	0	2	3	11	10	26	0.8	0.34
14	0	5	17	24	17	63	0.8	0.37
15	1	4	15	18	14	52	1	0.67
16	0	6	3	5	3	17	0.8	0.26
18	0	0	0	1	4	5	0.4	0.04
19	0	8	16	25	15	64	0.8	0.22

By performing a correlation test between the total number of misconceptions and the students' handicap levels as illustrated in Table 21, we found that the number of misconceptions does not capture a student's knowledge of concurrency well (a number under 0.5 indicates weak correlation). We then performed a correlation test between the breadth of misconceptions and students' handicap levels as illustrated in Table 22. It is clear that this simple count of the type (since the total number of types of misconceptions is a constant 5, the breadth is equivalent to a count in calculating correlation) of misconceptions has a stronger correlation with handicap levels.

However, the breadth of misconception still doesn't capture the essential character of a student's knowledge structure (a correlation value around 0.5 indicates only a weak correlation). To explore the different relations between different types of misconceptions and the handicap level of a student's knowledge structure, we performed linear regressions with regard to pair of handicap level and each type of misconception. The result is shown in Table 23, through which we could clearly see that for the first four types of misconceptions, the lower-level a misconception is, the more it causes student's handicap in understanding concurrency. A possible interpretation of the relatively high coefficient of correlation between uncertainty level mistakes and handicap can be explained as the result of generalized regression of two studies, but subjects from first study's low occurrences of uncertainty mistakes are not counted as handicaps. A separate test of regressions based just on the data of the second study is illustrated in Table 24. Another explanation, which is probably more possible, is that uncertainty is a major cause of students' handicaps in understanding concurrency and may result in misconceptions at other levels. This is actually also confirmed by empirical evidence that we term as a "fall back effect". We find that even the most advanced students had difficulty when reasoning about a large space of possibilities. When students are not quite able to manage the execution space (usually over 3-4 possibilities), they tend to reduce the complexity by falling back into one of the lower level misconceptions, perhaps as a result of increased cognitive load. At this time, they either give a correct

explanation but choose an incorrect answer or conflate two concepts in a way that reduces the execution space. Figure 11 illustrates one such “fall back” effect. The question under consideration involves the phenomenon of a message send followed by an un-received acknowledgement at the sender side. Two possibilities may explain this phenomenon. One is that the message has not been received yet. The other is that the message has been received already but the acknowledgement message has not been received yet. However, when a combination of such possibilities exists in a problem statement that causes the total possible state of the system under reasoning to exceed students’ cognitive load, they adopt the strategy of assuming some un-specified factor to reduce the total number of possible states. Although the “fall back effect” exists, we think the general hierarchical level of misconceptions still holds, which means that a pure novice should conquer most of the misconceptions at lower levels before understanding a higher level concept.

As a conclusion of this repeated study towards the theory that a misconception hierarchy exists in students’ knowledge of concurrent programs, we propose that two equivalent misconception hierarchies exist in the understanding of shared memory and message passing models of concurrent systems, respectively. This misconception hierarchy is constructed with five levels of misconception, including 1) description, 2) terminology, 3) concurrency, 4) implementation, and 5) uncertainty. For each individual student, one or more misconceptions may be demonstrated in a continued cluster that may span as far as covering all five types of misconceptions (as shown with shadings in Table 20). This continuation is only broken when a misconception fall back happens due to cognitive overload from reasoning about a large execution space (subject 109, 3, and 10 as shown in Table 20).

TABLE 21 CORRELATIONS BETWEEN TOTAL NUMBER OF MISCONCEPTION AND HANDICAP LEVEL

	Total	Unified Handicap
Total	1	
Unified Handicap	0.493637	1

TABLE 22 CORRELATIONS BETWEEN BREADTH OF MISCONCEPTIONS AND HANDICAP LEVEL

	Breadth	Unified Handicap
Breadth	1	
Unified Handicap	0.579926	1

TABLE 23 LINEAR REGRESSIONS OF MISCONCEPTIONS TO HANDICAP LEVEL

	Coefficients	Standard Error	P-value
Description	0.049	0.029307	0.001253
Terminology	0.019	0.006874	5.74E-06
Concurrency	0.019	0.003414	1.35E-06
Implementation	0.011	0.001645	5.08E-10
Uncertainty	0.024	0.00466	1.73E-05

TABLE 24 LINEAR REGRESSIONS OF MISCONCEPTION TO HANDICAP LEVEL (SECOND STUDY ONLY)

	Coefficients	Standard Error	P-value
Description	0.453	0.126792	0.002799
Terminology	0.062	0.016888	3.64661
Concurrency	0.018	0.003258	5.677726
Implementation	0.014	0.002047	6.964365
Uncertainty	0.024	0.002846	8.468347

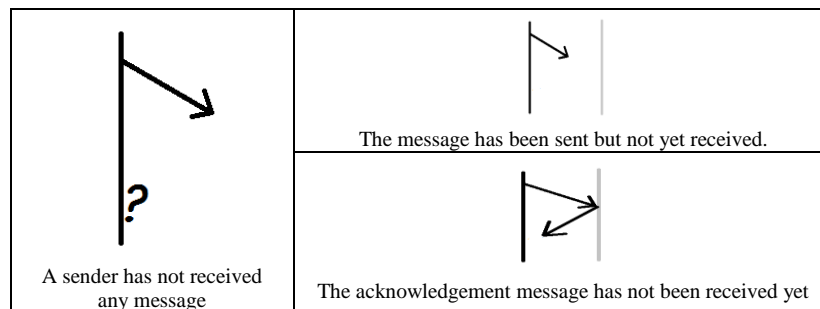


FIGURE 11 SAMPLE FALL BACK TO LOWER LEVEL MISCONCEPTION

3.3 OTHER BARRIERS

In providing the misconception hierarchy of concurrency-related concepts, we ask whether these are the only barriers to learning concurrency and the answer is obviously no. Based on our conceptual framework of the nature of programming expertise, we expect that the larger the knowledge repository possessed by programmers, the more expertise they will show in problem solving. This knowledge repository not only includes concurrency-related knowledge, but also a variety of other knowledge. To examine this idea, we present a case study of two subjects in our spring 2013 study. Both of the students implemented the same single-lane bridge problem with their preferred programming language during the final exam. Both students used Java and Eclipse as their programming language and development environment. The problem was modified so that students could not directly translate the pseudocode they saw on the midterm exam. A feature of Eclipse was used to record code histories from both subjects. These code histories were studied to reveal that concurrency-related concepts are not the only factors in successful problem solving in this domain.

The raw code histories recorded by Eclipse are randomly named txt files dumped in randomly named directories under the “.history” folder in the Eclipse workspace. Therefore, we devised and implemented a smart text reader with Python to search related code files, rename them and copy them into organized folders. After that, we examined and sorted the code files according to their last modified time to form a series of files recorded along the coding history. Then, the UNIX diff command with “-wic” output was used to elicit the differences between every two subsequent files in the history. For example, if three files exist in the coding history, two comparisons are made, between file 1 and 2 and between file 2 and 3. We then used a script to write the sequence of modifications (the output of the diff command) into a text file for our case study analysis.

The analysis took several paths. First, we identified the targets of each modification made by the subjects. Then we grouped their targets to infer their goals of making a sequence of modifications.

Finally, we infer the possession or lack of knowledge that causes their success or failure in accomplishing their goals. The two subjects in this study are regarded as intermediate (subject I) and novice (subject N) with respect to programming expertise in general.

TABLE 25 SAMPLE MODIFICATIONS IN CODE HISTORY

No	Content
1	<pre> *** intermediate_con/Bridge_2013_5_2_18_22.java 2013-05-02 14:22:28.000000000 -0400 --- intermediate_con/Bridge_2013_5_2_18_23.java 2013-05-02 14:23:20.000000000 -0400 ***** </pre>
2	<pre> *** 107,117 **** } public void statusCheck() { ! if (events checkFreq) System.out.println("Status Check of Usage: " + redCarsFinished + " " + blueCarsFinished); System.out.println("Status Check of Bridge: " + redCars.size() + " " + blueCars.size()); } public synchronized boolean isAllExit(int total) { if (redCarsFinished+blueCarsFinished == total) return true; </pre>
3	<pre> --- 107,119 ---- } public void statusCheck() { ! if (events > checkFreq) { System.out.println("Status Check of Usage: " + redCarsFinished + " " + blueCarsFinished); System.out.println("Status Check of Bridge: " + redCars.size() + " " + blueCars.size()); } + } + public synchronized boolean isAllExit(int total) { if (redCarsFinished+blueCarsFinished == total) return true; </pre>

Table 25 illustrates an example modification (one modification in the coding history; the whole history consists of tens of such modifications). The 1st part of this modification includes the names and last modified times of two subsequently recorded history files. For the example in Table 25, the first file is saved at 14:22:28 and the second file is saved at 14:23:20. A series of “*” at the end of the 1st part separates it from the later parts. The 2nd and 3rd parts of the modification have lines from both files involved in the modification (i.e., lines that are different in two subsequently recorded files.) A line with an exclamation mark indicates a change in that line. A line with a plus mark indicates a newly added line. A line with a minus mark indicates a removed line. Added lines only show up in the 3rd part (later file)

while removed lines only show up in the 2nd part (previous file). For the modification shown in Table 25, we conclude that the subject made a syntax correction of an if statement in the method `statusCheck()` in file `Bridge.java`. Some part of the code history is missing because subjects were not informed beforehand to turn off the default recording limitation of Eclipse, but we think the available slices of code history are rich enough for this case study and permit a peek into other barriers to programming with concurrency.

In this implementation of the single-lane bridge problem using the monitor pattern with Java threads and concurrency, the major files of interest under analysis are: `Bridge.java`, `RedCar.java` and `BlueCar.java`. The first file implements the monitor class `Bridge` and the other two files implement two thread classes, `RedCar` and `BlueCar`. These files are provided to students with a class declaration, necessary interfaces inherited and declaration of methods that are called in other helping codes, but no more details. Other helping codes are provided to subjects and do not require modifications if subjects correctly follow the implementation of a monitor pattern. A detailed specification is provided to subjects on the expected behavior and output rules of the final system, as seen in section 6.5. Generally, this single-lane bridge system should print out records for car arriving, car entering bridge, car exiting bridge and bridge status information. The bridge status information includes two pieces: 1) the current number of red cars and blue cars on the bridge (one must be zero for safety); 2) the number of red cars and blue cars in total that have entered (used) the bridge (the difference between these two numbers should not exceed the `waitDiff` specified as a fairness requirement). Also, the bridge status information should only be printed after a certain number (`checkFreq`) of other events are printed as specified in the document.

We analyzed a small piece of the available modification histories of two subjects in our class to illustrate the knowledge needed beyond the concurrency-related knowledge to successfully implement a concurrent system. Table 48 and Table 49 in appendix section 6.1 contain the behaviors we observed for each modification, the corresponding goals we inferred that the subject was trying to achieve with

the associated behaviors and the corresponding solid or fragile knowledge the subjects may have. Solid knowledge is shown without shading in the table. Fragile knowledge related to concurrency is lightly shaded and fragile knowledge not related to concurrency is darkly shaded. Table 48 comes from the code history of the intermediate level subject (subject I) and Table 49 comes from the code history of the novice level subject (subject N).

Through the analysis, we first notice that subject I did much less skipping around among the three files. In total, two shifts were observed. Subject I first implemented the two thread classes and then shifted to working on the monitor class. After he implemented the monitor class, he shifted back to refine the thread classes. We suspect that subject I has relatively solid strategy knowledge of the implementation of such concurrent systems with the monitor pattern: naming and calling the required monitor methods in the thread classes first and then implementing these methods in the monitor class. In contrast to subject I, subject N had a total of 22 shifts among three files. And unlike subject I, who made changes to multiple functions in one place before shifting to another place, subject N's shifts are almost all at the level of a single function. Apparently, subject N has very fragile knowledge of how to implement concurrency with Java in the monitor pattern.

When zooming out, we actually see subject N wander between two different implementation strategies. One strategy is to implement the system with a monitor class of several synchronized getter and setter methods and use different combinations of getter and setter methods together with calls to the `wait()` and `notify()/notifyAll()` functions in the thread class. The other strategy is to implement the system following the monitor pattern. However, knowledge of both of these strategies is fragile. They seem to come directly from the other sample code subject N has seen during previous programming experience (getter and setter are commonly seen in CS1 or CS2 when students are taught to use object oriented programming language and monitor patterns are pervasively seen in sample codes given in this

class) without the understanding of why a certain pattern should be used or the detailed mechanism for using it.

For subject I, one example of non-concurrency-related fragile knowledge we observed is the construction of a code structure that increases a counter variable and checks whether a preset interval has passed (required in implementation of printing out status check according to preset interval of printing out other events). Subject N is also lacking in this knowledge, but compared to subject I's five closely gathered attempts on this issue (see line 16-20 in Table 48), subject N has about four attempts scattering towards this issue (see line 17, 19, 21, 22 in Table 49). Between lines 17 and 19, subject N decides to call and probably tests the printing functions without a full implementation. This does not quite make sense since the specification clearly stated the correct number of events printed within the status check interval is under final testing. Then, between lines 19 and 21, subject N added the synchronized keyword to some methods in the Bridge class, which has nothing to do with the implementation of the `statusCheck()` method. However, subject N may hold some incorrect mental representation of the system in which the incorrect functionality of `statusCheck()` is due to a race condition. Another example of non-concurrency-related fragile knowledge we observed in both subjects was the schematic knowledge of the order of implementation. Both subjects tried to add randomization before testing that the whole system is functioning well (see line 7-9 in Table 48 and line 31, 38, 41 in Table 49)

Also, we infer that subject N may have much fragile knowledge, much of which is not related to concurrency or even to programming. As we stated in our conceptual framework of the nature of programming expertise, programming is an information-intensive activity during which the knowledge used to form a mental model that guides searching and fetching external information (data) is critical. Subject N's inferiority is not only caused by fragile knowledge of the monitor pattern (which we also observe from subject I), but is also related to much other knowledge. For example, subject N does not

quite understand 1) why and how getters and setters are used in object oriented design; 2) what is the inheritance between super class/object and sub class/object; 3) why and when to wrap functions and use the caller-callee pattern; 4) when to use global versus local variables; 5) why and how to declare and initialize class variables, etc. Subject N even has difficulties in reading, understanding and interpreting some simple requirement specifications written in natural language. For example, the `isAllExit()` method is supposed to check whether the number of cars that finally exit the bridge equals the total number of cars sent by thread generators. Therefore, according to subject N's implementation, a simple comparison of the passed-in total value and the sum of two variables he defined, `redCarOff` and `blueCarOff`, should serve the purpose. However, subject N first misused `redCarOn` and `blueCarOn` in the function. After fixing this error, subject N wrote code to return a true value, which indicates that all cars exit, when the total is found not equal to the sum, which indicates that some cars are still on bridge.

Therefore, we concluded that the current emphasis in courses that deal with concurrency on helping programmers appreciate performance gains and tackle the uncertainty and large space of possibilities in concurrent programming is definitely not the most pressing concern for novice and intermediate students. A solid understanding of the programming language they are using, the mechanisms of various constructs (mostly not concurrency related such as inheritance), the mechanisms of concurrency patterns (including control flow, data flow and relations among functions), and even the ability to read and understand natural language are far more important. Showing students sample code does build some knowledge in their minds (as we saw from subject N's wandering between using getters, setters and using the monitor pattern), but the knowledge is fragile and does not support them to make any solid achievements in real problem solving. As indicated by our conceptual framework, it is the mental representation, or part of it, or the procedure for formulating such a representation, that is internalized into knowledge for retrieval in future problem solving and seen as expertise. The processes

of assimilating data in the world through challenging practice rather than simply memorizing the data finally builds programming expertise.

We generalize the concurrency related and non-concurrency related fragile knowledge exhibited by both the intermediate and novice subjects in Table 26. Both intermediate and novice subjects demonstrated some fragile knowledge related to programming with concurrency. For the intermediate subject, most concurrency-related fragile knowledge is at the implementation level regarding the detailed control flow and data flow mechanisms of the monitor pattern. The novice subject also showed fragile knowledge at the implementation level but this was not apparent in comparison with the large amount of fragile knowledge seen at concurrency level regarding basic organization of threads and shared objects in a shared memory system. This again provides evidence of our proposed hierarchical structure of concurrency related misconceptions. However, compared to the intermediate subject, we think the major barrier for the novice subject to implement the concurrent program was his pervasive non-concurrency-related fragile knowledge. Through observation of the novice subject's behaviors and many goals he struggled to achieve during the procedure, which appeared much easier and more manageable for the intermediate subject, we suggest that the lack of prior programming knowledge (general knowledge, procedural knowledge, object oriented knowledge, etc.) greatly affects a novice subject's ability to learn and appreciate programming with concurrency.

TABLE 26 FRAGILE KNOWLEDGE EXHIBITED BY INTERMEDIATE AND NOVICE SUBJECTS

	Intermediate	Novice	
Concurrency related			
concurrency level	with shared memory, threads distinguish themselves by both calling different methods and passing different arguments to methods that modify shared object's state	with shared memory, synchronization achieves through modification and control of shared object's state instead of direct coordination of threads' behaviors	
		with shared memory, shared object's class methods changes its states	
		threads may be defined to iteratively execute a set of actions	
		initialization of thread object	
implementation level	detailed control flow of monitor pattern (how do threads call monitor methods to achieve actions)	detailed mechanism of race condition and synchronized keyword	
	detailed data flow of monitor pattern (how are values passed into and returned from monitor methods to interact monitor and thread objects)	detailed mechanism of notify() and notifyAll() functions	
	detailed monitor pattern of "conditional check, execution and notify"		
Non-concurrency related			
general abilities		reading and understanding of specification written in natural language	
general procedures	complex functionalities should be implemented after basic functionalities are guaranteed	complex functionalities should be implemented after basic functionalities are guaranteed	
general programming constructs		declaration of function signatures	
		knowledge of when to use caller-callee relations and purpose of organizing local functions into call hierarchy	
		translation of natural language conditions and corresponding returns to condition branches in high level programming language	
		declaration and initialization of variables	
object oriented constructs		purpose and usage of setters and getters methods (to hide implementation details with usage of private variables)	
		the inheritance relationship among super class and sub class	
code patterns	code pattern of continuously incrementing a counter to control the occurrence of some function based on a preset interval	code pattern of continuously incrementing a counter to control the occurrence of some function based on a preset interval	
		code pattern of a function that checks value of some variables	
		code pattern of using existed class variable to return some other value	

3.4 SUMMARY AND FUTURE WORK

In this section, we focus on the discussion of barriers to learning about programming with concurrency. We first review the empirical studies that provide discussion and insights into the nature of expertise, synthesize the available results, claims and theories, and unify and validate them with a proposed conceptual framework of programming expertise. Our framework reconciles the findings in psychological studies of programmers and theories grounded in empirical studies of both production and comprehension programming activities, which covers the whole span of the software development cycle. We believe that the research conclusions we surveyed align well with our newly proposed conceptual framework for the nature of programming expertise, yet with the ever-evolving software development ecosystem, more empirical evidence is definitely needed to support, challenge and refine our proposition.

Our conceptual framework of programming expertise targets individual expertise. However, much research has been done to study software development teams and organizations. With the growth of system scale, many real world development tasks require the cooperation of multiple individuals and even teams. Levesque et al. (Levesque, et al., 2001) studied shared mental representations of team members on their perception of their own team and project progress and discovered that team member's mental representations about the group's work and each other's expertise surprisingly did not become more similar over time. Crowston and Kammerer (Crowston & Kammerer, 1998) studied two software requirements teams on their development of requirements for large and complex real-time systems and pointed out that the construction of the team's collective minds on knowledge of itself and its project was critical. This is also observed in (Herbsleb & Grinter, 1999), who proposed that distance team work was unlikely to be efficient. Therefore, we think it will be interesting to ask, in team-work circumstances, does our concept of individual expertise apply to a group of people? If not, what is missing? And if so, what is the relation between group and individual expertise? The answers to these

questions can suggest strategies for developing organizational structures, external information structures for team members, and the training of individuals in a team.

Besides, with the development of various programming languages and development frameworks in recent decades, learning to program is no longer the same task as in the 1970-1980s, although the underlying core framework of expertise is quite similar. For example, we notice that (Fleming, et al., 2008), (Fleming, et al., 2008) and (Matthijssen, et al., 2010) all found empirical evidence that programmers found it difficult to follow simple but branching paths to comprehend programs. With the pervasive use of parallel and distributed systems, it will be interesting to explore what kinds of internal and external information are helpful for program comprehension and production tasks involved in these types of systems. Also, we discern the following questions to be interesting to answer: 1) What obstacles do students encounter in an environment characterized by a fast learning pace due to rapid change in languages and development frameworks? 2) Is there any influence of the choice of first programming language and programming paradigm, since as indicated by our framework, the initial internal knowledge is critical to subsequent problem solving processes?

Furthermore, in reviewing the literature, we notice that some software engineering activities are under-explored. One such activity is code inspection and review. The authors of (Letovsky, et al., 1987) studied an inspection session and stated that information about design decisions and design details were lost through time and needed to be reconstructed during a code inspection session. This statement interprets our framework from another perspective: that the internalized information needs to be externalized for others with less expertise to fulfill a programming task and for selves to permit reconstruction of mental representations. (Parnin & Rugaber, 2011) and (Parnin & Rugaber, 2012) further discuss what information is easily lost over time and is necessary for reconstruction of mental representations by discussing programmer's resumption strategies. Rigby et al. (Rigby, et al., 2008), examined peer review practices with archival records of emails and version control repositories, but

from a software engineering perspective that provided few insights into programming expertise. Porter et al. (Porter, et al., 1997), studied code review procedures of traditional software (in contrast to open source software), but largely focused on productivity and effectiveness implications with different techniques and organizations of code review. We suspect that code review is a procedure for communicating and exchanging mental representations between two programmers (code author and reviewer) who have different repositories of internal information (knowledge). The reviewer has more knowledge regarding the “big picture” while the author knows more detail about the portion under review. We suspect that studying this procedure could provide valuable empirical evidence on the acquisition of programming expertise from the perspectives of both knowledge internalization and mental representation evolution.

With the interpretation of the nature of expertise under our proposed conceptual framework, we posit that knowledge is an important factor in problem solving and that it is difficult to accumulate. Some work, such as the curriculum guides we discuss in section 4.3 of the next chapter, list required knowledge and concepts for programming with concurrency, but our misconception hierarchy identifies the difficulties and procedures necessary for acquiring the concurrency related knowledge. We categorize five types of misconceptions as in description, terminology, concurrency, implementation and uncertainty levels and further imply that these five types of misconceptions are organized in a hierarchical structure so that misconceptions at a lower level prevent one to acquire knowledge and concepts at a higher-level. Thus, concepts at one level may only be taught and learned after concepts at lower levels have been assimilated. The results of our two empirical studies serve both as data sources for our grounded theory of hierarchical knowledge structure for programming with concurrency (the reasoning protocols) but also serve as a source of validation of the hierarchical structure (the misconception count and performance metrics). Our misconception hierarchy covers both the shared memory and message passing models of concurrency (which is comprehensive according to

contemporary parallel and distributed software architectures). By shedding light on the hierarchical structure of misconceptions related to concurrency concepts, we suggest that future work may take the procedure and structure of knowledge acquisition into account in guiding pedagogical designs rather than purely focus on the content of required knowledge. Certainly, we recognize that our misconception hierarchy is still in a very primitive state, with limited misconceptions enumerated for each level and further work on providing empirical evidence to refine and enhance the hierarchy is needed.

For a better and more comprehensive understanding of student's barriers to learning programming with concurrency, we conducted a case study to explore student's fragile knowledge displayed during an actual production activity of implementing a concurrent system. We notice that, in addition to the misconceptions captured by our proposed hierarchy, other non-concurrency and even non-programming related knowledge is important for the development of programming expertise and success at programming tasks. With the interpretation of our conceptual framework, we suspect that since programming requires dealing with large volume and rapidly changing information, reading and understanding abilities that seem unrelated to concurrency or even to programming are important to the development of student's programming expertise. Certainly, our case study is just an initial step in the study of knowledge related to expertise in information-intensive activities. More empirical evidence is needed to further clear up the picture of the composition and development of such knowledge.

As indicated by the title of (Curtis, et al., 1986), "Software Psychology: The Need for an Interdisciplinary Program", we believe the study of programming expertise and human factors in software engineering should be carried out with input from many fields and perspectives and we hope our work provides a solid stepping stone for future research work in this and related fields.

CHAPTER 4.

EXPLORATIONS IN TEACHING

In this chapter, we present the work we carried out to explore better and more effective pedagogical approaches, materials and class designs for teaching programming with concurrency. We first review the literature studying pair programming as a pedagogical technique in section 4.1. In section 4.2, we discuss the pedagogical, engineering and cognitive impacts of pair programming observed within our own studies. These findings complement the previous work on the pedagogical impact of pair programming and illustrate the cognitive benefits of pair programming in complex problem solving procedures such as implementing a concurrent program. Then we present a survey of curriculum guides and essential elements in teaching programming with concurrency in section 4.3. We identify two concurrency models, three implementation approaches and several classic scenarios to cover in our teaching of an upper-level computer science course and illustrate how related curriculum items are covered by teaching these elements. In section 4.4, we present feedback on teaching programming with concurrency and discuss the benefits and drawbacks of various pedagogical innovations.

TABLE 27 OVERVIEW OF WORK CONTRIBUTING TO CONDUCT EXPLORATIONS IN TEACHING

Work	Data	Results	Section
2012 survey on the impact of pair programming	previous empirical study on impact of pair programming as a pedagogical technique	a conclusion of previous empirical studies and impact of pair programming as a pedagogical technique	4.1
quasi-experimental study in Spring 2012 observational study of Spring 2013 final exam	performance data and code submission from Spring 2012 observation notes from Spring 2013	pedagogical, engineering and cognitive impact of pair programming	4.2
2012 survey on teaching concurrency	various resources on teaching programming with concurrency	concurrency related curriculum guides, models of concurrency, details of language constructs, and classic concurrency scenarios	4.3
case study of Spring 2013 course	performance data and course feedback from Spring 2013	benefits and drawbacks of various approaches to teaching programming with concurrency	4.4

4.1 PAIR PROGRAMMING AS A PEDAGOGICAL TECHNIQUE

Pair programming is a practice in which two programmers work collaboratively at one computer on the same design, algorithm, code, or test (Williams & Kessler, 2002). The effectiveness of pair programming as a pedagogical approach has been widely researched (Preston, 2006) and (Salleh, et al., 2011) with the following findings:

- pair programming helps improve the retention rate and course pass rates in introductory computer science (CS) courses (Braught, et al., 2011) (Carver, et al., 2007) (Hanks, et al., 2004) (McDowell, et al., 2003) (McDowell, et al., 2006) (Mendes, et al., 2005) (Mendes, et al., 2006) (Nagappan, et al., 2003) (Williams, et al., 2003) and contributes to greater persistence in CS-related majors (McDowell, et al., 2003) (Williams, et al., 2003).
- pair programming helps to improve the quality of programs, programmers' confidence in their work and programmers' enjoyment (Carver, et al., 2007) (McDowell, et al., 2003) (McDowell, et al., 2006) (Williams, et al., 2000) (Williams, et al., 2003).
- pair programming helps students with lower SAT scores to gain better individual programming abilities than similar students who do not engage in pair programming (Braught, et al., 2008) (Braught, et al., 2011)
- pair programming is particularly beneficial for women because it addresses factors that potentially limit their participation in CS (Sax, n.d.).

Pair programming is an element of eXtreme programming that has been widely recognized in industry to improve productivity and programmer satisfaction (Williams & Kessler, 2002). Recognition of the benefits has also generated great interest in applying pair programming to CS education. The advantages of adopting it as a pedagogical approach have been widely researched in several empirical studies. Five major studies carried out in recent years are identified here.

Studies at University of Utah: '99

In 1999, L. Williams, et al. (Williams, et al., 2000) carried out a study of the benefits of pair programming in an advanced undergraduate level course at the University of Utah. Students who expressed interest in pair programming worked in pairs on standard assignments and also on additional assignments to guarantee the same amount of total workload among paired and solo students. Although pair programmers were assigned more work in this study, the result showed that they were more willing to work in pairs in the future. In addition, other hypotheses regarding assignment quality, exam scores, overall pass rate and programmers' enjoyment were confirmed.

Studies at UC-Santa Cruz: '00 – '01

A study carried out by McDowell, et al. (McDowell, et al., 2006) at UCSC looked at an introductory level CS1 course offered over four sections during the 2000-01 academic year. In three of the four sections, students were required to complete all assignments using pair programming techniques. Students were paired according to their preferences and worked with the same partner throughout the course. Another study (McDowell, et al., 2003) was conducted over three advanced CS courses. Students in those courses had the option of working in pairs. Pairs were assigned and changed twice in the first two courses. In the third course, students had the option to work in pairs for each programming assignment and in most cases continued to work with the same partner. The major findings of these two studies showed that pair programming improved course completion rates, persistence in CS related majors, the quality of students' products and students' programming abilities. This trend was particularly apparent in the retention of female CS students who participated in pair programming (Werner, et al., 2004).

Studies at North Carolina State: '01 – '02

L. Williams, et al. (Williams, et al., 2003) at NCSU carried out studies on pair programming in a CS1 course over two semesters during the 2001-02 academic years. In each semester, two nearly-identical

sections of the course were offered, with the main difference being that one of the sections involved a paired closed lab session and the other section involved a solo closed lab session. The students were paired randomly and the pairs were re-assigned every two to three weeks. Students also had the option to work in pairs on other programming projects assigned outside the lab session. Although this introductory course was offered to all students across the university, data analysis was carried out over freshmen and sophomores only. In contrast to previous studies, no significant differences were found between pair and solo sections on exam scores but the benefits of pair programming on retention rates and the quality of students' products remained. Students in this study were also found to have a more positive attitude toward working in collaborative environments, which is perceived as a long-term beneficial factor for their future professional life.

Studies at University of Auckland: '04 – '05

During 2004-2005, E. Mendes, et al. (Mendes, et al., 2005) (Mendes, et al., 2006) carried out a study of the effects of pair programming in an intermediate-level course in which software development and design are taught in the context of UML and Java OO. Students worked individually on all course activities that contributed to their final score. One section of the course employed pair programming in the closed lab session while the other section did not. In the pair group, the students were randomly paired and the pair assignment changed about three times. This study showed that students in the pair group not only produced higher quality code, but also used less time and experienced more enjoyment and were more confident about their work.

Studies at Dickinson College: '05 – '07

G. Braught, et al. (Braught, et al., 2008) (Braught, et al., 2011) carried out studies on the adoption of pair programming in an introductory CS1 course during academic years 2005-07. In each year, two sections of the course were offered. One section adopted pair programming on open laboratory assignments while the other did not. All other assignments and activities were completed individually.

Students were initially randomly assigned into pairs. Later, students with similar assignment scores were paired. This re-assignment occurred approximately three times. This study evaluated pair versus solo students' acquisition of programming ability. Students with lower SAT scores were able to achieve higher lab practica scores through the pair programming experience. Also, the benefits of pair programming in improving enjoyment level, confidence level and course completion rate were reinforced.

4.2 IMPACTS OF PAIR PROGRAMMING

The wide array of findings in prior work, some of which conflict, reinforces the notion that much remains to be explored and learned about in incorporating pair programming principles in the classroom. Our efforts in studying the impact of pair programming were carried out in two empirical studies during spring 2012 and spring 2013.

In spring 2012, we sought to investigate the effects of pair programming in an under-explored course, namely an intermediate-level programming course as in (Mendes, et al., 2005). The course in which we performed our study is devoted to the acquisition of a second programming language (C++) in the context of UNIX systems programming. The course takes CS1 as a prerequisite and CS2 as a co-requisite or pre-requisite. Thus, we had the opportunity to compare the effects of pair programming on students with varying levels of prior programming experience. This is different from most previous studies that were carried out either in an introductory CS course (Braught, et al., 2008) (Braught, et al., 2011) (McDowell, et al., 2003) (McDowell, et al., 2003) (Williams, et al., 2000) in which students have no programming experience or in an advanced level CS course (McDowell, et al., 2003) (Williams, et al., 2000) in which students have substantial programming experience.

In our spring 2012 study of the impact of pair programming, the pair group and the solo group were combined into one course offering. That is, the pair programmers and the solo programmers were all in the same class for two 75-minute lectures each week and met separately only for a 50-minute lab

meeting per week. Students were assured that any difference in lab averages between the sections would be addressed with an appropriate curve at the end of the semester and were satisfied with this approach. Compared with most of the previous studies, which took place over different course offerings or during different semesters (McDowell, et al., 2006) (McDowell, et al., 2003) (Nagappan, et al., 2003) (Werner, et al., 2004) (Williams, et al., 2003), we studied two groups with nearly identical experience on materials, lectures, and activities other than pair versus solo programming experience in the lab, thus eliminating many potential confounding factors.

We looked at a different type of lab experience in our spring 2012 study; while (Mendes, et al., 2006) (Mendes, et al., 2005) (Williams, et al., 2003) studied closed-lab experiences and (Braught, et al., 2008) (Braught, et al., 2011) (McDowell, et al., 2003) (McDowell, et al., 2006) (Williams, et al., 2000) studied open-assignment experience, our study involved programming projects that started during an initial 50-minute class period and were then open for completion after class. In addition to project completion times and subjective satisfaction ratings, we also collected self-reported usage of TA office hours. This data helps to characterize the benefits of pair programming in a resource-limited department. We collected students' preferences on course organization (lecture to lab ratio) and their subjective feedback on the pair programming experience. This feedback reveals the logistical problems encountered by pair programmers and suggests that a "flipped" classroom experience (Bergmann & Sams, 2012), through which students learn necessary knowledge from reading-at-home and practice in class, could be more helpful. We also analyze the code health level of student's lab submissions between pair programmers and solo programming, which reveals engineering impacts of pair programming with student programmers.

In spring 2013, we observed and compared the problem solving process of pair and solo programmers during a final exam and report a case study of the cognitive impacts of pair programming on problem solving in general.

4.2.1 PEDAGOGICAL IMPACT

Although pair programmers withdrew from our intermediate CS course half as often as solo programmers (13% vs. 27%), this trend was not significant. However, we found that pair programming served as a protective factor for retention of female students (14% vs. 67%) and of students concurrently enrolled in CS2 (20% vs. 57%). Using the NASA TLX subjective workload survey (Hart & Staveland, 1988), we also found that pair programmers reported less mental demand, temporal pressure, total effort and frustration than did solo programmers. Lower frequencies of withdrawal in the pair programming group also suggest that pair programmers were more confident in their ability to successfully complete the course.

After adopting pair programming into the course, we note several important impacts it has, especially towards both female students and less-experienced students in that they are less likely to withdraw from the course if they were in the pair programming group. The following discussion addresses the details of this and some other findings.

Both the pair and solo groups of CSCI 1730 consisted largely of students seeking a B.S. in CS as a sole major, with several students in each section opting to complete a double major between CS and another major. The solo group had fewer non-CS majors (5) than the pair group (9). The students were not told of any differences in lab sections before the start of class.

Withdrawal Rates

CSCI 1730's historical withdrawal rate is 21.1%. Across both conditions in our study, the combined withdrawal rate for spring 2012 was 20.0%, generally consistent with previous semesters. In the following discussion, we consider the impact of pair programming versus solo programming not only on the general population (all students in the two sections under study), but also on male students versus female students, and on less-experienced versus more experienced students.

We explored various types of statistical methods to analyze the different aspects of our observed data and to present the results in a comprehensive manner. The statistical methods we considered fall into two categories, null hypothesis significance testing (NHST) methods and effect size methods so that in our analysis the descriptive statistic measurements complement the inferential statistics. The sample size of our groups was relatively small and the withdrawal variable is dichotomous (students either withdrew or they did not). These characteristics may violate the underlying assumptions of NHST methods such as the t-test or ANOVA. Thus, we employed Fisher's Exact Test (Fisher, 1922) and Barnard's Exact Test (Barnard, 1945) (with MATLAB code (Trujillo-Ortiz, et al., 2004)) on 2x2 contingency tables to evaluate the significance of dependence on row (Pair Programming, PP vs. Solo Programming, SP) and column (withdrew vs. retained) variables. We present results from Barnard's Exact Test because it is more accurate than Fisher's on a 2x2 contingency table (Barnard, 1947). When considering using effect size methods to present a meaningful complement to the results of significance tests, we considered risk estimates, which compare the relative risk for a particular outcome between two or more groups. In our study, the relative "risk" is student withdrawal from the course and the two groups are PP (pair programming) and SP (solo programming). We chose these risk estimate methods because they are likely better estimates of effects for binomial data than methods of examining association among variables through calculating Pearson's r (Ferguson, 2009). We also present the strength of association of a 2x2 table with the Yule's Q measure (Edwards, 1963) to illustrate the strength of association in the range of $(-1, 1)$.

As shown in Table 28, the withdrawal rate for students in the pair group was 4 of 30 (13.3%) in contrast to 8 of 30 (26.7%) in the solo group. Fisher's Exact Test yields a p value of 0.3334, which is not significant. The p value of the more exact Barnard's Exact Test is 0.1162 and also not significant. The relative risk of the table is 2, which indicates that those in the SP group withdrew twice as often as those in the PP group. By considering the actual base rate of around 20% (withdrawal rate of general

population and historical withdrawal rate), we consider this relative risk to be a practically significant effect.

As seen in Table 29, we found that only 1 of 7 females (14.3%) withdrew from the pair group but that 2 of 3 (66.7%) of females withdrew from the solo group. In contrast, as seen in Table 3, we found that while a similar ratio of males (3 of 23, 13.04%) withdrew from the pair group, a much smaller ratio of males (6 of 27, 22.22%) withdrew from the solo group than did females. Although we do not see any statistical significance from the Exact Tests of Table 29, both the relative risk and the strength of association of Table 29 are high, especially compared to the same measurement of the general population in Table 28 and male students in Table 30. This suggests that pair programming has a much greater impact on retaining female students: female students in the solo group are more than four times more likely to withdraw from the course than those in the pair group and pair programming is 84% positively correlated with retaining female students. It is also worth noting that the average SAT Math score of female students in the pair group (588, $\sigma = 85.2$) was lower than that of female students in the solo group (650, $\sigma = 14.1$). However, more female students were retained in the pair group. This suggests that pair programming may serve as a protective factor in retaining female students and we speculate that this is a result of the collaborative rather than a competitive environment and more social aspect of what may be otherwise perceived as a non-social and isolating computing activity, as described in (Werner, et al., 2004). Our findings support the notion that pair programming may be a useful approach to attract and retain underrepresented groups, including women, in CS education.

We also studied the impact of pair programming on “less experienced” students (those concurrently enrolled in CS2, the pre- or co-requisite of this intermediate course) versus the “more experienced” students (those who completed CS2 in a prior semester). As shown in Table 31, while only 2 of the 10 (20%) of the “less experienced” students in the pair group withdrew, 8 of 14 (57.1%) of the “less experienced” students in the solo group withdrew. Moreover, the p values of Fisher’s and

Barnard's Exact Tests on Table 31 are 0.1041 and 0.0477, which indicates statistical significance of pair programming on retention of less experienced students. Although the values of relative risk and the strength of association between rows (PP vs. SP) and columns (withdrew vs. retained) of Table 31 are 2.857 and 0.684, these are not as strong as those seen for female students. Compared to the relative risk and the strength of association of "more experienced" students, which are 0.625 and -0.25 (indicates a reverse association between pair programming and retention), we conclude that pair programming is practically effective in retaining "less experienced" students in the course. This suggests that pair programming could be useful in shortening the long pre-requisite chain typically found in CS programs, reducing time to degree, and in promoting student productivity during undergraduate studies.

Since the midterm score is an important indicator for students in deciding whether to withdraw (midterm comprises 20% of final grade), we examined overall means on the midterm exam (PP: 74.7, $\sigma = 14.42$; SP: 75.7, $\sigma = 18.84$), but found no significant difference between the groups. However, the difference in withdrawal rates indicates a relative difference in the level of confidence in successfully completing the course, with members of the pair group exhibiting greater confidence.

TABLE 28 WITHDRAWAL COUNTS

	Withdrew	Retained
SP	8	22
PP	4	26
Fisher's Exact Test	two sided $p = 0.3334$	
Barnard's Exact Test	two sided $p = 0.1162$	
Relative Risk	RR = 2	
Yule's Q	Q = 0.405	

TABLE 29 WITHDRAWAL COUNTS: FEMALE

	Withdrawal	Retention
SP	2	1
PP	1	6
Fisher's Exact Test	two sided $p = 0.1833$	
Barnard's Exact Test	two sided $p = \mathbf{0.0912}$	
Relative Risk	RR = 4.667	
Yule's Q	Q = 0.846	

TABLE 30 WITHDRAWAL COUNTS: MALE

	Withdrawal	Retention
SP	6	21
PP	3	20
Fisher's Exact Test	two sided $p = 0.4790$	
Barnard's Exact Test	two sided $p = 0.2426$	
Relative Risk	RR = 1.704	
Yule's Q	Q = 0.311	

TABLE 31 WITHDRAWAL COUNTS: LESS EXPERIENCED

	Withdrawal	Retention
SP	8	6
PP	2	8
Fisher's Exact Test	two sided $p = 0.1041$	
Barnard's Exact Test	two sided $p = \mathbf{0.0477}$	
Relative Risk	RR = 2.857	
Yule's Q	Q = 0.684	

TABLE 32 WITHDRAWAL COUNTS: MORE EXPERIENCED

	Withdrawal	Retention
PP	2	18
SP	1	15
Fisher's Exact Test	two sided p = 1.0000	
Barnard's Exact Test	two sided p = 0.6258	
Relative Risk	RR = 0.625	
Yule's Q	Q = -0.250	

Lab Performance

We examined the scores of labs (15 labs in total comprise 40% of the final grade) and found that students in the pair group performed significantly better on the first five labs, carried out before the withdrawal date. We performed a one-tailed, homoscedastic t-test on lab scores of the PP and SP groups. For each of these first five labs, students in the pair group performed significantly better (all p values < 0.05) than students in the solo group. However, this effect did not persist in later labs nor did it appear in a comparison of the top 10 students in the two groups. We suspect that the disappearance of better lab performance from the pair group versus the solo group is due to relatively more of the weaker students withdrawing from the solo group, equalizing the performance of the two groups.

TABLE 33 LAB PERFORMANCES OF PAIR AND SOLO PROGRAMMERS

	Lab 2	Lab 3	Lab 4	Lab 6	Lab 7
PP	48.92	47.96	45.73	45.27	48.04
SP	41.95	40.77	36.77	36.82	39.91
p value =	0.022	0.039	0.022	0.024	0.037

Utilization of TA Office Hours

In an in-class survey, 21 members of the pair group reported a total of 31 visits to TA office hours for 975 minutes of assistance (1.47 visits per student; 46.42 minutes per student). In the same survey, 16 members of the solo group reported a total of 46 visits and 1485 minutes of assistance (2.875 visits per student; 92.8 minutes per student). This difference in utilization of TA resources likely stems from

the ability of pair programmers to talk through their difficulties and to have partners fill in the gaps in one another's knowledge. One potential impact of this phenomenon is that resource-constrained departments might better meet student needs or handle larger groups of students with existing resources through the use of pair programming.

TABLE 34 TA OFFICE HOUR USAGES BY PAIR AND SOLO PROGRAMMERS

	#students	total visits	visits/student	total minutes	min/student
PP	21	31	1.47	975	46.42
SP	16	46	2.875	1485	92.8

Course Preference

TABLE 35 COURSE ORGANIZATION PREFERENCES OF PAIR AND SOLO PROGRAMMERS

	150-min lecture + 50-min lab	200-min lecture	125-min lecture + 75-min lab
PP	4	1	20
SP	7	3	9

On the day of the final exam, students completed a survey on their preference for course organization. We provide three choices: 1) 150 minutes of lecture + 50 minutes of lab per week and continued work on the lab after class (as described in this paper); 2) 200 minutes of lecture per week and fully take-home project assignments; 3) 125 minutes of lecture + 75 minutes of lab per week and continued work on the lab after class. The ratios of students' preferences for the three different class organizations were quite different between pair and solo groups. We performed a chi square test on this preference data and found that significantly more pair programming students preferred more in-class lab time ($p < 0.001$). Similarly, in survey feedback students in the pair group expressed that lab is where they learned and actually practiced the knowledge they got from lectures. Pair programmers also reported difficulty in scheduling time to work with their partners after class, which may be at least partially responsible for pair programming students' preference for more time in lab. Moreover, this implies that a course using pair programming practices could make good use of a flipped classroom

format (i.e., provide online lecture materials for students to use at home and devote the in-class time to collaborative lab work).

Self-reported Workload Metrics

As expected, we found that the average time that pair programmers spent on the lab (412 min/person on labs 1-5) was less than that of solo programmers (631 min). However, the total student-hours devoted by pair programmers were greater. Solo programmers also felt more temporal pressure in most of the lab, though they were able to do their work at any time while the pair programmers had to coordinate with their partners. On the other hand, the pair programmers did not require as much actual time per person on the labs. We also found differences in perceived mental demand, effort to complete a lab and frustration levels. In the first five labs, students from the pair group consistently reported lower perceived mental demand, total effort and frustration while completing the labs. These metrics are graphically illustrated in Figure 12 to Figure 17

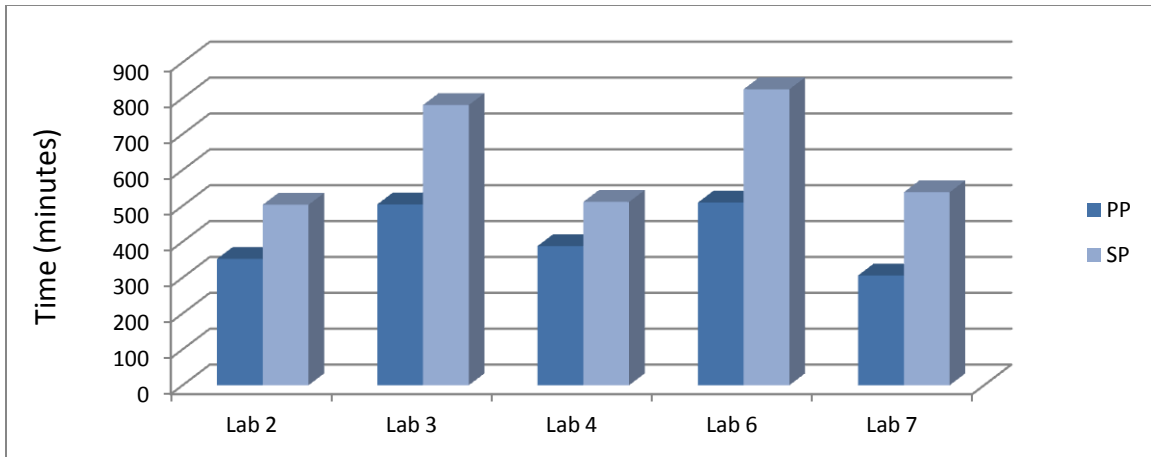


FIGURE 12 PAIR PROGRAMMER SPEND LESS TIME ON COMPLETING LABS

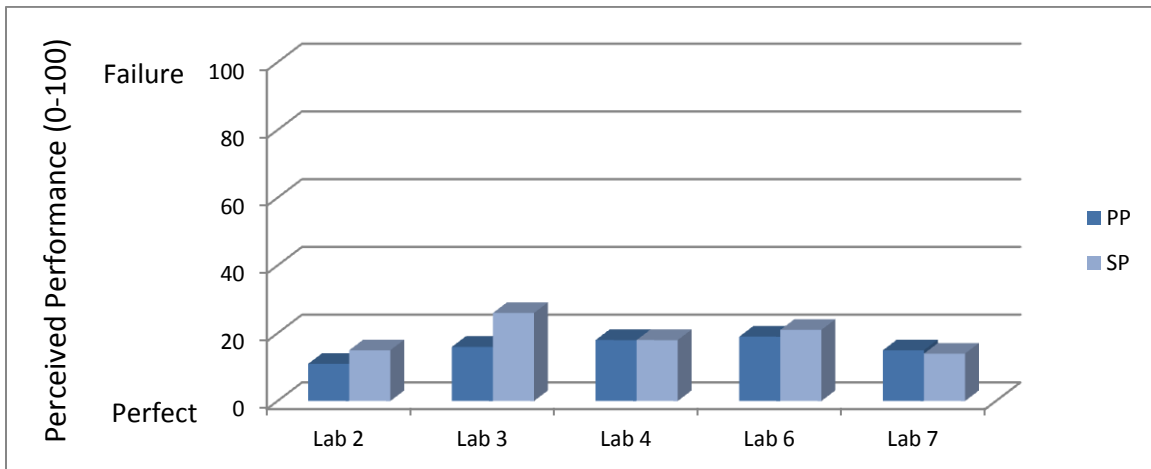


FIGURE 13 PAIR PROGRAMMERS ARE GENERALLY MORE CONFIDENT IN THEIR PERFORMANCE

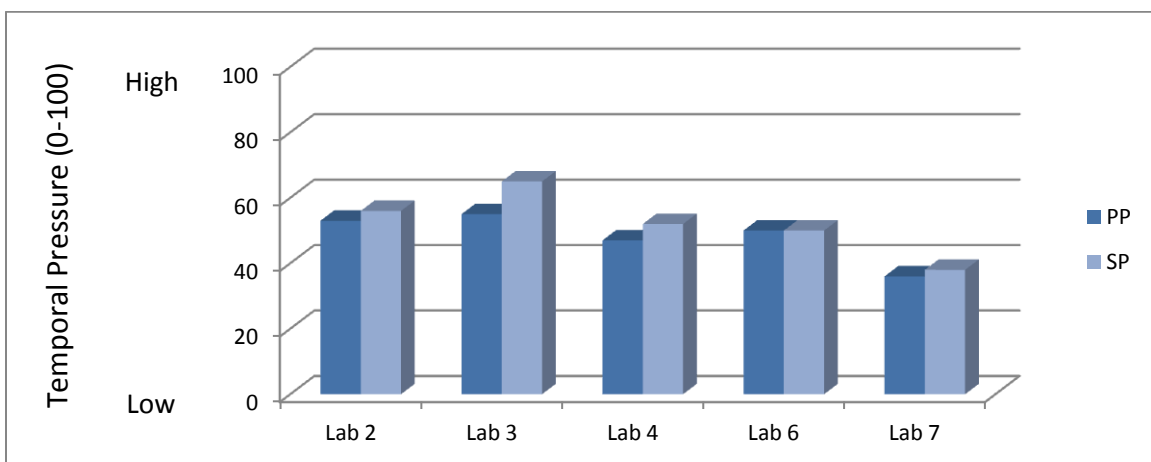


FIGURE 14 PAIR PROGRAMMERS FEEL LESS TEMPORAL PRESSURE FOR COMPLETION OF LAB

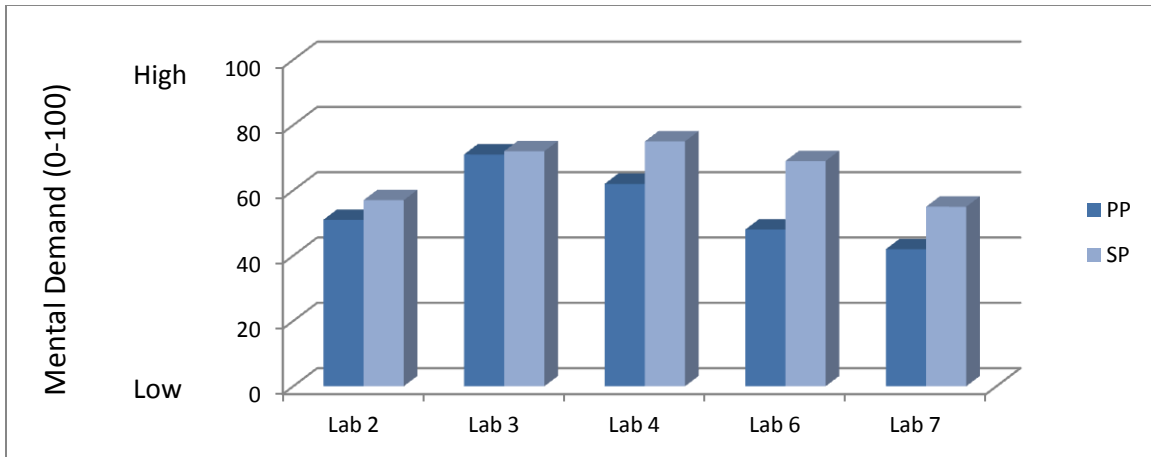


FIGURE 15 PAIR PROGRAMMERS FEEL LESS MENTAL DEMAND THAN SOLO PROGRAMMERS

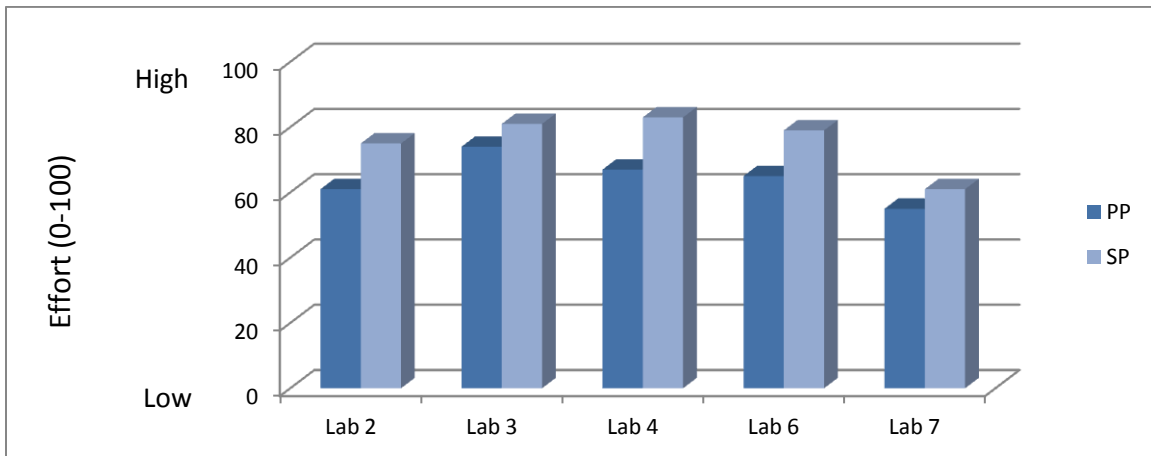


FIGURE 16 PAIR PROGRAMMERS COMPLETE LABS WITH LESS EFFORT THAN SOLO PROGRAMMERS

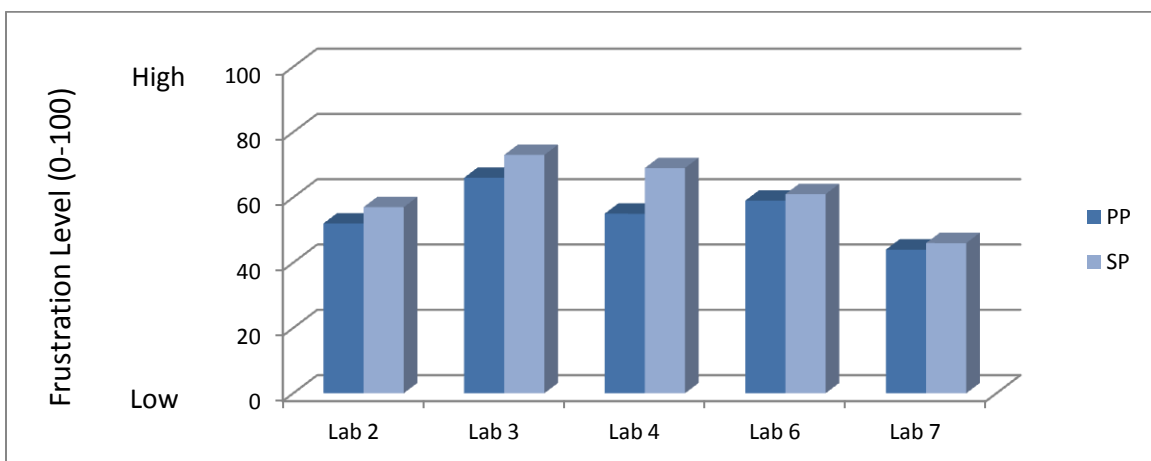


FIGURE 17 PAIR PROGRAMMER EXPERIENCE LESS FRUSTRATIONS THAN SOLO PROGRAMMERS

4.2.2 ENGINEERING IMPACT

Good coding style helps with future comprehension and modification of code. Students are required to practice writing clear and readable codes. In this study, we also instructed students to follow the Google C++ style guide (Weinberger, et al., 2010) while implementing their projects. However, the code style of a project was not assessed in our programs' grading rubrics. Therefore, we are interested to know how much students followed the given coding style guide and especially how differently do pair programmers and solo programmers follow the coding style guide.

We used Google's C++ lint checker to examine all student's submissions. This lint program reads through a code and outputs all possible style violations. The output of each violation from this checker program is illustrated in Table 36. According to the inline comments of the original Google lint checker, the filename is the name of the file containing the error. The "linenum" is the number of the line containing the error. The category is a string description of a category of the style bug. Five first level categories are: 1) build, 2) legal, 3) readability, 4) runtime and 5) whitespace. Each first level category has one or more detailed sub categories of errors as illustrated in Table 37. The confidence level is a confidence score produced by this checker regarding the error. It scales from 1 to 5 with 5 indicating the greatest confidence that an error has occurred. And finally the message records the details of the error, the codes involved and the suggestions. The two examples in Table 36 illustrate two style violations. The first one occurs in file "pair_sub/id/last1_last2_lab04_Thu_Feb_16_14_48_04_2012/pointers1.cpp" at line 0, indicating a level 5 confidence error of missing legal statement. The second one occurs at line 3 of the same file indicating a level 5 confidence error of missing whitespace before a curly brace.

TABLE 36 RAW LINT LOG FORMAT

Format	filename, linenum, category, confidence, message
Example1	pair_sub/id/last1_last2_lab04_Thu_Feb_16_14_48_04_2012/pointers1.cpp 0 legal/copyright 5 No copyright message found. You should have a line: "Copyright [year] <Copyright Owner>"
Example2	pair_sub/id/last1_last2_lab04_Thu_Feb_16_14_48_04_2012/pointers1.cpp 3 whitespace/braces 5 Missing space before {

TABLE 37 CATEGORIES OF LINT STYLE ERRORS

Category	Sub Category
build	class, deprecated, endif_comment, explicit_make_pair, forward_decl, header_guard, include, include_alpha, include_order, include_what_you_use, namespaces, printf_format, storage_class
legal	copyright
readability	alt_tokens, braces, casting, check, constructors, fn_size, function, multiline_comment, multiline_string, namespace, nonlint, streams, todo, utf8
runtime	arrays, casting, explicit, int, init, invalid_increment, member_string_references, memset, operator, printf, printf_format, references, rtti, sizeof, string, threadsafe_fun
whitespace	blank_line, braces, comma, comments, empty_loop_body, end_of_line, ending_newline, forcolon, indent, labels, line_length, newline, operators, parens, semicolon, tab, todo

TABLE 38 PARSED LINT ERROR DATA

Format	lab	confidence	category	line number	message	filename
Sample Rows	4	4	whitespace/operators	7	Missing spaces around =	pair_sub/id/last1_last2_lab04_Thu_Feb_16_14_48_04_2012/pointers5.cpp
	4	1	whitespace/tab	8	Tab found; better to use spaces	pair_sub/id/last1_last2_lab04_Thu_Feb_16_14_48_04_2012/pointers5.cpp

TABLE 39 NUMBER OF ERRORS PER FILE

All labs, Confidence level >= 3												
	build				readability			runtime			whitespace	
solo	502				217			40			836	
pair	391				130			32			598	
total	893				347			72			1434	
Concurrency-related labs, confidence >= 3												
	build				readability			runtime			whitespace	
solo	258				86			27			144	
pair	141				54			12			141	
total	399				140			39			285	
All categories, confidence >= 3, p= 0.009												
	lab2	lab3	lab4	lab6	lab7	lab8	lab9	lab10	lab11	lab13	lab14	lab15
solo	108	125	45	102	71	131	42	200	96	54	62	123
pair	52	97	45	66	36	75	97	114	60	51	54	48
total	160	222	90	168	107	206	139	314	156	105	116	171

The raw lint outputs across all student submissions, both from pair programmers and solo programmers were stored in two separate log files. Then we wrote a Python parser to parse these raw log records into two data files. The data files are organized according to labs, confidence of error, category of error, line number, error message and filename in which it is found, as illustrated in Table 38.

We wrote and used a Python query script to perform stats. Since each pair group is only required to make one submission as the result of their work but each solo programmer is required to make a submission, more solo submissions are found. Therefore, to simply compare the occurrence of style errors across the groups is not legitimate. Our query script, instead, counts and calculates the number of error occurrences per file (in which the errors are found) with a restriction of input parameters on confidence levels, specific labs as well as particular types of error.

Some interesting results are shown in Table 39. As seen in the table, across all labs, submissions of pair programmers consistently have fewer errors per file as compared to the submissions of solo programmers for each category of errors. This effect also occurs in the implementation of concurrency related labs (lab 13, 14 and 15). Across all categories of errors, submissions of pair programmers always have fewer errors per file as compared to the submissions of solo programmers for each lab ($p=0.009$), except lab 9. Lab 9 is an extension of lab 8 during which students are required to implement the C++ feature of operator overloading to overload the “+=” operator as a member function and the “+” operator as a friend non-member function that adds a shape to a complex shape.

We also observe that the most common style errors are in the whitespace category for all labs. The most common style error for concurrency related labs are build errors. Among all the labs, lab 10, in which students were required to practice C file and I/O interfaces had the most style errors. This is partially because the style checker is designed for C++ instead of C but is probably also because C is relatively unfamiliar to students (this is the only C graded lab). Lab 8 has the second greatest number of

style errors. This lab required students to work with C++ FLTK (fast and light toolkit) interfaces, which are relatively unfamiliar to students.

4.2.3 COGNITIVE IMPACT

To further explore the impact of pair programming on students' problem solving activity, we conducted a field study during the spring 2013 study. During the final exam of the spring 2013 study, students were required to implement a concurrent system, the single-lane bridge, with their preferred programming language. Students also had the choice to either work in pairs or individually. Seven enrolled and one auditing student voluntarily formed four pairs to work on the final. Other four students chose to work independently. Experimenter's notes were taken based on the observation of different activities students were engaged in at different times during the 3-hour test period. Details of the observation notes are in Table 40. For the convenience of discussion, we identify pair groups as P1 – P4 and solos as S1 to S4. The exam ran from 12:00 pm to 3:00 pm.

It is obvious to see that pair programmers start coding much later than solo programmers, instead, they invest a majority of their time understanding the specification and planning the overall structure of their system. In contrast, solo programmers almost rushed to coding activities and delayed the comprehension of the specification and development of a structural plan until later, together with the testing and debugging activities. This claim is supported by looking at the time point when questions were raised by pairs and solos as listed in Table 41. It is clear to see that all three questions that are critical to the design and implementation of the system were asked much earlier by pair programmers than solo programmers (12:35 vs. 2:20, 12:55 vs. 2:45 and 1:25 vs. 2:30). Although solo programmers asked more questions than pair programmers, 2 out of the 6 questions they asked, which were not raised by any pair, were clearly stated in specification.

To summarize, we think pair programming promotes pair partners to spend more time considering more on requirements rather than rushing to implementation in production activities. This is probably

because pair partners must convince each other of any specific detail as well as the general design. Therefore, fewer subjective assumptions can be made and more deliberation will be carried out. This is definitely helpful for problem solving under circumstances of intensive information searching and processing.

TABLE 40 OBSERVATION DETAILS

Time	Pair's Activity	Solo's Activity
12:25	P2-P4 discuss specification P1 have 1 partner start coding and the other skim through specification	4 solos all start coding
12:35	P1, P3, P4 elicit specification P2 ask for clarification of checkFreq and waitDiff ¹	4 solos work back and force between codes and specifications
12:45	P1, P2, P4 start coding P3 discuss program architecture (this pair involves the novice we described in section of Other Barriers)	S2 and S3 browse sample codes (available on course web page) for other concurrent system
12:55	P1, P2, P4 keep coding P3 start writing code on paper P2 found a typo ³ on specification one partner of P4 check online API document	S2, S3 ask whether a random travel time on bridge is expected ² S4 ask for clarification of the method finish() in main (S4 work with Scala, a different set of code than what we discussed in section of Other Barriers)
1:15	P1-P4 all coding drivers perform as simply coder navigators perform as information provider (API, sample code), document finder, code checker (for typo), teddy bear, activity logger and reminder as well as designer	solos work among requirements, codes, outputs and online resources (API and sample codes)
1:25	P1 start testing and code modification P2 ask for clarification of "using bridge" ⁴ P2 - P4 still code	S2-S4 are in testing and code modification S1 still code
1:30	P1-P2 are in testing and code modification P3-P4 still cod	4 solos are in testing and code modification
1:35	P1-P2 are in testing and code modification P2 ask tailing issue of status check ⁵ (whether status check should always appear at the very end regardless of previous number of events) P4 ask for clarification of waitDiff ¹ P3 start coding but at the same time discussing other possible system organization/pattern	
1:45	P4 do simulated execution for debugging P2 check API calls on peak, push and pull P1 discuss and analyze bug reason	S2 stuck on a runtime issue S3 ask clarification of information printed in status check ⁶ (which is clearly stated in specification) S1 searching previous code files (students are allowed to use any resource except phone and txt)
1:55	P2 reach a consistent version and check boundary cases	
2:00	P2 test on different inputs and final check specification P4 test on different inputs and encounter a bug P3 still coding	S1 finish and leave
2:15	P2 ask whether printed car id should be in order of the sequence of arriving bridge P1, P4 debug and change code P3 is still coding	S2 ask whether waiting is counted as an event ⁷ (which is clearly stated yes in specification) S3, S4 is still debugging
2:20	P4 check fairness policy P1's navigator explain "checkFreq" to driver P3 is still coding	S3 final check specification, ask for clarification of checkFreq ¹
2:30	P3 is reasoning the function and use of checkFreq P4 encounter stackOverflow exception and search to change maxPoolSize for Scala	S3 ask for clarification of "using a bridge"
2:45	P4 ask whether a random travel time on bridge is expected ²	S3 final check specification, found the typo ³ , ask about tailing issue of status check ⁵

TABLE 41 QUESTIONS ASKED BY PAIRS AND SOLOS

No.	Question	Comment	Pair	Solo
1	clarification of checkFreq and waitDiff	critical	12:35	2:20
2	random travel time	mild	2:45	12:55
3	typo in specification	critical	12:55	2:45
4	clarification of “using bridge”	critical	1:25	2:30
5	tailing issue of status check	fair	1:35	2:45
6	information in status check	stated in specification		1:45
7	whether waiting is counted as an event	stated in specification		2:15

4.3 CURRICULUM GUIDES AND ELEMENTS OF TEACHING CONCURRENCY

ACM and the IEEE Computer Society have sought to provide curriculum guidance on computing at approximately ten-year intervals. Thus 1968, 1991, and 2001 were the dates of publication of previous guidance on Computer Science. Around the time of the publication of curriculum guidance on computing in December 2001, a commitment was formed by the ACM and the IEEE Computer Society to provide curriculum guidance on a more regular basis in recognition of the rapid rate of change in the discipline and the consequent need for guidance to the community. Thus, the publishing interval was shortened to around 5 years. The two most recent publications are a revision in 2008 (Cassel, et al., 2008) and a draft in 2013 (Sahami, et al., 2013).

According to these two versions of the curriculum guide, we extract in Table 42 the elements that are related to concurrent computing and programs. These elements are mostly scattered under different categories in the version of 2008 but are gathered together in the Parallel and Distributed Computing section in the version of 2013. The topics and learning objectives are mainly from curriculum version 2013, except marked.

These concurrency elements are further explored and detailed in the *NSF/TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates* (Prasad, et al., 2012). Both the ACM/IEEE curriculum guidance and the NSF/TCPP curriculum propose a span of different concurrency elements in teaching a broad range of computer science course. We will address this later in our research work.

According to the more detailed guidance provided by *NSF/TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates*, we discern the topics in Table 43 to be covered in our research on pedagogical design.

TABLE 42 CONCURRENCY ELEMENTS IN ACM/IEEE CURRICULUM GUIDANCE

Topic	Learning Objectives
<p><from version 2008></p> <p>Distributed Algorithms [core]</p> <ul style="list-style-type: none"> Consensus and election Termination detection Fault tolerance Stabilization <p>AL/DistributedAlgorithm in version 2008</p> <p>PD/Distributed Systems in version 2013</p>	<p><from version 2008></p> <ul style="list-style-type: none"> Explain the distributed paradigm. Explain one simple distributed algorithm. Determine when to use consensus or election algorithms. Distinguish between logical and physical clocks. Describe the relative ordering of events in a distributed algorithm.
<p>Multiprocessing [core]</p> <ul style="list-style-type: none"> Power Law Example SIMD and MIMD instruction sets and architectures Interconnection networks (hypercube, shuffle-exchange, mesh, crossbar) Shared multiprocessor memory systems and memory consistency Multiprocessor cache coherence <p>AR/Multiprocessing in version 2008</p> <p>AR/Multiprocessing and alternative architecture in version 2013</p>	<ul style="list-style-type: none"> Discuss the concept of parallel processing beyond the classical von Neumann model [Familiarity] Describe alternative architectures such as SIMD and MIMD [Familiarity] Explain the concept of interconnection networks and characterize different approaches [Familiarity] Discuss the special concerns that multiprocessing systems present with respect to memory management and describe how these are addressed [Familiarity] Describe the differences between memory backplane, processor memory interconnect, and remote memory via networks [Familiarity]
<p>Concurrency [core]</p> <ul style="list-style-type: none"> States and state diagrams (cross reference SF/State-State Transition-State Machines) Structures (ready list, process control blocks, and so forth) Dispatching and context switching The role of interrupts Managing atomic access to OS objects Implementing synchronization primitives Multiprocessor issues (spin-locks, reentrancy) (cross reference SF/Parallelism) <p>OS/Concurrency in version 2008</p> <p>OS/Concurrency in version 2013</p>	<ul style="list-style-type: none"> Describe the need for concurrency within the framework of an operating system [Familiarity] Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks [Usage] Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each [Familiarity] Explain the different states that a task may pass through and the data structures needed to support the management of many tasks [Familiarity] Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives) [Familiarity] Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system [Familiarity] Create state and transition diagrams for simple problem domains [Usage]
<p>Parallel Algorithms [core]</p> <ul style="list-style-type: none"> Critical paths, work and span, and the relation to Amdahl's law (cross-reference SF/Performance) Speed-up and scalability Naturally (embarrassingly) parallel algorithms Parallel algorithmic patterns (divide-and-conquer, map and reduce, master-workers, others) 	<p>[core]</p> <ul style="list-style-type: none"> Define "critical path", "work", and "span" [Familiarity] Compute the work and span, and determine the critical path with respect to a parallel execution diagram [Usage] Define "speed-up" and explain the notion of an algorithm's scalability in this regard [Familiarity] Identify independent tasks in a program that may be parallelized [Usage] Characterize features of a workload that allow or

<p>Specific algorithms (e.g., parallel MergeSort)</p> <p>[elective]</p> <p>Parallel graph algorithms (e.g., parallel shortest path, parallel spanning tree) (cross-reference AL/Algorithmic Strategies/Divide-and-conquer)</p> <p>Parallel matrix computations</p> <p>Producer-consumer and pipelined algorithms</p> <p>AL/ParallelAlgorithms in version 2008</p> <p>PD/Parallel Algorithms, Analysis, and Programming in version 2013</p>	<p>prevent it from being naturally parallelized [Familiarity]</p> <ul style="list-style-type: none"> • Implement a parallel divide-and-conquer (and/or graph algorithm) and empirically measure its performance relative to its sequential analog [Usage] • Decompose a problem (e.g., counting the number of occurrences of some word in a document) via map and reduce operations [Usage] <p>[elective]</p> <ul style="list-style-type: none"> • Provide an example of a problem that fits the producer-consumer paradigm [Familiarity] • Give examples of problems where pipelining would be an effective means of parallelization [Familiarity] • Identify issues that arise in producer-consumer algorithms and mechanisms that may be used for addressing them [Familiarity]
<p>Performance Enhancements [elective]</p> <p>Superscalar architecture</p> <p>Branch prediction, Speculative execution, Out-of-order execution</p> <p>Prefetching</p> <p>Vector processors and GPUs</p> <p>Hardware support for Multithreading</p> <p>Scalability</p> <p>Alternative architectures, such as VLIW/EPIC, and Accelerators and other kinds of Special-Purpose Processors</p> <p>AR/PerformanceEnhancements in version 2008</p> <p>AR/Performance enhancements in version 2013</p>	<ul style="list-style-type: none"> • Describe superscalar architectures and their advantages [Familiarity] • Explain the concept of branch prediction and its utility [Familiarity] • Characterize the costs and benefits of prefetching [Familiarity] • Explain speculative execution and identify the conditions that justify it [Familiarity] • Discuss the performance advantages that multithreading offered in an architecture along with the factors that make it difficult to derive maximum benefits from this approach [Familiarity] • Describe the relevance of scalability to performance [Familiarity]

TABLE 43 CONCURRENCY RELATED TOPICS FROM NSF/TCPP CURRICULUM GUIDE COVERED IN OUR
TEACHING

Category	Topics	Learning Outcome
8.2 Architecture Topics	1. Taxonomy	Flynn's taxonomy, data vs. control parallelism, shared/distributed memory
	2. MIMD	Identify MIMD instances in practice (multicore, cluster, e.g.), and know the difference between execution of tasks and threads
8.3 Programming Topics	3. Shared memory	Be able to write correct thread-based programs (protecting shared data) and understand how to obtain speed up
	4. Task/thread spawning	Be able to write correct programs with threads, synchronize (fork-join, producer/consumer, etc.), use dynamic threads (in number and possibly recursively), thread creation (e.g. Pthreads, Java threads, etc.) builds on shared memory topic above
	5. Language extensions	Know about language extensions for parallel programming. Illustration from Cilk (spawn/join) and Java (Java threads)
	6. Tasks and threads	Understand what it means to create and assign work to threads/processes in a parallel program, and know of at least one way do that (e.g. OpenMp, Intel TBB, etc.)
	7. Synchronization	Be able to write shared memory programs with critical regions, producer-consumer communications, and get speedup; know the notions of mechanisms for concurrency (monitors, semaphores, etc.)
	8. Critical regions	Be able to write shared memory programs that use critical regions for synchronization
	9. Producer-consumer	Be able to write shared memory programs that use the producer-consumer pattern to share data and synchronize threads
	10. Monitors	Understand how to use monitors for synchronization
	11. Concurrency defects	Understand the notions of deadlock (detection, prevention), race conditions (definition), determinacy/non-determinacy in parallel programs (e.g. if there is a data race, the output may depend on the order of execution)
	12. Deadlocks	Understand what a deadlock is, and methods for detecting and preventing them
	13. Data Races	Know what a data race is, and how to use synchronization to prevent it
	14. Distributed Memory	Know basic notions of messaging among processes, different ways of message passing, collective operations
	15. Message passing	Know about the overall organization of an message passing program as well as point-to-point and collective communication primitives (e.g. MPI)
	16. Functional/logic languages	Understanding advantages and disadvantages of very different programming styles (e.g., parallel Haskell, Parlog, Erlang)
	17. Work stealing	Understand one way to do dynamic assignment of computation
	18. Tools to detect concurrency defects	Know the existence of tools to detect race conditions (e.g. Eraser)
8.4 Algorithm Topics	19. Synchronization	Be aware of methods for controlling race conditions
8.5 Cross Cutting and Advanced Topics	20. Why and what is parallel/distributed computing	Know the common issues and differences between parallel and distributed computing: history and applications. Microscopic level to macroscopic level parallelism in current architectures.
	21. Concurrency	The degree of inherent parallelism in an algorithm, independent of how it is executed on a machine
	22. Non-determinism	Different execution sequence can lead to different results hence algorithm design either be tolerant to such phenomena or be able to take advantage of this

Our efforts in teaching concurrency are carried out within the following dimensions and their interactions. First are the models of concurrent systems, including shared memory model and message passing model. Second are several concurrency approaches, such as the thread approach, actor approach and coroutine approach. Third are some classical problems and scenarios of concurrency, including producer-consumer, dining philosophers, etc. To better prepare for our specific purpose of informing the computer science education community, we also add to our investigation a fourth dimension of major concurrency concepts identified in curriculum guidance.

4.3.1 MODELS OF CONCURRENCY

Two concurrency models are pervasively used in contemporary computing software that differentiate two types of inter-process communication. One model is the shared memory model in which blocks of random access memory can be accessed by several different processing units so that different processes communicate through a single unified image of memory. A second model is the message passing model in which each process has its own private memory and communicates through the exchange of messages. In some large scale systems, usually distributed, both of these models are adopted at different levels of the system architecture. However, to limit our scope of informing undergraduate computer science education society, we eliminated the discussion of adopting a hybrid model in a single system.

4.3.2 APPROACHES TO CONCURRENCY

We restrict our research to three major approaches to achieving concurrency. These approaches are widely used in systems of different levels of complexity and with different functionalities. They are also efficiently supported by modern programming languages. This entitles them to be good candidates for teaching. Notice that although the characteristics of some approaches enable them to implement one type of concurrency model more naturally and easily, generally speaking, concurrency models and concurrency approaches are not tied by any means.

In this section, we discuss the origin of different concurrency approaches, our considerations in the selection of implementation languages, a brief summary of the features of different languages, and some simple efficiency tests and the visualizations of usage of multi-core architectures with our selected language implementations on three simple concurrent programs.

Java and Thread-Based Approach

First is the thread-based approach supported by programming languages such as C, C++, Java, Python, etc. Threads usually share the same memory space, which makes it easier to implement the shared memory model of concurrency. However, the thread-based approach may also be used to implement message passing systems. For example, Java threads may also be used to send and receive synchronous or asynchronous messages to achieve a message passing model of concurrent system.

We further select Java as a realization of the thread-based approach because of its pervasive use over a large array of devices and the fact that it is a popular introductory programming language in many CS curricula. However, an understanding of performance issues calls for a bit deeper exploration of the underlying implementation. A mainstream programming language, Java is implemented on top of the Java virtual machine. According to the documentation of the most recent version of Java implementation, the Java Development Kit (JDK) provides two kinds of Java virtual machines (VM) (Oracle, 2013):

- On platforms typically used for client applications, the JDK comes with a VM implementation called the Java HotSpot Client VM (client VM). The client VM is claimed to be tuned for reducing start-up time and memory footprint.
- On all platforms, the JDK comes with an implementation of the Java virtual machine called the Java HotSpot Server VM (server VM). The server VM is designed for maximum execution speed.

Supporting thread synchronization is claimed to be one of the Java HotSpot VM features that the Java programming language allows for use of multiple, concurrent paths of program execution (called "threads") and Java HotSpot technology provides a thread-handling capability that is designed to scale

readily for use in large, shared-memory multiprocessor servers (Lindholm, et al., 2013). However, detailed documentation is lacking for two thread mapping and handling mechanisms. One mechanism for which detail is lacking is the way in which the Java VM compiler maps threads defined in the program to threads running in the Java virtual machine. The other is how the Java VM maps threads running in it to actual tasks executed on multi-core architecture systems. For the first mapping mechanism, although it is widely believed that each thread defined in a Java program occupies a Java virtual machine thread dedicatedly while running, no specific and detailed documentation is available. All we know from the previous version of the Java virtual machine specification is that thread synchronization in the Java VM is realized through low-level machine implementation with semaphores and locks.

The Java default language package *java.lang* provides two artifacts that are related to concurrency. One is the Runnable interface and the other is the Thread class. By defining a class that implements Runnable or extends Thread, the Java language provides a whole set of thread manipulation functions including start(), resume(), stop(), and join(), etc. Also, a set of inter-thread communication methods are defined in the object class under the *java.lang* package, including wait(), notify() and notifyAll() which enables every object to cooperatively participate in a concurrent system. Therefore, threads may be regarded as an embedded feature of the Java language. Besides, Java also provides a utility library, *java.util.concurrent* which provides many practical artifacts such as CountdownLatch and CyclicBarrier. After years of development and usage, the Java thread approach is pretty mature with these provided language features and libraries. Also, due to the large amount of usage of Java threads, its documentation is more comprehensive and detailed compared to other two concurrency approaches that we consider. Furthermore, pervasive code examples and technical articles are available online. These are not official parts of the Java language feature or libraries by any means but may provide great relief to programmers.

Scala and Actor-Based Approach

The second approach we consider is the actor-based approach supported by programming languages such as Erlang and Scala. Before getting any deeper into this concurrency approach, it will be beneficial to first review some fundamental concepts. The first concept is the “happened before” relation (Lamport, 1978) among distinct events in universe which in turn defines the concept of time. This partial relation may then be extended to a full relation with an algorithm that results in a non-deterministic event sequence in a distributed system, a system in which the transmission time of messages among different tasks cannot be neglected when compared to the time between two events happen in the same task. In such a distributed system, tasks may be carried out on computational units that are either spatially separated or on a single processor. These fundamental concepts actually characterize a concurrent system with non-determinism of task executions.

These concepts are employed by the Actors approach. We differentiate the Actors-based approach from primitive message passing interfaces such as OpenMP and MPI in that the Actors-based approach illustrates a different mathematical theory of computation. In this theory, “Actors” are the universal primitives of concurrent digital computations. An Actor is a computational entity that in response to a message it receives can concurrently: 1) send messages to other actors, 2) create new actors, and 3) designate how to handle the next message it receives (Hewitt, 2010). The nature of the Actors-based approach makes it easy to implement message passing. However, Actors may also be designed and created to share common memory spaces to emulate a shared memory system.

Scala is a recently popularized general-purpose programming language that integrates features of object-oriented and functional languages. Scala programs also run on Java virtual machines and the program byte code is compatible with Java. Therefore, Scala fully allows usage of any existing Java libraries or application packages. Scala programs may be called from Java and vice versa, with seamless integration. Accordingly, it is fully possible to implement concurrency and synchronization in Scala by

using Java Threads artifacts with the *java.lang.Thread* and *java.util.concurrent* libraries which provide several thread definition mechanisms, inter-thread communication mechanisms and some high-level synchronized object classes. The thread synchronization and monitor patterns available in Java are also fully accessible in Scala.

However, Scala differs from the Java programming language in that it provides another means to implement concurrency -- the Actors approach. To support the Actors approach, Scala provides a set of language utilities to deal with sending, receiving, and handling messages, and creating and recognizing different actors. The Scala language provides a library package *scala.actors* for actor programming. Inside this package, Scala provides both asynchronous and synchronous messaging mechanisms and artifacts to allow concurrent programming without explicit synchronization. The most important member of the package is the Actor trait in which basic operations (sending, receiving, reacting and forwarding messages) that may be performed by an actor are defined. Besides this package, Scala also provides a *scala.concurrent* library. In this library, several utilities for concurrency are defined.

Scala is still a relatively young programming language. Scala documentation is lesser in both quality and quantity, as compared to Java. Many objects defined in Scala libraries lack detailed descriptions and deprecated or re-factored objects and functions are scattered around in the documentation. Virtually, no sample code can be found in the Scala documents and many functions are under-specified. Further, the language itself is still evolving rapidly and is somewhat unstable.

Python and Coroutine-Based Approach

The third approach we consider is the Coroutine-based approach supported by programming languages such as Haskell and Python. Coroutines support cooperative multitasking with the ability to suspend and resume executions. Coroutines may be implemented as a way of sharing data as in a shared memory model of concurrency or as a way of reacting to events (inputs of execution resumption) as in a message passing model of concurrency.

The concept of coroutines was introduced in the early 1960s and constitutes one of the oldest proposals of a general control abstraction. It is attributed to Conway, who described coroutines as “subroutines who act as the master program” (Conway, 1963). Then, the use of coroutines to express several useful control behaviors was widely explored during the next twenty years in several different contexts, including simulation, artificial intelligence, concurrent programming, text processing, and various kinds of data-structure manipulation (Knuth, 1981) (Marlin, 1980) (Pauli & Soffa, 1980). However, the power of coroutines is generally omitted from the design of general-purpose programming languages. The rare exceptions are Simula, BCPL, Modula-2, Icon and the generator in the Python language.

The fundamental characteristics of a coroutine are introduced in (Marlin, 1980) as follows:

- The values of data local to a coroutine persist between successive calls (to that coroutine)
- The execution of a coroutine is suspended as control leaves it, only to carry on where it left off when control re-enters the coroutine at some later stage.

In addition to this fundamental description, three further issues are identified in (de Mour & Ierusalimsky, 2004) for a coroutine:

- The control-transfer mechanism, which can provide *symmetric* or *asymmetric* coroutines
- Whether coroutines are provided in the language as *first-class* objects, which can be freely manipulated by the programmer, or as constrained constructs
- Whether a coroutine is a *stackful* construct, i.e., whether it is able to suspend its execution from within nested calls

Based on these three issues, the authors of (de Mour & Ierusalimsky, 2004) classify coroutines into different categories and claimed that a first-class stackful coroutine provides the same expressiveness as obtained with one-shot continuation which supports concurrency, as described in (Hieb & Dybvig, 1990). Therefore, a system that supports coroutines is totally capable of defining a concurrent system.

Python is a general-purpose, interpreted, high-level programming language that supports several programming paradigms including object-oriented, imperative and, to some extent, functional programming. Several different implementations exist. The most commonly seen is CPython, which is a mainstream Python implementation written in C that compiles Python programs into byte codes that are then executed on the CPython Interpreter. Part of this interpreter is a simple stack machine known as the Python VM (virtual machine), which actually executes the Python byte code. In terms of the threading in Python VM, Python threads are true operating system (OS) threads.

Coroutines are a rarely used concurrency concept due in part to the pervasiveness of implementation and wide acceptance of thread-based approach. Few supportive resources are available for implementing coroutines in Python. The most related resource is the PEP 342 document (van Rossum & Eby, 2005) on the implementation of coroutines through enhanced functionality of generators. This document contains all the essential elements for implementation of coroutines, yet it is difficult for novices to understand without sample codes and detailed explanations.

Some general patterns exist for the design and implementation of concurrent programs under these three different approaches. To design and implement a simple concurrent program with the Java Threads approach, one may first identify passive (shared) versus active (thread) objects. Then one would apply the monitor pattern to the data manipulation methods of shared objects and implements the Runnable interface for the active objects. To design and implement a simple concurrent program with the Scala Actors approach, one would first design a protocol of message types and the corresponding information carried by each type of message exchanged among actors, as well as each actor object's corresponding behavior on sending and receiving each type of message. Then one would apply the reactor pattern to all actors and implement messages as final classes (the case class as defined in Scala). To design and implement a simple concurrent program with Python Coroutines approach, one would first identify shared objects versus functional objects (coroutines) as well as the suspension and

resumption conditions of these functional objects. Then, one would apply the generator pattern to those functional objects through the use of yield. Table 44 generalizes the basic concurrency constructs and design procedures to implement a concurrent program with three different approaches.

TABLE 44 CONCURRENCY CONSTRUCTS AND DESIGN PROCEDURES OF THREE APPROACHES

Approach	Constructs	General Design Procedure
Java thread	java.lang.Object java.lang.Runnable Runnable interface java.lang.Thread wait(), notify(), notifyAll()	discern shared vs. thread object apply monitor pattern (Figure 18)
Scala actor	scala.actors._ Actor._ !, receive, react, mailbox related function	design protocols of message types and behaviors apply serializer pattern (Figure 19)
Python coroutine	PEP 342: enhanced generator function yield, send, next, StopIteration exception	discern shared object discern coroutine and progress conditions apply generator pattern (Figure 20)

Data object class Function lock while condition is false wait execution unlock end function end class	Active object class Run function invoking data object class's functions end run function end class
---	---

FIGURE 18 MONITOR PATTERN OF IMPLEMENTING CONCURRENCY

Data object class Receive message while condition is false delay processing processing End receive message End class	Active object class Run function send messages End run function End class
--	---

FIGURE 19 REACTOR PATTERN OF IMPLEMENTING CONCURRENCY

Data object class Function If condition is false return false Else processing return true End if End function End class	Active object class Run function while invoking data object class's function returns false yield yield End run function End class
--	--

FIGURE 20 GENERATOR PATTERN OF IMPLEMENTING CONCURRENCY

4.3.3 CONCURRENCY SCENARIOS

We explore a list of concurrency scenarios and problems for further research and we describe their problem settings here.

The **Ornamental garden** scenario involves an ornamental garden with two turnstiles through which tourists may enter and the problem is to correctly maintain a count of the total number of tourists in the garden at any time.

The **Sum and worker** scenario involves a group of workers trying to report the number of job they each completed and the problem is to correctly maintain the total number of jobs done at any time.

The **Bank account** scenario involves a checking account that supports both deposit and withdrawal operations. A number of customers make deposits to and withdrawal from this account. The problem is to correctly maintain the account balance and also guarantee that no overdraft ever occurs.

The **Bounded buffer** scenario involves a buffer with limited capacity and that supports both put and get operations. A put operation places an item into the buffer and a get operation removes an item from the buffer. A number of producers place items into the buffer and a number of consumers remove items from the buffer. The problem is to correctly maintain the state of the buffer and to guarantee that it is never overflows (by placing items into a full buffer) or underflows (by removing items from an empty buffer).

The **Dining philosopher** scenario involves five silent philosophers sitting at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, the philosopher needs to put down both forks so they become available to others. A philosopher can grab the fork on his right or the one on his left as they become available, but can't start eating before getting both of them. Eating is not limited

by the amount of spaghetti left. The problem is how to design a discipline of behavior such that each philosopher won't starve, i.e. can forever continue to alternate between eating and thinking, assuming that a philosopher cannot know when others may want to eat or think.

The **Readers and writers** scenario involves a number of readers trying to access some shared data without modifying it and a number of writers trying to access the same shared data and modify it. Therefore, readers may “read” data concurrently while writers need exclusive access to data. The problem is to design a discipline of behavior such that both readers and writers have opportunities to progress.

The **Party matching** scenario involves a number of boys and girls who arrive at a party and leave in pairs of a boy and a girl. The problem is to correctly maintain the status of the party and guarantee that each participant leaves with a partner.

The **Sleeping barber** scenario involves a barber's shop that has a number of barbers working in it and customers come to it to have different kinds of barbering services. These services take different amounts of time to finish. When no customer is in the shop, the barbers rest. If a customer comes and barbers are available (barbers who are resting), one of the available barbers should provide the customer the required barbering service. When all the barbers are serving customers, a newly arriving customer should wait in the shop. However, if the waiting space is already full, that customer will leave directly without having any service. Barbers keep a record of the total time of service they have provided and do not serve any more customers when that time reaches/exceeds their maximum working time. The problem is to keep barbers working when there are customers to serve and resting when there are no customers.

The **Book inventory** scenario involves a book stock management server that talks to multiple clients. Clients request to increase or decrease the stock of a number of books by making job requests. No negative stock is allowed and the server is responsible for holding and later applying “decrease jobs”

that could not be performed due to insufficient stock. Multiple jobs may be processed concurrently within the server. The problem is to maintain a correct inventory of book stocks and guarantee that all accepted job will eventually complete assuming limited concurrent processing power of the server and that, over time, the increases equal or exceed the decreases for any one book.

The **Single lane bridge** scenario involves a single-lane bridge that is wide enough to permit only a single lane of traffic. That is, the bridge permits only one-way traffic at any time and cars exit the bridge according to their order of entering the bridge. The problem is to correctly maintain the state of the bridge while guaranteeing all cars have a chance to utilize the bridge and proceed.

The concurrency related curriculum topics listed in Table 43 are covered from various perspectives by implementing concurrent programs with the three concurrency approaches (Java Threads, Scala Actors and Python Coroutines) and the ten concurrency scenarios listed above. Table 45 shows the coverage of curriculum topics by the three concurrency approaches and ten concurrency scenarios.

TABLE 45 COVERAGE OF CURRICULUM TOPICS BY CONCURRENCY APPROACHES AND SCENARIOS

Scenarios	Java Thread	Scala Actor	Python Coroutine
Ornamental Garden, Sum & Workers	3, 4, 5, 6, 7, 8, 11, 13, 22	4, 14, 15, 17	4, 16
bank account, bounded buffer	3, 4, 5, 6, 7, 8, 9, 10, 11, 22	4, 14, 15, 17	4, 16
dinning philosopher, readers and writers, party-matching, book inventory, single-lane bridge	3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 22	4, 14, 15, 17	4, 16

In our spring 2013 work, we designed and carried out our plan of teaching programming with concurrency in an exploratory, upper-level undergraduate CS course. We developed appropriate teaching materials for these concurrency programming models and implementation approaches. We carried out the course with the innovations of introducing a pseudocode for concurrency, flipped classroom organization and intensive problem solving practice with various implementation approaches for different concurrency programming models. Section 4.4 discusses the findings and feedback of our course design and pedagogical efforts.

4.4 TEACHING PROGRAMMING WITH CONCURRENCY

Current trends in multi-core and multi-processor architectures demand that students in Computer Science and Computer Engineering not only master concurrency concepts but also develop substantial practical skills in concurrent and parallel programming. However, even with recent updates to the undergraduate curriculum to include PDC concepts, Computer Science students are not systematically introduced to different development approaches. Difficulties in programming such systems correctly and efficiently are seen in both academia and industry. Improved pedagogical design on teaching concurrency related programming and a comprehensive study of how programmers use different programming language approaches to concurrency may help to provide guidance.

4.4.1 COURSE DESIGN AND MATERIALS

Table 46 describes how different course materials we designed cover the curriculum topics related to concurrency as discussed in section 4.3. Please reference Table 42 for the detailed learning outcomes for each topic.

TABLE 46 COVERAGE OF CONCURRENCY RELATED CURRICULUM TOPICS

Category	Topics	Course Material
8.2 Architecture Topics	1. Taxonomy	Lectures on topic of multi-core architecture and overview of Parallel and concurrent programming Reading: Introduction to Parallel Computing, Chapter 1-4 Reading: Parallel Computer Architecture, Flynn's Taxonomy Reading: Parallel Computer Architecture, Memory Organizations Reading: Parallel Computer Architecture, Caches and Memory Hierarchy Reading: Multicore processors and Systems, General-Purpose Multi-core Processors Reading: Parallel Programming, Interconnection Networks Reading: Parallel Programming, Routing and Switching
	2. MIMD	
8.3 Programming Topics	3. Shared memory	Reading: Parallel Programming, Thread Level Parallelism Reading: Introduction to Parallel Computing, Chapter 5.1-5.3 Reading: Parallel Programming, Parallel Programming Patterns Reading: Parallel Programming, Synchronization Mechanisms Reading: Mutexes and Semaphores, Part I – III (online blog) Reading: Livelock (online blog) Lecture: Shared Memory Concurrent Systems Race Condition Sum & Worker Example in Java, C++, Pseudo Code Ornamental Garden Videos Pseudo Code System Conditional Synchronization Bank Account Example in Pseudo Code Design Activity: Bounded-Buffer System Deadlock & Livelock Large Printing Job Example in Pseudo Code Four necessary condition for deadlock Design Activity: Dining Philosopher System Fairness Issue Readers and Writers Example in Pseudo Code Homework: Practice Pseudocode with Shared Memory Systems Project: Design Book Inventory as Shared Memory System Reading: Java Concurrency Tutorial Project: Party Matching with Java Threads Project: Sleeping Barber Simulation (in Java) Project: Debugging Contest on Book Inventory System (in Java)
	4. Task/thread spawning	
	5. Language extensions	
	6. Tasks and threads	
	7. Synchronization	
	8. Critical regions	
	9. Producer-consumer	
	10. Monitors	
	11. Concurrency defects	
	12. Deadlocks	
	13. Data Races	
	14. Distributed Memory	Reading: Introduction to Parallel Computing, Chapter 5.4 Reading: Parallel Programming, Message Passing Programming Lecture: Message Passing Concurrent Systems Non-deterministic Order of Messages Sum & Worker Example in Scala, Pseudo Code Ornamental Garden White Board Illustration Pseudo Code System Conditional Synchronization Bank Account Example in Pseudo Code Group Design Activity: Bounded-Buffer System Deadlock & Livelock Large Printing Job Example in Pseudo Code Group Design Activity: Dining Philosopher System Fairness Issue Readers and Writers Example in Pseudo Code Homework: Practice Pseudocode with Message Passing System Project: Design Book Inventory as Message Passing System Reading: Programming in Scala, Chapter 32
	15. Message passing	

		Project: Party Matching with Scala Actors Project: Sleeping Barber Simulation (in Scala) Project: Debugging Contest on Book Inventory System (in Scala)
	16. Functional/logic languages	Paper presentations Lecture: Cooperative Multi-tasking Concurrent Systems
	17. Work stealing	Homework: Practice Python with Cooperative Multi-tasking System
	18. Tools to detect concurrency defects	Reading: A Comprehensive Tutorial on Python Coroutines (online resource) Project: Party Matching with Python Coroutines Project: Sleeping Barber Simulation (in Python) Project: Debugging Contest on Book Inventory System (in Python)
8.4 Algorithm Topics	19. Synchronization	See materials cover topic 3-13
8.5 Cross Cutting and Advanced Topics	20. Why and what is parallel/distributed computing	See materials cover topic 1-15
	21. Concurrency	
	22. Non-determinism	See materials cover topic 3-15

4.4.2 COURSE FEEDBACK

Surveys on effort and preferences (see Figure 45 in section 6.5) were collected with each lab and homework assignment. Students consistently reported difficulties with the shared memory model. In homeworks 2 (shared memory) and 3 (message passing), students were asked to write pseudocode for the bounded-buffer and dining-philosopher problems discussed in class. In a survey conducted after homework 3, only one student indicated that the message-passing model was more difficult and 10 indicated that the shared memory model was more difficult. The remaining students either indicated that the two approaches were equally difficult, or they did not respond to the question. In lab 2 (shared memory) and lab 3 (message passing) students were asked to design a book inventory system. In the post-lab survey, 8 of 11 students who responded indicated that shared memory is more difficult, 1 indicated that message passing is more difficult, and 2 students found the assignments equally difficult.

In the cases of both the homeworks and the labs, students were asked to first solve the problem for the shared-memory case and then for the message-passing case. Thus, ordering effects may explain the preference for message-passing. Therefore, for Test 1, students were assigned into two groups S and D such that the groups had equivalent performance on previous assignments and were asked to complete the sections of the exam in opposite orders. In the 1st session group S took the shared-memory section of the exam and group D took the message-passing section of the exam. In the 2nd session, each group took the remaining section of the exam. The testing order is listed in Table 47.

TABLE 47 PERFORMANCE ON MIDTERM EXAM

Group	Shared Memory Mean	Message Passing Mean	Overall Mean
S (9 students)	56.67 / 100 (1 st)	81.72 / 100 (2 nd)	138.39 / 200
D (7 students)	76.14 / 100 (2 nd)	65.93 / 100 (1 st)	142.07 / 200
All	65.19 / 100	74.81 / 100	

Group S finishes shared memory first, message-passing then
Group D finishes message-passing first, shared memory later

After test 1, we again surveyed students on their perceived difficulty of the two different systems. In this survey, 11 of the 15 students who responded indicated that questions in the shared memory

section were harder to answer than those in the message passing section. In the same survey, students were given the opportunity (without knowing their scores) to choose which of the two sections of the exam would count as their midterm grade. (In fact, we always used the higher-scoring section to count toward their class grade). Of the respondents, 10 of the 15 choose the message passing section. Of the 5 students who chose the shared memory section, 4 took the shared memory portion in the 2nd session. Of these 15 students, 13 chose correctly, in that they selected the section in which they actually scored higher. The two students who chose incorrectly chose the shared memory section but actually scored slightly higher on the message-passing section.

Test results are listed in Table 47. We found no significant difference in performance between the shared-memory and message-passing sections. However, we did find that students performed better in the 2nd session (79.20%) than in the 1st session (60.71%) ($p=0.005$), likely as a result of learning that occurred during the exam and/or additional studying that may have occurred between sessions. However, the students' better raw scores on the message passing section than on the shared memory section supports the survey result that students found the shared memory model more difficult to understand. We suspect that one reason for this effect is that we introduced concurrent systems in shared memory model with the monitor pattern. Empirically evidence shows that data and control flow information of object oriented designs are difficult for novices to capture (Fix, et al., 1993), (Wiedenbeck, et al., 1993). However, this type of information is critical to both implement and understand the behaviors of a concurrent system. As we saw in the case study described in section 3.3, novice students may establish a raw impression of the structure of a monitor pattern, but it is hard for even intermediate students to reason about the data and control flow of a monitor implementation. On the other hand, the rules and principles of message passing are more analogous to daily activities such as a mail system and therefore are much easier for students to start with.

The students' preference for the message passing model with Scala Actors approach persisted through the last survey carried out before the final exam. As seen in Figure 21, students reported slightly more difficulties in both comprehension and implementation related tasks with Java Threads. However, the majority of students (10 out of 12) actually used Java Threads for the implementation of their final exam. We suspect that although threads and the monitor pattern create difficulties for students, most students are much more familiar with Java than Scala (Java is taught in CS1 or CS2 for these students, but Scala is freshly new). The momentum of familiarity and flexibility with a previously used language overcomes the slight difficulties that language and approach present. Another issue we noticed is that students constantly report Python Coroutines to be difficult to either implement or to understand. Consequently, no student chose to use Python Coroutines as to implement the final exam question. However, according to the lecturer's solution of these three implementations, the Python Coroutines approach only requires students to implement 92 out of the 182 lines of the code skeleton compared to Java Threads 115 out of 224 lines of the code skeleton and Scala Actors 165 out of the 250 lines of code. Yet, we suspect that students find it difficult to accept coroutines because their programming education has focused on the subroutine approach, in which caller and callee relationships can be distinctively identified in a function call. However, coroutines allow functions to call and suspend one another, which impose a dramatic cognitive load for students to understand, let alone plan a system in this paradigm. Some advanced students appreciated the introduction of coroutines, stating that the coroutine "helps them to think about things in another way". But for the majority of average students, this is not a simple approach.

As to the gaining of programming expertise with each of the three different programming languages respectively, most students report a moderate increase in their perceived expertise levels. The Likert scale we use is from no knowledge (0) to novice (1), intermediate (3) and expert (5). Most students were fairly familiar with Java before the class and so had a limited increase in general expertise

through practicing the programming of concurrent systems with Java Threads, as seen in the top bar chart of Figure 22. However, as seen in the bottom bar chart of Figure 22, all except two students (number 1 and 4) reported good learning outcome regarding the Threads approach. It is apparent that our course complements the missing part of students' past undergraduate courses in using Java to implement multi-thread programs. Scala is the most unfamiliar language for students in the class and through the practice of programming concurrent system in the message passing model, most students reported gaining expertise except one (number 5) as seen in the top bar chart of Figure 23. From the corresponding bottom bar chart of Figure 23, we could see that student number 5 was already pretty familiar with the Scala Actors approach and therefore the course content imposed only a limited challenge for this student to gain expertise. Python was relatively familiar to most of the students as they may have experience with it in previous undergraduate course. However, due to the confusion that resulted from introducing coroutine as we described above, students report only limited increase in expertise of programming with Python as seen in the top bar chart of Figure 24. It is worth noting that in the bottom bar chart of Figure 24, student 4 even reports a decrease in learning coroutine approach. But since our questions (see detailed questions in Figure 45 in section 6.5) were formulated to ask about students' perceived expertise, this may also imply that this student simply realized the complexity of coroutine only after taking the course.

Seven of the ten students who responded to the final survey reported that they have a moderate learning of concurrency concepts. Two of those ten students reported above moderate learning outcomes on concurrency concept and the last reported a basic learning outcome. All students reported above basic acquisition of programming capabilities. Six of them think they have a moderate gain and three of them indicate reaching the expert level. We understand that these surveys are subject to student's own perceptions, which are largely affected by their previous knowledge and capabilities.

However, the positive replies definitely reveal students' increased level of programming efficacy, which is important for their future learning and success in the computer science domain.

When asked about different elements in the course, most student reported that the scattered content of the course without a course textbook created many difficulties in their learning. One student (a novice) even proposed to have a book of concurrent concepts with pseudocode only. Other novice students reported that the use of pseudocode helped them in framing the rest of the course and tackling larger problems. At the same time, some other students did not like the concept of using pseudocode and expressed that it is even harder than programming languages (mostly more advanced students) since "it is not possible to compile and test pseudocode". Our original intention was to use the pseudocode approach to promote students' thinking and planning before coding. However, it turns out again that the nature of programming activity is information intensive and advanced programmers become well used to chasing down information. Considering this feedback, we think a textbook that weaves different concurrent concepts, models, and approaches is in need and a pseudocode that is both easier to understand for novices and able to be compiled for experts would be the sweet spot.

Students have two types of feedback regarding the use of different languages to solve a single problem. Novices' negative feedback indicated that it creates a large overhead to warm-up and learn each language while advanced students' negative feedbacks indicated that it discouraged their motivation since they've already have working knowledge of all languages and it is tedious to repeatedly implement a problem with these languages. Positive feedback reported that using different languages to solve a single problem promotes engagement and helps in understanding both concurrency concepts and gaining programming expertise with different languages (from both novice and expert students). Therefore, it seems a balance or synthesis of learning programming languages and concurrency related topics may be further explored. We propose that this course could be slightly reorganized according to the languages so that in each sub-portion of the course, students learn the basics of the language and

then the concurrency features to solve a concurrency problem with that language. Also, having students work on projects with different problem statements (scenarios) but with identical concepts may help with the issue of engagement.

In general, students report positive feedback on programming-oriented learning in this course and comment that this approach helps them to learn faster. But students also proposed that more debugging and testing practice should be included earlier. In our current course, only one debugging contest towards the end of the course was organized and most students felt not quite prepared for such a task.

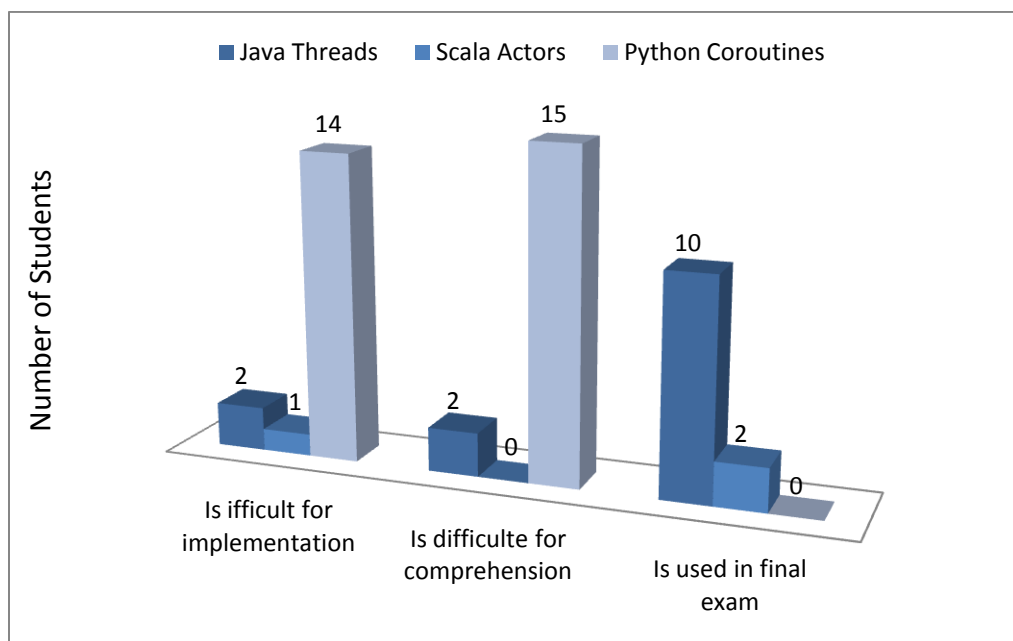


FIGURE 21 SUBJECTIVE DIFFICULTY LEVEL AND USAGE OF THREE PATTERNS

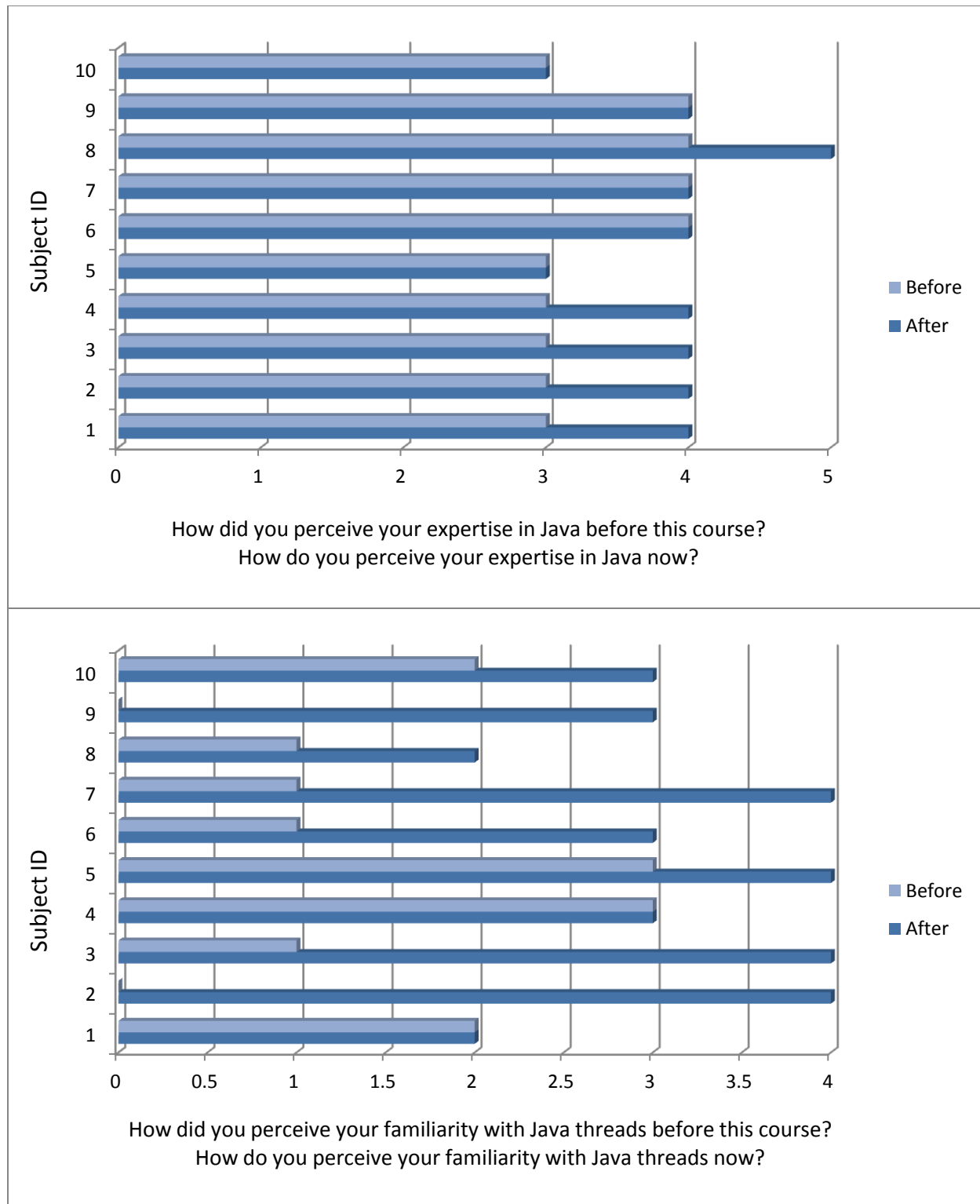


FIGURE 22 PERCEIVED EXPERTISE IN JAVA AND THREAD APPROACH BEFORE AND AFTER COURSE

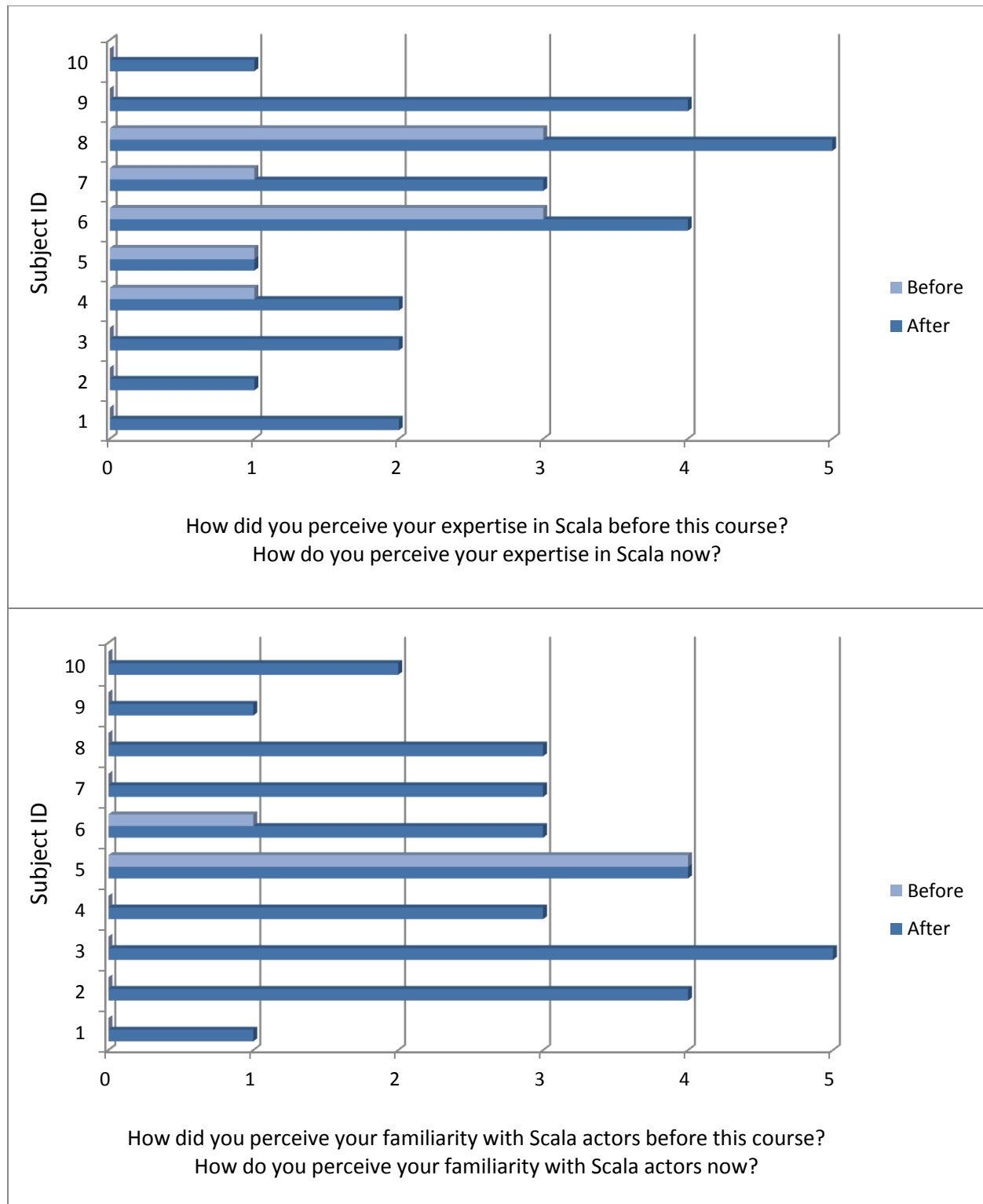


FIGURE 23 PERCEIVED EXPERTISE IN SCALA AND ACTOR APPROACH BEFORE AND AFTER COURSE

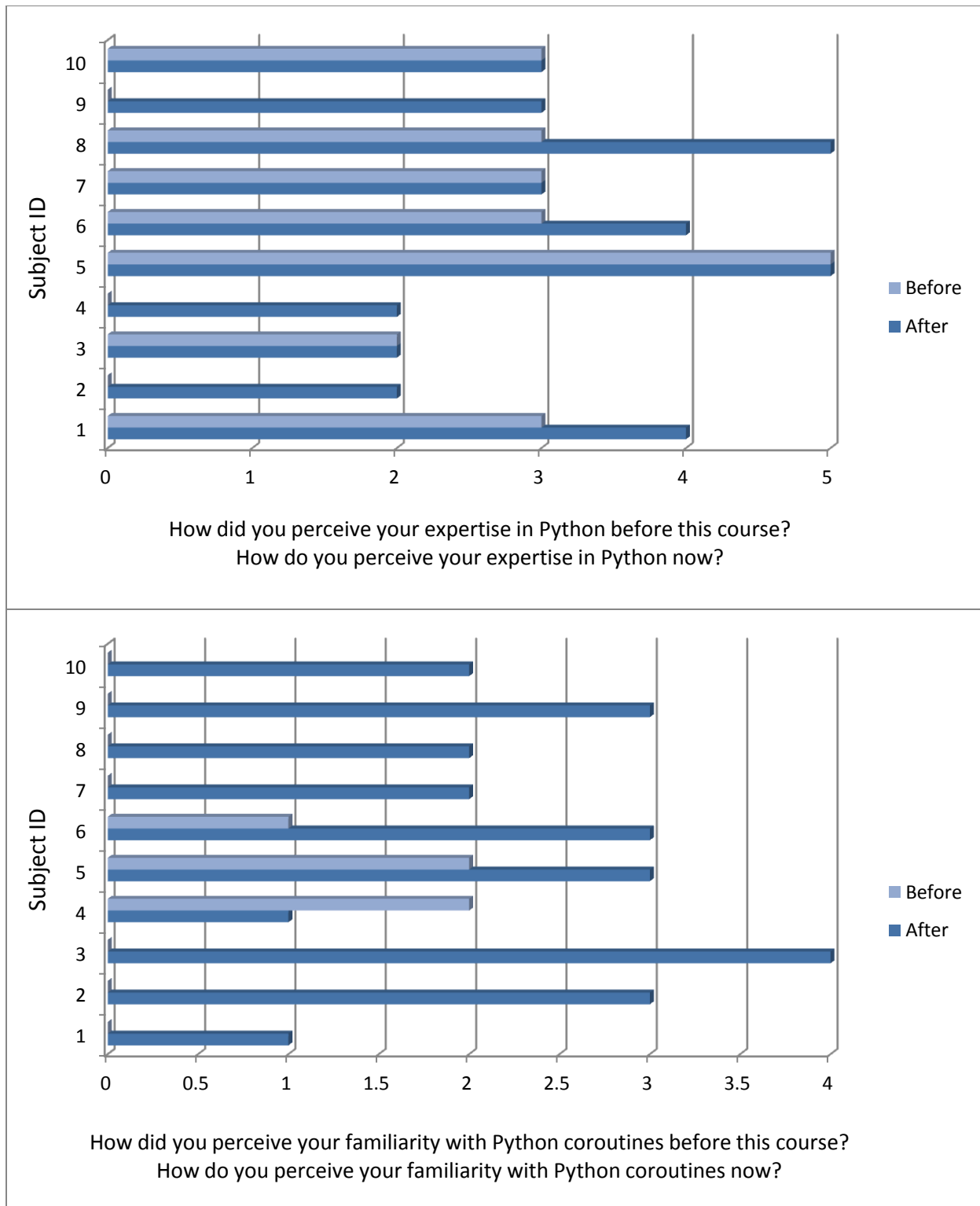


FIGURE 24 PERCEIVED EXPERTISE IN PYTHON AND COROUTINE APPROACH BEFORE AND AFTER COURSE

4.5 SUMMARY AND FUTURE WORKS

In summary, in this section, we describe a quasi-experimental study of pair programming versus solo programming, in the context of a required, four credit hour, intermediate-level course that focuses on the acquisition of a second language (C/C++). We report several important findings especially that both female students and less-experienced students were less likely to withdraw from the course if they were in the pair programming group and that pair programming helps improve the code health level. Thus, we provide evidence that builds upon prior findings and identifies pair programming as a promising protective factor as a pedagogical technique. We also reinforce the engineering impact of pair programming with novice students. As in any quasi-experimental study, however, we note several factors that may impact the interpretation of findings. First, the number of females in the CSCI 1730 course was low (10 out of 60), but consistent with previous courses. However, we believe the protective effect of pair programming that we observed provides further support for pair-programming in retention efforts. Second, while in-lab time was exclusively pair programming or solo programming, we did not have control over students' activities outside of the classroom. It is possible that out-of-class interactions might have varied. Third, we had students self-report the amount of time they spent on projects as well as the amount of time spent at TA office hours. We acknowledge that self-reporting may not accurately capture actual behavior. Future work should use a more empirically-measured approach. Finally, more solo programming students withdrew than did pair programming students. Specifically, more of the weaker solo students dropped than did the weaker pair programming students. This may have occluded a potential performance benefit of pair programming. However, there were no significant differences in the exam scores of the top third from students from each condition. Thus, we expect further empirical studies to reinforce these findings.

As an exploration on the impact of pair programming with production activities, we report a case study of a final exam with eight pair programmers working in four pairs and four solo programmers

working individually. Through analysis of experimenter's notes on observed behavior of pair and solo programmers, we conclude that pair programming pushes programmers to think in more detail about design requirements and system structures and to clarify critical questions and issues earlier. For pair programmers, all three questions that were critical to the design of the system were asked during their specification reading and system planning period. Although these questions are also asked by solo programmers, they were asked much later, during their testing and debugging phases. Therefore, we conclude that pair programming not only has positive impacts as a pedagogical technique to retain female and inexperienced students and promote code health, it is also powerful for design and code planning, which are some of the most difficult and tedious tasks in programming. Certainly, our case study was carried out in a limited manner in that only twelve programmers (four pairs, four solos) were observed. More empirical evidence is definitely in need to further explore the impact of pair programming across all software engineering activities.

We also describe our offering of an upper-level undergraduate computer science course that focuses on the concepts and programming of concurrent systems. We created a full set of materials that cover the curriculum guides and introduce both shared memory and message passing models of concurrent systems. The course included the usage of three distinctive programming approaches, Java Threads, Scala Actors and Python Coroutines, for the implementation of different concurrency models. Students practiced programming several classical concurrency scenarios such as dining philosopher, sleeping barber, single-lane bridge, etc. and report moderate gains in knowledge both regarding concurrency concepts and programming expertise. We also noticed and reported some caveats of our course design and possible changes to make in the future. We have an enhanced, purely online version of this course designed. It will be interesting to carry this course out in future over several offerings and modify its content and structure.

CHAPTER 5.

CONCLUSIONS AND CONTRIBUTIONS

In this thesis work, we carried out research of the intersection of psychology of programming and computer science education, focusing programming with concurrency. We first identified the barriers to learning programming with concurrency through three pieces of work: 1) a review of the literature and proposal of a general conceptual framework for the development and application of programming expertise, 2) two empirical studies using a combination of qualitative and quantitative methods that formulate and validate a “misconception hierarchy” that reveals the structure of concurrency-related knowledge and provides insight into the procedures for acquiring such knowledge, and 3) a case study of novice versus intermediate knowledge repository demonstrated during the task of implementing a concurrent program, which provides comprehensive arguments on other non-concurrency-related knowledge required for learning programming with concurrency. We then conducted explorations in teaching programming with concurrency through several pieces of work: 1) pair programming: a survey of previous work, a quasi-experimental study of students performing pair versus solo programming, a case study on the problem solving procedures of pair and solo programmers provides a comprehensive discussion of the impact of pair programming from pedagogical, engineering and cognitive perspectives, which provides insights into pedagogical design and course implementation especially for challenging topics such as programming with concurrency, and 2) course design: a realization of an upper-level computer science course that provides feedback about the benefits and drawbacks of various course materials, organizations, and teaching approaches. Through our discussion of barriers to learning and explorations in teaching programming with concurrency, this thesis work unifies previous research

efforts in teaching and learning programming with a focus on concurrency that meets current trends in computing and the increasingly prevalent use of concurrency.

The first contribution of this thesis work is a conceptual framework for the nature of programming expertise, how such expertise impacts the problem-solving process, and how such expertise is developed. Specifically, we identify the inter-relationship among three entities in the framework: a knowledge base, the mental representation of a problem, and external data. We identify the processes by which 1) both the knowledge base and external data are used to develop and evolve the mental representation, 2) the problem-solving process employs the evolving mental representation to guide a search of the knowledge base and for additional data, and 3) the repeated construction and subsequent internalization of mental representations builds the persistent, structured, and connected knowledge that is the basis of expertise.

This thesis work also describes a hierarchical organization of misconceptions exhibited by students engaged in learning about concurrency and how to program with concurrency; this hierarchical structure explains the content of and development mechanism for expertise in programming with concurrency. Specifically, we identify five levels of knowledge (description, terminology, concurrency, implementation and uncertainty). Our work reveals 1) that a lack of lower level knowledge prevents the acquisition of higher level knowledge, and 2) that a necessary phase exists in the knowledge acquisition process in which an apparent mastery of concepts is sacrificed to create a simpler solution space and the resulting incorrect solution helps the learner to reexamine and reconfirm the correct solution and the better internalize the related concepts.

Empirical evidence of student barriers to learning about programming with concurrency is identified and discussed in this thesis. Specifically, we found that non-concurrency related programming knowledge and even natural language related knowledge are critical in the problem solving process of programming concurrent systems.

A comprehensive evaluation of pair programming from pedagogical, engineering and cognitive perspectives for the development of programming expertise is presented in this thesis work. Specifically, we found that pair programming helps retain less-experienced and female students as a pedagogical intervention, encourages all students to write better styled code as an engineering technique, and stimulates students to devote cognitive effort earlier in the software design phase as a problem solving practice.

Several innovative pedagogical techniques are introduced and an evaluation of the benefits and caveats for using these techniques in teaching programming with concurrency are discussed in this thesis work. Specifically, we found the benefits of repeated programming practice in developing expertise but proposed the use of conceptually-identical-superficially-different problems to remedy the issue of potential discouragement caused by repeatedly practicing with the same problem. We also developed a recommendation for integration of teaching programming of concurrent systems into different computer science courses to better meet the needs of students with different levels of expertise.

A comparison of three distinct approaches to concurrency, based on empirical studies with popular programming languages is produced in this thesis work. Specifically, we found that although the Actors approach is reported to be easier for implementation and comprehension tasks, the familiarity of the Java language drives the students' choice to use Threads approach. We also found that students exhibit a poor mastery of the Coroutines approach, likely due to the dominance of the subroutine paradigm in their prior experience.

An extended pseudocode system is suggested and an evaluation of its use is also carried out in this thesis work. Specifically, we extended a pseudocode system to cover concurrency related concepts of both shared memory and message passing models and used it to test students' comprehension of concurrency concepts independently of any programming language. We evaluated and identified

caveats of imperfect syntactic design and the lack of a compiler for this pseudocode system during its usage in implementation and comprehension tasks.

Based on our thesis work, we propose following tracks of future work:

1. Further exploration of the validity and refinement of the conceptual framework. Through our literature review, much research effort has been devoted to studying the observable behaviors (how programmers search and fetch data), but less has been devoted to studying the invisible “behaviors” (how programmers access and retrieve knowledge) and the knowledge internalization procedures. Interdisciplinary work with cognitive scientists and psychologists would be appropriate.
2. A further understanding of knowledge acquisition procedures in programming and other domains and a more general understanding of knowledge accumulation and expertise development is needed. Specifically, we suspect that initial knowledge interferes with and or promotes the future knowledge building procedures as the knowledge base shapes the mental model which is in turn internalized into further knowledge. Therefore, towards the development of general programming expertise, it would be interesting to study the impact of choice of first programming language, programming paradigms and programming problems. Also, it is valuable to propose a knowledge acquisition hierarchy for learning of general programming knowledge.
3. A further study and empirical evidence on the cognitive impact of pair programming are needed. Specifically, it would be interesting to compare the evolution of the mental representations on a given problem but with two different knowledge bases, from the cognition and psychology perspectives. It would also be interesting to investigate the applications of such cognitive impact in the management, organization and education fields.
4. Innovations and refinements of pedagogical techniques and course designs on teaching programming with concurrency, moving forward. These research efforts will provide valuable

feedbacks not only from the educational perspective, but may also provide insights into human cognition and learning procedures.

REFERENCES

- Adelson, B., 1981. Problem Solving and the development of abstract categories in programming languages. *Memory & Cognition*, 9(4), pp. 422-433.
- Adelson, B., 1984. When Novices Surpass Experts: The Difficulty of a Task May Increase With Expertise. *Journal of Experimental Psychology*, 10(3), pp. 483-495.
- Adelson, B. & Soloway, E., 1985. The Role of Domain Experience in Software Design. *Software Engineering, IEEE Transactions on*, Volume SE-11, pp. 1351-1360.
- Ahmadzadeh, M., Elliman, D. & Higgins, C., 2005. *An Analysis of Patterns of Debugging Among Novice Computer Science Students*. Monte de Caparica, Portugal, ACM, pp. 84-88.
- Ball, L. J. & Ormerod, T. C., 1995. Structured and opportunistic processing in design: a critical discussion. *Int. J. Human-Computer Studies*, July, 43(1), pp. 131-151.
- Ball, T. et al., 2010. Preemption Sealing for Efficient Concurrency Testing. Volume 6015, pp. 420-434.
- Barnard, G. A., 1945. A New Test for 2×2 Tables. *Nature*, 12, 156(3974), pp. 783-784.
- Barnard, G. A., 1947. SIGNIFICANCE TESTS FOR 2×2 TABLES. *Biometrika*, Volume 34, pp. 123-138.
- Barney, B., 2013. *Message Passing Interface (MPI)*. [Online]
Available at: <https://computing.llnl.gov/tutorials/mpi/>
- Barney, B., 2013. *OpenMP*. [Online]
Available at: <https://computing.llnl.gov/tutorials/openMP/>
- Bergmann, J. & Sams, A., 2012. *Flip Your Classroom: Reach Every Student in Every Class Every Day*. s.l.:International Society for Technology in Education.
- Boehm-Davis, D. A., Holt, R. W. & Schultz, A. C., 1992. The Role of Program Structure in Software Maintenance. *Int. J. Man-Mach. Stud.*, January, 36(1), pp. 21-63.
- Bonar, J. & Soloway, E., 1985. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Hum.-Comput. Interact.*, June, 1(2), pp. 133-161.
- Brought, G., Eby, L. M. & Wahls, T., 2008. The effects of pair-programming on individual programming skill. *SIGCSE Bull.*, mar, Volume 40, pp. 200--204.
- Brought, G., Wahls, T. & Eby, L. M., 2011. The Case for Pair Programming in the Computer Science Classroom. *Trans. Comput. Educ.*, feb, Volume 11, pp. 2:1--2:21.

- Brooks, R., 1983. Towards a Theory of the Comprehension of Computer Programs. *Int'l J. Man-Machine Studies*, Volume 18, pp. 543-554.
- Carver, J. C. et al., 2007. Increased Retention of Early Computer Science and Software Engineering Students Using Pair Programming. *Software Engineering Education and Training, Conference on*, Volume 0, pp. 115-122.
- Cassel, L. et al., 2008. *Current Curricula -- Association for Computing Machinery*. [Online] Available at: <http://www.acm.org/education/curricula/ComputerScience2008.pdf>
- Chase, W. C. & Simon, H. A., 1973. Perception in Chess. *Cognitive Psychology*, Volume 4, pp. 55-81.
- Conway, M. E., 1963. Design of a separable transition-diagram compiler. *Commun. ACM*, jul, Volume 6, pp. 396--408.
- Corritore, C. L. & Wiedenbeck, S., 1999. Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *Int. J. Hum.-Comput. Stud.*, January, 50(1), pp. 61-83.
- Corritore, C. L. & Wiedenbeck, S., 2001. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *Int. J. Hum.-Comput. Stud.*, January, 54(1), pp. 1--23.
- Crowston, K. & Kammerer, E. E., 1998. Coordination and collective mind in software requirements development. *IBM Systems Journal*, 37(2), pp. 227-245.
- Curtis, B. et al., 1986. Software psychology: The need for an interdisciplinary program. *Proceedings of the IEEE*, Volume 74, pp. 1092-1106.
- de Mour, A. L. & Ierusalimsky, R., 2004. *Revisiting Coroutines*, s.l.: s.n.
- Edwards, A. W. F., 1963. The Measure of Association in a 2×2 Table. *Journal of the Royal Statistical Society. Series A (General)*, 126(1), pp. 109-114.
- Eisenstadt, M., 1993. *Tales of Debugging from The Front Lines*. s.l., s.n., pp. 86-112.
- Ferguson, C. J., 2009. An effect size primer: A guide for clinicians and researchers.. *Professional Psychology: Research and Practice*, Volume 40, pp. 532--538.
- Fisher, R. A., 1922. On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society*, jan, Volume 85, pp. 87--94.
- Fitzgerald, S. et al., 2008. Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(24), pp. 93-116.
- Fix, V., Wiedenbeck, S. & Scholtz, J., 1993. *Mental representations of programs by novices and experts*. New York, NY, ACM, pp. 74-79.

Fleming, S. D. et al., 2008. *Refining Existing Theories of Program Comprehension During Maintenance for Concurrent Software*. Amsterdam, Netherlands, IEEE Computer Society, pp. 23--32.

Fleming, S. D. et al., 2008. *A study of student strategies for the corrective maintenance of concurrent software*. Leipzig, Germany, ACM, pp. 759-768.

Gilgun, J. F., 1992. Definitions, methodologies, and methods in qualitative family research. In: J. F. Gilgun, K. Daly & G. Handel, eds. *Qualitative methods in family research*. Thousand Oaks, CA, US: Sage Publications, Inc., pp. 22-39.

Gilmore, D. J. & Green, T. R. G., 1984. Comprehension and recall of miniature programs. *Int. J. Man-Machine Studies*, Volume 21, pp. 31-48.

Gilmore, D. J. & Green, T. R. G., 1988. Programming Plans and Programming Expertise. *THE QUARTERLY JOURNAL OF EXPERIMENTAL PSYCHOLOGY*, 40A(3), pp. 423-442.

Gligoric, M., Jagannath, V. & Marinov, D., 2010. *MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code*. s.l., s.n., pp. 55-64.

Gould, J. D., 1975. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, March, 7(2), pp. 151-182.

Gould, J. D. & Drongowski, P., 1974. An Exploratory Study of Computer Program Debugging. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, June, 16(3), pp. 258-277.

Gray, W. D. & Anderson, J. R., 1987. Change-Episodes in Coding: When and How Do Programmers Change Their Codes?. In: G. M. Olson, S. Sheppard & E. Soloway, eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corp., pp. 185--197.

Gugerty, L. & Olson, G. M., 1986. Comprehension differences in debugging by skilled and novice programmers. In: E. Soloway & S. Iyengar, eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishing Corp., pp. 13--27.

Guindon, R., 1990. Designing the design process: exploiting opportunistic thoughts. *Hum.-Comput. Interact.*, jun, Volume 5, pp. 305--344.

Guindon, R., 1990. Knowledge exploited by experts during software system design. *Int. J. Man-Machine Studies*, September, 33(3), pp. 279-304.

Guindon, R., Krasner, H. & Curtis, B., 1987. Breakdowns and Processes During the Early Activities of Software Design by Professional. In: G. M. Olson, S. Sheppard & E. Soloway, eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corp., pp. 65-82.

Hanks, B., McDowell, C., Draper, D. & Krnjajic, M., 2004. Program quality with pair programming in CS1. *SIGCSE Bull.*, jun, Volume 36, pp. 176--180.

- Hart, S. & Staveland, L. E., 1988. Development of NASA-TLX (Task Load Index): results of empirical and theoretical research. In: P. A. N. Hancock, ed. *Human Mental Workload*. s.l.:Amsterdam, North-Holland, pp. 139-183.
- Herbsleb, J. D. & Grinter, R. E., 1999. Architectures, Coordination, and Distance: Conway's Law and Beyond. *IEEE Software*, September, 16(5), pp. 663-70.
- Herlihy, M. & Moss, J. E., 1993. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, may, Volume 21, pp. 289--300.
- Hewitt, C., 2010. Actor Model of Computation. *Arxiv preprint arXiv10081459*, pp. 1-29.
- Hieb, R. & Dybvig, R. K., 1990. Continuations and concurrency. *SIGPLAN Not.*, feb, Volume 25, pp. 128--136.
- Hollan, J., Hutchins, E. & Kirsh, D., 2000. Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research. *ACM Transaction on Computer-Human Interaction*, Jun, 7(2), pp. 174-196.
- Holt, R. W. & Boehm-Davis, D. A., 1987. Mental Representations of Programs for Student and Professional Programmers. *Empirical studies of programmers: second workshop*, pp. 33--46.
- Jeffries, R., Turner, A., Polson, P. & Atwood, M., 1981. The process involved in designing software. In: s.l.:Lawrence Erlbaum, pp. 255--283.
- Kant, E., 1985. Understanding and Automating Algorithm Design. *IEEE Trans. Softw. Eng.*, nov, Volume 11, pp. 1361--1374.
- Katz, I. R. & Anderson, J. R., 1987-1988. Debugging: An Analysis of Bug-Location Strategies. *Human Computer Interaction*, Volume 3, pp. 351-399.
- Knuth, D. E., 1981. *Seminumerical Algorithms*. s.l.:Addison-Wesley.
- Ko, A., Aung, H. & Myers, B., 2005. *Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks*. s.l., s.n., pp. 126-135.
- Ko, A. J. et al., 2011. The state of the art in end-user software engineering. *ACM Comput. Surv.*, April, Volume 43, pp. 21:1--21:44.
- Ko, A. J., Myers, B. A., Coblenz, M. J. & Aung, H. H., 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.*, December, 32(12), pp. 971-987.
- Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, jul, Volume 21, pp. 558--565.
- Larson, J., 2009. Erlang for concurrent programming. *Commun. ACM*, mar, Volume 52, pp. 48--56.

- LaToza, T. D., Garlan, D., Herbsleb, J. D. & Myers, B. A., 2007. *Program comprehension as fact finding*. Dubrovnik, Croatia, ACM, pp. 361-370.
- LaToza, T. D. & Myers, B. A., 2010. *Developers ask reachability questions*. Cape Town, South Africa, ACM, pp. 185-194.
- LaToza, T. D., Venolia, G. & DeLine, R., 2006. *Maintaining mental models: a study of developer work habits*. Shanghai, China, ACM.
- Lawrance, J. et al., 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering*, Feb, 39(2), pp. 197-215.
- Leontjev, A., 1978. *Activity, Consciousness, and Personality*. s.l.:Prentice-Hall.
- Lesani, M. & Palsberg, J., 2011. Communicating memory transactions. *SIGPLAN Not.*, feb, Volume 46, pp. 157--168.
- Letovsky, S., 1986. Cognitive processes in program comprehension. In: E. Soloway & S. Iyengar, eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishing Corp., pp. 58--79.
- Letovsky, S., Pinto, J., Lampert, R. & Soloway, E., 1987. A Cognitive Analysis of a Code Inspection. In: G. M. Olson, S. Sheppard & E. Soloway, eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corp..
- Levesque, L. L., Wilson, J. M. & Wholey, D. R., 2001. Cognitive divergence and shared mental models in software development project teams. *Journal of Organizational Behavior*, March, 22(SI), pp. 135-144.
- Lindholm, T., Yellin, F., Bracha, G. & Buckley, A., 2013. *The Java® Virtual Machine Specification*. [Online] Available at: [The Java® Virtual Machine Specification](#)
- Littman, D. C., Pinto, J., Letovsky, S. & Soloway, E., 1987. Mental models and software maintenance. *J. Syst. Softw.*, December, 7(4), pp. 341--355.
- Marlin, C. D., 1980. *Coroutines: A Programming Methodology, a Language Design and an Implementation*. s.l.:Springer.
- Matthijssen, N. et al., 2010. *Connecting Traces: Understanding Client-Server Interactions in Ajax Applications*. s.l., s.n., pp. 216-225.
- McCauley, R. et al., 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2), pp. 67-92.
- McDowell, C., Hanks, B. & Werner, L., 2003. Experimenting with pair programming in the classroom. *SIGCSE Bull.*, jun, Volume 35, pp. 60--64.
- McDowell, C., Werner, L., Bullock, H. .. & Fernald, J., 2006. Pair programming improves student retention, confidence, and program quality. *Commun. ACM*, aug, Volume 49, pp. 90--95.

McDowell, C., Werner, L., Bullock, H. E. & Fernald, J., 2003. *The impact of pair programming on student performance, perception and persistence*. Washington, DC, USA, IEEE Computer Society, pp. 602--607.

Mckeithen, K. B. & Reitman, J. S., 1981. Knowledge Organization and Skill Differences in Computer Programs. *Cognitive Psychology*, Volume 13, pp. 307-325.

Mendes, E., Al-Fakhri, L. B. & Luxton-Reilly, A., 2005. Investigating pair-programming in a 2nd-year software development and design computer science course. *SIGCSE Bull.*, jun, Volume 37, pp. 296--300.

Mendes, E., Al-Fakhri, L. & Luxton-Reilly, A., 2006. A replicated experiment of pair-programming in a 2nd-year software development and design computer science course. *SIGCSE Bull.*, jun, Volume 38, pp. 108--112.

Meredith, A., 2009. *N2869 State of C++ Evolution (Post San Francisco 2008)*. [Online] Available at: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2869.html>

Moher, T. & Schneider, G. M., 1982. Methodology and experimental research in software engineering. *International Journal of Man-Machine Studies*, January, 16(1), pp. 65-87.

Nagappan, N. et al., 2003. Improving the CS1 experience with pair programming. *SIGCSE Bull.*, jan, Volume 35, pp. 359--362.

Nanja, M. & Cook, C. R., 1987. An Analysis of the On-Line Debugging Process. In: G. M. Olson, S. Sheppard & E. Soloway, eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corp., pp. 172-184.

Nistor, A. et al., 2012. *Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code*. s.l., s.n., pp. 727-737.

OHara, J., 2007. Toward a Commodity Enterprise Middleware. *Queue*, may, Volume 5, pp. 48--55.

Olszewski, M., Ansel, J. & Amarasinghe, S., 2009. Kendo: efficient deterministic multithreading in software. *SIGPLAN Not.*, mar, Volume 44, pp. 97--108.

Oracle, 2009. *Lesson: Concurrency (The Java Tutorials > Essential Classes)*. [Online] Available at: <http://docs.oracle.com/javase/tutorial/essential/concurrency/>

Oracle, 2013. *The Java® Virtual Machine Specification*. [Online] Available at: <http://docs.oracle.com/javase/specs/jvms/se7/html/index.html>

Parnin, C. & Rugaber, S., 2011. Resumption strategies for interrupted programming tasks. *Software Quality Control*, March, 19(1), pp. 5-34.

Parnin, C. & Rugaber, S., 2012. *Programmer Information Needs After Memory Failure*. Passan, Germany, IEEE Computer Society, pp. 123-132.

- Pauli, W. & Soffa, M. L., 1980. Coroutine behaviour and implementation. *Software: Practice and Experience*, Volume 10, pp. 189--204.
- Pea, R. D., 1986. Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1), pp. 25-36.
- Pennington, N., 1987. Comprehension Strategies in Programming. In: G. M. Olson, S. Sheppard & E. Soloway, eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corp., pp. 100-113.
- Pennington, N., 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, Volume 19, pp. 295 - 341.
- Pennington, N., Lee, A. Y. & Rehder, B., 1995. Cognitive Activities and Levels of Abstraction in Procedural and Object-Oriented Design. *Human-Computer Interaction*, 10(2-3), pp. 171-226.
- Perkins, D. N. & Martin, F., 1986. Fragile knowledge and neglected strategies in novice programmers. In: E. Soloway & I. Sitharama, eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishing Corp., pp. 213--229.
- Pirolli, P. & Card, S., 1999. Information Foraging. *Psychology Review*, 106(4), pp. 643-675.
- Porter, A. A., Siy, H. P. & Votta, L. G., 1997. *Understanding the effects of developer activities on inspection interval*. Boston, MA, USA, ACM, pp. 128-138.
- Prasad, S. K. et al., 2012. *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing – Core Topics for Undergraduates*. [Online] Available at: <http://www.cs.gsu.edu/~tcpp/curriculum/sites/default/files/NSF-TCPP-curriculum-version1.pdf>
- Preston, D., 2006. Using collaborative learning research to enhance pair programming pedagogy. *SIGITE Newsl.*, jan, Volume 3, pp. 16--21.
- Ramalingam, V. & Wiedenbeck, S., 1997. *An empirical study of novice program comprehension in the imperative and object-oriented styles*. s.l., ACM, pp. 124--139.
- Rigby, P. C., German, D. M. & Storey, M.-A. D., 2008. *Open Source Software Peer Review Practices: A Case Study of the Apache Server*. Leipzig, Germany, ACM, pp. 541-550.
- Rist, R. S., 1986. Plans in programming: definition, demonstration, and development. In: E. Soloway & S. Iyengar, eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishing Corp., pp. 28--47.
- Rist, R. S., 1989. Schema Creation in Programming. *Cognitive Science*, September, 13(3), pp. 389-414.
- Rist, R. S., 1990. Variability in program design: the interaction of process with knowledge. *Int. J. Man-Machine Studies*, Volume 33, pp. 305-322.

Rist, R. S., 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human Computer Interaction*, Volume 6, pp. 1-46.

Robillard, M. P., Coelho, W. & Murphy, G. C., 2004. How Effective Developers Investigate Source Code: An Exploratory Study. *IEEE Trans. Softw. Eng.*, December, 30(12), pp. 889-903.

Robins, A., Rountree, J. & Rountree, N., 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), pp. 137-172.

Romero, P., du Boulay, B., Cox, R. & Lutz, R., 2003. *Java debugging strategies in multi-representational environments*. s.l., s.n., pp. 421-435.

Romero, P., Lutz, R., Cox, R. & du Boulay, B., 2002. *Co-ordination of multiple external representations during Java program debugging*. s.l., s.n., pp. 207-214.

Sahami, M. et al., 2013. *Computer Science Curricula 2013*. [Online] Available at: <http://ai.stanford.edu/users/sahami/CS2013/ironman-draft/cs2013-ironman-v1.0.pdf>

Salleh, N., Mendes, E. & Grundy, J., 2011. Empirical Studies of Pair Programming for CS/SE Teaching in Higher Education: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, July-Aug, 37(4), pp. 509 -- 525.

Sax, L. J., n.d. *Examining the Underrepresentation of Women in STEM Fields: Early Findings from the Field of Computer Science*. s.l.:s.n.

Schneiderman, B., 1977. Measuring computer program quality and comprehension. *Int. J. Man-Machine Studies*, Volume 9, pp. 465-478.

Schneiderman, B., 1986. Empirical studies of programmers: The territory, paths, and destinations, Keynote address for workshop. In: E. Soloway & R. Iyengar, eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishers, pp. 1-12.

Scholtz, J. & Wiedenbeck, S., 1992. The role of planning in learning a new programming language. *Int. J. Man-Machine Studies*, August, 37(2), pp. 191-214.

Scholtz, J. & Wiedenbeck, S., 1993. Using unfamiliar programming languages: the effects on expertise. *Interacting with Computers*, March, 5(1), pp. 13-30.

Sheil, B. A., 1981. The Psychological Study of Programming. *ACM Comput. Surv.*, March, 13(1), pp. 101--120.

Shneiderman, B., 1976. Exploratory experiments in programmer behavior. *International Journal of Parallel Programming*, 5(2), pp. 123-143.

Shneiderman, B. & Mayer, R., 1979. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *Int'l J. Computer and Information Sciences*, 8(3), pp. 219-238.

Sillito, J., Murphy, G. C. & De Volder, K., 2006. *Questions programmers ask during software evolution tasks*. Portland, OR, ACM, pp. 23-34.

Sillito, J., Murphy, G. C. & De Volder, K., 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng.*, July, 34(4), pp. 434--451.

Sillito, J., Volder, K. D., Fisher, B. & Murphy, G. C., 2005. *Managing software change tasks: An exploratory study*. Noosa Heads, Australia, IEEE Computer Society, pp. 23-32.

Simon, H. A., 1973. The Structure of Ill Structured Problems. *Artif. Intell.*, pp. 181-201.

Soloway, E., Adelson, B. & Ehrlich, K., 1988. Knowledge and Processes in the Comprehension of Computer Programs. In: M. Chi, R. Glaser & M. Farr, eds. *The Nature of Expertise*. Hillsdale, NJ: A. Lawrence Erlbaum Associates, pp. 129-152.

Soloway, E. & Ehrlich, K., 1984. Empirical Studies of Programming Knowledge. *Software Engineering, IEEE Transactions on*, Volume SE-10, pp. 595-609.

Storey, M.-A. D., 2005. *Theories, Methods and Tools in Program Comprehension: Past, Present and Future*. St. Louis, MI, IEEE Computer Society, pp. 181-191.

Strauss, A. & Juliet, C., 1994. *Handbook of qualitative research*. s.l.:Sage Publications.

Suchman, L., 1987. *Plans and Situated Actions: The Problem of Human-Machine Communication*. s.l.:Cambridge University Press.

Taylor, S. J. & Bogdan, R., 1984. *Introduction to Qualitative Research Methods: The Search for Meaning*. s.l.:John Wiley & Sons Inc..

Tew, A. E. & Guzdial, M., 2011. *The FCS1: a language independent assessment of CS1 knowledge*. s.l., ACM, pp. 111--116.

Thomas, G. & James, D., 2006. Reinventing grounded theory: some questions about theory, ground and discovery. *British Educational Research Journal*, 32(6), pp. 767-795.

Trujillo-Ortiz, A., Hernandez-Walls, R., Castro-Perez, A. & Rodriguez-Cardozo, L., 2004. [Online] Available at: <http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=6198>

van Rossum, G. & Eby, P. J., 2005. [Online] Available at: <http://www.python.org/dev/peps/pep-0342/>

Vessey, I., 1985. Expertise in debugging computer programs: A process analysis. *Int. J. Man-Machine Studies*, Volume 23, pp. 459-494.

Vessey, I., 1986. Expertise in debugging computer programs: an analysis of the content of verbal protocols. *IEEE Trans. Syst. Man Cybern.*, September, 16(5), pp. 621--637.

Visser, W., 1987. Strategies in Programming Programmable Controllers: A Field Study on a Professional Programmer. In: G. M. Olson, S. Sheppard & E. Soloway, eds. *Empirical Studies of Programmers: Second Workshop*. Norwood, NJ: Ablex Publishing Corp., pp. 217-230.

Visser, W., 1994. Organisation of design activities: opportunistic, with hierarchical episodes. *Interacting with Computers*, 6(3), pp. 235-238.

Weinberger, B. et al., 2010. *Google C++ Style Guide*. [Online] Available at: http://smacked.org/docs/cpp_style_google.pdf

Werner, L. L., Hanks, B. & McDowell, C., 2004. Pair-programming helps female computer science students. *J. Educ. Resour. Comput.*, mar. Volume 4.

Wiedenbeck, S., 1986. Processes in computer program comprehension. In: E. Soloway & S. Iyengar, eds. *Empirical Studies of Programmers*. Norwood, NJ: Ablex Publishing Corp., pp. 48-57.

Wiedenbeck, S., Fix, V. & Scholtz, J., 1993. Characteristics of the Mental Representations of Novice and Expert Programmers: An Empirical Study. *Int. J. Man-Mach. Stud.*, November, 39(5), pp. 793-812.

Williams, L. & Kessler, R. R., 2002. *Pair Programming Illuminated*. s.l.:Addison-Wesley Longman Publishing Co., Inc..

Williams, L., Kessler, R. R., Cunningham, W. & Jeffries, R., 2000. Strengthening the Case for Pair Programming. *IEEE Softw.*, jul, Volume 17, pp. 19--25.

Williams, L. et al., 2003. *Building Pair Programming Knowledge through a Family of Experiments*. Washington, DC, USA, IEEE Computer Society, pp. 143--152.

Ye, N. & Salvendy, G., 1996. An objective approach to exploring skill differences in strategies of computer program comprehension. *Behaviour & Information Technology*, May, 15(3), pp. 139-147.

CHAPTER 6.

APPENDIX

6.1 DATA AND ANALYSIS PROCESSES

Data analysis processes of case study described in section 3.3.

TABLE 48 BEHAVIORS, GOALS AND KNOWLEDGE REPOSITORY OF INTERMEDIATE SUBJECT

Behavior	Goal	Knowledge Solid or Fragile
RedCar.java (BlueCar.java)		
declare run() method	implement necessary interface method	run() method is required for a class implements Runnable interface
implement constructor method by assignment of passed in bridge object	implement constructor of a thread class according to monitor pattern	with monitor pattern, threads are constructed with corresponding monitor objects
add comment "red car is 1"	figure out whether distinction needs to be made between different thread types	with monitor pattern, threads distinguish themselves by calling different monitor methods
modify comment to "red car is 0"		
remove comment "red car is 0"		
call bridge.redEnter() and bridge.redExit() in run() method	implement run() method according to monitor model	
add Thread.sleep(500) between bridge.redEnter() and bridge.redExit() calls	add randomization to run() method	complex functionalities should be implemented after basic functionalities are guaranteed
modify Thread.sleep() with a random number generated		
correct syntax of random number generation		
re-organize line break	clean up code	good code style is easier for future comprehension
remove blank line		
remove unused class variable		
add class variable id	?	?
Bridge.java		
implement bridge class <detail history lost>	implement bridge class according to monitor pattern	monitor class should implement methods called correspondingly by different threads
correct syntax of if statement in statusCheck() method	Syntax modification <most probably due to compilation errors>	Java compilation errors and their corresponding meanings
add event value reset and corresponding comment in if statement in statusCheck() method	increase event count after every event and print out statusCheck messages according to number of event since last print of statusCheck messages	code pattern of continuously incrementing a counter to control the occurrence of some function based on a preset interval
add call of statusCheck() to redEnter(), blueEnter(), redExit(), blueExit() methods		
add else branch in statusCheck() method and increase event value inside it		
move event value increment to beginning of statusCheck() method		

to blueCars.add(id) <blueCars also records blue cars on bridge according to comments in code> remove blank lines		
remove import java.util.ArrayList statement remove redCheck and blueCheck variables remove redCheck and blueCheck related print statements add if (DEBUG) to print statements of redCar and blueCar variables		
RedCar.java (BlueCar.java)		
add pass of id argument to bridge.redEnter() and bridge.redExit() calls	use id to identify car threads calling monitor methods in bridge <this is unnecessary, since when monitor methods are called, an id may be assigned and returned to each thread through the return value of monitor methods>	monitor methods may also return values data flow of monitor pattern
Bridge.java		
Modify println of "Red car " + redWaiting.poll() + "enters bridge" to "Red car " + id + "enters bridge" Modify println of "Blue car " + blueWaiting.poll() + "enters bridge" to "Blue car " + id + "enters bridge"		
add while loop and thread waiting statements to blueEnter() method	implement detailed monitor pattern <this should be implemented much earlier without the struggles of monitoring cars with complex list structures	detailed monitor pattern for each method being called by threads the conditional check, execution and notify pattern
add while loop and thread wait statements to redEnter() method add call of notifyAll() to redEnter() and blueEnter() methods		
modify the condition of while loop surrounded wait statements in blueEnter() and redEnter() method		
exam time up		

*lighter gray cells are concurrency-related knowledge which IS captured by misconception hierarchy

*darker gray cells are other programming or non-programming knowledge

TABLE 49 BEHAVIORS, GOALS AND KNOWLEDGE REPOSITORY OF NOVICE SUBJECT

Behavior	Goal	Knowledge Solid or Fragile
BlueCar.java		
add blueEnter(), blueExit(), blueWaiting() method stubs	implement enter, exit and waiting in thread class	with monitor pattern, actions happen with monitor object instead of thread object
Bridge.java		
add class variables numBlueOn, numBlueUsed, numRedOn, numRedUsed, blueQueue, redQueue	implement setters and getters method for class variables <this is not quite necessary with relatively simple design structure of the single lane bridge problem and actually the subject declare these class variables without private keyword which implicitly make them protected and therefore, all other classes within the same package bridge could access the variables directly>	in object oriented design, setters and getters methods are for private variables to hide implementation details
add synchronized setters for numBlueUsed, numRedUsed variables: numBlueUsedPlus(), numBlueU		
add synchronized setters for numBlueOn, numRedOn, variables: numBlueOnPlus(), numBlueOnMinus(), numRedOnPlus(), numRedOnMinus()		
add synchronized getters for, numBlueUsed, numRedUsed variables which simply return values numBlueOn and numRedOn variables implement non-synchronized enterBridge(), exitBridge() methods with boolean variable color, integer variable id passed into them and a print statement print out "color + ' car ' + id + ' exits bridge'"		
add synchronized method addBlueCar(), addRedCar(), removeBlueCar(), removeRedCar() methods for variable blueQueue, redQueue		
BlueCar.java (RedCar.java)		
implement blueEnter() with calls of bridge.blueOnPlus() and bridge.enterBridge(color, id) implement blueExit() with calls of bridge.blueOnMinus(), bridge.blueUsed() and bridge.exitBridge()	implement enter/exit functions in thread class so that they change the state variables of bridge object	with monitor pattern, status of monitor object should be changed with monitor class methods
Bridge.java		
pass argument Thread t to addBlueCar() and addRedCar() methods return redQueue, blueQueue's size from addBlueCar() and addRedCar() methods	record "threads" on bridge	the inheritance relationship among super class and sub class in object oriented design
BlueCar.java (RedCar.java)		
add call of bridge.popBlueCar() after bridge.blueUsed() to blueExit() method add call of bridge.waitBridge(color, id, position) to blueWaiting()	implement exit/waiting functions in thread class so that they change the state variables of bridge object	with monitor pattern, status of monitor object should be changed with monitor class methods
pass the return value of bridge.addBlueCar(Thread.currentThread()) as position to call of bridge.waitBridge(color, id, position)		
add package bridge	add package information <probably due to a failed compilation with other helping codes defined in bridge package>	Java compilation errors and their corresponding meanings
Bridge.java		
add package bridge	add package information <probably due to a failed compilation with other helping codes defined in bridge package>	Java compilation errors and their corresponding meanings

	package>	
add method waitDiff(), calculate totalBlue as the sum of numBlueOn and numBlueUsed, totalRed as the sum of numRedOn and numRedUsed, write if statement to check whether the absolute difference of totalBlue and totalRed is greater than class variable waitDiff, if so, call wait() in waitBridge() method	implement thread waiting condition <subject mis-understands the specification of waitDiff, it should be only compared with the absolute difference between numRedOn and numBlueOn according to current code setting> <subject mis-implements waiting in a separate monitor class method>	natural language meaning of specification with the monitor pattern, blocking conditions and blocking statements should be implemented in the synchronized class methods that modify monitor states
add implementation of previous if statement to call wait()		
add notifyAll() to the end of exitBridge() method	implement notifyAll() <subject again mis-implement notifying in a separate monitor class method>	with the monitor pattern, notifying should happen after the variables recording monitor object state have been changed
implement constructor by setting passed in values as initial values of waitDiff and checkFreq variables	implement constructor	initialize class variables according to values passed into constructor
implement statusCheck() method with just two required print outs	initial implementation of statusCheck() method <subject probably do not quite understand how event intervals decides the print outs of status as defined in specification> <subject may also lack of the knowledge about how to check intervals>	natural language meaning of specification code pattern of continuously incrementing a counter to control the occurrence of some function based on a preset interval
BlueCar.java (RedCar.java)		
make call of statusCheck() method in blueEnter(), blueExit() and blueWait() methods	call status check in corresponding functions <subjects mis-placed action functions as mentioned before> <subject probably decide to delay the detail implementation of status check but first just confirm it prints out something when called>	general strategy of implementing functions general knowledge of function related information (caller-callee relations, etc.)
Bridge.java		
declare an integer class variable statusChecker	<subject probably realizes the detailed implementation of statusCheck() method needs some counter variable>	code pattern of continuously incrementing a counter to control the occurrence of some function based on a preset interval
add synchronized keyword to enterBridge(), exitBridge() and waitBridge() methods	<subject probably realizes that these methods also access class variables and should be synchronized>	mechanism of race condition mechanism of synchronized keyword
add if statement to statusCheck() method so that only when statusChecker%checkFreq==0, print outs	implement some detail of statusCheck() method <this implementation is wrong since statusChecker is never increased and the if statement is always true with statusChecker's initial value of 0>	code pattern of continuously incrementing a counter to control the occurrence of some function based on a preset interval
add else return directly to statusCheck()		

implement isAllExit method, return true when sum of numRedUsed and numBlueUsed does not equal to the passed in total value and false otherwise	implement isAllExit() method <subject either mis-understands the functionality of this method, which is to check whether all cars sent by thread generator finish using the bridge or mis-implements the if condition>	natural language meaning programming language translation of natural language conditions and corresponding returns
BlueCar.java (RedCar.java)		
implement run() method with calls of blueWaiting(), blueEnter(), blueExit() methods	implement run() method <subject's local implementation of waiting, enter, exit methods are not quite necessary considering that most of them have just one-two lines of codes>	purpose of organizing local functions and call hierarchy
Bridge.java		
modify generic data structure blueQueue, redQueue to take BlueCar, RedCar instead of Thread	<probably due to a compilation error>	the inheritance relationship among super class and sub class in object oriented design
BlueCar.java (RedCar.java)		
modify to pass this instead of Thread.currentThread() to call of briedg.addBlueCar()	<tag-along to previous change>	
remove the initialization to 0 from the declaration of id and position class variables	?	variable declaration, initialization
Bridge.java		
modify setter methods blueOnPlus() and redOnPlus() to increase both numBlueOn, numBlueUsed variables and numRedOn, numRedUsed variables	modify setter, getter methods <subject mistakenly update two different variables in one setter method>	in object oriented design, setters and getters methods are for private variables to hide implementation details
BlueCar.java (RedCar.java)		
wrap all method calls in run() method with a while(true) loop	add iteration to run() method <this is not necessary since the helping code generates car threads to simulate multiple cars, each thread object in this system emulates just one car that never comes back>	thread may be defined to have recurring actions or not initialization of thread object
Bridge.java		
change notifyAll() to notify() in exitBridge() method	?	functionality of and differences between notify and notifyAll statements
BlueCar.java (RedCar.java)		
import java.util.Random initialize a random generator in run() method use random generator to generate a random waitTime, bridgeTime call Thread.currentThread() and Thread.sleep(waitTime) between redWaiting and redEnter() call Thread.currentThread() and Thread.sleep(bridgeTime) between redEnter() and redExit() change bridgeTime to a fixed number	add randomization to implementation of run() method	complex functionalities should be implemented after basic functionalities are guaranteed
change comment "true=red – false=red" to "true=red – false=blue" in RedCar.java only	correct comment contradiction modify variable value according to comment	programming language translation of natural language conditions and corresponding returns
set variable color to false in RedCar.java	<subject makes incorrect assignment	

	to color variable>	
modify range of random number waitTime	?	?
Bridge.java		
add boolean class variable carOn and comment “True=red – False=blue” change notify() to notifyAll() in exitBridge() method declare and implement an onBridge() method to set carOn variable according to numBlueOn and numRedOn’s values	implement onBridge() method to check the type of car on bridge change notify to notifyAll <subject does not need to declare a class variable for the purpose of implement an onBridge() method>	functionality of and differences between notify and notifyAll statements
modify onBridge() method to directly return a boolean value instead of setting value of carOn variable	modify onBridge() method	code pattern of a function that checks value of some variables code pattern of using existed class variable to return some other value
remove class variable carOn add a local boolean variable carOn to onBriedg() method modify onBridge() method to set the value of local carOn variable and then return its value		
BlueCar.java		
comment out the use of random sleep in run() methods	<subject probably realizes that randomization causes difficulty to test>	complex functionalities should be implemented after basic functionalities are guaranteed
RedCar.java		
modify run() method to call redEnter() first, then check whether return value of bridge.onBridge() equals false, if so, call bridge.waitDiff(), then redEnter() again	coordinate thread waiting through the use of onBridge() method in thread class	with monitor pattern, thread’s activity are coordinated by methods defined in monitor class
change previous check condition to whether return value of bridge.onBridge() equals true		
comment out and then remove the use of random sleep in run() methods	<subject probably realizes that randomization causes difficulty to test>	complex functionalities should be implemented after basic functionalities are guaranteed
Bridge.java		
remove all methods except constructor and isAllExit() remove implementation of isAllExit() method remove all class variables except waitDiff, checkFreq, numBlueOn, numBlueUsed, numRedOn, numRedUsed, which are all integers	give up previous design	monitor pattern
rename numRedUsed to numRedOff, numBlueUsed to numBlueOff	rename variable	variables name should reflect their functionality
add method declaration of synchronized blueEnter(), blueExit(), redEnter(), redExit() implement redEnter() method with first a while loop check whether numBlueOn is greater 0 and call of wait() inside loop, then an increase of numRedOn and finally a return of numRedOn value	implement monitor class methods according to monitor pattern <subject recall to follow monitor pattern but still have some difficulties such as figuring out the condition of waiting block>	monitor pattern
implement redExit() method with a while loop check whether numRedOff not equals to positionVal passed into the method and call of wait() inside loop		programming language translation of natural language conditions
modify the while loop condition in redEnter() to check whether numBlueOn is greater than 0 or absolute difference between numRedOn and		

numBlueOn is greater than waitDiff		
modify the while loop condition in redExit to check whether numRedOff no equals to positionVal-1		
implement blueEnter(), blueExit() methods according to corresponding red methods add notify() to the end of all enter exit methods		
pass BlueCar object to blueEnter() and blueExit() methods	modify argument passing	data flow in monitor pattern
implement isAllExit() method, return true if the passed in total does not equal to the sum of numRedOn and numBlueOn	implement isAllExit() method <subject takes time to figure out comparison should be between total and the sum of numRedOff and numBlueOff>	natural language meaning of specification programming language translation of natural language conditions
modify the condition in isAllExit to the sum of numRedOff and numBlueOff		
BlueCar.java (RedCar.java)		
restore to the given skeleton version		
implement run() method by first call bridge.blueEnter(this), record return value of this call in a variable order and then call bridge.blueExit(order, this)	implement thread class according to monitor pattern	monitor pattern
add package bridge to RedCar.java		
Bridge.java		
add package bridge	fix compilation errors	Java compilation error and its meaning
add try/catch block to all wait() statements		
add print statement about "enter bridge" before wait() statement in while loop of redEnter() method	add print statement <subject is actually try to figure out the thread behavior with monitor class methods on whether a car (thread) entered the bridge before the call of wait()>	control flow mechanism of monitor pattern
modify previous print statement to be about "arrives at waiting position"		
remove passing a RedCar object to redEnter() method	modify passing arguments to monitor class methods	data flow mechanism of monitor pattern
add a print statement about "enter bridge" after increasing numRedOn in redEnter() method	add print statement	control flow mechanism of monitor pattern
BlueCar.java (RedCar.java)		
wrap all method calls in run() method with a while(true) loop	add iteration to run() method <this is not necessary as mentioned before>	thread may be defined to have recurring actions or not initialization of thread object
remove passing a BlueCar object to the call of bridge.blueEnter() in run() method	modify passing arguments to monitor class methods	data flow mechanism of monitor pattern
Bridge.java		
remove passing a BlueCar object to blueEnter(), blueExit() methods	modify passing arguments to monitor class methods	data flow mechanism of monitor pattern
remove passing a RedCar object to redExit() method		
add print statement "Red car enter bridge" in blueEnter() method		
add print statement "red car exits bridge" in redExit() method add print statement "blue car exits bridge" in blueExit() method	add print statement <subject add some wrong statements>	programming language translation of natural language conditions
restore the previous version with synchronized getters, setters	give up monitor model and restore the version with synchronized getters, setters	general implementation procedure monitor pattern

BlueCar.java (RedCar.java)		
restore the previous version that work with synchronized getters, setters in Bridge.java	give up monitor model and restore the version with synchronized getters, setters	general implementation procedure
		monitor pattern
remove the first call of redEnter()in run() method in RedCar.java	?	?
remove statements on random sleep remove call of bridge.waitDiff() in run() method in RedCar.java		
remove call of bridge.onBridge() in run() method in RedCar.java		
Bridge.java		
remove declaration and implementation of onBridge() method	?	?
exam time up		

*lighter gray cells are concurrency-related knowledge which IS captured by misconception hierarchy

*darker gray cells are other programming or non-programming knowledge

6.2 PROGRAMS USED IN THESIS WORK

```

import os, shutil, sys, getopt, time, re

def sema_copy(src_dir, dst_dir):
    for top, dirs, files in os.walk(src_dir):
        for nm in files:
            filename = os.path.join(top, nm)
            filetime = time.gmtime(os.path.getmtime(filename))
            if filetime[0] > 2012:
                f = open(filename, 'rU')
                content = f.read()
                # Java concurrency-related files
                if 'Runnable' in content or 'Thread' in content or 'synchronized' in
content:
                    cn_match = re.search(r'class\s(?P<classname>[0-9a-zA-Z./_]+)', content)
                    if cn_match:
                        javaname = cn_match.group('classname') + '.java' + ('_').join(map(str,
filetime[0:5]))
                        shutil.copy2(filename, os.path.join(dst_dir, javaname))
                    # Scala concurrency-related files
                    elif 'Actor' in content:
                        cn_match = re.search(r'object\s(?P<objectname>[0-9a-zA-Z./_]+)', content)
                        if cn_match:
                            scalaname = cn_match.group('objectname') + '.scala' +
('_').join(map(str, filetime[0:5]))
                            shutil.copy2(filename, os.path.join(dst_dir, scalaname))
                    # Python concurrency-related files
                    elif 'yield' in content or 'def' in content:
                        cn_match = re.search(r'usage="\s"\s"(?P<name>[0-9a-zA-Z./_]+)', content)
                        if cn_match:
                            pythonname = cn_match.group('name') + '.py' + ('_').join(map(str,
filetime[0:5]))
                            shutil.copy2(filename, os.path.join(dst_dir, pythonname))
                        else:
                            newname = ('_').join(map(str, filetime[0:5]))
                            shutil.copy2(filename, os.path.join(dst_dir, newname))

def parse_arguments(args):
    try:
        (opts, others) = getopt.getopt(args, '', ["src=", "dst="])
    except getopt.GetoptError:
        print 'Invalid arguments.'

    for opt, val in opts:
        if opt in ('--src'):
            src = val
        elif opt in ('--dst'):
            dst = val
        else:
            print 'Invalid option: %s=%s' %(opt, val)

    return (src, dst)

def main():
    (src, dst) = parse_arguments(sys.argv[1:])
    sema_copy(src, dst)

```

FIGURE 25 PYTHON HISTORY FILE ORGANIZER FOR STUDYING CODE HISTORY

```

import time, os, getopt, sys

def diff_com(src_dir, file_name, start_date, end_date):
    filelist = []
    for top, dirs, files in os.walk(src_dir):
        for nm in files:
            filename = os.path.join(top, nm)
            filetime = time.gmtime(os.path.getmtime(filename))
            name_match = file_name+'_' in filename
            year_match = filetime[0] >= int(start_date[0]) and filetime[0] <=
int(end_date[0])
            month_match = filetime[1] >= int(start_date[1]) and filetime[1] <=
int(end_date[1])
            day_match = filetime[2] >= int(start_date[2]) and filetime[2] <=
int(end_date[2])
            if name_match and year_match and month_match and day_match:
                filelist.append((filetime, filename))

    filelist.sort()
    prev = filelist.pop(0)
    for file in filelist:
        print 'diff -wic %s %s' %(prev[1], file[1])
        prev = filelist.pop(0)

def parse_arguments(args):
    try:
        (opts, others) = getopt.getopt(args, '', ["src=", "name=", "start=", "end="])
    except getopt.GetoptError:
        print 'Invalid arguments.'

    for opt, val in opts:
        if opt in ('--src'):
            src = val
        elif opt in ('--name'):
            name = val
        elif opt in ('--start'):
            start = val.split('-')
        elif opt in ('--end'):
            end = val.split('-')
        else:
            print 'Invalid option: %s=%s' %(opt, val)

    return (src, name, start, end)

def main():
    (src, name, start, end) = parse_arguments(sys.argv[1:])
    diff_com(src, name, start, end)

```

FIGURE 26 PYTHON CODE HISTORY GENERATOR FOR STUDYING CODE HISTORY

```

import sys, re

def parse_line(line):
    l = line.strip().split()
    name_num_info = l[0]
    conf_info = l[-1]
    cate_info = l[-2]
    msg_info = l[1:-2]

    m = re.search(r'(?P<filename>[0-9a-z./_]+):(?P<linenum>[\d]+):', name_num_info,
re.I)
    if m:
        filename = m.group('filename')
        linenum = m.group('linenum')

        m_lab = re.search(r'(.*)lab(?P<lab>[\d]+)(.*)', filename, re.I)
        if m_lab:
            lab = int(m_lab.group('lab'))

            message = (' ').join(msg_info)

            m_cate = re.search(r'\[(?P<category>.+)\]', cate_info, re.I)
            category = m_cate.group('category')

            m_conf = re.search(r'\[(?P<confidence>[0-5])\]', conf_info, re.I)
            confidence = m_conf.group('confidence')

            print '%s %s %s %s %s %s' %(lab, confidence, category, linenum, message,
filename)
# m = re.search(r'(?P<filename>[0-9a-z./_]+):(?P<linenum>[\d]+):\s:(?P<message>.+)\s\[ (?P<category>.+)\]\s\[ (?P<confidence>[0-5])\]', line.strip(), re.I)

def parse_file(filename):
    f = open(filename, 'rU')
    for line in f:
        parse_line(line)

def main():
    print 'lab confidence category linenum message filename'
    args = sys.argv[1:]
    parse_file(args[0])

```

FIGURE 27 PYTHON LINT OUTPUT PARSER FOR STUDYING CODE HEALTH

```

import sys, getopt

def execute_query(filenamees, count, labs, confs, cates):
    print 'Query: labs = %s, confidences = %s, categories = %s' %(labs, confs, cates)
    if count:
        count_list = []
        file_list = []
        for filename in filenamees:
            occurrence = 0
            f = open(filename, 'rU')
            for line in f:
                l = line.split('\t')
                if l[0] in labs and l[1] in confs:
                    for cate in cates:
                        if cate in l[2]:
                            if l[4] not in file_list:
                                file_list.append(l[4])
                                occurrence += 1
            if len(file_list) == 0:
                count_list.append((filename, 0))
            else:
                count_list.append((filename, occurrence/len(file_list)))
            file_list = []
        print count_list
    else:
        for filename in filenamees:
            f = open(filename, 'rU')
            summary = filename + ':\n'
            for line in f:
                l = line.split()
                if l[0] in labs and l[1] in confs:
                    for cate in cates:
                        if cate in l[2]:
                            summary += line + '\n'
            print summary

def parse_arguments(args):
    try:
        (opts, filename) = getopt.getopt(args, '', ["count", "lab=", "confidence=",
"category="])
    except getopt.GetoptError:
        print 'Invalid arguments.'

    for opt, val in opts:
        if opt in ('--count'):
            count = True
        elif opt in ('--lab'):
            labs = val.split(',')
        elif opt in ('--confidence'):
            confs = val.split(',')
        elif opt in ('--category'):
            cates = val.split(',')
        else:
            print 'Invalid option: %s=%s' %(opt, val)
    return (filename, count, labs, confs, cates)

def main():
    (filenamees, count, labs, confs, cates) = parse_arguments(sys.argv[1:])
    execute_query(filenamees, count, labs, confs, cates)

```

FIGURE 28 PYTHON LINT QUERY EXECUTOR FOR STUDYING CODE HEALTH

6.3 MATERIALS FROM SPRING 2010 WORK

Demographic Survey

1. Choose your gender
 - a) male
 - b) female
2. Choose your current grade
 - a) Freshman
 - b) Sophomore
 - c) Junior
 - d) Senior
 - e) Graduate
3. Check courses you have taken or been enrolled in to date from the following list.
 - ☐ CSCI 1100-1100L Introduction to Personal Computing
 - ☐ CSCI 1130 Hands-On Programming for Beginners
 - ☐ CSCI 1210 Computer Modeling and Science
 - ☐ CSCI 1301-1301L Introduction to Computing and Programming
 - ☐ CSCI 1302 Software Development
 - ☐ CSCI 1303H Programming and Software Development (Honors)
 - ☐ CSCI 1710-1710L Introduction to Computer Science and Computer Systems
 - ☐ CSCI 1730 Systems Programming
 - ☐ CSCI 1900 Computer Science Special Topic
 - ☐ CSCI 2150-2150L Introduction to Computational Science
 - ☐ CSCI(MATH) 2610 Discrete Mathematics for Computer Science
 - ☐ CSCI 2670 Introduction to Theory of Computing
 - ☐ CSCI 2720 Data Structures
 - ☐ CSCI 3030 Computing, Ethics, and Society
 - ☐ CSCI 4050/6050 Software Engineering
 - ☐ CSCI 4070/6070 Introduction to Game Programming
 - ☐ CSCI 4140/6140 Numerical Methods and Computing
 - ☐ CSCI 4150/6150 Numerical Simulations in Science and Engineering
 - ☐ CSCI 4210/6210 Simulation and Modeling
 - ☐ CSCI 4250/6250 Computer Security
 - ☐ CSCI 4300 Web Programming
 - ☐ CSCI 4330/6330 Artificial Intelligence and the Web
 - ☐ CSCI 4350/6350 Global Information Systems
 - ☐ CSCI 4370/6370 Database Management
 - ☐ CSCI 4470/6470 Algorithms
 - ☐ CSCI 4490/6490 Algorithms for Computational Biology
 - ☐ CSCI 4500/6500 Programming Languages
 - ☐ CSCI 4520/6520 Functional Programming
 - ☐ CSCI(ARTI) 4530/6530 Introduction to Robotics
 - ☐ CSCI(ARTI) 4540/6540 Symbolic Programming
 - ☐ CSCI(PHIL) 4550/6550 Artificial Intelligence
 - ☐ CSCI 4560/6560 Evolutionary Computation and Its Applications
 - ☐ CSCI 4570/6570 Compilers
 - ☐ CSCI(MATH)(PHYS) 4612/6612 Introduction to Quantum Computation

- ☐ (CSCI)MATH 4630/6630 Mathematical Analysis of Computer Algorithms
- ☐ (CSCI)MATH 4670/6670 Combinatorics
- ☐ (CSCI)MATH 4690/6690 Graph Theory
- ☐ CSCI 4720 Computer Architecture and Organization
- ☐ CSCI 4730/6730 Operating Systems
- ☐ CSCI 4740/6740 Real-Time Systems
- ☐ CSCI 4750/6750 VLSI System Design
- ☐ CSCI 4760/6760 Computer Networks
- ☐ CSCI 4770/6770 Ubiquitous Computing
- ☐ CSCI 4780/6780 Distributed Computing Systems
- ☐ CSCI 4800/6800 Human-Computer Interaction
- ☐ CSCI 4810/6810 Computer Graphics
- ☐ CSCI 4830/6830 Virtual Reality
- ☐ CSCI 4850/6850 Biomedical Image Analysis
- ☐ CSCI 4900/6900 Special Topics in Computer Science
- ☐ (CSCI)ENGR 4922 Computer Systems Engineering Design Project
- ☐ CSCI 4950/6950 Directed Study in Computer Science
- ☐ CSCI 5007/7007 Internship in Computer Science Business/Industry
- ☐ CSCI 5080/7080-5080L/7080L Personal Computer System Administration
- ☐ CSCI 5310/7310 Web Composing and Scripting
- ☐ CSCI 6610 Automata and Formal Languages
- ☐ CSCI 6720 Computer Systems Architecture
- ☐ (CSCI)ENGR 6922 Computer Systems Engineering Design Project
- ☐ CSCI 7000 Master's Research
- ☐ CSCI 7005 Graduate Student Seminar
- ☐ CSCI 7010 Computer Programming
- ☐ CSCI 7100 Technical Report
- ☐ CSCI 7300 Master's Thesis
- ☐ CSCI 8050 Knowledge-Based Systems
- ☐ CSCI 8060 Advanced Software Engineering
- ☐ CSCI 8140 Parallel Processing and Computational Science
- ☐ CSCI 8150 Advanced Numerical Methods and Scientific Computing
- ☐ CSCI 8220 Parallel and Distributed Simulation Systems
- ☐ CSCI 8250 Advanced Network and Security Systems
- ☐ CSCI 8350 Enterprise Integration
- ☐ CSCI 8351 Enterprise Integration Issues
- ☐ CSCI 8370 Advanced Database Systems
- ☐ CSCI 8380 Advanced Topics in Information Systems
- ☐ CSCI 8470 Advanced Algorithms
- ☐ CSCI(LING) 8570 Natural Language Processing Techniques
- ☐ CSCI 8610 Topics in Theoretical Computer Science
- ☐ CSCI(MATH)(PHYS) 8612 Topics in Quantum Computation
- ☐ CSCI(PHIL) 8650 Logic and Logic Programming
- ☐ CSCI 8710 Computer System Performance Evaluation
- ☐ CSCI 8720 Advanced Computer Architecture
- ☐ CSCI 8730 Software Systems for Parallel and Distributed Computing
- ☐ CSCI 8740 Advanced Topics in Real-Time Systems
- ☐ CSCI 8750 Advanced VLSI Systems Design

- ☐ CSCI 8770 Computer-Aided Design
 - ☐ CSCI 8780 Advanced Topics in Distributed Systems
 - ☐ CSCI 8810 Image Processing and Computer Graphics
 - ☐ CSCI 8820 Computer Vision and Pattern Recognition
 - ☐ CSCI 8850 Advanced Biomedical Image Analysis
 - ☐ CSCI(ENGR) 8940 Computational Intelligence
 - ☐ CSCI(ARTI) 8950 Machine Learning
 - ☐ CSCI 8990 Research Seminar
 - ☐ CSCI 9000 Doctoral Research
 - ☐ CSCI 9005 Doctoral Graduate Student Seminar
 - ☐ CSCI 9300 Doctoral Dissertation
 - ☐ Others, please specify
-
-

4. Please list up to 3 programming languages that you are most familiar with. For each language, please say whether you are a novice, intermediate or expert.

-

5. One estimation of programming experience is based on the total number of lines of code ever written by a programmer. The DSI LOC (Delivered Source Instructions – Lines of Code) specification uses the following guidelines:

- Only source lines that are delivered as part of the product are included -- test drivers and other support software is excluded
- Source lines are created by the project staff -- code created by applications generators is excluded
- One instruction is one line of code or card image
- Declarations are counted as instructions
- Comments are not counted as instructions

Based on these guidelines how would you classify your programming experience?

- a) less than 1,000 LOC
 - b) more than 1,000 LOC, less than 10,000 LOC
 - c) more than 10,000 LOC, less than 100,000 LOC
 - d) more than 100,000 LOC
6. How long have you been programming?
- a) less than 1 year
 - b) 1 – 3 years
 - c) 4 – 5 years
 - d) 6 – 10 years
 - e) more than 10 years
7. Do you have any working experience in industry (internships are not counted)?
- a) Yes
 - b) No, go to question 10
8. How long did you work in industry?
- a) less than 3 years
 - b) 3 – 5 years

- c) 6 – 10 years
 - d) more than 10 years
9. What field have you worked in?
- a) Research and Development
 - b) Sales and Marketing
 - c) Consultant
 - d) System Maintenance and Customer Services
 - e) Others, please specify_____
10. Do you have any experience in developing multi-threaded programs (Have you dealt with thread synchronization, shared objects, deadlock, etc. in developing that software/program)?
- a) Yes
 - b) No, go to question 17
11. How many multi-thread program have you developed or helped to develop?
- a) less than 3
 - b) 3 – 5
 - c) 6 – 10
 - d) more than 10
12. How would you describe the total size of the multi-threaded programs you participated in developing according to DSI LOC specification described in question 6?
- a) less than 500 LOC
 - b) more than 500 LOC, less than 5,000 LOC
 - c) more than 5,000 LOC, less than 10,000 LOC
 - d) more than 10,000 LOC
13. What kind of concurrency scenarios have you dealt with in developing multi-threaded programs? Check all applicable ones.
- a) Shared objects
 - b) Conditional thread synchronization
 - c) Deadlock
 - d) Fairness and thread scheduling
14. List the programming packages you've used in developing multi-threaded programs.
- _____
- _____
- _____
15. Do you have experience in developing Object-Oriented multi-thread software/program?
- a) Yes
 - b) No, go to question 17
16. Do you have experience in monitor programming?
- a) Yes
 - b) No
17. Do you have any experience in using modeling languages?
- a) Yes
 - b) No, go to question 23
18. List the modeling languages you've used.
- _____
- _____
- _____
19. Do you have any experience in modeling concurrent software?
- a) Yes

b) No, go to question 23

20. What modeling languages or notations have you used in modeling concurrent software?

21. Which modeling language or notation do you find most helpful for understanding the behavior of concurrent software?

22. Which modeling language or notation do you find most helpful for implementing and maintaining concurrent software?

23. How much do you know about modeling concurrent software?

- a) no idea
- b) very little
- c) some
- d) quite a lot

Thanks!

FIGURE 29 DEMOGRAPHIC SURVEYS FOR SPRING 2010 STUDY

Concurrency:

http://www.cs.uga.edu/~eileen/Concurrency_tutorials/Concurrency/Concurrency_Quiz/Concurrency_quiz.html

UML:

http://www.cs.uga.edu/~eileen/Concurrency_tutorials/Modeling/UML_Modeling_quiz/UML_Modeling_quiz.html

Implementation:

http://www.cs.uga.edu/~eileen/Concurrency_tutorials/Implementation/Implementation_quiz/Implementation_quiz.html

State Diagram:

http://www.cs.uga.edu/~eileen/Concurrency_tutorials/StateDiagrams/StateDiagram_quiz/StateDiagram_quiz.html

Sequence Diagram:

http://www.cs.uga.edu/~eileen/Concurrency_tutorials/SequenceDiagrams/Sequence_quiz/Sequence_quiz.html

Modeling Concurrency:

http://www.cs.uga.edu/~eileen/Concurrency_tutorials/ModelingConcurrency/ModelingConcurrency/Model_Concurrency.html

FIGURE 30 MULTI-MEDIA TUTORIALS FOR SPRING 2010 STUDY

Please answer the questions in order. You may refer to the answers of previous questions, **but are not allowed to modify the previous answers.**

I. Single-Lane bridge problem

The Single-Lane Bridge is a typical problem in the study of concurrent system. A bridge over a river is wide enough to permit only a single lane of traffic. That is, the bridge permits only one-way traffic at any one time. To simplify this problem, we will define the cars that move from left to right as red cars and those that move from right to left as blue cars. To avoid a safety violation, only one kind of car is allowed to be on the bridge at a time.

The bridge may be empty or occupied. If occupied, it may contain red cars or may contain blue cars. The bridge will never contain both red and blue cars at the same time. The bridge is initially empty.

When a car arrives, the bridge may be empty or occupied. If the bridge is empty, the car will enter the bridge. If the bridge is occupied, it might be occupied by cars of the same color or by cars of the different color. If the bridge is occupied by a car or cars of the same color as the arriving car, the arriving car may enter the bridge. If the bridge is occupied by a car or cars of a different color from the arriving car, the arriving car must wait.

Cars exit the bridge in the order in which they entered. The leading car may exit the bridge at any time.

In the implementation of the Single-Lane Bridge, each color of car is implemented as a thread, and the shared bridge object is implemented as a monitor. Two condition variables **okToEnter** and **okToExit** are associated with this monitor. The basic function for entering and exiting the bridge are *redEnter()*, *redExit()*, *blueEnter()* and *blueExit()*. The cars invoke *redEnter()* or *blueEnter()* to occupy the bridge and invoke *redExit()* or *blueExit()* to leave it.

Based on the above information, please answer the following questions.

1. Suppose that only two threads exist in the system: **redCar1** and **redCar2**. Suppose further that **redCar1** has invoked the *redEnter()* method, and has returned. A context switch occurs and the **redCar2** thread starts to run.

Which of the following event sequences could happen next?

Circle YES if the sequence is possible; otherwise, circle NO.

Then please provide a brief explanation of your reasoning.

- (a) **redCar2** invokes *redEnter()*, then returns.

YES NO

Explanation:

- (b) **redCar2** invokes *redEnter()*, then blocks on the monitor lock.

YES NO

Explanation:

- (c) **redCar2** invokes *redEnter()*, returns, and then invokes *redExit()*, then returns.

YES NO

Explanation:

- (d) **redCar2** invokes *redEnter()* and is context-switched out before the call returns.

YES NO

Explanation:

2. Suppose the only two threads exist in the system: **redCar1** and **redCar2**. Suppose further that **redCar1** has invoked the *redEnter()* method, but has not returned. A context switch occurs and the **redCar2** thread starts to run.

Which of the following event sequences could happen next?

Circle YES if the sequence is possible; otherwise, circle NO.

Then please provide a brief explanation of your reasoning.

- (a) **redCar2** invokes *redEnter()*, then returns.

YES NO

Explanation:

- (b) **redCar2** invokes *redEnter()*, then blocks on the monitor lock.

YES NO

Explanation:

- (c) **redCar2** invokes *redEnter()*, returns, and then invokes *redExit()*, then returns.

YES NO

Explanation:

- (d) **redCar2** invokes *redEnter()* and is context-switched out before the call returns.

YES NO

Explanation:

3. Suppose that only three threads exist in the system: **redCar1**, **redCar2**, and **blueCar1**. Suppose further that **redCar1** is running, that it has invoked the *redEnter()* method, and that the *redEnter()* method has returned. A context switch occurs and **blueCar1** thread begins to run and invokes the *blueEnter()* method. The *blueEnter()* method has not returned.

Which of the following event sequences could happen next?

Circle YES if the sequence is possible; otherwise, circle NO.

Then please provide a brief explanation of your reasoning.

- (a) The *blueEnter()* returns, followed by *blueExit()*.

YES NO

Explanation:

- (b) Another context switch happens, **redCar1** is in the ready state.

YES NO

Explanation:

- (c) Another context switch occurs and the **redCar2** thread begins to run. The **redCar2** invokes *redEnter()*, and returns, and then invokes *redExit()*, and returns.

YES NO

Explanation:

- (d) Another context switch occurs and the **redCar2** thread begins to run. The **redCar2** invokes *redEnter()* and then blocks on the monitor lock.

YES NO

Explanation:

4. Suppose that only three threads exist in the system: **redCar1**, **redCar2**, and **blueCar1**. Suppose further that **redCar1** is running and has just invoked the *redEnter()* method and the *redEnter()* method has returned. A context switch occurs and the **redCar2** thread begins running and invokes the *redEnter()* method. **redCar2**'s invocation of the *redEnter()* method has not returned.

Which of the following event sequences could happen next?

Circle YES if the sequence is possible; otherwise, circle NO.

Then please provide a brief explanation of your reasoning.

- (a) **redCar2**'s invocation of *redEnter()* returns. **redCar2** then invokes *redExit()* and this invocation returns.

YES NO

Explanation:

- (b) **redCar2**'s invocation of *redEnter()* returns. **redCar2** then invokes *redExit()* and blocks on the **okToExit** condition variable.

YES NO

Explanation:

- (c) **redCar2**'s invocation of *redEnter()* returns. **redCar2** then invokes *redExit()* and blocks on the monitor lock.

YES NO

Explanation:

- (d) A context switch occurs, and the **redCar1** thread begins to run. **redCar1** then invokes *redExit()* and this invocation returns.

YES NO

Explanation:

- (e) A context switch occurs, the **redCar1** thread begins to run. **redCar1** then invokes the *redExit()* method and blocks on the monitor lock.

YES NO

Explanation:

- (f) A context switch occurs, the **redCar1** thread begins to run. **redCar1** then invokes *redExit()* and blocks on the **okToExit** condition variable.

YES NO

Explanation:

- (g) A context switch occurs, the **blueCar1** thread begins to run, invokes the *blueEnter()* method and returns.

YES NO

Explanation:

(h) A context switch occurs, the **blueCar1** thread begins to run, invokes the *blueEnter()* method and blocks on the monitor lock.

YES NO

Explanation:

(i) A context switch occurs, and the **blueCar1** thread begins to run, and invokes the *blueEnter()* method and then blocks on the **okToEnter** condition variable.

YES NO

Explanation:

5. Suppose that only three threads exist in the system: **redCar1**, **redCar2**, and **blueCar1**. Suppose further that **redCar1** is running and has just invoked the *redEnter()* method and the *redEnter()* method has not returned. A context switch occurs and **redCar2** thread begins running and invokes the *redEnter()* method. **redCar2**'s invocation of the *redEnter()* method has not yet returned.

Which of the following event sequences could happen next?

Circle YES if the sequence is possible; otherwise, circle NO.

Then please provide a brief explanation of your reasoning.

(a) **redCar2**'s invocation of *redEnter()* returns. **redCar2** then invokes *redExit()* and this invocation returns.

YES NO

Explanation:

(b) **redCar2**'s invocation of *redEnter()* returns. **redCar2** then invokes *redExit()* and blocks on the **okToExit** condition variable.

YES NO

Explanation:

(c) **redCar2**'s invocation of *redEnter()* returns. **redCar2** then invokes *redExit()* and blocks on the monitor lock.

YES NO

Explanation:

(d) A context switch occurs, the **redCar1** thread begins to run, and its invocation of the *redEnter()* method returns. **redCar1** then invokes *redExit()* and this invocation returns.

YES NO

Explanation:

(e) A context switch occurs, the **redCar1** thread begins to run, and its invocation of the *redEnter()* method returns. **redCar1** then invokes *redExit()* and blocks on the monitor lock.

YES NO

Explanation:

(f) A context switch occurs, the **redCar1** thread begins to run, and its invocation of the *redEnter()* method returns. **redCar1** then invokes *redExit()* and blocks on the **okToExit** condition variable.

YES NO

Explanation:

- (g) A context switch occurs, the **blueCar1** thread begins to run, invokes the *blueEnter()* method and returns.

YES NO

Explanation:

- (h) A context switch occurs, the **blueCar1** thread begins to run, invokes the *blueEnter()* method and blocks on the monitor lock.

YES NO

Explanation:

- (i) A context switch occurs, and the **blueCar1** thread begins to run, and invokes the *blueEnter()* method and then blocks on the **okToEnter** condition variable.

YES NO

Explanation:

6. Suppose that only three threads exist in the system: **redCar1**, **redCar2**, and **blueCar1**. Suppose that the following sequence of invocations have occurred with the listed results:

- a. **redCar1** invokes *redEnter()* and returns
- b. **redCar2** invokes *redEnter()* and returns
- c. **blueCar1** invokes *blueEnter()* and blocks on **okToEnter**

Which of the following event sequences could happen next?

Circle YES if the sequence is possible; otherwise, circle NO.

Then please provide a brief explanation of your reasoning.

- (a) **redCar2** invokes *redExit()* and this invocation returns.

YES NO

Explanation:

- (b) **redCar2** invokes *redExit()* and blocks on the monitor lock.

YES NO

Explanation:

- (c) **redCar2** invokes *redExit()* and blocks on **okToExit**.

YES NO

Explanation:

- (d) **redCar1** invokes *redExit()* and returns, **redCar2** invokes *redExit()* and returns, **blueCar1** remains blocked on the monitor lock.

YES NO

Explanation:

- (e) **redCar1** invokes *redExit()* and returns, **redCar2** invokes *redExit()* and returns, **blueCar1** remains blocked

on the **okToEnter** condition variable.

YES NO

Explanation:

(f) **redCar1** invokes *redExit()* and returns, **redCar2** invokes *redExit()* and returns, **blueCar1** is signaled and eventually returns from *blueEnter()*.

YES NO

Explanation:

(g) **redCar1** invokes *redExit()* and returns, **redCar2** invokes *redExit()* and returns, **blueCar1** invokes *blueExit()* and returns.

YES NO

Explanation:

7. Figure 1 contains a partial implementation of the Single-Lane Bridge problem. **Identify any elements of the implementation of the *redEnter()* method that are missing or incorrect, or write that the implementation is correct.** If any errors exist, write down the line number where the errors occur, explain their effects and show how to correct them.

```

-----
// global declaration
1  typedef struct {
2      pthread_mutex_t mut;
3      int redEntered, blueEntered, redExited, blueExited;
4      pthread_cond_t okToEnter, okToExit;
5  } SLB_LOCK_S;
6
7  int redEnter(SLB_LOCK_S *sLock) {
8
9      int redNum = 0;
10     if (NULL == sLock)
11         return -1
12
13     pthread_mutex_lock(sLock->mut);
14     while (((sLock->blueEntered) - (sLock->blueExited)) != 0)
15         pthread_cond_wait(sLock->okToEnter, sLock->mut);
16     redNum = sLock->redEnter++;
17
18     return redNum;
19 }
20 main() {
21     SLB_LOCK_S sLock;
22     sLock->redEntered = 0; sLock->blueEntered = 0;
23     sLock->redExited = 0; sLock->blueExited = 0;
24
25     // create redCar thread and blueCar thread
26     ... ...
27 }
-----

```

Figure 1: Implementation of the *redEnter()* method for the Red Car

8. In the space below, write the POSIX Thread code to implement the *redExit()* method.

II. Readers-Writers problem

The readers-writers problem is a classic synchronization problem in which two distinct classes of threads, readers and writers, share access to a database. Multiple reader threads can be present in the database simultaneously. However, the writer threads must have exclusive access. That is, no other writer thread, nor any reader thread, may be present in the database while a given writer thread is present. Note: the reader thread must call *startRead()* to enter the database and it must call *endRead()* to exit the database. Similarly, the writer thread must call *startWrite()* to enter the database and it must call *endWrite()* to exit the database. Assume that state variables *numReaders* and *numWriters* are used to keep track of the number of client processes currently in the database.

9. In the space below, write the POSIX Thread code to implement the *startRead()* and *endRead()* methods for the Readers and Writers problem.

10. Please use UML 2.0 notation to draw state diagrams of the shared database for the Readers-Writers problem.

For your references, we have included sample state diagrams for the Bounded Buffer example from the tutorial, figures 2-4 on the following pages.

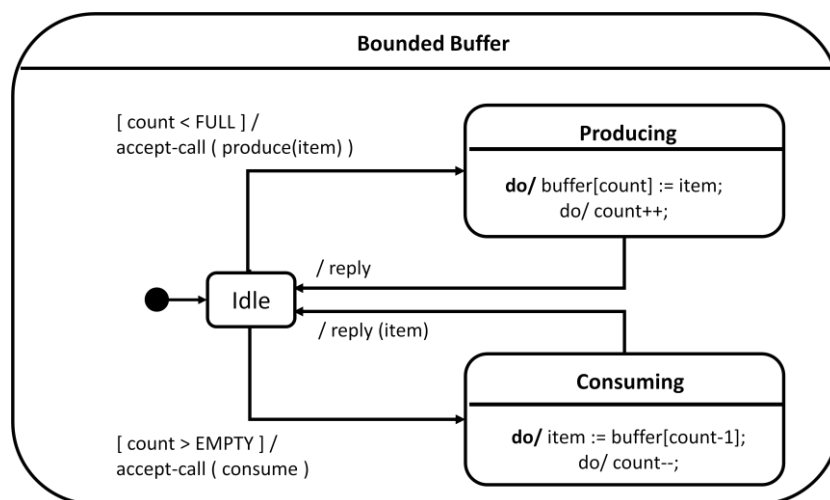


Figure 2

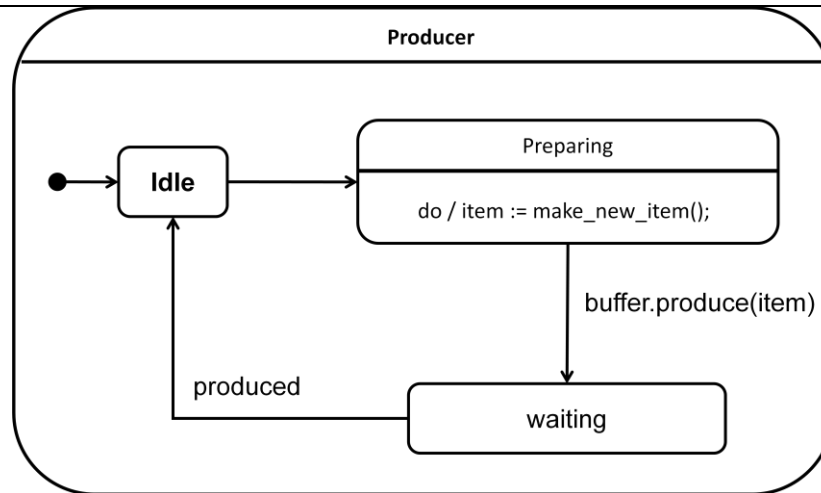


Figure 3

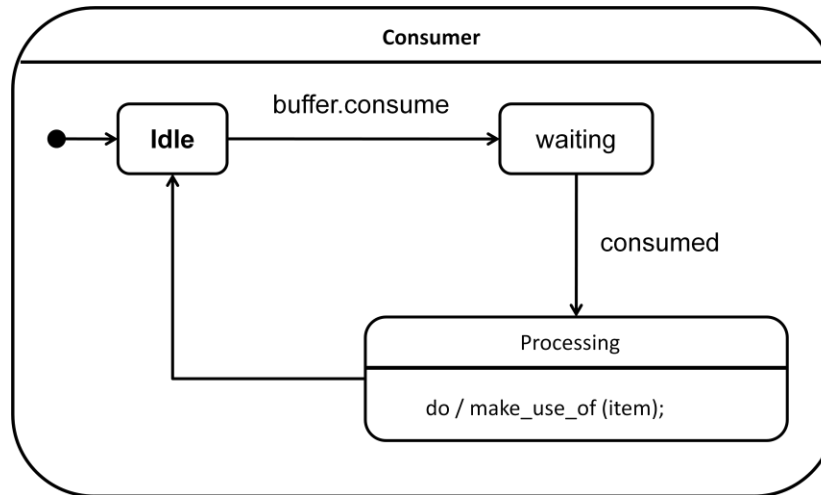


Figure 4

6.4 MATERIALS FROM SPRING 2012 WORK

Concurrency II

Threads & Conditional Synchronization

What we have learned so far...

- Threads (light) versus Processes (heavy)
- Java threads & C++ Pthread library
- Thread management
 - create, join, start
- Shared object
 - Race conditions
 - Atomic operations
 - Locks and synchronization
 - the synchronized keyword in java
 - the pthread_mutex_t type in c++

Other issues in concurrency?

- ▶ So, we guarantee the atomic operations on shared objects and avoid race conditions, any other issues??
- ▶ Think about a simple savings account:
 - A bank account object (shared object) records an amount of money left
 - This account object supports two operations, deposit and withdraw
 - No over-withdrawn is allowed
 - A number of customers (threads) share this account

Code Speaks

- ▶ Implement bank account program in Java:
 - makefile
 - Account.java
 - Customer.java
 - Bank.java (driver program)
- ▶ An Implementation on nuke...

Java Implementation v1

```
// Account.java
public class Account {
    private int amount;

    public Account(int amount){
        this.amount = amount;
    }

    public synchronized void deposit(int id, int amount){
        System.out.println(id + ": " + this.amount + " + " + amount + " = " +
            (this.amount+=amount));
    }

    public synchronized void withdraw(int id, int amount){
        System.out.println(id + ": try to " + this.amount + " - " + amount + " = " +
            (this.amount-=amount));
    }
}
```



Java Implementation v1 issue

```
// Account.java
public class Account {
    private int amount;

    public Account(int amount){
        this.amount = amount;
    }

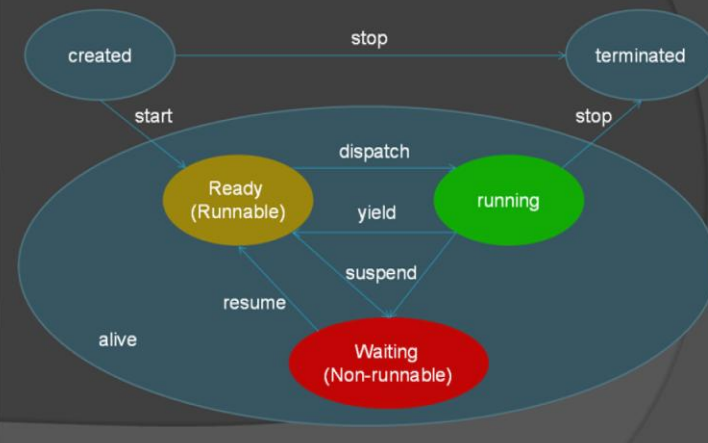
    public synchronized void deposit(int id, int amount){
        System.out.println(id + ": " + this.amount + " + " + amount + " = " +
            (this.amount+=amount));
    }

    public synchronized void withdraw(int id, int amount){
        System.out.println(id + ": try to " + this.amount + " - " + amount + " = " +
            (this.amount-=amount));
    }
}
```

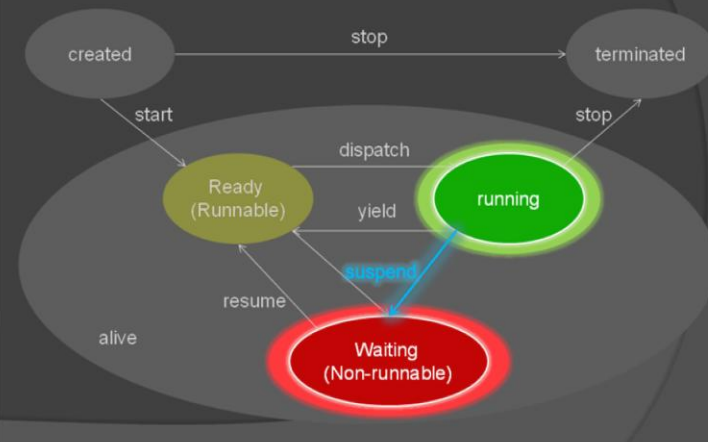


**Conditional
Synchronization**

Thread Life Cycle



Thread Life Cycle



Java Implementation v2

```
// Account.java
public class Account {
    private int amount;

    public Account(int amount){
        this.amount = amount;
    }

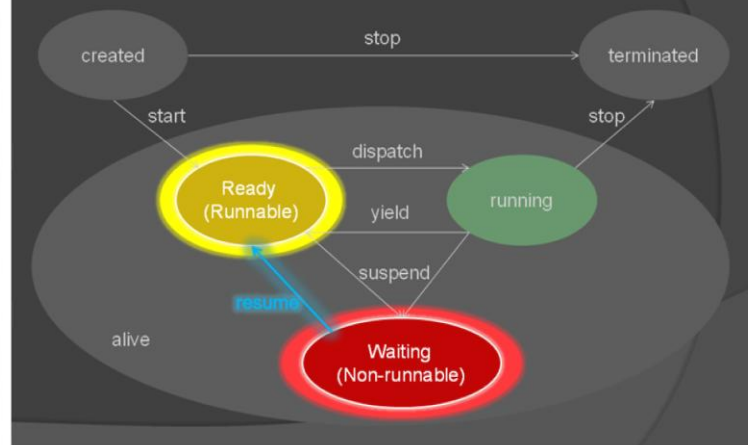
    public synchronized void deposit(int id, int amount){
        System.out.println(id + ": " + this.amount + " + " + amount + " = " +
            (this.amount+=amount));
    }

    public synchronized void withdraw(int id, int amount){
        if (this.amount < amount){
            System.out.println(id + ": try to " + this.amount + " - " + amount + " --> wait");
            try( wait(); ) catch (InterruptedException ie){}
        }
        System.out.println(id + ": try to " + this.amount + " - " + amount + " = " +
            (this.amount-=amount));
    }
}
```

A Run of Implementation v2

```
-bash-4.1$ make run
java Bank
2: 0 + 300 = 300
0: 300 + 100 = 400
6: 400 + 700 = 1100
9: try to 1100 - 900 = 200
1: try to 200 - 100 = 100
7: try to 100 - 700 --> wait
4: 100 + 500 = 600
3: try to 600 - 300 = 300
8: 300 + 900 = 1200
5: try to 1200 - 500 = 700
// program stuck here; main function doesn't return
```

Thread Life Cycle



Java Implementation v2 issue

```
// Account.java
public class Account {
    private int amount;

    public Account(int amount){
        this.amount = amount;
    }

    public synchronized void deposit(int id, int amount){
        System.out.println(id + ": " + this.amount + " + " + amount + " = " +
            (this.amount+=amount));
        // amount changed
    }

    public synchronized void withdraw(int id, int amount){
        if (this.amount < amount){
            System.out.println(id + ": try to " + this.amount + " - " + amount + " --> wait");
            try{ wait(); } catch (InterruptedException ie){}
        }
        System.out.println(id + ": try to " + this.amount + " - " + amount + " = " +
            (this.amount-=amount));
        // amount changed
    }
}
```

Java Implementation v3

```
// Account.java
public class Account {
    private int amount;

    public Account(int amount){
        this.amount = amount;
    }

    public synchronized void deposit(int id, int amount){
        System.out.println(id + ": " + this.amount + " + " + amount + " = " +
            (this.amount+=amount));
        notifyAll();
    }

    public synchronized void withdraw(int id, int amount){
        if (this.amount < amount){
            System.out.println(id + ": try to " + this.amount + " - " + amount + " --> wait");
            try{ wait(); } catch (InterruptedException ie){}
        }
        System.out.println(id + ": try to " + this.amount + " - " + amount + " = " +
            (this.amount-=amount));
        notifyAll();
    }
}
```

A Run of Implementation v3

```
-bash-4.1$ make run
java Bank
5: try to 0 - 500 --> wait
2: 0 + 300 = 300
5: try to 300 - 500 = -200
1: try to -200 - 100 --> wait
7: try to -200 - 700 --> wait
0: -200 + 100 = -100
7: try to -100 - 700 = -800
1: try to -800 - 100 = -900
4: -900 + 500 = -400
6: -400 + 700 = 300
3: try to 300 - 300 = 0
8: 0 + 900 = 900
9: try to 900 - 900 = 0
```

What
Happened
??!!

Look Closer

amount = 0

```
withdraw(id, amt){  
  if (amount < amt){  
    5: try to amount - amt --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

Look Closer

amount = 0

```
withdraw(5, amt){  
  if (amount < amt){  
    5: try to amount - amt --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (amount < amt){  
    5: try to amount - amt --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to amount - amt --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

```
deposit(id, amt){  
  2: amount + amt = sum;  
  notifyAll();  
}
```

Thread 2

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

```
deposit(2, amt){  
  2: amount + amt = sum;  
  notifyAll();  
}
```

Thread 2

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

```
deposit(2, 300){  
  2: amount + amt = sum;  
  notifyAll();  
}
```

Thread 2

Look Closer

amount = 0

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

```
deposit(2, 300){  
  2: 0 + 300 = 300;  
  notifyAll();  
}
```

Thread 2

Look Closer

amount = 300

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

```
deposit(2, 300){  
  2: 0 + 300 = 300;  
  notifyAll();  
}
```

Thread 2

Look Closer

amount = 300

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

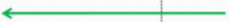
```
deposit(2, 300){  
  2: 0 + 300 = 300;  
  notifyAll();  
}
```

Thread 2

Look Closer

amount = 300

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```



Thread 5

Look Closer

amount = 300

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to amount - amt = diff;  
  notifyAll();  
}
```

Thread 5

*Resume
from where
it stopped!*

Look Closer

amount = 300

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to 300 - 500 = -200;  
  notifyAll();  
}
```

Thread 5

Look Closer

amount = 300

```
withdraw(5, 500){  
  if (0 < 500){  
    5: try to 0 - 500 --> wait;  
    wait();  
  }  
  5: try to 300 - 500 = -200;  
  notifyAll();  
}
```

Thread 5

At this point, we would like the thread to re-check whether it is runnable (whether the condition is satisfied)

Java Implementation v3 issue

```
// Account.java
public class Account {
    private int amount;

    public Account(int amount){
        this.amount = amount;
    }

    public synchronized void deposit(int id, int amount){
        System.out.println(id + ": " + this.amount + " + " + amount + " = " +
        (this.amount+=amount));
        notifyAll();
    }

    public synchronized void withdraw(int id, int amount){
        if(this.amount < amount){
            System.out.println(id + ": try to " + this.amount + " - " + amount + " --> wait");
            try{ wait(); } catch (InterruptedException ie){}
        }
        System.out.println(id + ": try to " + this.amount + " - " + amount + " = " +
        (this.amount-=amount));
        notifyAll();
    }
}
```

Java Implementation v4

```
// Account.java
public class Account {
    private int amount;

    public Account(int amount){
        this.amount = amount;
    }

    public synchronized void deposit(int id, int amount){
        System.out.println(id + ": " + this.amount + " + " + amount + " = " +
        (this.amount+=amount));
        notifyAll();
    }

    public synchronized void withdraw(int id, int amount){
        while(this.amount < amount){
            System.out.println(id + ": try to " + this.amount + " - " + amount + " --> wait");
            try{ wait(); } catch (InterruptedException ie){}
        }
        System.out.println(id + ": try to " + this.amount + " - " + amount + " = " +
        (this.amount-=amount));
        notifyAll();
    }
}
```


Run Java Implementation v4

- It seems OK...
- No more unreasonable printouts
- Is the program really correct?
- This is a research question
- Bugs of concurrent programs do NOT show up in all runs
- For this simple program, it IS correct



Implementation with Pthreads

- Implement bank account program in Java:
 - makefile
 - Account.java
 - Customer.java
 - Bank.java (driver program)



Implementation in C++

- ▶ **Implement bank account program in Java:**
 - makefile
 - Account.java
 - Customer.java
 - Bank.java (driver program)
- ▶ **Implement bank account program in C++:**
 - makefile
 - account.h, account.cpp
 - customer.h, customer.cpp
 - bank.cpp (driver program)

Pthread Conditional Variable

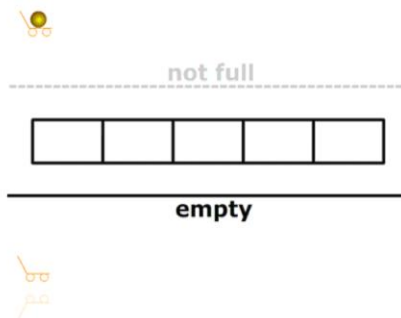
- ▶ **Type**
 - pthread_cond_t condv
- ▶ **Create and destroy conditional variable**
 - pthread_cond_create(&condv)
 - pthread_cond_destroy(&condv)
- ▶ **Wait on conditional variable**
 - pthread_cond_wait(&condv, &mutex);
- ▶ **A side by side implementation in nike...**

A More Complicated Program

▶ Bounded Buffer

- a buffer object (shared object) has limited number of slots to hold products
- producers (threads) produce products onto the buffer when there are still available empty slots
- consumers (threads) consume things from the buffer when there are still available things in buffer

Animation of Bounded Buffer



To Be Expected

- ▶ You will implement both Java and C++ versions of bounded buffer program in tomorrows lab
- ▶ Detailed specifications will be released tomorrow



FIGURE 32 LECTURE NOTES FOR SPRING 2012 STUDY

CSCI 1730 Lab/Project Specification (Week 2)

Programming Project #1

C++ Hands On with UNIX (Nike) Environment

Goals

In this project, you will use C++ to implement a command line based calculator program that runs in the UNIX environment. The goal of this lab is to allow you to practice basic C++ programming knowledge and to become familiar with the process of compiling and running a C++ program in UNIX.

Due Date

Jan 24, 2012 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project, and according to pair programming policies.

Project Description

Step 1:

You are given a skeleton source code file **calculator.cpp**. To complete the command-line calculator you are to implement the following functions, stubs for which are found in the file:

- *myAdd (operand1, operand2)*, which adds two operands;
- *mySubtract (operand1, operand2)*, which subtracts operand2 from operand1;
- *myMultiply (operand1, operand2)*, which multiplies two operands;
- *myDivide (operand1, operand2)*, which divides operand1 by operand2;
- *mySin (operand)*, which returns the sine of the operand (in degrees);
- *myCos (operand)*, which returns the cosine of the operand (in degrees);
- *myFib (operand)*, which returns the value of Fibonacci function with an input value of operand; e.g. myFib(1) = 1, myFib(2)=1, myFib(3)=2, myFib(4)=3, myFib(5)=5, etc.
- *main*, which continually displays a menu, accepts an integer to indicate the operation, requests the operand or operands needed to perform the operation, and displays a result until the user selects the integer corresponding to the "quit" operation

Use g++ to compile the program into the default output binary.

Here is a sample run of the program:

```
-bash-3.2$ ./a
Welcome to command calculator!
1: add
2: subtract
3: multiply
4: divide
5: sin
```

```

6: cos
7: Fibonacci
9: quit

1
Please enter operand1: 12
Please enter operand2: 34
The result is: 46
-----

1: add
2: subtract
3: multiply
4: divide
5: sin
6: cos
7: Fibonacci
9: quit

7
Please enter operand: 20
The result is: 6765
-----

1: add
2: subtract
3: multiply
4: divide
5: sin
6: cos
7: Fibonacci
9: quit

3
Please enter operand1: 25
Please enter operand2: 4.8
The result is: 120
-----

1: add
2: subtract
3: multiply
4: divide
5: sin
6: cos
7: Fibonacci
9: quit

9
-bash-3.2$

```

Step 2:

Create a **makefile** with three targets: compile, run and clean. The command “make compile” should compile the **calculator.cpp** program to create a **calc** executable. The command “make run” should execute the **calc** program. The “make clean” command should remove the **calc** file.

Step 3:

Break the **calculator.cpp** program into two files: **calculator.cpp** and **operations.cpp**. The **calculator.cpp** file should contain the main method and other utility methods that you define. The **operations.cpp** file should contain the methods for add, subtract, multiply, divide, sin, cos, and fib operations.

Complete the given header file **operations.h** and include it in **calculator.cpp**. Now modify your

makefile to compile the project (**calculator.cpp**, **operations.cpp**) and run the calc program.

Step 4:

Modify the *readOperand()* and *readOperation()* functions in your program to deal with possible input exceptions.

Step 5:

Write a **readme** file with brief instructions on how to compile and run your program as well as anything you would like TAs to note.

Use the `mkdir` command to create a new folder (named "lastname1_lastname2_lab02") in your local Nike account. Use the `cp` command to copy the **calculator.cpp**, **operations.cpp**, **operations.h**, **makefile** files and the **readme** to the newly created folder and submit the folder to cs1730. (submit cs1730 lastname1_lastname2_lab02)

Grading Rubric

Deliverables	Total	Points	Comments
Subjective Survey	5		
Source Code	45		
function add()	3		
function subtract()	3		
function multiply()	3		
function divide()	3		
function sin()	3		
function cos()	3		
function fib()	3		
function main()	5		
readme file	4		
makefile that successfully compiles linked header file	5		
exception handling	10		

CSCI 1730 Lab/Project Specification (Week 3)

Programming Project #2

C++ Arrays, Vectors & Control Structures Revision

Goal

In this project, you will use C++ to implement a command line matrix multiplication program to gain familiarity with basic C++ control structures, the array and vector data structures, as well as basic file reading /writing with C++.

Due Date

Jan 31, 2012 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

Project Description

Step 1: Implement and test matrix read and write methods

You are given **io.h** and **io.cpp**, which contains functions for reading a matrix from a file and for writing a matrix to a file. The **io.h** contains the function prototypes and should not be modified. The **io.cpp** file contains stubs for these functions. You are to replace these stubs with implementations.

- `vector<vector<double>> readMatrix(const char *filename)`, which reads and parses a text file into a two dimensional matrix
- `void writeMatrix(vector<vector<double>> const &matrix, const char *filename)`, which writes a two dimensional matrix into a file

Copy the makefile **makefile** and test files **matrix1**, **matrix2**, **matrix3**, **matrix_empty**, **matrix_single**, **matrix_normal** and **matrix_abnormal** to your working folder. Do NOT make any modifications to these files; do NOT rename them. Use the command “make testio” to compile the program **testio.cpp** (do NOT modify **testio.cpp**). Use the command “./testio” to run the program that tests the functions you will write in **io.cpp**. Make sure your program passes all four test cases specified there. The expected output is as follow:

```
-bash-3.2$ ./testio
testcase1: empty matrix passed
testcase2: single element matrix passed
testcase3: normal matrix passed
testcase4: abnormal matrix passed
all testcases passed
-bash-3.2$
```

Step 2: Implement and test matrix multiplication operation

You are given **op.cpp** which deals with the multiplication of two matrices. Implement the following methods, stubs for which are found in **op.cpp**. File **op.h** contains the prototypes for these methods, and should not be modified.

- `vector<vector<double> > multiplyMatrix(vector<vector<double> > const &matrix1, vector<vector<double> > const &matrix2)`, which multiplies two matrices
- `vector<vector<double> > transpose(vector<vector<double> > const &matrix)`, which transposes a matrix
- `double multiplyVector(vector<double> const &vector1, vector<double> const &vector2)`, a helper method that calculates the dot product of two one-dimensional vectors

To multiply two matrices, the matrices must be compatible. This means that the number of columns in the first matrix must be the same as the number of rows in the second matrix. In general, the result of attempting to multiply incompatible matrices is undefined. For purposes of this lab, please follow the rules below in your implementation of the *multiplyMatrix* method:

- multiplication that involves an empty matrix should result in an empty matrix;
- multiplication that involves two incompatible matrices (such as the first matrix with dimension 3*2 and the second matrix with dimension 4*3) should result a matrix with a single element 0;
- multiplication that involves two compatible matrices, such as the first matrix with dimension 3*2 and the second matrix with dimension 2*4, should result a matrix with dimension 3*4;

Here is a simple review of how to multiply two compatible matrices:

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \times \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} = \begin{bmatrix} ag + bj & ah + bk & ai + bl \\ cg + dj & ch + dk & ci + dl \\ eg + fj & eh + fk & ei + fl \end{bmatrix}$$

However, since matrices are stored as a vector of vectors (`vector<vector<double> >`) in our program, it is much easier to retrieve a line of the matrix than retrieve a column of the matrix. So, we could implement the above matrix multiplication using the transpose of the second matrix as illustrated below:

$$\begin{aligned} \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \times \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix} &= \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix} \times \begin{bmatrix} g & j \\ h & k \\ i & l \end{bmatrix}^T \\ &= \begin{bmatrix} [a \ b] \cdot [g \ j] & [a \ b] \cdot [h \ k] & [a \ b] \cdot [i \ l] \\ [c \ d] \cdot [g \ j] & [c \ d] \cdot [h \ k] & [c \ d] \cdot [i \ l] \\ [e \ f] \cdot [g \ j] & [e \ f] \cdot [h \ k] & [e \ f] \cdot [i \ l] \end{bmatrix} \\ &= \begin{bmatrix} ag + bj & ah + bk & ai + bl \\ cg + dj & ch + dk & ci + dl \\ eg + fj & eh + fk & ei + fl \end{bmatrix} \end{aligned}$$

Use command “make testop” to compile the program **testop.cpp**. Do NOT modify **testop.cpp** or **op.h**. Use command “./testop” to run the program that tests the functions you write in **op.cpp**. Make sure your program passes all six test cases specified there. The expected output is as follow:

```
-bash-3.2$ ./testop
testcase1: empty*empty passed
testcase2: empty*single passed
testcase3: empty*normal passed
testcase4: abnormal*normal (uncompatible) passed
testcase5: normal*normal (uncompatible) passed
testcase6: normal*normal (compatible) passed
all testcases passed
```

Step 3: Create driver program

Complete the *main* function in **main.cpp** so that the main program will take three arguments:

- filename of the first matrix to be read
- filename of the second matrix to be read
- filename of the result matrix to be written

The *main* function should deal with the problem of insufficient arguments and print out corresponding

hint messages for users. Use command “`make matrix`” to compile the main program and test it with your own test files.

Step 4: Documentation and Submission

Write a **readme** file with very brief instructions on how to compile and run your program as well as anything you would like the TAs to note.

Create a new folder (named “`lastname1_lastname2_lab03`”) in your local Nike account. Copy **main.cpp**, **io.h**, **io.cpp**, **op.h**, **op.cpp**, **makefile** and the **readme** to the newly created folder and submit the folder to cs1730. **Do not include any *.o or executable files.** (submit `lastname1_lastname2_lab03 cs1730`)

Step 5: (bonus)

Write new files **iosparse.h**, **iosparse.cpp**, **opsparses.h**, **opsparses.cpp** and use them in **main.cpp** to deal with multiplication of sparse matrices (matrices populated primarily with zeros) to boost the performance of your program when dealing with very big but sparse matrices. You could get the running time of a program with the “`time`” command in UNIX. Here is an example:

```
-bash-3.2$ time ls
lab01 lab02

real    0m0.003s
user    0m0.001s
sys     0m0.002s
-bash-3.2$
```

Feel free to use new data structures other than vectors if necessary. Test your program performance on matrices with dimensions more than 10^3 and report your program’s performances on different dimensions of sparse matrices in the **readme** file with a table.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Source Code	45		
function read_matrix()	10		
function write_matrix()	10		
function multiply_matrix	15		
function transpose()	5		
function multiply_vector	5		
function main()	5		
Bonus	10		

CSCI 1730 Lab/Project Specification (Week 4)

Programming Homework #1

Pointers and Its Related Messy Stuff in C++

Goal

In this homework, you will use C++ to implement several pieces of code that involve using pointers. This lab will help you to strengthen concepts of memory models you have learned/will learn in lectures and also familiarize you with using explicit pointers in C++. After this lab, you should be able to:

- Understand and use pointers and pointer operators
- Understand and use array and pointers, and understand the relationships between arrays and pointers
- Understand and use function pointers
- Understand and use the **const** modifier with pointers
- Understand and use pointers to manipulate strings

Due Date

Feb 14, 2012 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

Project Description**Step 1: Warm Up with Pointers of Basic Types and Operators**

In this first step, you will become familiar with pointers to basic C++ data types and some basic pointer operations (referencing, dereferencing, arithmetic operations):

1. Compile and run the following program:

```

1 #include <iostream>
2
3 int main(int argc, char** argv){
4
5     int integer1, *p1, **p2;
6
7     integer1 = 10;
8     p1 = &integer1;
9     p2 = &p1;
10
11     std::cout << "integer1=" << integer1 << std::endl;
12     std::cout << "p1=" << p1 << std::endl;
13     std::cout << "p2=" << p2 << std::endl;
14
15     return 0;
16 }

```

Suppose after the declaration of variables in line 5, we have the following symbol table and memory chunk:

Name	Type	Address
integer1	int	0x7fff8c1ff994
p1	int *	0x7fff8c1ff988
p2	int **	0x7fff8c1ff980

0x7fff8c1ff994	
0x7fff8c1ff988	
0x7fff8c1ff980	

- a) Draw three diagrams that reflect how memory changes after the execution of each line of code in lines 7, 8 and 9.
- b) If we substitute line 11-13 with following two statements:

```
(*p1)++;
std::cout << "integer1=" << *p1 << std::endl;
```

Then what will be the output of the program?

- c) Will the output of the program be same if we substitute above two lines with following two statements?

```
integer1++;
std::cout << "integer1=" << *p1 << std::endl;
```

- d) Will the output of the program be same if we substitute above two lines with following two statements?

```
*p1++;
std::cout << "integer1=" << integer1 << std::endl;
```

- e) Explain why above outputs are same or different from each other.

2. Consider the following program and answer the questions below.

```
1 #include <iostream>
2
3 int main(int argc, char** argv){
4
5     int *p1;
6     char *p2;
7
8     p2 = p1;
9
10    return 0;
11 }
```

What does variable p2 represent? Will this program successfully compile? Why or why not? (Try it.)

Step 2: Arrays and Pointers

In this second step, you are going to relate pointer representations with array representations.

3. Compile and run following program and answer the corresponding questions.

```

1 #include <iostream>
2
3 int main(int argc, char** argv){
4
5     int array_of_integer[10], *pointer_of_array, *pointer_of_first_element;
6
7     pointer_of_array = array_of_integer;
8     pointer_of_first_element = &array_of_integer[0];
9
10    for (int i=0; i<10; i++){
11        array_of_integer[i] = i;
12    }
13
14    for (int i=0; i<10; i++){
15        std::cout << *(pointer_of_first_element+i) << std::endl;
16    }
17
18    return 0;
19 }

```

- a) What does `array_of_integer` evaluate to? If you dereference it, what value do you get? What is the output of the program?
- b) Suppose we change lines 10-12 to following statements:

```

for (int i=0; i<10; i++){
    *(pointer_of_array+i) = i;
}

```

Will the output change? Why or why not?

- c) Suppose we change lines 14-16 of the original program to the following statements:

```

for (int i=0; i<10; i++){
    std::cout << *(pointer_of_array+i) << std::endl;
}

```

Will the output change? Why or why not?

- d) Suppose we change lines 10-12 of the original program to the following statements:

```

for (int i=0; i<10; i++){
    *(pointer_of_array++) = i;
}

```

Will the output change? Why or why not? Now, does the variable `pointer_of_array` have the same value as the variable `pointer_of_first_element`? Why or why not?

4. Consider the following code:

```

1#include <iostream>
2
3void increment(int i, int *pi, int &ri);
4
5int main(int argc, char** argv){
6
7    int x = 10;
8    int *px = &x;
9
10   int a[10] = {100, 100, 100, 100, 100, 100, 100, 100, 100, 100};
11   int b[5] = {1000, 1000, 1000, 1000, 1000};
12
13   int *p1 = a;
14   int *p2 = b;
15
16   int (*parray)[10] = &a;
17
18   int *ap1[3];
19   ap1[0] = &x;
20   ap1[1] = p1;
21   ap1[2] = b;
22
23   int (*ap2[2])[10];
24   ap2[0] = &a;
25   ap2[1] = parray;
26
27   return 0;
28 }

```

Fill out this memory model table:

Label of variable	Address of variable	Content of variable	Meaning of variable
x	addr_x	10	an integer
px	addr_px	addr_x	a pointer to an integer
a	addr_a		
a[0]	addr_a[0]		
a[1]	addr_a[1]		
a[2]	addr_a[2]		
a[3]	addr_a[3]		
a[4]	addr_a[4]		
a[5]	addr_a[5]		
a[6]	addr_a[6]		
a[7]	addr_a[7]		
a[8]	addr_a[8]		
a[9]	addr_a[9]		
b	addr_b		
b[0]	addr_b[0]		
b[1]	addr_b[1]		
b[2]	addr_b[2]		
b[3]	addr_b[3]		
b[4]	addr_b[4]		

p1	addr_p1		
p2	addr_p2		
parray	addr_parray		
ap1	addr_ap1		
ap1[0]	addr_ap1[0]		
ap1[1]	addr_ap1[1]		
ap1[2]	addr_ap1[2]		
ap2	addr_ap2		
ap2[0]	addr_ap2[0]		
ap2[1]	addr_ap2[1]		

a) Will the program successfully compile if we substitute line 16 of the program with following statement? Explain why or why not.

```
int (*parray)[10] = &b;
```

b) What if we substitute line 16 of the program with following statement? Explain.

```
int (*parray)[] = &b;
```

c) Will the program successfully compile if we substitute line 24 of the program with following statement? Explain why.

```
ap2[0] = &(*parray);
```

Step3: Functions and Pointers

In this part, you will get to play with pointers as function parameters and practice pass-by-reference. Also, you will become familiar with complicated mixed representation of pointers, arrays and functions.

5. What is the output of the following code? Why? Which variable in the code is “pass-by-reference”?

```

1 #include <iostream>
2
3 void increment(int i, int *pi, int &ri);
4
5 int main(int argc, char** argv){
6
7     int a = 1, b = 2, c = 3;
8
9     std::cout << a << " " << b << " " << c << std::endl;
10    increment(a, &b, c);
11    std::cout << a << " " << b << " " << c << std::endl;
12
13    return 0;
14 }
15
16 void increment(int i, int *pi, int &ri){
17     i = i+1;        *pi = *pi+1;        ri = ri+1;
18     std::cout << i << " " << *pi << " " << ri << std::endl;
19 }

```

6. Here is a golden *Right-Left* rule of how to read C++ declarations involving pointers, arrays and functions. Use this rule to interpret the meaning of the following variables in the table by saying x is something.

Start at the variable name (or innermost construct if no identifier is present). Look right without jumping over a right parenthesis; say what you see. Look left again without jumping over a parenthesis; say what you see. Jump out a level of parentheses if any. Look right; say what you see. Look left; say what you see. Continue in this manner until you say the variable type or return type.

Declaration	Meaning
int x;	x is an integer
int *x;	
char **x;	
int *x[5];	
int (*x)[5];	
int (*x[5])[5];	
int *(*x[5])[5];	
int x();	
int x(int);	
int *x();	
int *x(int *);	
int (*x)();	
int *(*x)(int *);	
int **(*x)(int **);	
int (*x[5])();	
int *(*x[5])();	
int (*(x()))[5];	
int *(*(*x()))[5];	

int *(*(*x[5]))())()	
int *(*(*x[10])(int &))[5];	

Step 4: The Const Modifier and Pointers

In this part, you will play with codes involving the **const** modifier with pointers.

7. Write and try to compile the following code:

```

1 #include <iostream>
2
3 void increment(int i, int *pi, int &ri);
4
5 int main(int argc, char** argv){
6
7     int x = 10;           // an integer x
8     x++;                  // x could be modified
9
10    const int c = 200;     // a constant integer c
11
12    const int *p1c = &c;   // a pointer pc points to a constant integer
13    p1c++;
14    (*p1c)++;
15
16    int const *p2c = &c;   // alternative expression
17    p2c++;
18    (*p2c)++;
19
20    char ch = 'a';         // a character ch
21    char * const cpc = &ch; // a constant pointer cpc points to a character
22    cpc++;
23    (*cpc)++;
24
25    return 0;
26 }

```

- a) What lines of the code cannot be successfully compiled? What are the error messages given by compiler? Explain why.

8. Explain how the following lines of code differ:

```

void function(data_type &parameter);
void function(data_type const &parameter);

```

When will the second expression become useful? Explain why.

Step5: Strings and Pointers

In this part, you will implement some functions from the `<cstring>` library to practice manipulating strings as pointers. Please read Section 21.8 in your textbook carefully before writing any code.

9. Implement your own versions of the following library functions defined in `<cstring>`:

a) `char *strcpy(char *s1, const char *s2);`

- b) `char *strncat(char *s1, const char *s2, size_t n);`
 c) `int strncmp(const char *s1, const char *s2, size_t n);`
 d) `size_t strlen(const char *s);`

Bonus:

- e) `char *strtok(char *s1, const char *s2);`

Write a header file named **mycstring.h** and finish your implementation in **mycstring.cpp**. Notice that you should implement the functions instead of using `<cstring>` library directly. Compile and test your implementations with **makefile** and **main.cpp** (do not modify them).

Step 6: Documentation and Submission

Write a **readme** file with anything you would like the TAs to note.

Create a new folder (named "lastname1_lastname2_lab04") in your local Odin account. Copy all **mycstring.h**, **mystring.cpp**, **makefile** and the **readme** to the newly created folder and submit the folder to cs1730. **Do not include any *.o or executable files.** (submit lastname1_lastname2_lab04 cs1730)

Submit a ***.pdf** file with all answers to homework questions for step 1-4.

You are not required to submit any code appeared in step 1-4.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Questions and Answers (Step 1-4)	32		
4 points per question Any severe mistake cause -1 Any minor mistake cause -.5			
Code Implementation (Step 5)	13		
a) strcpy function	2		
b) strncat function	4		
c) strncmp function	4		
d) strlen function	3		
Bonus (Step 5)	10		
e) strtok function	10		

Lab 05 Practice on GDB – A Command Line Debugger

Debugging crash1.cpp

- Use the given makefile and command "make crash1" to compile program. What is the command used by the system to compile the program? What does each argument in the command mean?
- Run the program, what is the error message?
- Use command "gdb" to start the debugger. Try "list", what is the message given by the debugger? Type command "file crash1" and then "list", now what you see?
- Use command "quit" to quit the debugger and then restart the debugger with command "gdb crash1". Try "list", now what you get? Try "list" again, what you see? Try "list 10",

what you see? Try to change the number with the list command, what you know now?

- Type command “run”, from the message given by debugger, what information do you get? Do you know which line of code fails now? If you do a little bit inference, could you know which call of the divint function fails? (The first or the second call?)
- Let’s kill the program with command “kill”. When prompted, please type “y”. Now we set a breakpoint at line 8 (which is the first line of main function) with command “break 8”. Now run the program again, notice that it stops before executing line 8? Try command “info breakpoints”, what information do you get?
- Use command “print x” and “print y” to print the value of variable x and y, what value do you get? Try command “next” to execute one line of code, now what are the values of x and y? Type “next” command again, notice that the debugger ignores all the lines in divint function and directly execute whole line 9 in main function?
- Now print value of x and y, then use “step” command and print value of x and y again. What are the values of these two variables? Use the “step” command again and notice that the debugger actually forward into the divint function? Notice the parameters of divint function (i and j) are having the values copied from x and y.
- Use command “set j = 1” to set the value of j. Now print the value of j again, notice that its value has been changed. Use the “step” command and see that the return statement where previously raises SIGFPE now could be executed normally. So could you guess why the program fails before?
- Use command “continue” to finish executing the program and notice that this time program exited normally. Search online a little bit about SIGFPE and make sure you know what this signal means.

Now, let’s draw some conclusion from our experience of debugging this crash1.cpp program:

1. To enable gdb debugging of a program, we have to use the -g flag during compilation.
2. To start gdb, we simply use “gdb”. We could load file with “file filename” command. We could also start gdb with a specific file with command “gdb filename”
3. To set a breakpoint, we use “break line_number”.
4. To retrieve the information of breakpoints, we use “info breakpoints”.
5. To list the source code of a program, we use “list” or “list line_number”.
6. To run a program, we use “run” and program will stop before the line with breakpoint or until it encounters some errors.
7. To execute the line and ignoring any function calls, we use “next”.
8. To execute the next step without ignoring function calls, we use “step”.
9. To set the value of a variable, we use “set variable_name = some_value”.
10. To kill a program running in debugger, we use “kill” and we could restart it by using “run” again.

Debugging crash2.cpp

- Compile and run the program outside of debugger, what kind of error you get? Now run the program inside the debugger, any more information you get?
- Use the strategies and commands we learned from debugging crash1.cpp, could you predict which call of setint function actually fails (the first or the second)?
- Let’s set a breakpoint at line 10. Now restart the program, when it stops at line 10, try to find out the following: 1) What is the value of integer1? 2) What is the address of integer1 (use “print &integer1”)? 3) What is the address of pointer1? 4) What is the value of pointer1? If you feel confused, draw a sketch of the memory down on a scratch paper as you getting values of above items.
- Now, let’s set the pointer to point to integer1 (use “set pointer1 = &integer1” or “set

pointer1 = 0x7fffffffe494"/use the actual address you get from debugger) and then step into the setint function. Notice that the value passed into the function now is actually the address of integer1. Step out of the setint function and before executing line 12 in main function, print out the value of what pointer1 points to (use "print *pointer1"). What is that value now?

- Let's put our hand even deeper into manipulating memories here. Say the address of integer1 is 0x7fffffffe494, use command "set *(int *)0x7fffffffe494 = 20". Now try to print the value of integer1, what is it? Use "next" and what is printed out?
- Continue and you should see program exited normally this time. So could you infer what the bug of the program is? (We actually talked about this during yesterday's lecture.)

Another conclusion time:

1. Besides basic commands, gdb actually allows you directly modify and examine the memories during execution of a program.
2. When a piece of memory is associated with a symbol, we could use "set symbol=value" to change the content of the memory. But even when no symbol is assigned to a chunk of memory, we could still specify the type of that memory and set the value it by using "set *(type *)address = value".
3. As to examine the content of a chunk of memory, we could use "print symbol" when a symbol is assigned to it or "x address" when no symbol is assigned to it. Try with the crash2.cpp program and examine the content of the memory where integer1 is with following different flags passing to the "x" command: 1) "x 0x7fffffffe494"; 2) "x /d 0x7fffffffe494"; 3) "x /c 0x7fffffffe494"; 4) "x /f 0x7fffffffe494". Substitute 0x7fffffffe494 with the actual address of integer1 when you execute the program in debugger.
4. Another tip, gdb is just like bash where you could use the up/down arrow keys to find previous executed commands. This could save you a lot time on repeatedly typing "step" or "next".

Debugging crash3.cpp

- This program involves more than one file. Except the crash3.cpp which is the driver program, it also utilize sort.cpp and sort.h file to finish selection sort on an array. If you are not familiar with selection sort, please refer to chapter 8 in your textbook.
- Take a look at three different files of this program first. Make sure you understand what the program is doing by reading codes and comments.
- Now, compile the program with "make crash3" command and run it by command "./crash3 a 20 50". Run this command several times, and you should see that each time the program generates a random length array with random values (but the length is bounded by 20 and values are bounded by 50), prints the original array, sorts it ascending, and prints the sorted array.
- Now, comment out the code section that are labeled as correct ways of doing things, instead, use the code section that is syntactically correct but semantically wrong (both code sections are in generateArray function in crash3.cpp). Compile (the program should still be free of compilation error) and run the program again, what did you get?
- Now, using the debugger to run through the correct and incorrect versions of the program. Try to figure out why the syntactically correct code doesn't work.

Some Tips:

1. To run a program with command line arguments with gdb, a simple way of doing that is to use command "gdb --args ./crash3 a 20 50"
2. To examine the source code of other files, for example sort.cpp, you use command "list sort.cpp:line_number".

3. To set breakpoint in other files, for example sort.cpp, you use command "break sort.cpp:line_number".

Take home practice: debugging the matrix multiplication program

The crash4.cpp program is a simplified matrix multiplication program where it randomly initializes two compatible matrices and try to multiply them. The seeded bugs in this program are all coming from Q&As. Use gdb debugger to fix the segmentation fault of this program.

CSCI 1730 Lab/Project Specification (Week 6)

Programming Project #3

Object-Oriented Programming in C++

--Building Object Hierarchy for a Drawing System

Goal

In this project, you will use C++ to implement part of a drawing system. In this part, you have to implement an object hierarchy of shapes. This part will be used to build further parts of the system. By implementing this shape hierarchy, you are expected to learn basic C++ object-oriented programming design and implementation of object-oriented programming concepts such as inheritance, polymorphism, composition, etc.

Due Date

Feb 28, 2012 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

Project Description

Step 1: Design the shape objects hierarchy

In this first step, you are going to design the shape objects hierarchy according to following requirements:

- Any object in this hierarchy is a **shape** object. A shape object calls its *draw()* method to draw itself. An *addShape(Shape* s)* method will add another shape *s* to the current shape to make it more complicated. (e.g., *s1.addShape(s2)* added shape *s2* to shape *s1*) An *explodeShape()* method will return all shape components (all shapes added to current shape) in a list format.
- A **finalized** shape is a shape to which no more shapes may be added. Thus, calling *addShape(Shape* s)* on a finalized shape should throw an exception. Calling the *explodeShape()* methods on a finalized shape will return a list of length 1, containing the shape itself.
- A **non-finalized** shape is a shape that other shapes (either finalized or not) may be added to it to make it more complicated (e.g., *nonfinal.addShape(s)* is permitted)

- All basic shapes, including point, line, rectangle, round, and triangle, are finalized shape. They should be protected so that users cannot change them to non-finalized shapes.
- A **point** shape is a shape of a single pixel. It has a specified *color* and integer coordinate *x* and *y* define its position.

```
// pseudo code
Shape* point = new Point(color, x, y);
```
- A **line** shape is a shape of an undirected line. A *startpoint* and an *endpoint* which are both points define its position and the color of startpoint also specifies the color of the line.

```
// pseudo code
Shape* line = new Line(point1, point2);
```
- A **rectangle** shape is a shape of a rectangle. A Boolean property *filled* will decide whether the shaped is drawn filled or non-filled. A *basepoint* (point of its left-top corner) specifies its position and color, and two integers, *width* and a *height*, define its shape.

```
// pseudo code
Shape* rectangle = new Rectangle(basepoint, width, height, filled);
```
- A **round** shape is a shape that is part of a *filled* or non-filled ellipse. A *boundingbox* which is a rectangle, specifies the shape, color, and fill property of the ellipse. Doubles *startdegree* and *enddegree* define the drawing part of the shape. An ellipse is drawn counterclockwise from start to end.

```
// pseudo code
Shape* round = new Round(boundingbox, 30.5, 100.8);
```
- A **triangle** shape is a shape of a filled triangle. Three *points* defines its position and shape, in which the first point specifies its color.

```
// pseudo code
Shape* triangle = new Triangle(point1, point2, point3);
```
- A complex shape, **complexshape** should also be defined. It is not finalized and is left for the user to add other shapes (either basic or complex) to compose it.

```
// pseudo code
Shape* myshape = new ComplexShape();
myshape.addShape(new Triangle(point1, point2, point3));
List<Shape*> components = myshape.explodeShape();
...
```

Leave flexibility on the implementation of **color** since we will port this code to work with a drawing toolkit when implementing the later part of this system. In this part, since we don't actually "draw" anything but just build the shape hierarchy, you may just set and retrieve the name of a color.

Simplify the implementation of "throw exception". You could simply print out a line of message saying that an exception happens. We will perfect this part in the exception handling lab.

Think about these requirements and design an object hierarchy of the shapes according to principles of object oriented design. You could find a brief introduction to SOLID design principles on Wikipedia: http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29.

Draw a class diagram of the shape hierarchy that reflects your design choices. Save your class diagram with common image format (jpeg, png) and name it "diagram.*".

Step 2: Implement and test shape objects

You are given **shape.h** as shown below, which has the class prototypes defined in it.

```

...
class Shape {
    public:
        Shape(bool isFinal);
        void addShape(Shape* s);
        std::vector<Shape*> explodeShape();
        virtual void draw();
        virtual ~Shape();

    private:
        bool finalized;
        std::vector<Shape*> components;
};
...

```

Implement the following methods, stubs for which are found in [shape.cpp](#).

- `Shape::Shape(bool isFinal)`, the constructor of a shape
- `Shape::~~Shape()`, the abstract destructor of a shape
- `void Shape::addShape(Shape* s)`, the *addShape(Shape* s)* method
- `std::vector<Shape*> Shape::explodeShape()`, the *explodeShape()* method
- `void Shape::draw()`, the abstract drawing method of a shape

Implement [point.h](#), [point.cpp](#), [line.h](#), [line.cpp](#), [rectangle.h](#), [rectangle.cpp](#), [round.h](#), [round.cpp](#), [triangle.h](#), [triangle.cpp](#), [complexshape.h](#), [complexshape.cpp](#) according to your class design.

Leave the *draw()* methods for all shape classes as simple as printing a line that says a corresponding draw method is called to draw a specific shape with its position and other properties such as width, height, etc through a standard output: `std::cout`. For example:

```

// pseudo code
void Point::draw() {
    std::cout
        << "Point::draw() is called to draw point: "
        << getX() << ", " << getY()
        << " with color " << getColor()
        << std::endl;
}

```

We will implement this method with actual drawing toolkit library methods in later parts of the system.

To compile this shape hierarchy, you will write a [makefile](#) with following format.

First we define some parameters of compiler, flags and linker. This part will be useful when later we start using toolkit library since it simplifies the compiling commands defined in later part of the makefile.

```

CXX = $(shell fltk-config --cxx)
DEBUG = -g
CXXFLAGS = $(shell fltk-config --use-gl --use-images --cxxflags)
-Wall -I
LDFLAGS = $(shell fltk-config --use-gl --use-images --ldflags)

```

```
LDSTATIC = $(shell fltk-config --use-gl --use-images --ldstaticflags)
LINK = $(CXX)
```

Then we define parameters of target (executable file), objects (.o files) and source files (.cpp files)

```
TARGET = test
OBJS = // list all object .o files you are going to compiled here
SRCS = // list all your source .cpp files here
```

Then we define rule of compilation and linking of different files.

```
.SUFFIXES: .o .cpp
%.o: %.cpp
    $(CXX) $(CXXFLAGS) $(DEBUG) -c $<

$(TARGET): $(OBJS)
test.o: test.cpp //add other header files here
shape.o: shape.cpp shape.h
// add rules for other objects (point, rectangle, line, etc) here
```

Finally, we define makefile targets and their corresponding commands.

```
all: $(TARGET)
    $(LINK) -o $(TARGET) $(OBJS) $(LDSTATIC)

clean: $(TARGET) $(OBJS)
    rm -f *.o 2> /dev/null
    rm -f $(TARGET) 2> /dev/null
```

You are given an unfinished **makefile** in the above style. Complete it according to the hints in comments and use it to compile your shape hierarchy. Notice that the **test.cpp** is the driver program you will create in step 3. For the purpose of compiling your shape hierarchy, you could simply used the given empty **test.cpp** file which includes all shapes' header files as seen below.

```
#include "shape.h"
#include "point.h"
#include "line.h"
#include "rectangle.h"
#include "round.h"
#include "triangle.h"
#include "complexshape.h"

int main(int argc, char** argv) {

    return 0;

}
```

Step 3: Create driver program

Complete the *main* function in **test.cpp** so that the driver program will perform following actions:

- create new points **p1**, **p2**, **p3**, **p4**, draw all of them
- create a new line **l1** with points **p1** and **p2**, draw it
- create a new line **l2** with points **p3** and **p4**
- create a new filled rectangle **r1** with basepoint **p1**, draw it
- create a new non filled rectangle **r2** with basepoint **p2**, draw it

- create a new filled rectangle **r3** with basepoint **p3**
- create a round **rd1** with boundingbox **r2**, draw it
- create a round **rd2** with boundingbox **r3**
- create a triangle with **p1**, **p2** and **p3**, draw it
- create a complexshape **myshape**
- add **r3**, **l2**, and **rd2** to myshape
- draw **myshape**
- explode **myshape** and draw each of its components
- delete all shapes

For unmentioned parameters of a shape such as width and height for a rectangle, use any value. Compile and run your program.

Step 4: Documentation and Submission

Write a **readme** file with very brief instructions on how to compile and run your program as well as anything you would like the TAs to note.

Create a new folder (named "lastname1_lastname2_lab06") in your local Nike account. Copy all ***.h**, ***.cpp**, the **image file** of your class diagram design and the **readme** to the newly created folder and submit the folder to cs1730. **Do not include any *.o or executable files.** (submit lastname1_lastname2_lab06 cs1730)

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Source Code	45		
Shape class	10		
Point class	5		
Line class	5		
Rectangle class	5		
Round class	5		
Triangle class	5		
ComplexShape class	10		

Lab 07 Practice on Basic C++ Exception Handling

Goal

In this lab, you will use C++ to modify a piece of code so that it appropriately handles all possible exceptions. Then you will add exception handling in the shape.cxx and test.cxx files of your previous lab, to handle possible exceptions in the drawing system.

Due Date

Feb 24, 2012 (Friday) 11:59 pm on part 1.

Feb 28, 2012 (Tuesday) 11:59 pm on part 2.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

Part 1: Handling basic C++ exceptions

Look at **handle.cpp** file and read the comments. Inside the main function, each comment indicates a possible exception that you will need to handle.

bad_alloc exception

Comment out code blocks 2, 3, 4 and leave only code block 1 in the main() function. Compile and run the program. What is the output you get?

Search a little bit online with key words “c++ exception bad_alloc” to find code examples and explanations.

Add a try/catch block around code block 1 to appropriately handle the exception. Then compile and re-run the program. Make sure your program prints a line saying “All code executed” before you proceed.

bad_cast exception

Uncomment code block 2 in the main() function. Compile and run the program. What is the output you get now?

Figure out what key words to search online and look up that exception.

Add a try/catch block around code block 2 that appropriately handles the exception. Then compile and re-run the program. Make sure your program prints a line saying “All code executed” before you proceed.

bad_typeid exception

Uncomment the code block 3 in the main() function. Compile and run the program. What is the output you get now?

Figure out the cause of that exception by searching online.

Add a try/catch block around code block 3 that appropriately handles the exception. Then compile and re-run the program. Make sure your program prints a line saying “All code executed” before you proceed.

other exception

Uncomment the code block 4 in the main() function. Compile and run the program. What is the output you get now?

Figure out the cause of that exception by searching online.

Modify myfunction() so that it throws the correct type of exception (say an integer 20). Compile and run the program again. What did you get now?

Add a try/catch block around code block 4 that appropriately handles the integer exception. Then compile and re-run the program. Make sure your program prints a line saying “All code executed” before you proceed.

bad_exception exception

Modify myfunction() again so that it still throws an exception with the incorrect type (say a character 'x'). Compile and run the program. What did you get now?

Now search online with the keywords “c++ exception bad_exception”.

Register the bad exception handler, the unexp_hd() function, at the very beginning of code block 4 by writing: set_unexpected(unexp_hd);

Compile and re-run the program. What did you get now?

Add another catch block after the one that catches the integer exception in code block 4, to

catch the `bad_exception` exception. Appropriately handle this exception. Now your program should be able to reach the final statement in the `main()` function and print a line saying "All code executed".

Part 2: Implement Exception Handling for Drawing System

In this step, you should revisit the `addShape()` function in your `shape.cxx` file and actually throw a string exception that saying "no shape may be added to a finalized shape". Also modify your `test.cxx` file to use try/catch blocks to catch and handle possible exceptions.

Part 3: Documentation and Submission

For part 1: (Due on Feb 24, Friday, 11:59pm)

Create a new folder (named "lastname1_lastname2_lab07") in your local nike account. Copy the **handle.cpp** file you modified in step 1 to the newly created folder and submit it to cs1730. **Do not include any *.o or executable files.** (`submit lastname1_lastname2_lab07 cs1730`)

For part 2: (Due on Feb 28, Tuesday, 11:59pm)

Please refer to lab06 specification for submission. You just submit your lab06 source code.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Exception Handling (Step 1)	45		
bad_alloc	10		
bad_cast	10		
bad_typeid	10		
bad_exception	15		

CSCI 1730 Lab/Project Specification (Week 8)

Programming Project #4

Object-Oriented Programming in C++

--Working with FLTK for a Drawing System

Goal

In this project, you will learn to use external library functions in C++ to implement part II of the drawing system. In this part, you will implement the drawing methods of shape objects with functions provided by FLTK library. You are expected to learn how to work with external library functions in C++. This project also furthers your understanding of object-oriented programming concepts such as inheritance, function overloading, polymorphism, composition, etc. This part will be used to build further parts of the drawing system.

Due Date

Mar 20, 2012 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

Project Description

Step 1: Working with Graphic Interfaces through nike System

In this first step, you are going to setup some basic environments which enable you to work with graphical interfaces through the nike system.

- If you are working on Mac:

You do not need to install any third party software to work with a graphical interface through the nike system. All you need to do is to connect to nike with the '-x' option. Use the following command to connect to nike:

```
ssh -l [your username] nike.cs.uga.edu -x
```

After you see the command prompt, type command:

```
firefox &
```

If everything is set up correctly, a firefox browser window will start from the nike server.

- If you are working on Windows:

You need to install third party software (x windows) to work with a graphical interface through the nike system with ssh.

1. Go to <http://x.cygwin.com/> and scroll down to the "downloading and installing" section. Click on the setup.exe link, download and save the file.
2. Double click the setup.exe stored on your local machine and start installation. Use the default settings to finish the installation of the cygwin-x system. This may take quite some time depending on your network connection. So please be patient.
3. After installation, you may have to reboot your system. Then you will see the Cygwin X folder from your start menu. Go to that folder and run the XWin Server program.
4. After XWin Server is running, you should see an X icon in your windows notification area (lower right corner of the screen). The Cygwin X window system works with both windows XP and windows 7 systems.
5. Now start your ssh-client program. (For downloading and installing of ssh-client program, please refer to AccessNikeGuide file on ELC home page.)
6. Click on "Profiles->Edit Profiles". Select the Tunneling panel on the right side and make sure to choose "Tunnel X11 connections" option. Then select the Authentication panel and make sure to choose "Enable to SSH2 connections" option. Save your settings.
7. Connect to nike normally. After you see the command prompt, type command:
firefox &

If everything is set up correctly, the firefox window should start from the nike server.

Step 2: Implement Color Property

In previous lab, we kept the property of a shape object's color as a string. In this lab, we are going to work with FLTK colors and you will use the FL_Color type. First, modify the point.h file as follows:

```

#include <FL/Fl.H>
#include <FL/fl_draw.H>
...
class Point : public Shape {
public:
    Point(int, int, Fl_Color);
    // some other methods
private:
    int x;
    int y;
    Fl_Color color;
};
...

```

Then, modify the actual implementation of the constructor and the accessors of color property:

```

#include <FL/Fl.H>
#include <FL/fl_draw.H>
...
Point::Point(int newx, int newy, Fl_Color newcolor):Shape
(true) {
    x = newx;
    y = new y;
    color = newcolor;
}
Fl_Color Point::getColor(){return color;}
void Point::setColor(Fl_Color newcolor){color = newcolor;}
...

```

Finally, if necessary, make modifications to files of other shapes.

Step 3: Implement the Draw() Method

In this part of the lab, we are going to actually draw out shapes. Here is an example of how to use FLTK methods to draw a point (a single pixel):

```

// point draw method
void Point::draw(){
    // set the color of pen to the point's color
    fl_color(getColor());
    // draw the point
    fl_point(getX(), getY());
}

```

To draw other shapes, you need to use following functions. Look up the FLTK documents (<http://www.fltk.org/documentation.php>) and use these functions appropriately in different shape object's draw method.

```

fl_line(int x1, int y1, int x2, int y2);
fl_rect(int x, int y, int width, int height);
fl_rectf(int x, int y, int width, int height);
fl_pie(int x, int y, int width, int height, double start, double
end);
fl_arc(int x, int y, int width, int height, double start, double
end);
fl_polygon(int x1, int y1, int x2, int y2, int x3, int y3);

```

To draw a complex shape, you just call the corresponding draw method of each of its components.

Step 4: Create a Canvas Object

To draw shapes, we need a canvas object. It will extend the `Fl_Widget` class and maintain a list of shapes (finalized or non-finalized) to draw. A prototype of it (`canvas.h`) looks like this:

```
#ifndef CANVAS_H
#define CANVAS_H

#include "shape.h"
#include <FL/Fl.H>
#include <FL/Fl_Double_Window.H>
#include <FL/fl_draw.H>
#include <vector>

class Canvas : public Fl_Widget {
public:
    Canvas(int X, int Y, int W, int H, Fl_Color
B);

    void draw();
    void enqueueShape(Shape* s);
private:
    Fl_Color background;
    std::vector<Shape*> shapes;
};
#endif
```

The constructor of the canvas specifies the position and size of the canvas in a window as well as its background color. The canvas object maintains a list of shapes to draw. To implement the draw method of the canvas, we draw the canvas itself as well as all the objects it holds in the shapes vector:

```
// draw method
void Canvas::draw(){
    fl_push_clip(x(), y(), w(), h());

    fl_color(background);
    fl_rectf(x(), y(), w(), h());

    //Actually draw shapes here
    for (unsigned int i=0; i<shapes.size(); i++){
        shapes[i]->draw();
    }

    fl_pop_clip();
}
```

Step 5: Modify Driver Program

Now in our driver program, we create shapes and add them to the drawing list of the canvas. Then we call the redraw method which is defined for all `Fl_Widget` objects and calls the draw method of that widget. To initialize and start the FLTK window, we need to create, initialize, show and run it in our main function.

```
//testdraw.cxx
int main(int argc, char** argv) {
    Fl_Double_Window window(600, 600);
```

```

        Canvas* canvas = new Canvas(0, 0, 600, 600,
        FL_WHITE);

        Point* bp = new Point(10,10,FL_BLACK);
        Shape* rectangle = new Rectangle(bp, 100, 100,
        true);

        canvas->enqueueShape(rectangle);

        canvas->redraw();

        window.end();
        window.show(argc,argv);
        return Fl::run();
    }

```

With these settings, when you run the driver program, a GUI window should start and you may be creative what you would like to draw in your program. Our test program will use each shape at least once.

Step 6: Documentation and Submission

Write a **readme** file with very brief instructions on how to compile and run your program as well as anything you would like the TAs to note.

Create a new folder (named "lastname1_lastname2_lab08") in your local Nike account. Copy all *.h, *.cxx and the **readme** to the newly created folder and submit the folder to cs1730. **Do not include any *.o or executable files.** (submit lastname1_lastname2_lab08 cs1730)

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Source Code	45		
Point draw & color	10		
Line draw	5		
Rectangle draw	5		
Round draw	5		
Triangle draw	5		
ComplexShape draw	5		
Driver program	10		

Lab 09 Practice on C++ Operator Overloading

Goal

In this lab, you will use C++ operator overloading feature to modify part of the drawing system -- the complex shape class. You are going to implement the "+=" and "+" operator so that user may add shapes to a complex shape using these operators rather than calling the *addShape* method.

Due Date

Mar 20, 2012 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

Step 1: Reviewing Operator Overloading

Review the corresponding material of operator overloading in textbook and lecture slides.

Here are some simple rules of thumb for you to remember:

- 1) While designing operator overloading, you are making the user's life easier but not yours.
- 2) If an operator's semantic meaning is not clear or straight forward in application domain, use a function with well-chosen name instead.
- 3) Provide all out of a set of related operators such as "=", "+=", "+".
- 4) A unary operator is usually implemented as a member function.
- 5) A binary operator that treats both operands equally (such as +, -) is usually implemented as a non-member function.
- 6) A binary operator that does not treat both operands equally (such as =, +=, -=), may be implemented as a member function of its left hand side operand.

The operator "=" is already declared and implemented in the given **complexshape.h** and **complexshape.cxx** files. Please take a look at it and make sure you understand the general syntax of overloading an operator.

Step 2: Implement Operator "+=" as a Member Function

The header files and object files of all basic shape types (point, line, triangle, rectangle and round) and the canvas are given for your use. You are also given a slightly modified makefile which will only build the complexshape object and testdraw programs. In **testdraw.cxx** the program main function utilizes the += operator to add basic shapes to a complex shape.

Leave code block 2 uncommented and comment out code blocks 3 and 4. Declare and implement the += operator overloading in the **complexshape.h** and **complexshape.cxx** files. Compile and run the program. You should see a red Minnie drawn under the black Mickey.

Then leave code block 3 uncommented and comment out code blocks 2 and 4. Compile and run the program, it should achieve the same effect. If not, you should modify your += operator overloading.

Step 3: Implement Operator "+" as a Friend Non-Member Function

Leave code block 4 uncommented and comment out code blocks 2 and 3. Declare and implement the + operator overloading in the given **complexshape.h** and **complexshape.cxx** files. Compile and run the program. You should see the same effect as described in the previous step.

Part 3: Documentation and Submission

Please submit this work together with your lab 08. No separate submission is required. This lab will be graded together with lab 08.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Operator Overloading	45		
+=	25		
+	20		

CSCI 1730 Lab/Project Specification (Week 7)

Programming Project #5

Shell, I/O, Directory, File and Memory

Goal

In this project, you will write three pieces of programs. The first program should list files and directories in current working directory. The second program will encode plain text from standard I/O or text file. The third program will decode encrypted text from standard I/O by using UNIX pipes. The goal of the project is:

- 1) Practice using C DIR interfaces to extract file and directory information
- 2) Practice using C I/O functions and FILE interfaces to read/write standard I/O and files
- 3) Practice using UNIX pipe (popen(), pclose()) in C
- 3) Practice using basic memory management functions (memset(), malloc(), free())
- 4) Get familiar with basic shell redirection and shell level piping

Due Date

Feb 28 (Tuesday) at 11:59 pm

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

Project Description

Step 1: Create Listdir Program with C DIR interface

In this first step, you will create a program to extracting file and directory info under current directory. Your program should explore the current directory where it is executed, print out the name, size and file number of each file/directory, each as a line and using tab to separate name, size and file number information. Use opendir(), readdir(), closedir(), rewinddir() and stat() functions appropriately to achieve this.

Name your program as **listdir.c**. Here is a sample run of the program.

```
-bash-4.1$ ./ls
decipher      11314      89524144
```

```
ls      8025      89525269
test.cpp      227      89525085
makefile      271      89525439
note.c  183      89525267
.        4096      89522255
cipher.c      1432      89525087
..        4096      89525054
encode  214920  89524162
cipher  9560    89524140
origin  18      89525230
comp    214920  89525092
listdir.c      567      89525220
alien    214920  89524163
decipher.c      3176      89525245
note     7245      89525056
-bash-4.1$
```

Step 2: Create Encryption Program with Standard I/O

In this first step, you will create a program for encryption. The encryption is a simple cipher, replacing each alphanumeric symbol with a shifted value. Here are two examples:

```
Dog3=> shifted 1 => Eph4
Cat...9  => shifted 2 => Ecv...1
```

Only letter (upper- and lowercase) and numeric (0-9) symbols should be affected. All other symbols should pass through encryption unaffected. The shifting of a symbol should wrap around its set. For example, the symbol “z” shifted 1 should become “a”. The symbol “Z” shifted 2 should become “B”. The symbol “9” shifted 3 should become “2”;

The program should encode text. It should prompt for the user to enter a string, encode it, and print out the encoded version to screen. An EOF char (which could be input by hitting Ctrl-D) should terminate the program. The program should also accept a single command line argument defining the shift delta. The value of delta must be an integer between 0 and 9, inclusive.

Name your program as **cipher.c**. Here is a sample run of the program.

```
-bash-4.1$ ./cipher 4
this is an apple.      // user input
xlmw mw er ettpi.      // print by program
-bash-4.1$              // user input Ctrl-D
```

Step 3: Run Encryption Program with Shell Stream Redirection

In this step, you should run your program with following different shell commands:

```
UNIX> ./cipher 4
=> read from standard input, encrypt the text and print out the
    encrypted text to screen
```

```
UNIX> ./cipher 4 < origin
=> redirect standard input to file origin, encrypt the text in
    it and print out the encrypted text to screen
```

```
UNIX> ./cipher 4 > encode
=> read from standard input, redirect standard output to start
    of file encode and write out the encrypted text to it
```



```

UNIX> ./cipher 4 >> encode_all
=> read from standard input, redirect standard output to end of
file encode_all and append encrypted text to it

UNIX> ./cipher 4 < origin > encode
=> redirect both standard input and output to file origin and
encode

```

Step 4: Modify Encryption Program to Work with Files and Chomd

In this step, you should modify the **cipher.c** program you created in step 1 so that besides working with standard input/output, a user could also pass filenames as command line arguments to the program. The program should check the `argc` value. When it is 2, the program will use standard input/output. When it is 4, the program will take last two arguments as input and output filenames. Otherwise, the program should exit with printing out a corresponding error message. Here are examples:

```

UNIX> ./cipher 4
=> read from standard input, encrypt the text and print out the
encrypted text to screen

UNIX> ./cipher 4 original encoded
=> read file original, encrypt the text in it and save the
encrypted text in file encoded

```

Use the “`chmod 000 original`” command to change the umask of file original and try to re-execute the above commands. See what will happen. Use the “`chmod -rw-r--r--original`” command to change the umask of file original back to what it was.

Step 5: Create Decryption Program with Standard I/O, using UNIX Pipe and Memory Management Functions

In this step, you will create a second program for decrypt encoded text. The program should be unaware of the value of delta used to encode the text. Instead, it must figure out the value of delta by trying to decrypt using all possible values for delta and examining the resulting text. To examine the result, the program must use the dictionary stored in the `linux.words` file (in `/usr/share/dict/` directory on Nike). It should compare every potential decrypted word with the dictionary, looking for a match. Whichever value for delta produces the most matches with words in the dictionary should be assumed to be the correct value of delta. The program should print out the decrypted text using that value of delta (and it should not print out anything else). An EOF char (which could be input by hitting Ctrl-D) should terminate the program. Use `popen()`, `fscanf()` and `pclose()` functions appropriately to achieve these.

Name your program as **decipher.c**. Here is a sample run of the program.

```

-bash-4.1$ ./decipher
xlmw mw er ettpi.    // user input
this is an apple.    // user input Ctrl-D
-bash-4.1$

```

Step 6: Test Programs with Shell Pipelining

In this step, run both of your program with shell pipelining techniques to pipe standard output from the cipher to standard input on the decipher.

You don't need to write any more code and you should be able to achieve following commands:

```
UNIX> ./cipher 4 < origin | ./decipher > decoded
```

=> read file origin and pipe output of cipher program to the input of decipher program and the decipher program write output to file decoded

```
UNIX> ./ls | ./cipher 4 | ./decipher > list
```

=> list all files and directories, pipe the output to cipher program, pipe the encrypted list to decipher program and write output to file list

Step 7: Documentation and Submission

Write a **readme** file with anything you would like the TAs to note.

Create a new folder (named "lastname1_lastname2_lab07") in your local Nike account. Copy **listdir.c**, **cipher.c**, **decipher.c** and **readme** files to the newly created folder and submit the folder to cs1730. **Do not include any *.o or executable files.** (submit lastname1_lastname2_lab07 cs1730)

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
List program	10		
Encode program with Standard I/O	5		
Encode program with File	5		
Memory Allocations in Decode Program	10		
Unix Pipe in Decode Program	10		
I/O Redirection and Piping	5		

CSCI 1730 Lab Specification (Week 11)

Files and Streams in C and C++

--Adding Files and Streams Features for the Drawing System

Goal

In this lab, you will finish two parts of work. First part is a hands-on practice specified at: <http://www.cs.uga.edu/~eileen/1730/Notes/Mar28/Lab11.html>.

In second part, you will use C++ file and streaming features (which are built upon C basic file and streaming functions) to enhance the drawing system – adding shape parsing and file read/write features. You are going to review the usage of file input/output streams while learning other advanced stream features of C++.

Due Date

Part 1: Apr 3, 2012 (Tuesday) 11:59 pm.

Part 2: Apr 10, 2012 (Tuesday) 11:59 pm.**Late Penalty**

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

As solo programmers, you should work on your own.

Step 1: Serialization of Shape Objects

Serialization is a whole complicated topic in object oriented programming. Thought of in a simplest way, it is the technique of transforming objects to bits and bytes that could be transmitted through files, networks or other stream transmission methods.

In this lab, we will NOT implement a full serialization feature for the drawing system. However, we are going to add features to the project so that the drawing system may draw shapes by reading files and write files according to the shapes it has drawn.

We define following records for all concrete shapes in our hierarchy:

1. Point
point [color] [coordinate]
2. Rectangle
rectangle [color] [coordinate_of_basepoint] [width] [height] [fill]
3. Line
line [color] [coordinate_of_startpoint] [coordinate_of_endpoint]
4. Round
round [rectangle] [start_degree] [end_degree]
5. Triangle
triangle [color] [coordinate_of_p1] [coordinate_of_p2][coordinate_of_p3]

For a color, we use three integer numbers (corresponds to red, green and blue values of a color) to represent it. For example, the color black will be represented as "0 0 0" (without the quotes) and color white will be represented as "255 255 255".

For a coordinate, we use two integer numbers (corresponds to x and y values of a coordinate) to represent it. So "20 30" (without the quotes) represent the coordinate (20, 30).

Here are a bunch of example records that represents different shapes:

point 255 0 0 20 30

→ a red point at coordinate (20, 30)

rectangle 0 255 0 20 30 100 200 1

→ a filled green rectangle start at (20, 30) which is 100 wide and 200 tall

round 0 255 255 20 30 50 50 1 30 360

→ a filled yellow round shape bounded by a rectangle which start at (20, 30) and is 50*50; while the round shape is a portion of the round from degree 30 to degree 360

triangle 0 0 255 0 0 20 20 100 100

→ a blue triangle whose three points are (0, 0), (20, 20) and (100, 100)

You are not expected to write out any code in this step. Your major concern is to understand the corresponding format of a record for each shape.

Step 2: Implement an Abstract `toString()` Method for Shape Hierarchy

Modify the **shape.h** file and add a virtual `toString()` method to the shape class. Then implement the actual `toString()` methods in each concrete class: point, line, rectangle, round, and triangle.

Here is an example of using the stringstream library functions to deal with string concatenations and conversions:

```
// toString method of Point class
std::string Point::toString(){
    // create a new stringstream object ss
    std::stringstream ss;

    // attach the string "point " to stream
    ss << "point ";

    // get the red, green, blue components of color
    uchar r, g, b;
    Fl::get_color(color, r, g, b);
    // attach the color components value as unsigned int
    ss << (unsigned int)r << " " << (unsigned int)g << " " <<
(unsigned int)b << " ";

    // attach the x, y coordinate values to stream
    ss << x << " " << y << std::endl;

    // convert stringstream object to standard string
    return ss.str();
}
```

Step 3: Implement Static `parseShape()` Method for Shape Hierarchy

In your **shape.h** file, add the following function prototype to shape class:

```
static Shape * parseShape(std::string message);
```

This method will parse the message that describe a shape and return the pointer to that shape. Here is a simple example of how you could use stringstream object to tokenize a string and parse different types of values:

```
#include <cstdlib>
#include <string.h>
#include <iostream>
#include <sstream>

int main(){
    std::string s("hello 20 23.5");
    std::string token;
    std::stringstream ss(s);

    std::string str;
    int i;
    double d;
```

```

        int count = 0;

        while (getline(ss, token, ' ')){
            std::stringstream stm;
            stm.str(token);

            switch (count){
                case 0:
                    stm >> str;
                    break;

                case 1:
                    stm >> i;
                    break;

                case 2:
                    stm >> d;
                    break;
            }
            ++count;
        }

        std::cout << str << std::endl;
        std::cout << i << std::endl;
        std::cout << d << std::endl;
    }
}

```

Step 4: Implement the Driver Program

Modify the driver program **testdraw.cxx** in your lab 09. Your program should now be able to deal with command line arguments. When there's no arguments, the program draw the shapes you defined in the driver program and after drawing all the shapes onto canvas, it writes messages of all shapes to a file named data. When there is one argument supplied to the program, the program should take it as a data file and draw shapes defined in that file.

Test your program by exchanging the data files with your classmates.

Step 5: Submission

For Part 1: Create a new folder **lab11_lastnames_part1**. Copy files **basic.c**, **basic2.c**, **basic3.c** and **basic4.c** to this folder and submit the folder to nike (Due date: Apr. 3th).

For Part 2: Create a new folder **lab11_lastnames_part2**. Copy all source files of drawing system (*.h and *.cxx files) to this folder and submit the folder to nike (Due Date: Apr. 10th).

Do not include any *.o or executable files.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Files and System Calls	45		
basic.c	10		
basic2.c	15		
basic3.c	15		

basic4.c	5		
Drawing System	50		
toString() methods	15		
parseShape() method	20		
driver program	15		

CSCI 1730 Lab Specification (Week 12)

Processes and Signals

Goal

In this lab, you will finish a series of hands-on practice about manipulating processes and inter-process communication with C system calls and signals. After the lab, you should be familiar with the UNIX process mechanism.

Due Date

Apr 17, 2012 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

As solo programmers, you should work on your own.

Step 1: Parent and Child Processes

1. Copy the following codes into **test_fork.c** file. Create a **makefile** to compile and run the program.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(){
    pid_t child_pid;

    switch (child_pid = fork()){
        case (pid_t) -1:
            break;
        case (pid_t) 0:
            // child
            exit(0);
        default:
            // parent
            exit(0);
    }
}
```

```

        return 0;
    }

```

2. Use “perror” to add error-handling for the case in which fork fails.
3. Add to the “parent” case so that the parent process reports its own process id (use *getpid()* function) and its child’s process id.
4. Add to the “child” case so that the child reports its own process id (use *getpid()* function) and its parent’s process id (use *getppid()* function).
5. Why does the child report that its parent’s process id is 1? Add a line of code in “parent” case to solve that (use *waitpid()* function).
6. The “*_exit(int status)*” system call terminates a process, but keeps the process table entry for that process. It is now a “zombie” process. The parent process can retrieve info about it. Write a test program **test_exit.c**. The program should print out “Test program for *_exit*”, and then call *_exit(0)*. Compile and run the program. Then type “echo \$?” at the command line. Now change *_exit(0)* to *_exit(23)*. Compile and run the program again. Then type “echo \$?” at the command line.
7. Copy your **test_fork.c** program to one named **test_waitpid.c**. In this new program, have the parent process wait for the child process (use *waitpid()*). Check if the child has exited with an error code (*WIFEXITED*, *WEXITSTATUS*), was stopped by a signal (*WIFSTOPPED*, *WSTOPSIG*), or was killed by a signal (*WIFSIGNALED*, *WTERMSIG*). Use *_exit(0)* to exit the child process. Compile and run the program. What is the output? Now change *_exit(0)* to *_exit(23)*, what is the output now?
8. Comment out the *_exit()* call in the child process. Use a *while(1)* loop to let the child run infinitely. Now compile and run your program in the background with the & sign. Send a termination signal with the kill command to the child process. Now what is the output of the program?
9. Copy the following code to **test_exec.c** file. Now modify this program to implement the “System” method. The program should be a simple shell that takes in a command, executes it, and then waits for another command (in a loop).

```

#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int System(const char *cmd){
    // execute command
}

int main(){

```

```

int rc = 0;
char buf[256];

do {
    printf("sh> ");
    fflush(stdout);
    if (!fgets(buf, 256, stdin)) break;
    rc = System(buf);
}while (!rc);

exit(rc);
}

```

Step 2: Signals

1. Write a small program **test_sig.c** that installs a signal handler for SIGUSR1 and SIGUSR2. The signal handler should 1) reinstall itself and 2) print out the identity of the signal received. The main method should install the signal handler for SIGUSR1 and SIGUSR2 (checking that they were properly installed) and then loop, invoking "pause()" within the loop. Compile and run the program in the background.
2. From the command line, send the process above the SIGUSR1 signal. (Use the kill command.)
3. Restart the process if necessary. From the command line, send the process above the SIGUSR2 signal. (Use the kill command.)
4. Restart the process if necessary. From the command line, send the process above the SIGINT signal. (type Ctrl-C or use the kill command).
5. Restart the process if necessary. From the command line, send the process above the SIGQUIT signal. (type Ctrl-\ or use the kill command).
6. Write a small program **test_alarm.c** that reports the elapsed time in 5 second intervals. Use the alarm() call and a handler for SIGALRM to accomplish this.
7. Write a small program called **test_cleanup.c**. The main program should fork off three child processes. Each child should report out (display a message "child process \$pid reporting in" and then enter a while (1) loop in which it reports a loop count and its pid, and then "sleeps" for a few seconds. (See "man -s3c sleep"). The parent process should "pause()" (waiting for a signal). Install a signal handler for the parent (you can choose the signal that you want to use) that sends a kill signal to the children, "wait()" for them to terminate, and then prints out a "parent terminating" message. Install a signal handler for the children that causes them to print out a message "child \$pid exiting" and then exit normally. Compile and run the program in the background. Then send the selected signal to the parent process.

Step 3: Submission

Create a new folder lab12_lastnames. Copy all source files (*.c) and your **makefile** to this folder and submit the folder to nike (Due Date: Apr. 11th).

Do not include any *.o or executable files.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		

Processes	20		
test_fork.c	5		
test_exit.c	3		
test_waitpid.c	10		
test_exec.c	2		
Signals	25		
test_sig.c	10		
test_alarm.c	5		
test_cleanup.c	10		

CSCI 1730 Lab Specification (Week 13)

Threads and Shared Objects in Java and C++

Goal

In this lab, you will finish a series of hands-on practice with threads and monitor models. After the lab, you should be familiar with Java thread mechanism and PThread library in C++.

Due Date

Apr 19, 2012 (Thursday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

As solo programmers, you should work on your own.

Step 1: Java Threads

- Copy the following codes into file **Worker.java**. Create a driver program **Factory.java** that initialize two workers with different ids and let each of them check in 10 times.

```
public class Worker {

    private int id;

    public Worker(int id){
        this.id = id;
    }

    public void checkIn(){
        System.out.println("Worker: " + id);
    }

}
```

11. Read <http://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html> and modify **Worker.java** into a thread object so that its *run* method calls the *checkIn* method 10 times.
12. Now modify the **Factory.java** file and start two worker threads in the *main* method. Run the program several times, what kind of results do you have?
13. Now, read <http://docs.oracle.com/javase/tutorial/essential/concurrency/sleep.html> and modify your **Worker.java** file so that after each check in, the worker will sleep for a random number of milliseconds (0-20). Compile and re-run the program. What kind of effect do you see? Create a folder `java1` and copy all your current *.java files to it.
14. Copy the following code to **Sum.java** file.

```
public class Sum {
    private int sum;

    public Sum() {
        sum = 0;
    }

    public void increase() {
        sum++;
    }

    public void printSum() {
        System.out.println("Sum is: " + sum);
    }
}
```

15. Copy your **Worker.java** file and modify it to the following:

```
public class Worker implements Runnable {

    private int id;
    private Sum sum;

    public Worker(int id, Sum sum) {
        this.id = id;
        this.sum = sum;
    }

    public void checkIn() {
        sum.increase();
    }

    public void run() {
        for (int i=0; i<10; ++i) {
            checkIn();
        }
    }
}
```

```

    }
}

```

Now what does a worker do when it starts?

16. Copy the **Factory.java** file and modify it to the following:

```

public class Factory {
    public static void main(String[] args){
        Sum sum = new Sum();
        Thread[] t = new Thread[100];
        for (int i=0; i<t.length; ++i){
            t[i]=new Thread(new Worker(i, sum));
        }
        for (int i=0; i<t.length; ++i){
            t[i].start();
        }
        for (int i=0; i<t.length; ++i){
            try{
                t[i].join();
            }
            catch (InterruptedException ie){}
        }
        sum.printSum();
    }
}

```

Explain what happens in each **for** loop in the above code. What is the expected output of the program?

17. Create a **makefile** with three targets: all, run and clean. Compile the above program and run it several times. What outputs do you get? How can you explain the outputs? Now read <http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html> and add one keyword to the *increase()* method of the **Sum** class to correct the program so that it gives the expected output. What is that keyword?

18. Copy all your *.java files to a new folder java2.

It will be good for you to finish reading the Java Concurrency Tutorial online: <http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>

Step 2: PThreads

8. Copy the following code into a **sum.h** file:

```

#include <stdlib.h>
#include <pthread.h>
#ifndef SUM_H
#define SUM_H
class Sum {

    public:
        Sum ();
        ~Sum ();

```

```

        void increase();
        void printSum();

    private:
        pthread_mutex_t mutex;
        int sum;
};
#endif

```

9. Read <https://computing.llnl.gov/tutorials/pthreads/#Mutexes> to understand how to create, destroy, lock and unlock a mutex. Then create a **sum.cpp** file to implement the Sum object. It should perform in the same way as the object you defined in **Sum.java**.

10. Copy the following code into **worker.h** file:

```

#include <cstdlib>
#include <pthread.h>

#include "sum.h"

#ifndef WORKER_H
#define WORKER_H

class Worker {

    public:
        Worker(int, Sum *);
        void checkIn();
        static void *run(void *);

    private:
        int id;
        Sum *sum;
};
#endif

```

11. Copy the following code into a **worker.cpp** file and complete its implementation so that a worker object functions in the same way as the one defined in **java2/Worker.java**.

```

#include <cstdlib>
#include <iostream>
#include <time.h>
#include <unistd.h>

#include "sum.h"
#include "worker.h"

Worker::Worker(int identity, Sum *psum){
    // TODO: constructor
}

void Worker::checkIn(){
    //TODO: increase sum
}

```

```

void *Worker::run(void *arg){
    // initialize a singleton instance
    Worker *obj = (Worker *)arg;
    for (int i=0; i<10; ++i){
        obj->checkIn();
        // sleep up to 20 milliseconds
        sleep(rand()%20/1000);
    }
}

```

12. Read “Thread Management” section at <https://computing.llnl.gov/tutorials/pthreads/>. Pay attention to the `pthread_create()` and `pthread_join()` methods. Write a driver program **factory.cpp** that spawns 100 workers.
13. Copy all *.cpp and *.h files to a folder `cpp`. Create a **makefile** to compile and run your program. Your program should not have race conditions. Use the following flag to compile the program with PThread library calls: `-pthread`

Step 3: Submission

Write down brief answers to the above questions in an **answer.txt** file. Create a new folder `lab13_lastnames`. Copy all files in the `java1`, `java2` and `cpp` folders to this folder. Move **answer.txt** to this folder and submit it to nike (Due Date: Apr. 19th).

Do not include any *.o, *.class or other executable files.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Step 1 Java	25		
Q1	2		
Q2	5		
Q3	5		
Q4	3		
Q6	1		
Q7	4		
Q8	5		
Step 2 C++	20		
Q2	8		
Q4	2		
Q5	8		
Q6	2		

CSCI 1730 Lab Specification (Week 14)

Threads and Conditional Synchronization in Java and C++

Goal

In this lab, you will implement a bounded buffer program in both Java and C++ that performs the same functionality. You will practice more with threads and monitor models. You will also practice using the *wait()* call and conditional variables to enable conditional synchronization. After the lab, you should be more familiar with the Java thread mechanism and the PThread library in C++.

Due Date

Apr 26, 2012 (Thursday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

As solo programmers, you should work on your own.

Step 1: Review

Read all previous lecture slides on concurrency topics. Make sure you understand following concepts:

1. Threads
2. Shared Objects
3. Race Conditions
4. Atomic Operations
5. Locks and Synchronizations
6. Conditional Waiting

Read sample codes in slides and codes you wrote for previous lab. Make sure that you are able to recall the code implementation regarding each of the above concepts.

Step 2: Implement Bounded-Buffer Programs in Java and C++

1. You are given code skeletons in both Java and C++ for implementing the bounded-buffer program. You may make your own decisions on the implementation order (Java first, C++ first or implement both at the same time).
2. Your bounded-buffer should have the capacity of holding up to 10 characters. A producer should produce a random character into the buffer when empty slots are available. A consumer should consume a character from the buffer. Characters should be consumed in the order in which they were placed into the buffer.
3. A driver program should spawn 20 producers (totally 20 characters will be produced) and 20 consumers (each consume one character and finally all characters will be consumed).
4. Your program should be free of race conditions. Producer and consumer threads should be synchronized according to the condition of buffer (full or empty).
5. (Bonus) Implementing a generic buffer will be counted as a bonus (a buffer that could be initialized to take different types of objects). You could use Java Generic or C++ Templates to achieve this.

Step 3: Submission

Create a new folder lab14_lastnames. Copy all your source files into this folder and submit it to

nike (Due Date: Apr. 26th).

Do not include any *.o, *.class or other executable files.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Java Implementation	35		
Bounded Buffer	15		
Producer & Consumer	10		
Driver Program	10		
C++ Implementation	60		
Bounded Buffer	20		
Producer	15		
Consumer	15		
Driver Program	10		
Bonus	30		
Java Generic	15		
C++ Template Class	15		

CSCI 1730 Lab Specification (Week 15)

Sockets

Goal

In this lab, you will practice basic function calls used to set up a socket communication between processes. After this lab, you should be familiar with basic C system calls used to set up sockets and be able to build a client/server system.

Due Date

Apr 30, 2012 (Monday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

As pair programmers, you may only collaborate with your assigned pair programming partner for this project according to pair programming policies.

As solo programmers, you should work on your own.

Step 1: Review

Read the sample code of a client/server system at

<http://www.cs.uga.edu/~eileen/1730/Notes/Apr19/Apr19.html> .

Step 2:

14. Download the code skeleton for this lab.
15. In **part 1**, the code skeleton is to set up a server/client system. The server accepts connections from clients and prints out messages send by clients. The client sets up a connection to the server and sends user input messages to the server. When the user types “close”, the client program closes the connection to the server and quits. Where you find comments with a “TODO” note, add the appropriate function calls that you find in the sample code.
16. In **part 2**, a client/server drawing system is already implemented. Read the code, especially the socket related files, client.cxx and server.cxx, then compile and run it to answer the following questions.
 - 1) Which lines in which file make use of the system calls to create a socket, bind a socket, listen on a socket and accept connections of a socket?
 - 2) What is the behavior of the client drawing program? Write a brief user manual.
 - 3) How many threads are running in the server? What responsibilities do they have?

Step 3: Submission

Create a new folder lab15_lastnames. Copy all your source files in part1 and your answer to part2 into this folder and submit it to nike (Due Date: Apr. 30th).

Do not include any *.o, *.class or other executable files.

Grading Rubric:

Deliverables	Total	Points	Comments
Subjective Survey	5		
Part 1	25		
server.c	15		
client.c	10		
Part 2	20		
Question 1)	8		
Question 2)	8		
Question 3)	4		

FIGURE 33 LAB MATERIALS FOR SPRING 2012 STUDY

6.5 MATERIALS FROM SPRING 2013 WORK

CSCI 4900 Programming in Concurrency

(Spring 2013, T/Th 11:00 Boyd 307A, W 11:15 Poultry Science 136)

Description

This four-hour course covers knowledge of programs and programming techniques used in building concurrent systems, including multi-core architectures, concurrency and synchronization issues, programming tactics with Java threads and Scala Actor and Python Coroutine models for shared memory and message passing systems.

This will be a programming-based course with intensive lab sessions. Students are expected to finish reading related background materials and complete quizzes and warm-up programming exercises before class. Students are encouraged to bring a laptop for participating in in-class lab sessions.

This course will also focus on promoting research work. Students are required to present a research paper on the topic of “human factors/software engineering issues on concurrency/parallel programming” during the course. Students will choose papers in consultation with the instructor.

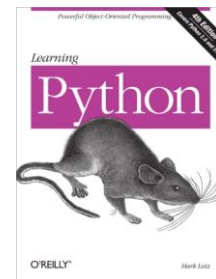
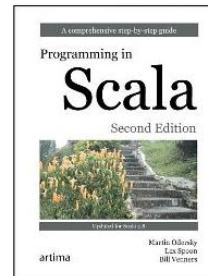
For a frequently updated course calendar and information, please refer to:
<http://www.cs.uga.edu/~zhen/TA/4900/index.html>

For enrollment in our Piazza course, please follow this link:
<https://piazza.com/class#spring2013/csci4900>

For the Piazza course homepage, please go to:
<https://piazza.com/uga/spring2013/csci4900/home>

Textbooks

1. Programming in Scala, Second Edition
 Martin Odersky, Lex Spoon, Bill Venners
 ISBN: 0981531644, ISBN-13: 978-0981531649
2. Learning Python, Fourth Edition
 Mark Lutz
 ISBN: 0596158068, ISBN-13: 978-0596158064



People

Instructor: Eileen Kraemer, eileen@cs.uga.edu.

Assistant Instructor: Zhen (Jane) Li, janeli@uga.edu

Teaching Assistants: Jordan Marchetto, jmarc937@uga.edu

Mayur Chandrakant Jadhav, mayur27@uga.edu

Times and Places

Tuesday/ Thursday 11:00 - 12:15 in Boyd 307A

Wednesday 11:15 - 12:05 in Poultry Science 136

Office hours with Dr. Kraemer: TBA

Office hours with Zhen Li : Boyd 536, Tues 2:00 – 3:30, Thurs 3:30 – 5:00

Office hours with Jordan : Boyd 536, Wednesday 1:30 – 2:45

Office hours with Mayur : Boyd 536, Monday 2:00 – 3:15

Course Policies

1. Policy on Attendance:

Students are expected to attend class. Online materials are designed to supplement, rather than replace, classroom experience. Essential information about assignments, extensions of due dates,

format of exam questions, etc. may be announced or discussed in class with no accompanying posting on eLC or the web. Attendance will be taken periodically in lectures and in labs, and will comprise an element of the course grade.

Without prior arrangement, any missed exam will result in a grade of zero. Absences from exams are excused only in the case of serious illness, as documented by a doctor's diagnostic note. A makeup exam, if offered, will occur at the time of the final exam.

2. Policy on Collaboration:

The purpose of course projects is to familiarize the student with concepts and details of programming shared memory and message passing concurrent systems. These may be pair-programming projects or individual projects, as assigned. Individual projects should be worked on only by that individual. Pair projects should be worked on only by members of the pair.

We recognize that students' interaction can facilitate learning. Accordingly, students are both permitted and encouraged to ask certain types of questions of one another but should be aware that direct exchange of code is prohibited, as is line-by-line assistance.

Examples of allowable questions:

- What does this compiler error mean?
- Why am I not getting any output?
- How do I submit my project?
- Would you help me with my makefile?

Examples of prohibited questions:

- Can I see your code?
- Would you send me your code (or code snippet)?
- How should I design this class?
- How did you implement function X?

If in doubt, please contact Dr. Kraemer or Jane Li for assistance in deciding what is or is not an allowable interaction.

Exams are closed-book. No outside assistance is permitted. No additional materials may be used.

3. Late Policy:

Late submissions of projects will be accepted with a penalty computed as follows:

$$score = MaxScore \times Percentage \times 0.9^{daysLate}$$

A day is a 24 hour period, rounded up to the nearest day.

For example:

You turn your project in 3 days late, and received a 95% score based on the work done. Your recorded score will be:

- $score = 100 \times 0.95 \times 0.9^3$
- $score = 100 \times 0.95 \times 0.729$
- $score = 69$

Exception: In the case that a solution is distributed, no project submissions will be accepted after distribution of the solution.

4. Grading Policy:

Your grade in this course will be calculated as follows:

- Exams: 30% (midterm I = 15%, midterm II = 15%)
- Homework and Quizzes: 10%
- Projects: 50%
- Paper Presentation: 10%

Letter Grades assigned as follows:

- 93 <= Grade A
- 89 <= Grade < 93 A-
- 86 <= Grade < 89 B+

- 83 ≤ Grade < 86 B
- 79 ≤ Grade < 83 B-
- 76 ≤ Grade < 79 C+
- 73 ≤ Grade < 76 C
- 69 ≤ Grade < 73 C-
- 65 ≤ Grade < 69 D
- Grade < 65 F

FIGURE 34 SYLLABUS OF CSCI4900 FOR SPRING 2013 STUDY

Syntax	Example
Values <i>value</i> The simplest component in our pseudo code system.	0 3.3 "number" True, False NULL
Operators <i>Math Operators: +, -, *, /, %</i> <i>Relational Operators: ==, <, <=, >, >=, !=</i> <i>Conditional Operators: AND, OR, NOT</i> <i>Expression: a well-formed combination of operators and values</i>	5 + 3 → 8 5 - 3 → 2 5 * 3 → 15 5 / 3 → 1 5 % 3 → 2 3 == 5 → False 3 < 5 → True 3 <= 5 → True 3 > 5 → False 3 >= 5 → False 3 != 5 → True True AND True → True True AND False → False False AND False → False True OR True → True True OR False → True False OR False → False NOT True → False NOT False → True 3 <= 5 OR False → True
Simple Statement <i>variable = expression</i> Simple statements are executed atomically. Assignment is an example of a simple statement	total = 0 name = "John Smith" condition = True height = 3.3
Print Statement PRINTLN <i>value, value, ...</i> PRINT <i>value, value, ...</i> Prints out values with or without new lines. Print statements are executed atomically.	PRINT "hello" PRINTLN "world" PRINTLN "hello", "world" Output hello world hello world
Comments // <i>comment(s)</i> Comments are not part of the executable	// print "hey there" PRINT "hello world"

code. They exist to illustrate the code.	Output hello world
Function Definitions <pre> DEFINE function-name(parameter, parameter, ...) statement(s) RETURN value ENDDDEF </pre>	<pre> DEFINE myFun(num1) PRINT num1 RETURN num1 + 1 ENDDDEF </pre>
Function Call Statement <pre>function-name(value, value, ...)</pre> <p>A function call statement triggers the execution of the function specified by <i>function-name</i>.</p> <p>The number of values passed into a function call statement must be the same as the number of parameters defined for that function.</p> <p>A function call statement is not necessarily executed atomically. However, the assignment of the value returned from a function call is a simple statement, and thus is executed atomically.</p>	<pre> DEFINE myFun(num1) PRINT num1 RETURN num1 + 1 ENDDDEF num = myFun(3) PRINTLN num </pre> Output 3 4
Random Generating Function <pre>randNum(start, end)</pre> <p>Generates a random number value that is greater than or equal to <i>start</i> and smaller than or equal to <i>end</i>.</p> <p>This function call is executed atomically.</p>	<pre>num = randNum(0, 2)</pre> <pre>PRINTLN num</pre> Output (all possibilities) possibility 1: 0 possibility 2: 1 possibility 3: 2
If Statement (Conditional) <pre> IF condition THEN statement(s) ELSE IF condition THEN statement(s) ELSE statement(s) ENDIF </pre> <p>The calculation of <i>condition</i> is not necessarily atomic if it involves function call statements. However, the choice of branch based on a calculated <i>condition</i> value is executed atomically.</p>	<pre> IF testScore >= 90 THEN PRINTLN "A" ELSE IF testScore >= 80 THEN PRINTLN "B" ELSE IF testScore >= 70 THEN PRINTLN "C" ELSE PRINTLN "F" ENDIF </pre> <pre>testScore = 88</pre> Output B
While Loop (Indefinite) <pre> WHILE condition DO statement(s) ENDWHILE </pre> <p>The calculation of <i>condition</i> is not necessarily atomic if it involves function call statements. However, the decision to enter or remain in the loop body based on the calculated <i>condition</i> value is executed atomically.</p>	<pre> count = 5 WHILE count < 9 DO PRINT count, " " count = count + 1 ENDWHILE </pre> Output 5 6 7 8

<p>Exit Statement</p> <pre>EXIT</pre> <p>A statement that terminates the current execution of a loop.</p>	<pre>DEFINE myPrint() WHILE True DO PRINTLN "hello" EXIT ENDWHILE ENDDEF myPrint()</pre> <p>Output hello</p>
<p>Classes and Objects</p> <pre>CLASS class-name DEFINE initialize class-name() variable = initial value ENDDF DEFINE function-name(parameter, ...) statement(s) ENDDF ENDCLASS</pre> <pre>object-name = new class-name() object-name.function-name()</pre> <p>Object creation is atomic. Member function calls are not atomic.</p>	<pre>CLASS Car DEFINE initialize Car() wheels = 4 speed = 100 ENDDF DEFINE speedUp(newSpeed) speed = newSpeed ENDDF ENDCLASS</pre> <pre>car1 = new Car() car1.speedUp(200)</pre>
<p>Lists</p> <pre>list-variable = [item1, item2, ...]</pre> <pre>length(list-variable)</pre> <pre>list-variable[index]</pre> <pre>add(list-variable[index], value)</pre> <pre>del(list-variable[index])</pre>	<pre>sports = ["soccer", "football", "hockey"]</pre> <pre>length(sports) → 3</pre> <pre>sports[0] → "soccer" sports[1] → "football" sports[2] → "hockey" sports[3] → NULL</pre> <pre>add(sports[3], "baseball") → sports now contains ["soccer", "football", "hockey", "baseball"]</pre> <pre>del(sports[2]) → sports now contains ["soccer", "football", "baseball"]</pre>
<p>Parallel Execution Statements</p> <pre>PARA statement(s) ENDPARA</pre> <p>Statements within the PARA/ENDPARA block are executed concurrently.</p> <p>Atomic statements within PARA/ENDPARA are executed in any order.</p>	<pre>PARA PRINT "hello " PRINT "world " ENDPARA</pre> <p>Output possibility 1: hello world possibility 2: world hello</p>

<p>Statements defined in a function that is called within the PARA/ENDPARA block are executed sequentially.</p>	<pre> DEFINE print() PRINT "hi" PRINT "there" ENDDEF PARA print() ENDPARA Output hi there </pre>
<p>Statements defined in functions that are called within a PARA/ENDPARA block are executed in any order of interleaving with simple statements within the same PARA/ENDPARA block.</p>	<pre> DEFINE print() PRINT "hi" PRINT "there" ENDDEF PARA print() PRINT "world" ENDPARA Output possibility 1: world hi there possibility 2: hi world there possibility 3: hi there world </pre>
<p>Statements defined in two functions that are called within the same PARA/ENDPARA block are executed in any order of interleaving while statements from any one of the functions are executed in their order of definition.</p>	<pre> DEFINE print1() PRINT "hello" PRINTLN "world" ENDDEF DEFINE print2() PRINT "hi" PRINTLN "there" ENDDEF PARA print1() print2() ENDPARA Output possibility 1: hi there hello world possibility 2: hi hello there world possibility 3: hi hello world there possibility 4: hello hi there world possibility 5: hello hi world there possibility 6: hello world hi there </pre>
Shared Memory Concurrency	
<p>Exclusively Accessed Statement</p> <pre> EXC_ACC statement(s) END_EXC_ACC </pre>	<pre> x = 10 DEFINE changeX(diff) EXC_ACC x = x + diff END_EXC_ACC </pre>

<p>Only appears within a function definition.</p> <p>When one function call modifies a variable within an EXC_ACC/END_EXC_ACC block, statements in other function calls that read or modify the same variable may not be executed until the first function call completes its statement or executes a WAIT function.</p>	<pre> ENDDDEF PARA changeX(1) changeX(-2) ENDPARA PRINTLN x Output 9 </pre>
<p>Wait and Notify Functions</p> <p>WAIT() NOTIFY()</p> <p>Only be called inside a EXC_ACC/END_EXC_ACC block.</p> <p>Once a WAIT() function starts execution, another function call that reads or modifies variables inside the EXC_ACC/END_EXC_ACC block may execute.</p> <p>Once a NOTIFY() function is executed, all WAIT() functions finish their execution.</p> <p>Both WAIT() and NOTIFY() functions are atomic.</p>	<pre> x = 10 DEFINE changeX(diff) EXC_ACC WHILE x + diff < 0 DO WAIT() ENDWHILE x = x + diff NOTIFY() END_EXC_ACC ENDDDEF PARA changeX(-11) changeX(1) ENDPARA PRINTLN x Output 0 </pre>
Message Passing Concurrency	
<p>Message Variable</p> <p>MESSAGE.<i>message-name</i>(value...)</p> <p>A special message variable that carries a collection of values. The <i>message-name</i> is used to distinguish message variables from one another.</p>	<pre> m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world") </pre>
<p>Send Statement</p> <p>Send(<i>message variable</i>).To(<i>object</i>)</p> <p>Send a message specified by message variable to a receiver object.</p> <p>A send statement is asynchronous, which means that the order in which messages are received may differ from the order in which they were sent.</p>	<pre> m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world") Send(m1).To(r1) Send(m2).To(r1) </pre>
<p>Receive Statement</p> <p>ON_RECEIVING <i>message</i> <i>statement(s)</i> <i>message</i> <i>statement(s)</i> ...</p> <p>Accept the next message and execute <i>statement(s)</i> according to the type of the</p>	<pre> CLASS Receiver DEFINE receive ON_RECEIVING MESSAGE.h(var) PRINT var MESSAGE.w(var) PRINTLN var ENDDDEF ENDCLASS </pre>

<pre>message.</pre>	<pre>m1 = MESSAGE.h("hello") m2 = MESSAGE.w("world") r1 = new Receiver() r1.receive() Send(m1).To(r1) Send(m2).To(r1)</pre> <p>Output possibility1: hello world possibility2: world hello</p>
<p>Self Value</p> <pre>self</pre> <p>This value can be carried by a message or be used in the To portion of a Send statement to indicate the creator of a message or the executor of a send statement.</p>	<pre>CLASS Receiver DEFINE receive ON_RECEIVING MESSAGE.h(var) PRINTLN var Send(MESSAGE.h(var)) .To(self) ENDDDEF ENDCLASS m1 = MESSAGE.h("hello") r1 = new Receiver() r1.receive() Send(m1).To(r1)</pre> <p>Output hello hello ...</p>

FIGURE 35 PSEUDOCODE GUIDE FOR SPRING 2013 STUDY

<pre>// Ornamental Garden CLASS Garden DEFINE initialize Garden population = 0 ENDDDEF DEFINE enter() EXC_ACC population = population + 1 END_EXC_ACC ENDDDEF ENDCLASS CLASS Turnstile DEFINE initialize Turnstile(gardenVal) garden = gardenVal ENDDDEF DEFINE run() count = 20 WHILE count > 0 DO garden.enter() count = count - 1 ENDWHILE ENDDDEF ENDCLASS</pre>
--


```

garden = new Garden()
east = new Turnstile(garden)
west = new Turnstile(garden)

PARA
    east.run()
    west.run()
ENDPARA

// Bank Account
CLASS Account
    DEFINE initialize Account()
        balance = 0
    ENDDF

    DEFINE deposit(amount)
        EXC_ACC
            balance = balance + amount
            NOTIFY()
        END_EXC_ACC
    ENDDF

    DEFINE withdraw(amount)
        EXC_ACC
            WHILE balance < amount DO
                WAIT()
            ENDWHILE
            balance = balance - amount
            NOTIFY()
        END_EXC_ACC
    ENDDF
ENDCLASS

CLASS Customer
    DEFINE initialize Customer(accountVal)
        account = accountVal
    ENDDF

    DEFINE run()
        WHILE True DO
            IF (randNum(0,1) == 0) THEN
                account.deposit(randNum(1,1000))
            ELSE
                account.withdraw(randNum(1,1000))
            ENDIF
        ENDWHILE
    ENDDF
ENDCLASS

account = new Account()
customer1 = new Customer(account)
customer2 = new Customer(account)

PARA
    customer1.run()
    customer2.run()
END_PARA

// Large Printing Job
CLASS Printer
    DEFINE initialize Printer()
        working = False

```

```

ENDDEF

DEFINE require()
  EXC_ACC
  WHILE working DO
    WAIT()
  ENDWHILE
  working = True
END_EXC_ACC
ENDDEF

DEFINE release()
  EXC_ACC
  working = False
  NOTIFY()
END_EXC_ACC
ENDDEF
ENDCLASS

CLASS Terminal
  DEFINE initialize Terminal(printer1Val, printer2Val)
    printer1 = printer1Val
    printer2 = printer2Val
  ENDEF

  DEFINE run()
    WHILE True DO
      printer1.require()
      printer2.require()
      // printing
      printer1.release()
      printer2.release()
    ENDWHILE
  ENDEF
ENDCLASS

printer1 = new Printer()
printer2 = new Printer()
terminal1 = new Terminal(printer1, printer2)
terminal2 = new Terminal(printer1, printer2)

PARA
  terminal1.run()
  terminal2.run()
END_PARA

// Bounded Buffer
CLASS Buffer
  DEFINE initialize Buffer(capacityVal)
    items = []
    capacity = capacityVal
  ENDEF

  DEFINE produce(itemVal)
    EXC_ACC
    WHILE length(items) > capacity DO
      WAIT()
    ENDWHILE
    items[length(items)] = itemVal
    NOTIFY()
  END_EXC_ACC
ENDDEF

```

```

    DEFINE consume()
        EXC_ACC
        WHILE length(items) < 1 DO
            WAIT()
        ENDWHILE
        item = items[0]
        del items[0]
        NOTIFY()
    END_EXC_ACC
    return item
ENDDEF
ENDCLASS

CLASS Producer
    DEFINE initialize Producer(bufferVal)
        buffer = bufferVal
    ENDDEF

    DEFINE run()
        WHILE True DO
            buffer.produce(randNum(0,10))
        ENDWHILE
    ENDDEF
ENDCLASS

CLASS Consumer
    DEFINE initialize Consumer(bufferVal)
        buffer = bufferVal
    ENDDEF

    DEFINE run()
        WHILE True DO
            PRINTLN buffer.consume()
        ENDWHILE
    ENDDEF
ENDCLASS

// Dining Philosopher
CLASS Forks
    DEFINE initialize Forks(numVal)
        num = numVal
        forks = []
        WHILE num > 0 DO
            add(forks.size, True)
            num = num - 1
        ENDWHILE
    ENDDEF

    DEFINE getLeftFork(id)
        EXC_ACC
        WHILE !forks[id] DO
            WAIT()
        ENDWHILE
        forks[id] = False
    END_EXC_ACC
    ENDDEF

    DEFINE getRightFork(id)
        EXC_ACC
        WHILE !forks[(id+1)%forks.size] DO
            WAIT()
        ENDWHILE
        forks[(id+1)%forks.size] = False

```

```

        END_EXC_ACC
    ENDDDEF

    DEFINE putLeftFork(id)
        EXC_ACC
        forks[id] = True
        NOTIFY()
    END_EXC_ACC
    ENDDDEF

    DEFINE putRightFork(id)
        EXC_ACC
        forks[(id+1)%forks.size] = True
        NOTIFY()
    END_EXC_ACC
    ENDDDEF
ENDCLASS

CLASS Philosopher
    DEFINE initialize Philosopher(idVal, forksVal)
        id = idVal
        forks = forksVal
    ENDDDEF

    DEFINE eat()
        IF id % 2 == 0 THEN
            forks.getLeftFork(id)
            forks.getRightFork(id)
        ELSE
            forks.getRightFork(id)
            forks.getLeftFork(id)
        ENDIF
    ENDDDEF

    DEFINE think()
        forks.putLeftFork(id)
        forks.putRightFork(id)
    ENDDDEF

    DEFINE run()
        WHILE True DO
            eat()
            think()
        ENDWHILE
    ENDDDEF
ENDCLASS

// Readers Writers
CLASS Database
    DEFINE initialize Database
        numReader = 0
        writing = False
        writerWait = 0
        readerWait = 0
        readTurn = False
    ENDDDEF

    DEFINE acquireRead()
        EXC_ACC
        readerWait = readerWait + 1
        WHILE writing OR (writerWait > 0 AND (NOT readTurn)) DO
            WAIT()
        ENDWHILE

```

```

        readerWait = readerWait - 1
        numReader = numReader + 1
    END_EXC_ACC
ENDDEF

DEFINE releaseRead()
    EXC_ACC
        numReader = numReader - 1
        readTurn = false
        IF numReader == 0 THEN
            NOTIFY()
        ENDIF
    END_EXC_ACC
ENDDEF

DEFINE acquireWrite()
    EXC_ACC
        writerWait = writerWait + 1
        WHILE (writing OR numReader > 0) OR (readerWait > 0 AND readTurn) DO
            WAIT()
        ENDWHILE
        writerWait = writerWait - 1
        writing = True
    END_EXC_ACC
ENDDEF

DEFINE releaseWrite()
    EXC_ACC
        writing = False
        readTurn = True
        NOTIFY()
    END_EXC_ACC
ENDDEF
ENDCLASS

CLASS Reader
    DEFINE initialize Reader(databaseVal)
        database = databaseVal
    ENDDF

    DEFINE run()
        WHILE True DO
            database.acquireRead()
            database.releaseRead()
        ENDWHILE
    ENDDF
ENDCLASS

CLASS Writer
    DEFINE initialize Writer(databaseVal)
        database = databaseVal
    ENDDF

    DEFINE run()
        WHILE True DO
            database.acquireWrite()
            database.releaseWrite()
        ENDWHILE
    ENDDF
ENDCLASS

// Book Inventory
CLASS Inventory

```

```

DEFINE initialize Inventory()
    stock = [0, 0, 0, 0] // suppose only 4 kinds of books exist
ENDDEF

DEFINE restock(idx, quantity)
    EXC_ACC
        stock[idx] = stock[idx] + quantity
    NOTIFY()
    END_EXC_ACC
ENDDEFINE

DEFINE ship(idx, quantity)
    EXC_ACC
        IF stock[idx] < quantity THEN
            RETURN False
        ELES
            stock[idx] = stock[idx] - quantity
            RETURN True
        ENDIF
    END_EXC_ACC
ENDDEF
ENDCLASS

CLASS Job {
    DEFINE initialize Job(idxsVal, quantitiesVal, typeVal)
        indexes = idxsVal
        quantities = quantitiesVal
        type = typeVal
    ENDDEF

    DEFINE getIdxes()
        RETURN indexes
    ENDDEF

    DEFINE getQuantities()
        RETURN quantities
    ENDDEF

    DEFINE getType()
        RETURN type
    ENDDEF
}

CLASS JobQueue
    DEFINE initialize Jobs()
        jobs = []
        MAX_NUM_JOBS = 100
    ENDDEF

    DEFINE addJob(job)
        EXC_ACC
            WHILE length(jobs) >= MAX_NUM_JOBS DO
                WAIT()
            ENDWHILE
            jobs[length(jobs)] = job
            NOTIFY()
        END_EXC_ACC
    ENDDEF

    DEFINE getJob()
        EXC_ACC
            WHILE length(jobs) <= 0 DO
                WAIT()
            ENDWHILE
        END_EXC_ACC
    ENDDEF

```

```

        ENDWHILE
        job = jobs[0]
        del(jobs[0])
        NOTIFY()
    END_EXC_ACC
    RETURN job
ENDDEF
ENDCLASS

CLASS Worker
    DEFINE initialize Worker(inventoryVal, jobQueueVal)
        inventory = inventoryVal
        jobQueue = jobQueueVal
    ENDDEF

    DEFINE run()
        WHILE True DO
            job = jobQueue.getJob()
            IF job.getType() THEN
                i = 0
                WHILE i < length(job.getIndex()) DO
                    inventory.restock(
                        job.getIdxes()[i], job.getQuantities()[i])
                    i = i + 1
                ENDWHILE
            ELSE
                i = 0
                WHILE i < length(job.getIdxes()) AND
                    inventory.ship(
                        job.getIdxes()[i], job.getQuantities()[i])
                DO
                    i = i + 1
                ENDWHILE
                IF i != length(job.getIdxes()) THEN
                    WHILE i > 0 DO
                        inventory.restock(
                            job.getIdxes()[i-1], job.getQuantities()[i-1])
                        i = i - 1
                    ENDWHILE
                    jobQueue.addJob(job)
                ENDIF
            ENDIF
        ENDWHILE
    ENDDEF
ENDCLASS

CLASS Requester
    DEFINE initialize(jobQueueVal)
        jobQueue = jobQueueVal
    ENDDEF

    DEFINE run()
        WHILE True DO
            joblength = randNum(1, 5)
            indexes = []
            quantities = []
            WHILE joblength > 0 DO
                add(indexes[length(indexes)-1], randNum(0, 3))
                add(quantities[length(quantities)-1], randNum(10, 50))
                joblength = joblength - 1
            ENDWHILE
            jobQueue.addJob(new Job(indexes, quantities, False))
        ENDWHILE
    ENDDEF
ENDCLASS

```

```

    ENDDDEF
ENDCLASS

CLASS Restocker
    DEFINE initialize(jobQueueVal)
        jobQueue = jobQueueVal
    ENDDDEF

    DEFINE run()
        WHILE True DO
            joblength = randNum(1, 5)
            indexes = []
            quantities = []
            WHILE joblength > 0 DO
                add(indexes[length(indexes)-1], randNum(0, 3))
                add(quantities[length(quantities)-1], randNum(10, 50))
                joblength = joblength - 1
            ENDWHILE
            jobQueue.addJob(new Job(indexes, quantities, True))
        ENDWHILE
    ENDDDEF
ENDCLASS

// Single-Lane Bridge
CLASS Bridge
    DEFINE initialize Bridge()
        redEntered = 0
        redExited = 0
        blueEntered = 0
        blueExited = 0
    ENDDDEF

    DEFINE redEnter()
        EXC_ACC
        WHILE (blueEntered - blueExited) > 0 DO
            WAIT()
        ENDWHILE
        redEntered = redEntered + 1
    END_EXC_ACC
    RETURN redEntered
    ENDDDEF

    DEFINE redExit(orderVal)
        EXC_ACC
        WHILE redExited != (orderVal - 1) DO
            WAIT()
        ENDWHILE
        PRINTLN "red ", orderVal
        redExited = redExited + 1
        NOTIFY()
    END_EXC_ACC
    ENDDDEF

    DEFINE blueEnter()
        EXC_ACC
        WHILE (redEntered - redExited) > 0 DO
            WAIT()
        ENDWHILE
        blueEntered = blueEntered + 1
    END_EXC_ACC
    RETURN blueEntered
    ENDDDEF

```



```

    DEFINE blueExit(orderVal)
        EXC_ACC
        WHILE blueExited != (orderVal - 1) DO
            WAIT()
        ENDWHILE
        PRINTLN "blue ", orderVal
        blueExited = blueExited + 1
        NOTIFY()
    END_EXC_ACC
    ENNDEF
ENDCLASS

CLASS RedCar
    DEFINE initialize RedCar(bridgeVal)
        bridge = bridgeVal
    ENNDEF

    DEFINE run()
        WHILE True DO
            order = bridge.redEnter()
            bridge.redExit(order)
        ENDWHILE
    ENNDEF
ENDCLASS

CLASS blueCar
    DEFINE initialize BlueCar(bridgeVal)
        bridge = bridgeVal
    ENNDEF

    DEFINE run()
        WHILE True DO
            order = bridge.blueEnter()
            bridge.blueExit(order)
        ENDWHILE
    ENNDEF
ENDCLASS

// Sleeping Barber
CLASS Barber
    DEFINE initialize Barber
        work = False
        customerWait = 0
    ENNDEF

    DEFINE inquire()
        EXC_ACC
        IF (customerWait >= 3) THEN
            RETURN False
        ELSE
            customerWait = customerWait + 1
            RETURN True
        ENDIF
    END_EXC_ACC
    ENNDEF

    DEFINE barber()
        EXC_ACC
        WHILE work DO
            WAIT()
        ENDWHILE
        work = True
        customerWait = customerWait - 1

```

```

        END_EXC_ACC
    ENDDDEF

    DEFINE finish()
        EXC_ACC
        work = False
        NOTIFY()
        END_EXC_ACC
    ENDDDEF
ENDCLASS

CLASS Customer
    DEFINE initialize Customer(barberVal)
        barber = barberVal
    ENDDDEF

    DEFINE run() {
        WHILE True DO
            IF barber.inquire() THEN
                barber.barber()
                barber.finish()
            ENDIF
        ENDWHILE
    ENDDDEF
ENDCLASS

```

FIGURE 36 PSEUDOCODE IMPLEMENTATIONS OF SHARED MEMORY PROGRAMS FOR SPRING 2013 STUDY

```

// Ornamental Garden
CLASS Garden
    DEFINE initialize Garden
        population = 0
    ENDDF

    DEFINE start()
        WHILE True DO
            ON_RECEIVING
                Message.enter()
                population = population + 1
            ENDWHILE
        ENDDF
    ENDClass

CLASS Turnstile
    DEFINE initialize Turnstile(gardenVal)
        garden = gardenVal
    ENDDF

    DEFINE start()
        count = 20
        WHILE count > 0 DO
            Send(MESSAGE.enter()).To(garden)
            count = count - 1
        ENDWHILE
    ENDDF
    ENDClass

garden = new Garden()
east = new Turnstile(garden)
west = new Turnstile(garden)

PARA
    east.start()
    west.start()
ENDPARA

// Bank Account
CLASS Accountant
    DEFINE initialize Accountant()
        balance = 0
    ENDDF

    DEFINE start()
        WHILE True DO
            ON_RECEIVING
                MESSAGE.deposit(customer, amount)
                balance = balance + amount
                Send(MESSAGE.succeed()).To(customer)

                MESSAGE.withdraw(customer, amount)
                IF balance > amount THEN
                    balance = balance - amount
                    Send(MESSAGE.succeed()).To(customer)
                ELSE
                    Send(MESSAGE.fail()).To(customer)
                ENDIF
            ENDWHILE
        ENDDF
    ENDClass

CLASS Customer

```

```

DEFINE initialize Customer(accountantVal)
    accountant = accountantVal
ENDDEF

DEFINE nextRequest()
    IF randNum(0,1) == 0 THEN
        RETURN MESSAGE.deposit(self, randNum(0,1000))
    ELSE
        RETURN MESSAGE.withdraw(self, randNum(0,1000))
    ENDIF
ENDDEF

DEFINE start()
    message = nextRequest()
    Send(message).To(accountant)
    WHILE True DO
        ON_RECEIVING
            MESSAGE.succeed()
            message = nextRequest()
            Send(message).To(accountant)

            MESSAGE.fail()
            Send(message).To(accountant)
        ENDWHILE
    ENDDEF
ENDCLASS

accountant = new Accountant()
customer1 = new Customer(accountant)
customer2 = new Customer(accountant)

PARA
    accountant.start()
    customer1.start()
    customer2.start()
END_PARA

// Large Printing Job
CLASS Printer
    DEFINE initialize Printer()
        working = False
    ENDDEF

    DEFINE start()
        WHILE True DO
            ON_RECEIVING
                MESSAGE.require(terminal)
                IF working THEN
                    Send(MESSAGE.fail()).To(terminal)
                ELSE
                    working = True
                    Send(MESSAGE.succeed()).To(terminal)
                ENDIF

                MESSAGE.release()
                working = False
            ENDWHILE
        ENDDEF
    ENDCLASS

CLASS Terminal
    DEFINE initialize Terminal(printer1Val, printer2Val)
        printer1 = printer1Val

```

```

        printer2 = printer2Val
        status = 0
    ENDDDEF

    DEFINE start()
        Send(MESSAGE.require(self)).To(printer1)
        WHILE True DO
            ON_RECEIVING
                MESSAGE.succeed()
                IF status = 0 THEN
                    Send(MESSAGE.require(self)).To(printer2)
                    status = 1
                ELSE
                    // printing
                    Send(MESSAGE.release()).To(printer1)
                    Send(MESSAGE.release()).To(printer2)
                    status = 0
                ENDIF
                MESSAGE.fail()
                Send(MESSAGE.require(self)).To(printer1)
            ENDWHILE
        ENDDDEF
    ENDCCLASS

printer1 = new Printer()
printer2 = new Printer()
terminal1 = new Terminal(printer1, printer2)
terminal2 = new Terminal(printer1, printer2)

PARA
    printer1.start()
    printer2.start()
    terminal1.start()
    terminal2.start()
END_PARA

// Bounded Buffer
CLASS Buffer
    DEFINE initialize Buffer(capacityVal)
        items = []
        capacity = capacityVal
    ENDDDEF

    DEFINE start()
        WHILE True DO
            ON_RECEIVING
                MESSAGE.produce(item, producer)
                IF items.size <= capacity THEN
                    items[size] = item
                    Send(MESSAGE.succeed()).To(producer)
                ELSE
                    Send(MESSAGE.fail()).To(producer)
                ENDIF

                MESSAGE.consume(consumer)
                IF items.size > 0 THEN
                    Send(MESSAGE.cargo(items[0])).To(consumer)
                    del items[0]
                ELSE
                    Send(MESSAGE.fail()).To(consumer)
                ENDIF
            ENDWHILE
        ENDDDEF

```

```

ENDCLASS

CLASS Producer
  DEFINE initialize Producer(bufferVal)
    buffer = bufferVal
  ENDDF

  DEFINE start()
    item = randNum(0, 10)
    Send(MESSAGE.produce(item, self)).To(buffer)
    WHILE True DO
      ON_RECEIVING
        MESSAGE.succeed()
        item = randNum(0, 10)
        Send(MESSAGE.produce(item, self)).To(buffer)

        MESSAGE.fail()
        Send(MESSAGE.produce(item, self)).To(buffer)
      ENDWHILE
    ENDDF
ENDCLASS

CLASS Consumer
  DEFINE initialize Consumer(bufferVal)
    buffer = bufferVal
  ENDDF

  DEFINE start()
    Send(MESSAGE.consume(self))
    WHILE True DO
      ON_RECEIVING
        MESSAGE.cargo(item)
        PRINTLN item
        Send(MESSAGE.consume(self))

        MESSAGE.fail()
        Send(MESSAGE.consume(self))
      ENDWHILE
    ENDDF
ENDCLASS

// Dining Philosopher
CLASS Forks
  DEFINE initialize Forks(numVal)
    num = numVal
    forks = []
    WHILE num > 0 DO
      add(forks[forks.size], True)
      num = num - 1
    ENDWHILE
  ENDDF

  DEFINE start()
    WHILE True DO
      ON_RECEIVING
        MESSAGE.requireLeft(phil, id)
        IF forks[id] THEN
          Send(MESSAGE.succeedLeft()).TO(phil)
        ELSE
          Send(MESSAGE.failLeft()).TO(phil)
        ENDIF

        MESSAGE.requireRight(phil, id)

```

```

        IF forks[(id+1)%forks.size] THEN
            Send(MESSAGE.succeedRight()).TO(phil)
        ELSE
            Send(MESSAGE.failRight()).TO(phil)
        ENDIF

        MESSAGE.relinquishLeft(id)
        forks[id] = True

        MESSAGE.relinquishRight(id)
        forks[(id+1)%forks.size] = True
    ENDWHILE
ENDDEF
ENDCLASS

CLASS Philosopher
    DEFINE initialize Philosopher(idVal, forksVal)
        id = idVal
        forks = forksVal
    ENDDEF

    DEFINE getFirst()
        IF id % 2 == 0 THEN
            Send(MESSAGE.requireLeft(self, id)).TO(forks)
        ELSE
            Send(MESSAGE.requireRight(self, id)).TO(forks)
        ENDIF
    ENDDEF

    DEFINE getSecond()
        IF id % 2 == 0 THEN
            Send(MESSAGE.requireRight(self, id)).TO(forks)
        ELSE
            Send(MESSAGE.requireLeft(self, id)).TO(forks)
        ENDIF
    ENDDEF

    DEFINE think()
        Send(MESSAGE.relinquishLeft(id)).TO(forks)
        Send(MESSAGE.relinquishRight(id)).TO(forks)
    ENDDEF

    DEFINE start()
        getFirst()
        WHILE True DO
            ON_RECEIVING
                MESSAGE.succeedLeft()
                IF id % 2 == 0 THEN
                    getSecond()
                ELSE
                    think()
                    getFirst()
                ENDIF

                MESSAGE.failLeft()
                IF id % 2 == 0 THEN
                    getFirst()
                ELSE
                    getSecond()
                ENDIF

                MESSAGE.succeedRight()
                IF id % 2 == 0 THEN

```

```

        think()
        getFirst()
    ELSE
        getSecond()
    ENDIF

    MESSAGE.failRight()
    IF id % 2 == 0 THEN
        getSecond()
    ELSE
        getFirst()
    ENDIF
ENDWHILE
ENDDEF
ENDCLASS

// Readers Writers
CLASS Database
    DEFINE initialize Database
        numReader = 0
        writing = False
        writerWait = 0
        readerWait = 0
        readTurn = False
    ENDEF

    DEFINE start()
        WHILE
            ON_RECEIVING
                MESSAGE.acquireRead(reader)
                IF (NOT writing) AND (writerWait == 0 OR readTurn) THEN
                    numReader = numReader + 1
                    Send(MESSAGE.succeedRead()).TO(reader)
                ELSE
                    readerWait = readerWait + 1
                    Send(MESSAGE.failRead()).TO(reader)
                ENDIF

                MESSAGE.releaseRead(reader)
                numReader = numReader - 1
                readTurn = False
                Send(MESSAGE.release()).TO(reader)

                MESSAGE.acquireWrite(writer)
                IF (NOT writing) AND (numReader == 0) AND (readerWait == 0 OR (NOT
readTurn)) THEN
                    writing = True
                    Send(MESSAGE.succeedWrite()).TO(writer)
                ELSE
                    writerWait = writerWait + 1
                    Send(MESSAGE.failWrite()).TO(writer)
                ENDIF

                MESSAGE.releaseWrite(writer)
                writing = False
                readTurn = True
                Send(MESSAGE.release()).TO(writer)
            ENDWHILE
        ENDEF
    ENDCLASS

CLASS Reader
    DEFINE initialize Reader(databaseVal)

```



```

        database = databaseVal
    ENDDDEF

    DEFINE start()
        Send(MESSAGE.acquireRead(self)).TO(database)
        WHILE True DO
            ON_RECEIVING
                MESSAGE.succeedRead()
                Send(MESSAGE.releaseRead(self)).TO(database)

                MESSAGE.failRead()
                Send(MESSAGE.acquireRead(self)).TO(database)

                MESSAGE.release()
                Send(MESSAGE.acquireRead(self)).TO(database)
            ENDWHILE
        ENDDDEF
    ENDCCLASS

    CLASS Writer
        DEFINE initialize Writer(databaseVal)
            database = databaseVal
        ENDDDEF

        DEFINE start()
            Send(MESSAGE.acquireWrite(self)).TO(database)
            WHILE True DO
                ON_RECEIVING
                    MESSAGE.succeedWrite()
                    Send(MESSAGE.releaseWrite(self)).TO(database)

                    MESSAGE.failWrite()
                    Send(MESSAGE.acquireWrite(self)).TO(database)

                    MESSAGE.release()
                    Send(MESSAGE.acquireWrite(self)).TO(database)
                ENDWHILE
            ENDDDEF
        ENDCCLASS

    // Sleeping Barber
    CLASS Barber
        DEFINE initialize Barber
            work = False
            customerWait = 0
        ENDDDEF

        DEFINE start()
            WHILE True DO
                ON_RECEIVING
                    MESSAGE.inquire(customer)
                    IF customerWait >= 3 THEN
                        customer ! MESSAGE.noSpace()
                    ELSE
                        customer ! MESSAGE.seat()
                        customerWait = customerWait + 1
                    ENDIF

                    MESSAGE.barber(customer)
                    IF work THEN
                        customer ! MESSAGE.seat()
                    ELSE
                        work = True
                    ENDIF
                ENDWHILE
            ENDDDEF
        ENDCCLASS

```

```

        customer ! MESSAGE.barbering()
    ENDIF

    MESSAGE.finish(customer)
    work = False
ENDWHILE
ENDDEF
ENDCLASS

CLASS Customer
    DEFINE initialize Customer(barberVal)
        barber = barberVal
    ENDDEF

    DEFINE start()
        barber ! MESSAGE.inquire(self)
        WHILE True DO
            ON_RECEIVING
                MESSAGE.onSpace()
                barber ! MESSAGE.inquire(self)

                MESSAGE.seat()
                barber ! barber(self)

                MESSAGE.barbering()
                barber ! MESSAGE.finsih()
                barber ! MESSAGE.inquire(self)
            ENDWHILE
        ENDDEF
    ENDCLASS

// Book Inventory
CLASS Inventory
    DEFINE initialize Inventory()
        stock = [0, 0, 0]
    ENDDEF

    DEFINE start
        WHILE True DO
            ON_RECEIVE
                MESSAGE.increase(idx, quantity)
                stock[idx] = stock[idx] + quantity

                MESSAGE.decrease(idx, quantity, worker)
                IF stock[idx] >= quantity THEN
                    stock[idx] = stock[idx] - quantity
                    Send(MESSAGE.invSucceed(idx, quantity)).To(worker)
                ELSE
                    Send(MESSAGE.invFail()).To(worker)
                ENDIF
            ENDWHILE
        ENDDEF
    ENDCLASS

CLASS Job {
    DEFINE initialize Job(idxesVal, quantitiesVal, typeVal)
        indexes = idxesVal
        quantities = quantitiesVal
        type = typeVal
    ENDDEF

    DEFINE getIdxes()
        RETURN index

```

```

ENDDEF

DEFINE getQuantities()
    RETURN quantity
ENDDEF

DEFINE getType()
    RETURN type
ENDDEF

}

CLASS JobQueue
    DEFINE initialize JobQueue()
        jobs = []
        MAX_NUM_JOBS = 100
    ENDDF

    DEFINE start()
        WHILE True DO
            ON_RECEIVE
                MESSAGE.request(job, sender)
                IF length(jobs) >= MAX_NUM_JOBS THEN
                    Send(MESSAGE.queFail()).To(sender)
                ELSE
                    jobs[length(jobs)] = job
                    Send(MESSAGE.queSucceed()).To(sender)
                ENDIF

                MESSAGE.retrieve(worker)
                IF length(jobs) > 0 THEN
                    Send(MESSAGE.job(jobs[0]).To(worker)
                del(jobs[0])
                ELSE
                    Send(MESSAGE.nojob()).To(worker)
                ENDIF
            ENDWHILE
        ENDDF
    ENDCCLASS

CLASS Worker
    DEFINE initialize Worker(inventoryVal, jobQueueVal)
        inventory = inventoryVal
        jobQueue = jobQueueVal
        succeedPool = []
        failCount = 0
        currentJob = NULL
        status = False // waiting for retrieve (not request) feedback
    ENDDF

    DEFINE start()
        Send(MESSAGE.retrieve(self)).To(jobQueue)
        WHILE True DO
            ON_RECEIVE
                MESSAGE.job(jobVal)
                currentJob = jobVal
                IF job.getType() THEN
                    i = 0
                    WHILE i < length(jobVal.getIdxes()) DO
                        Send(
                            MESSAGE.increase(jobVal.getIdxes()[i],
                                jobVal.getQuantities()[i]))
                            .To(inventory)
                        i = i + 1
                    
```

```

        ENDWHILE
    ELSE
        currentJobLength = length(jobVal)
        i = 0
        WHILE i < length(jobVal.getIdxes()) DO
            Send(
                MESSAGE.decrease(jobVal.getIdxes()[i],
                    jobVal.getQuantities()[i], self)
                    .To(inventory)
                i = i + 1
            )
        ENDWHILE

        MESSAGE.invSucceed(idx, quantity)
        add(succeedPool[length(succeedPool)], [idx, quantity])
        IF length(succeedPool) + failCount
            == length(currentJob)
        THEN
            IF failCount != 0 THEN
                i = 0
                WHILE i < length(succeedPool) DO
                    Send(
                        MESSAGE.increase(succeedPool[i][0], succeedPool[i][1])
                            .To(inventory)
                        i = i + 1
                    )
                ENDWHILE
                succeedPool = []
                failCount = 0
                Send(MESSAGE.request(currentJob, self))
                    .To(jobQueue)
                status = True
            ELSE
                Send(MESSAGE.retrieve(self)).To(jobQueue)
            ENDIF
        ENDIF

        MESSAGE.invFail()
        failCount = failCount + 1
        IF length(succeedPool) + failCount
            == length(currentJob)
        THEN
            i = 0
            WHILE i < length(succeedPool) DO
                Send(
                    MESSAGE.increase(succeedPool[i][0], succeedPool[i][1])
                        .To(inventory)
                    i = i + 1
                )
            ENDWHILE
            succeedPool = []
            failCount = 0
            Send(MESSAGE.request(currentJob, self)).To(jobQueue)
            status = True
        ENDIF

        MESSAGE.nojob()
        Send(MESSAGE.retrieve(self)).To(jobQueue)

        MESSAGE.queSucceed()
        status = False
        Send(MESSAGE.retrieve(self)).To(jobQueue)

        MESSAGE.queFail()
        Send(MESSAGE.request(currentJob, self)).To(jobQueue)
    ENDWHILE

```

```

    ENDDDEF
ENDCLASS

CLASS Requester
    DEFINE initialize Requester(jobQueueVal)
        jobQueue = jobQueueVal
    ENDDDEF

    DEFINE nextJob()
        joblength = randNum(1, 5)
        indexes = []
        quantities = []
        WHILE joblength > 0 DO
            add(indexes[length(indexes)-1], randNum(0, 3))
            add(quantities[length(quantities)-1], randNum(10, 50))
            joblength = joblength - 1
        ENDWHILE
        RETURN (new Job(indexes, quantities, False))
    ENDDDEF

    DEFINE start()
        job = nextJob()
        Send(MESSAGE.request(job, self).To(jobQueue))
        WHILE True DO
            ON_RECEIVING
                MESSAGE.queSucceed()
                job = nextJob()
                Send(MESSAGE.request(job, self).To(jobQueue))

                MESSAGE.queFail()
                Send(MESSAGE.request(job, self).To(jobQueue))
            ENDWHILE
        ENDDDEF
ENDCLASS

CLASS Restocker
    DEFINE initialize Restocker(inventoryVal)
        inventory = inventoryVal
    ENDDDEF

    DEFINE nextJob()
        joblength = randNum(1, 5)
        indexes = []
        quantities = []
        WHILE joblength > 0 DO
            add(indexes[length(indexes)-1], randNum(0, 3))
            add(quantities[length(quantities)-1], randNum(10, 50))
            joblength = joblength - 1
        ENDWHILE
        RETURN (new Job(indexes, quantities, True))
    ENDDDEF

    DEFINE start()
        job = nextJob()
        Send(MESSAGE.request(job, self).To(jobQueue))
        WHILE True DO
            ON_RECEIVING
                MESSAGE.queSucceed()
                job = nextJob()
                Send(MESSAGE.request(job, self).To(jobQueue))

                MESSAGE.queFail()
                Send(MESSAGE.request(job, self).To(jobQueue))
            ENDWHILE
        ENDDDEF
ENDCLASS

```

```

        ENDWHILE
    ENDDEF
ENDCLASS

// Single-Lane Bridge
CLASS Bridge
    DEFINE initialize Bridge()
        redEntered = 0
        redExited = 0
        blueEntered = 0
        blueExited = 0
    ENDDEF

    DEFINE start()
        WHILE True DO
            ON_RECEIVING
                MESSAGE.redEnter(red)
                IF (blueEntered - blueExited) == 0 THEN
                    redEntered = redEntered + 1
                    Send(MESSAGE.succeedEnter(redEntered)).To(red)
                ELSE
                    Send(MESSAGE.failEnter()).To(red)
                ENDIF

                MESSAGE.redExit(red, order)
                IF order == (redExited + 1) THEN
                    redExited = redExited + 1
                    Send(MESSAGE.succeedExit(order)).To(red)
                ELSE
                    Send(MESSAGE.failExit(order)).To(red)
                ENDIF

                MESSAGE.blueEnter(blue)
                IF (redEntered - redExited) == 0 THEN
                    blueEntered = blueEntered + 1
                    Send(MESSAGE.succeedEnter(blueEntered)).To(blue)
                ELSE
                    Send(MESSAGE.failEnter()).To(blue)
                ENDIF

                MESSAGE.blueExit(blue, order)
                IF order == (blueExited + 1) THEN
                    blueExited = blueExited + 1
                    Send(MESSAGE.succeedExit(order)).To(blue)
                ELSE
                    Send(MESSAGE.failExit(order)).To(blue)
                ENDIF
            ENDWHILE
        ENDDEF
ENDCLASS

CLASS RedCar
    DEFINE initialize RedCar(bridgeVal)
        bridge = bridgeVal
    ENDDEF

    DEFINE start()
        Send(MESSAGE.redEnter(self)).To(bridge)
        WHILE True DO
            ON_RECEIVING
                MESSAGE.succeedEnter(order)
                Send(MESSAGE.redExit(self, order)).To(bridge)
            ENDWHILE
        ENDDEF
ENDCLASS

```

```

        MESSAGE.failEnter()
        Send(MESSAGE.redEnter(self)).To(bridge)

        MESSAGE.succeedExit(order)
        PRINTLN "red ", order
        Send(MESSAGE.redEnter(self)).To(bridge)

        MESSAGE.failExit(order)
        Send(MESSAGE.redExit(self, order)).To(bridge)
    ENDWHILE
ENDDEF
ENDCLASS

CLASS BlueCar
    DEFINE initialize BlueCar(bridgeVal)
        bridge = bridgeVal
    ENDDF

    DEFINE start()
        Send(MESSAGE.blueEnter(self)).To(bridge)
        WHILE True DO
            ON_RECEIVING
                MESSAGE.succeedEnter(order)
                Send(MESSAGE.blueExit(self, order)).to(bridge)

                MESSAGE.failEnter()
                Send(MESSAGE.blueEnter(self)).to(bridge)

                MESSAGE.succeedExit(order)
                PRINTLN "blue ", order
                Send(MESSAGE.blueEnter(self)).To(bridge)

                MESSAGE.failExit(order)
                Send(MESSAGE.blueExit(self, order)).to(bridge)
            ENDWHILE
        ENDDEF
    ENDCLASS

```

FIGURE 37 PSEUDOCODE IMPLEMENTATIONS OF MESSAGE PASSING PROGRAMS FOR SPRING 2013 STUDY

CSCI 4900 Lab/Project Specification (Week 2)

Lab #1

Observing Multi-core Architecture

Goals

In this project, you will use your laptop to run two concurrent programs, the dining philosopher and thread pool. You are given the runnable jar files of these two programs and required to monitor the system performance while running these two programs. The goal of this lab is to allow you experience the multi-core computer systems accessible on common laptops and explore basic performance monitoring tools available with different operating systems.

Due Date

Jan 22, 2013 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab. You have to finish this lab on your own.

Lab Description

Step 1:

Dining Philosopher is a famous concurrent problem. Search online or use other resources to understand the problem. Write a short description (within 200 words) of the problem **in your own word**.

Step 2:

Read following resources according to the type of your laptop's system and familiarize yourself with corresponding performance monitoring tools.

Mac

Mac Performance Guide:

<http://macperformanceguide.com/Mac-MonitoringTips.html>

Apple Support:

<http://support.apple.com/kb/HT1342>

Windows

Windows Support:

<http://windows.microsoft.com/en-US/windows-vista/See-details-about-your-computers-performance-using-Task-Manager>

Linux/Unix

Top Command:

"man top"

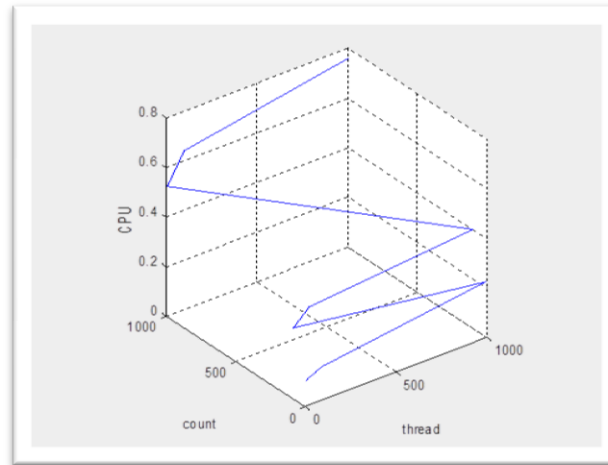
Report your system's property, including processor, memory and operating system.

Run **dining.jar** (explore online if you don't know how to execute a runnable jar file) and observe the change of CPU and memory usage while running the program. The program takes two arguments, the number of philosophers (i.e. the number of forks) and the number of meals each of them will dine.

Snapshot a CPU and memory usage picture when running the program with 5 philosophers that each dining for 1,000,000 times.

Step 3:

You are given a thread pool program that simply spawns a specified number of threads and each count down from a given number. Run **pool.jar**, change the arguments of the program (number of threads and the number to count down from) and record the CPU usage and execution time (reported by the program itself after finishing in milliseconds). Generate two graphs that correlate two arguments passed into the program with CPU usage and execution time respectively (figure 1 is an example of a graph that correlates two arguments with CPU usage with faked data).



Write a short conclusion (within 200 words) on how the two different arguments affect the CPU usage and execution time of the program.

Step 4:

Put all your work, including reports, graphs and answers to the questions together into one file **answer.pdf**.

Use the `mkdir` command to create a new folder (named "lastname_lab01") in your local Nike account. Use the `cp` command to copy **answer.pdf** to the newly created folder and submit the folder to cs4900. (`submit cs4900a lastname_lab01`)

Grading Rubric

Deliverables	Total	Points	Comments
Subjective Survey	5		
Answers	45		
step 1	15		
step 2	10		
step 3	20		

CSCI 4900 Lab/Project Specification (Week 3)

Lab #2**Modeling Book Inventory System Part I****Modeling Shared Memory Form with UML and Pseudocode****Goals**

During this course, you will work to design and implement a book inventory system using both shared memory and message passing approaches. In this lab project, you will finish the first part of understanding the criteria of the system and model it as a shared memory system with the pseudocode introduced in class.

Due Date

Feb. 1, 2013 (Friday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab. You must complete this lab on your own.

Lab Description**Step 1:**

Read and understand the following description of a book inventory system:

A book inventory system is a warehouse management system for the shipment and restocking of a book depository. The system processes two kinds of jobs, shipping and restocking. Multiple client programs are connected concurrently to this system and send these two jobs to the system for processing. The clients keep sending jobs to the system without knowledge of how previous jobs are fulfilled or the current status of the warehouse. The client only gets notification if the system is overloaded with too many jobs waiting to be processed and cannot take a specific job. In this case, the client should pause for a while and then re-send the previous job.

Both shipping and restocking jobs should be viewed as transactions, which means that a job is fulfilled either completely or not at all, i.e. a job should not be partially fulfilled. A shipping job may only involve decreasing the amount of one or more kinds of books (a shipping job ships books out of the warehouse). When the stock of any one type of book in the order is insufficient, the job should be automatically retried later by the system (the server). A restocking job may only involve increasing the amount of one or more kinds of books (a restocking job ships books into the warehouse) and this should always be fulfilled on the first attempt.

To increase the throughput (number of jobs processed in a certain amount of time) of this book inventory system, we would like the jobs to be processed concurrently in the system.

For those who enrolled in 4900 level, you could assume that each shipping or restocking job only involve one kind of book.

Step 2:

Model the book inventory system described above as a shared memory system with UML diagrams. You are free to choose and decide what data should be shared. But remember, the system is

closed in its whole. Client programs may only send job requirements and receive job rejection notifications, but never get to see or modify the system status.

Create a UML class diagram for each class as you designed, and provide information on "important" attributes. You need not list parameters of methods or utility methods.

Create a UML sequence diagram for each of the following important interactions:

- Sending a job with no other jobs waiting to be processed
- Sending a job when some jobs are waiting to be processed
- Sending a job when the system is overloaded by the number of waiting jobs
- Processing the only shipping job inside the system
- Processing a shipping job with another job concurrently inside the system
- Processing a restocking job with another job concurrently inside the system

Step 3:

Model the book inventory system with the pseudocode system. According to your UML modeling (especially class diagrams) in step 2, write pseudo code for each of the classes you designed. Write some statements to initialize and start the system with the following client program defined by pseudocode.

```

CLASS Client
  DEFINE initialize Client(inventoryVal)
    inventory = inventoryVal
  ENDDF

  DEFINE run()
    job = // some job, data structure depends on design
    WHILE True DO
      IF inventory.issue(job) THEN
        job = // some other job
      ELSE
        // pause for a while
        // Thread.sleep(100)
      ENDIF
    ENDWHILE
  ENDDF
ENDCLASS
  
```

Step 4:

Put all your work, including diagrams and pseudocode into one file **answer.pdf**.

Use the mkdir command to create a new folder (named "lastname_lab02") in your local Nike account. Use the cp command to copy **answer.pdf** to the newly created folder and submit the folder to cs4900. (submit cs4900a lastname_lab02)

Grading Rubric

Deliverables	Total	Points	Comments
--------------	-------	--------	----------

Subjective Survey	5		
Answers	95		
step 2	40		
UML class diagram	25		
UML sequence diagram	15		
step 3	45		
Pseudocode for classes	35		
Pseudocode for system initialization and start	10		

CSCI 4900 Homework Specification (Week 3)

Homework #2

Practice Using Pseudo Codes

Goals

In this homework, you will write pseudocode for several concurrency scenarios as discussed in class. You will also do some background research on the Therac-25 accident on your own and write a brief summary of the accident and its relation to race conditions.

The goal of this homework is to practice using the pseudocode system described in class to model several concurrency scenarios. Later in the course you will read and use this pseudocode to comprehend and represent more complicated programs.

Due Date

Jan 25, 2013 (Friday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual homework. You have to finish this homework on your own.

Homework Description

Task 1: Survey Therac-25 Accident

Therac-25 is a well-known medical radiation accident related to race conditions in the control software. Search online or use other resources to understand how the accident happened and how it was related to race conditions. Write a short description (within one page) of what you find **in your own words**.

Task 2: Write Pseudo Code for Sum & Worker System

Assume 100 workers exist in a sum & worker system. They each increment the sum value by 1 for 10 times. Use the pseudocode system we introduced in class (you may also refer to the pseudocode guide posted on ELC) to model this system. In your modeling, you should write out the definition of a

class Sum and a class Worker. You should also write some statements to initialize the sum and worker objects and start running thread objects.

Task 3: Write Pseudo Code for Bounded-Buffer System

A bounded buffer system is a system with a buffer object that only contains limited capacity to hold items. Producer and consumer threads produce and consume items to and from the buffer. Assume the buffer in the system has capacity of 10, i.e. it could hold a maximum of 10 items and the producers and consumers keep producing and consuming infinitely. Write out the pseudocode of this system. The definition of a class Buffer, a class Producer and a class Consumer should be included. You should also write some statements to initialize the buffer, two producers and two consumers and start running thread objects.

Hint:

- Use a list data structure to model the buffer object and use True/False values to denote the usage of a buffer slot.
- Use WAIT() and NOTIFY() functions appropriately for conditional synchronization.

Task 4: Write Pseudo Code for Dining Philosopher System

Write out the pseudo code for the dining philosopher problem that you surveyed in Lab 01. Assume five philosophers exist in the system (i.e. five forks exist and one between each pair of philosophers) and they shift between eating and thinking infinitely. The following pseudocode skeleton uses the asymmetry technique to solve the potential deadlock issue. Fill out the missing parts (comments) to finish the definition of fork and philosopher class.

Hint:

- A List data structure is used to denote the usage of the fork. A True value indicates that the fork with a particular index is available.
- Asymmetry is introduced in that philosophers with an even number ID pick up the fork on their left hand side first while those with an odd number ID pick up the fork on their right hand side first.

```

CLASS Forks
  DEFINE initialize Forks(numVal)
    num = numVal
    forks = []
    WHILE num > 0 DO
      // initialize value of forks here
    ENDWHILE
  ENDDF

  DEFINE getLeftFork(id)
    EXC_ACC
    // get the left fork for particular philosopher
    END_EXC_ACC
  ENDDF

  DEFINE getRightFork(id)
    EXC_ACC
    // get the right fork for particular philosopher
    END_EXC_ACC
  ENDDF

```

```

    DEFINE putLeftFork(id)
        EXC_ACC
        // put particular philosopher's left fork back
    END_EXC_ACC
    ENDDDEF

    DEFINE putRightFork(id)
        EXC_ACC
        // put particular philosopher's right fork back
    END_EXC_ACC
    ENDDDEF
ENDCLASS

CLASS Philosopher
    DEFINE initialize Philosopher(idVal, forksVal)
        id = idVal
        forks = forksVal
    ENDDDEF

    DEFINE eat()
        IF id % 2 == 0 THEN
            // pick up left fork first and then right one
        ELSE
            // pick up right fork first and then left one
        ENDIF
    ENDDDEF

    DEFINE think()
        // return both forks
    ENDDDEF

    DEFINE run()
        WHILE True DO
            eat()
            think()
        ENDWHILE
    ENDDDEF
ENDCLASS

```

Submission:

Put all your work, including summary and pseudo codes into one file **answer.pdf**. Use the `mkdir` command to create a new folder (named "lastname_hw01") in your local Nike account. Use the `cp` command to copy **answer.pdf** to the newly created folder and submit the folder to cs4900. (submit cs4900a lastname_hw01)

Grading Rubric

Deliverables	Total	Points	Comments
Subjective Survey	5		
Answers	45		
task 1	10		
task 2	10		
task 3	15		

task 4

10

CSCI 4900 Homework Specification (Week 4)

Homework #3

Practice Using Pseudo Codes for Message Passing Systems

Goals

In this homework, you will write pseudocode for several concurrency scenarios as discussed in class. The goal of this homework is to practice using the pseudocode system described in class to model several concurrency scenarios as message passing systems.

Due Date

Feb 4, 2013 (Monday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual homework. You have to finish this homework on your own.

No exchange or copy of pseudocode is allowed!

Homework Description

Task 1: Modeling Bounded Buffer as Message Passing System

If you have participated in class design discussion activity, write out the pseudocode that reflects the design of your team. Use comments to list all members in your design team at the first line of your pseudocode.

If you haven't participated in class design discussion activity, please refer to page #34-36 in Jan30's slides and write up answers to the discussion questions on your own. Then write out the pseudocode that reflects your design.

Task 2: Modeling Dining Philosophers as Message Passing System

If you have participated in class design discussion activity, write out the pseudocode that reflects the design of your team. Use comments to list all members in your design team at the first line of your pseudocode.

If you haven't participated in class design discussion activity, please refer to page #48-50 in Jan31's slides and write up answers to the discussion questions on your own. Then write out the pseudocode that reflects your design.

Submission:

Put all your work, including answers (if any) and pseudo codes into one file **answer.pdf**. Use the `mkdir` command to create a new folder (named "lastname_hw01") in your local Nike account. Use the `cp` command to copy **answer.pdf** to the newly created folder and submit the folder to cs4900. (`submit cs4900a lastname_hw03`)

Grading Rubric

Deliverables	Total	Points	Comments
Subjective Survey	5		
Answers	45		
task 1	20		
task 2	25		

CSCI 4900 Lab/Project Specification (Week 4)

Lab #3**Modeling a Book Inventory System, Part II****Modeling a Message Passing Solution with UML and Pseudocode****Goals**

Last week, you designed and modeled a book inventory system using a shared memory approach. In this lab project, you will model the same book inventory system, but using a message passing approach. You will again use UML and the pseudocode introduced in class.

Due Date

Feb. 7, 2013 (Thursday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab. You must complete this lab on your own.

No exchange or copying of pseudocode is allowed!

Lab Description**Step 1:**

Review the description of a book inventory system:

A book inventory system is a warehouse management system for recording shipments from and the restocking of a book depository. The system processes two kinds of jobs, shipping and restocking. Multiple client programs may be connected concurrently to this system and sending these two types jobs to the system for processing. The clients keep sending jobs to the system without knowledge of how previous jobs are fulfilled or the current status of the warehouse. The client only gets notification if the system is overloaded with too many jobs waiting to be processed and cannot take a specific job. In this case, the client should pause for a while and then re-send the previous job.

Both shipping and restocking jobs should be viewed as transactions, which means that a job is fulfilled either completely or not at all, i.e. a job should not be partially fulfilled. A shipping job may only involve decreasing the amount of one or more kinds of books (a shipping job ships books out of the warehouse). When the stock of any one type of book in the order is insufficient, the job should be

automatically retried later by the system (the server). A restocking job may only involve increasing the amount of one or more kinds of books (a restocking job ships books into the warehouse) and this should always be fulfilled on the first attempt.

To increase the throughput (number of jobs processed in a certain amount of time) of this book inventory system, we would like the jobs to be processed concurrently in the system.

Based on last week's project, you are likely now confident about how to realize this system as a shared memory system. Now, you must think about how to realize it as a message passing system. In another word, you are required to consider using only exchanged messages and corresponding behaviors of different entities to achieve the requirements of the system. Only private data but no shared data will be present in this message passing form of the system.

Step 2:

Model the book inventory system as a message passing system with UML diagrams. You are free to choose and decide what kind of messages should be exchanged. You are also free to design and decide what entities are involved in this system and their corresponding behaviors based on the messages being exchanged. But remember, the functionality and requirements of the system should not be different from its shared memory form.

Create a UML class diagram for each class you design (they may be actor entities or complex data structures), and provide information on "important" attributes. You need not list parameters of methods or utility methods.

Create a UML sequence diagram for each of the following important interactions:

- Sending a job with no other jobs waiting to be processed
- Sending a job when some jobs are waiting to be processed
- Sending a job when the system is overloaded by the number of waiting jobs
- Processing the only shipping job inside the system
- Processing a shipping job with another job concurrently inside the system
- Processing a restocking job with another job concurrently inside the system

Step 3:

Model the book inventory system with the pseudocode system. According to your UML modeling (especially class diagrams) in step 2, write pseudocode for each of the classes you designed. Write some statements to initialize and start the system with the following client program defined by pseudocode.

```

CLASS Client
  DEFINE initialize Client(inventoryVal)
    inventory = inventoryVal
  ENDDF

  DEFINE start()
    message = // some job message, details depends on design
    Send(message).To(inventory)
    WHILE True DO
      ON_RECEIVING
        MESSAGE.accept()
        message = // some other job message
    
```

```

        Send(message).To(inventory)

        MESSAGE.reject()
        // pause for a while
        Send(message).To(inventory)
    ENDWHILE
ENDDEF
ENDCLASS

```

Step 4:

Put all your work, including diagrams and pseudocode into one file **answer.pdf**.

Use the mkdir command to create a new folder (named “lastname_lab03”) in your local Nike account. Use the cp command to copy **answer.pdf** to the newly created folder and submit the folder to cs4900a. (submit cs4900a lastname_lab03)

Grading Rubric

Deliverables	Total	Points	Comments
Subjective Survey	5		
Answers	95		
step 2	50		
UML class diagram	25		
UML sequence diagram	25		
step 3	45		
Pseudocode for classes	35		
Pseudocode for system initialization and start	10		

CSCI 4900/6900 Lab/Project Specification (Week 5)

Lab #5**Practice Java Basics****Goals**

Before start our exploration in Java Thread model for concurrent programming, this lab is designed for you to warm-up with Java programming language by implementing some classic algorithms and reviewing data structures defined and used in Java. Students enrolled in CSCI 4900 should finish basic function definition, a backtracking algorithm and a class definition of remote keypad. Students enrolled in CSCI 6900 should finish an extra practice on defining a new data structure. This task is counted as a bonus for CSCI 4900 students.

Due Date

Feb. 12, 2013 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab. You must complete this lab on your own. **No copy of or detailed line by line cooperation on code is allowed.**

Lab Description**Step 1: Project Set Up**

Please create a Java project, name “lab05”, with Eclipse. Create a class “Main” in default package with a main method (copy and paste following codes).

```
class Main {
    public void static main(String args[]) {
        System.out.println("step2: print function");
        myPrint();

        System.out.println("step3: backtracking");
        EightQueenSolver solver = new EightQueenSolver();
        solver.solve();
        solver.printBoard();

        System.out.println("step4: remote keypad");
        RemoteKeypad keypad
            = new RemoteKeypad(Integer.parseInt(args[0]));
        for (int i = 1; i < args.length; i++) {
            keypad.getMoves(args[i]);
        }
    }
}
```

This step sets up the input and output for functions and classes defined in step 2 to step 4.

Step 2: A Simple Print Function

In the Main class defined in step 1, define another method *myPrint()* which prints out the following output:

```

      *
     ***
    *****
   ********
  *********
 *****
```

Step 3: A Backtracking Algorithm

An eight queen problem is to place eight queens on a chess board so that any two of them won't attack each other.

In the same default package, define a class, name “**EightQueenSolver**”. This class should have a two-dimensional array **board[][]** which records the placement of queens. A value of 0 indicates an empty cell in the board and a value 1 indicates a cell with a queen placed on it. Define following member functions:

`solve()` – use backtracking algorithm to solve the eight queen problem
`printBoard()` – print out the current layout of the board

Hint:

To solve the eight queen problem with backtracking, following helping functions might be necessary:

`solve(int row)` – solve a particular row on the board

`clearRow()` – clear the placement of a row

`boolean checkRow(int row)` – check whether a particular row is valid or not

`boolean checkCol(int col)` – check whether a particular column is valid or not

`boolean checkDiag(int row, int col)` – check whether a particular cell is diagonally valid or not

Step 4: Define a Remote Keypad Class

A remote keypad has 26 English characters arranged on it according to different width (1 to 26) of the keypad. The following examples show how different widths affect the arrangement of characters:

```
a b c d e
f g h i j
k l m n o
p q r s t
u v w x y
z
width = 5
```

```
a b c d e f g h
i j k l m n o p
q r s t u v w x
y z
width = 8
```

```
a b c d e f
g h i j k l
m n o p q r
s t u v w x
y z
width = 6
```

A cursor on the remote keypad indicates which of the character is selected. In above three examples of different remote keypads, character “g”, “t” and “l” are currently selected characters. Move the cursor left, right, up or down could re-select a new character. For first keypad (width = 5), move up the cursor, character “b” will be selected. Move down the cursor, character “l” will be selected. Move left the cursor, character “f” will be selected and move right the cursor, character “h” will be selected. For cursor in the second keypad (width = 8), it can be moved up, left, right but not down. For cursor in the third keypad (width = 6), it can be moved up, left, down but not right. When press enter, the character being selected by the cursor will actually being typed out by the remote keypad.

Now please write a remote keypad class, name “RemoteKeypad”. It should have **width**, **cursorRow**, **cursorCol** defined as private data to indicate its property and current state. A remote keypad object should be initialized with following constructor:

```
RemoteKeypad(int width) {
    this.width = width;
    cursorRow = 0;
    cursorCol = 0;
}
```

Now define a member function of the remote keypad class, `getMoves(String str)`. This function prints out all necessary moves and enter pressed to type the given string.

Take a width 5 remote keypad as an example:

```

RemoteKeypad keypad = new RemoteKeypad(5);
keypad.getMoves("abc");
keypad.getMoves("xyzrts");

// Output
enter // cursor at
(0,0)
right enter // cursor at (0,1)
right enter // cursor at (0,2)

down down down down right enter // cursor at (4,3)
right enter // cursor at (4,4)
left left left left down enter // cursor at (5,0)
up up right right enter // cursor at (3,2)
right right enter // cursor at (3,4)
left enter // cursor at
(3,3)

```

Hint:

You could use following statements to get the position of a character on a remote keypad with particular width.

```

int getRow(char c) {
    return (c - 'a') / width
}

int getCol(char c) {
    return (c - 'a') % width
}

```

Step 5: Define a Data Structure

Define a data structure that supports constant time for inserting an item, remove a particular item and randomly pick an item in the data structure. Define the class with Java Templates.

```

class MyDataStructure<T> {
    // essential data parts

    // constructor
    public MyDataStructure() {
        // initialization
    }

    ...
}

```

The class should have following three public member functions defined and all of them should have $O(1)$ time to complete.

void insert(T item) – insert an item into the data structure

void remove(T item) – remove the particular item from the data structure

T getRandom() – randomly pick an item from those in the data structure

Hint:

You are free to use any pre-defined data structures in Java language package such as lists, hashes, arrays, etc.

Step 6:

Create a readme file which includes your name and the course number (CSCI 4900 or CSCI 6900) you enrolled in. You could also put any necessary notice for grading in this file too.

Put **Readme**, **Main.java**, **EightQueenSolver.java**, **RemoteKeyPad.java** and **MyDataStructure.java** (optional for CSCI 4900 and required for CSCI 6900) into one folder lastname_lab05 and submit the folder to cs4900a through Nike(submit cs4900a lastname_lab05).

Grading Rubric (CSCI 4900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	95		
step 1	10		
step 2	20		
step 3	25		
step 4	40		
step 5 (bonus)	20		

Grading Rubric (CSCI 6900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	95		
step 1	5		
step 2	10		
step 3	20		
step 4	25		
step 5	35		

CSCI 4900/6900 Lab/Project Specification (Week 6)

Lab #6**Programming in Scala****Goals**

In this course, we will explore using the Scala Actor model to program concurrency. However, the Scala programming language is not a major teaching topic in the course

schedule. Instead, you are expected to learn Scala by reading the textbook, discussing with your classmates and finishing this lab. After completing this lab, you should gain knowledge of Scala programming, its syntax and basic programming conventions as preparation for using the Actor model and its corresponding language features to program concurrency. For students enrolled in CSCI 6900, you should finish all the exercises. For students enrolled in CSCI 4900, you should finish only those exercises that are not marked with “CSCI 6900 Only”.

Due Date

Feb. 20, 2013 (Wednesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab, but you are welcome to post any questions or start a discussion on piazza. Please refer to the course policy for questions that are allowed. You may also discuss general strategies with your classmates. However, you must complete coding on your own. **No copy of or detailed line by line cooperation on code is allowed.**

You are encouraged to look at the Scala API documentation while solving this exercise, which can be found here: <http://www.scala-lang.org/api/current/index.html>.

Note that Scala uses the String from Java, therefore the documentation for strings is found in the Javadoc API: <http://docs.oracle.com/javase/6/docs/api/java/lang/String.html>

Lab Description

Step 1: Project Set Up

Please create a Scala project, name “lab06”, with Eclipse. Create an object “Main” in the default package, then copy and paste the codes given in **Main.scala**, which sets up the input and output for functions and classes defined in step 2 to step 4.

The main function defined in **Main.scala** takes a series of parameters separated by white space:

`args(0)` – the number of lines of the Pascal triangle to print

`args(1)` – the amount of money for which to calculate the number of combinations for change

`args(2)`, `args(3)` – parenthesized expressions to be validated

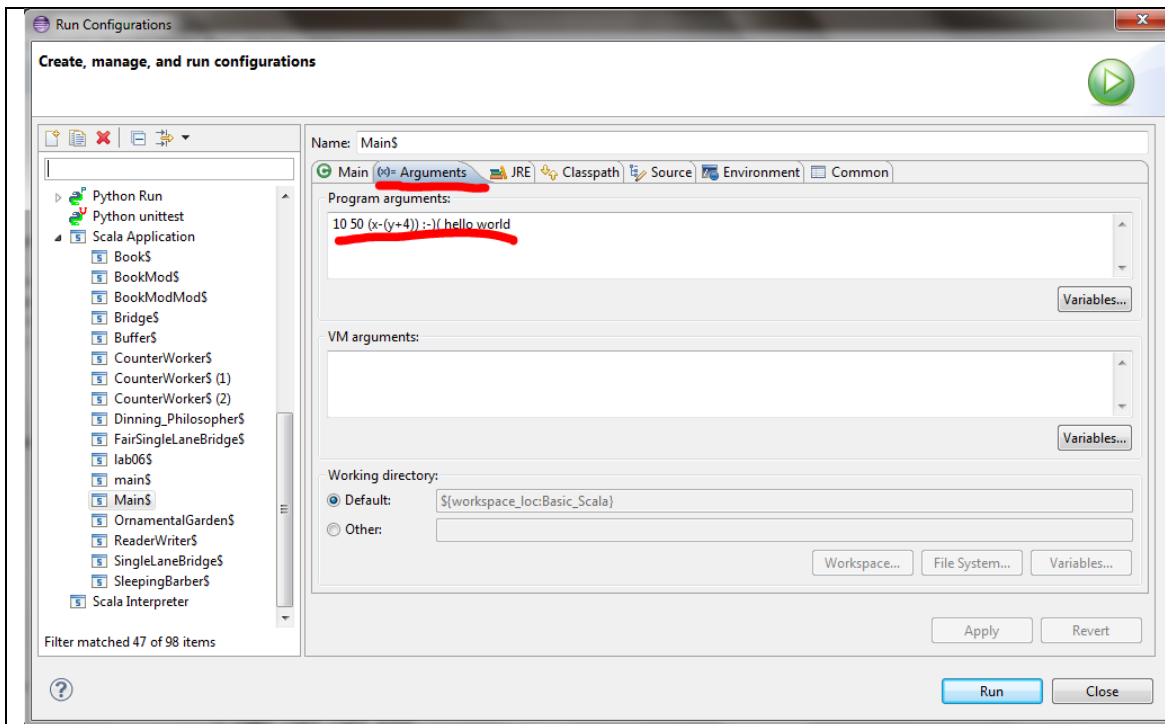
`args(4)`, `args(5)` – strings of alphabetic characters

A sample series of parameters is:

```
10 50 (x+(y+4)) :-) ( hello world
```

which will make the program to print 10 lines of Pascal triangle, count number of ways to change 50¢, verify whether parenthesis are balanced in “(x+(y-5))” and “:-)” expressions, and print moves for “hello” and “world”.

To configure different input parameters for the Main object, please left click the down arrow of run button, select “Run Configurations” and then select Arguments tab as shown below. You could input any number of arguments into the textbox provided. The arguments are separated by white space.



Following is a sample run output with the above input arguments:

step 2: print function

```

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

```

step 3: recursive algorithm

Pascal's Triangle

```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1

```

Number of change for 50 cents is: 36

expression: (x-(y+4)) is balanced

expression: :-(is not balanced

step 4: remote keypad

down right right enter


```

up right right enter
left left left down down enter
enter
right right right enter

left left down down enter
up up right right enter
left left down enter
left up enter
up up right right enter

```

Step 2: A Simple Print Function

In the Main object defined in step 1, define another method `myPrint()` which prints out the following output:

```

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

```

Step 3: Recursive Algorithms

Create a new object Recursion in the same default package as Main. Copy the codes in [Recursion.scala](#).

3.1. Pascal's Triangle

The following pattern of numbers is called *Pascal's Triangle*. The numbers at the edge of the triangle are all 1 and each number inside the triangle is the sum of the two numbers above it. Write a function that computes elements of Pascal's Triangle by means of a recursive process.

```

      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1

```

Finish this exercise by implementing the `pascal(c: Int, r: Int): Int` function, which takes a column `c` and a row `r`, counting from 0, and returns the number at that spot in the triangle. For example, `pascal(0, 2)=1`, `pascal(1, 2)=2`, `pascal(1, 3)=3`.

NOTE: the actual display of a Pascal Triangle won't be nicely centered. Rather, it will be left justified:

1 1
 1 2 1
 ... and so on.

3.2. Counting Change

Write a recursive function that counts how many different ways you can make change for an amount, given a list of coin denominations. For example, there are 3 ways to give change for 10¢ if you have coins with denomination 1¢, 5¢, and 10¢:

- 1 of 10¢
- 10 of 1¢
- 1 of 5¢ and 5 of 1¢
- 2 of 5¢

Finish this exercise by implementing the `countChange(money: Int, coins: List[Int]): Int` function. This function takes an amount to change, and a list of denominations for the coins.

Hint: Three functions of List type in Scala might be helpful:

`coins.isEmpty(): Boolean` – returns whether the list is empty
`coins.head(): Int` – returns the first element of the list
`coins.tail(): List[Int]` – returns the list without the first element

3.3. Parenthesis Balancing (CSCI 6900 Only)

Design a recursive algorithm to verify the balancing of parenthesis in a statement. For example, the following two statements have balanced parenthesis:

- `(if(zero?x)max(/1x))`
- `((x+3)/(y-4)+33)%2`

The following statements do **NOT** have balanced parenthesis:

- `:-)`
- `()()`

Notice that the last example above showed that only counting the number of left and right parenthesis is not enough for verification.

Finish this exercise by completing the implementation of `balance(chars: List[Char]): Boolean` function.

Hint: The `isEmpty`, `head`, `tail` functions may still be useful. You could define an inner function if you want to pass extra parameters to the `balance` function.

Step 4: Define a Remote Keypad Class

Implement a Remote Keypad Class that has the same properties and functionality as described in Step4 of Lab05 by completing the code skeleton in `RemoteKeypad.scala`. You are not supposed to modify any function signatures, i.e. the names, parameters and return value of functions.

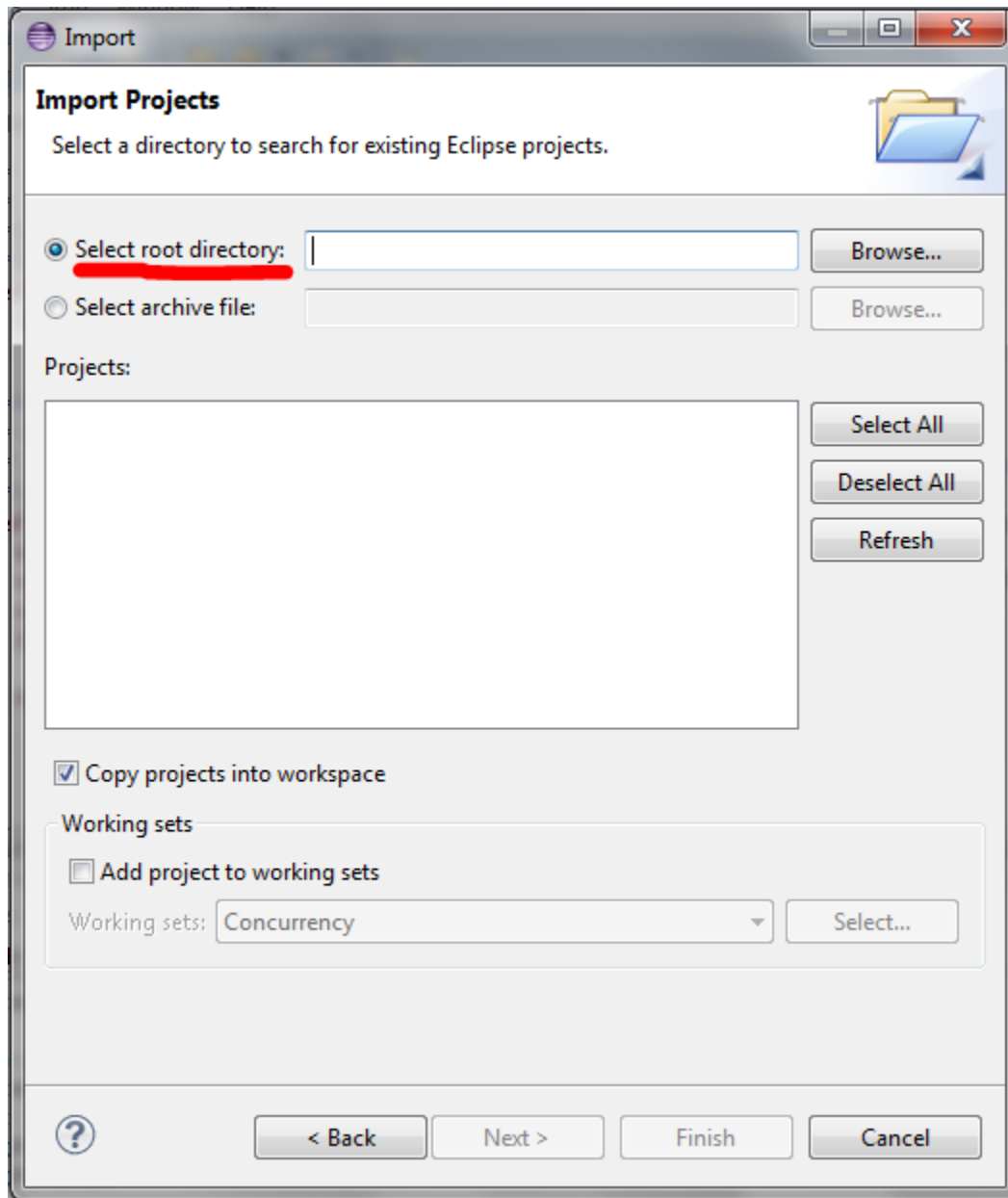
Hint: You could use the `toList` and `map` function in `getMoves(s:String):Unit` function. This function could be as simple as two lines.

Step 5: For-comprehension and Anagram (CSCI 6900 Only)

In this step you will solve combinatorial problem of finding all the anagrams of a sentence using the Scala Collections API and for-comprehensions.

5.0. Project Setup

Download [forcomp.zip](#) and extract it. In Eclipse, you can import an existing Scala project by specifying the root directory, which is the directory to which you extract the zip file. Work with `src/main/scala` and `src/test/scala` (unit tests). The only file you should make changes to is [Anagrams.scala](#) and your goal is to pass all 13 unit tests defined in [AnagramsSuite.scala](#).



5.1. The Problem

An anagram of a word is a rearrangement of its letters such that a word with a different meaning is formed. For example, if we rearrange the letters of the word `Elvis` we can obtain the word `lives`, which is one of its anagrams.

In a similar way, an anagram of a sentence is a rearrangement of all the characters in the sentence

such that a new sentence is formed. The new sentence consists of meaningful words, the number of which may or may not correspond to the number of words in the original sentence.

For example, the sentence:

`I love you`

is an anagram of the sentence:

`You olive`.

In this exercise, we will consider permutations of words anagrams of the sentence. In the above example:

`You I love`

is considered a separate anagram. When producing anagrams we will ignore the character casing and the punctuation characters.

Your ultimate goal is to implement a method `sentenceAnagrams`, which, given a list of words representing a sentence, finds all the anagrams of that sentence. Note that we used the term **meaningful** in defining what anagrams are. You will be given a dictionary, i.e. a list of words indicating words that have a meaning.

Here is the general idea. We will transform the characters of the sentence into a list saying how often each character appears. We will call this list **the occurrence list**. To find anagrams of a word we will find all the words from the dictionary which have the same occurrence list. Finding an anagram of a sentence is slightly more difficult. We will transform the sentence into its occurrence list, and then try to extract any subset of characters from it to see if we can form any meaningful words. From the remaining characters we will solve the problem recursively and then combine all the meaningful words we have found with the recursive solution.

Let's apply this idea to our example, the sentence `You olive`. Let's represent this sentence as an occurrence list of characters `eiloouvy`. We start by subtracting some subset of the characters, say `l`. We are left with the characters `eloouvy`.

Looking into the dictionary we see that `l` corresponds to word `I` in the English language, so we found one meaningful word. We now solve the problem recursively for the rest of the characters `eloouvy` and obtain a list of solutions `List(List(love, you), List(you, love))`. We can combine `I` with that list to obtain sentences `I love you` and `I you love`, which are both valid anagrams.

5.2. Representation

We represent the words of a sentence with the `String` data type:

```
type Word = String
```

Words contain lowercase and uppercase characters, and no whitespace, punctuation or other special characters. Since we are ignoring the punctuation characters of the sentence as well as the whitespace characters, we will represent sentences as lists of words:

```
type Sentence = List[Word]
```

We mentioned previously that we will transform words and sentences into occurrence lists. We represent the occurrence lists as sorted lists of character and integers pairs:

```
type Occurrences = List[(Char, Int)]
```

The list should be sorted by the characters in an ascending order. Since we ignore the character casing, all the characters in the occurrence list have to be lowercase. The integer in each pair denotes how often the character appears in a particular word or a sentence. This integer must be positive. Note that positive also means non-zero – characters that do not appear in the sentence do not appear in the occurrence list either.

Finally, the dictionary of all the meaningful English words is represented as a List of words:

```
val dictionary: List[Word] = loadDictionary
```

The dictionary already exists for this exercise and is loaded for you using the `loadDictionary` utility

method.

5.3. Computing Occurrence Lists

The `groupBy` method takes a function mapping an element of a collection to a key of some other type, and produces a Map of keys and collections of elements which mapped to the same key. This method **groups** the elements, hence its name.

Here is one example:

```
List("Every", "student", "likes", "Scala").groupBy((element: String)
=> element.length)
```

produces:

```
Map(
  5 -> List("Every", "likes", "Scala"),
  7 -> List("student")
)
```

Above, the key is the length of the string and the type of the key is `Int`. Every String with the same length is grouped under the same key – its length.

Here is another example:

```
List(0, 1, 2, 1, 0).groupBy((element: Int) => element)
```

produces:

```
Map(
  0 -> List(0, 0),
  1 -> List(1, 1),
  2 -> List(2)
)
```

Maps provide efficient lookup of all the values mapped to a certain key. Any collection of pairs can be transformed into a Map using the `toMap` method. Similarly, any Map can be transformed into a List of pairs using the `toList` method.

In our case the collection will be a Word (i.e. a String) and its elements are characters, so the `groupBy` method takes a function mapping characters into a desired key type.

In the first part of this exercise, we will use the `groupBy` method from the Collections API (you may additionally use other methods, such as `map` and `toList`) to implement the following method, which given a word produces its occurrence list.

```
def wordOccurrences(w: Word): Occurrences
```

Next, we implement another version of the method for entire sentences. We can concatenate the words of the sentence into a single word and then reuse the method `wordOccurrences` that we already have.

```
def sentenceOccurrences(s: Sentence): Occurrences
```

5.4. Computing Anagrams of a Word

To compute the anagrams of a word we use the simple observation that all the anagrams of a word have the same occurrence list. To allow efficient lookup of all the words with the same occurrence list, we will have to **group** the words of the dictionary according to their occurrence lists.

```
lazy val dictionaryByOccurrences: Map[Occurrences, List[Word]]
```

We then implement the method `wordAnagrams` which returns the list of anagrams of a single word:

```
def wordAnagrams(word: Word): List[Word]
```

5.5. Computing Subsets of a Set

To compute all the anagrams of a sentence, we will need a helper method which, given an

occurrence list, produces all the subsets of that occurrence list.

```
def combinations(occurrences: Occurrences): List[Occurrences]
```

The *combinations* method should return all possible ways in which we can pick a subset of characters from occurrences. For example, given the occurrence list:

```
List(('a', 2), ('b', 2))
```

the list of all subsets is:

```
List(
  List(),
  List(('a', 1)),
  List(('a', 2)),
  List(('b', 1)),
  List(('a', 1), ('b', 1)),
  List(('a', 2), ('b', 1)),
  List(('b', 2)),
  List(('a', 1), ('b', 2)),
  List(('a', 2), ('b', 2))
)
```

The order in which you return the subsets does not matter as long as they are all included. Note that there is only one subset of an empty occurrence list, and that is the empty occurrence list itself.

Hint: investigate how you can use for-comprehensions to implement parts of this method.

5.6. Computing Anagrams of a Sentence

We now implement another helper method called *subtract* which, given two occurrence lists *x* and *y*, subtracts the frequencies of the occurrence list *y* from the frequencies of the occurrence list *x*:

```
def subtract(x: Occurrences, y: Occurrences): Occurrences
```

For example, given two occurrence lists for words *lard* and *r*:

```
val x = List(('a', 1), ('d', 1), ('l', 1), ('r', 1))
val y = List(('r', 1))
```

the *subtract*(*x*, *y*) is *List*(('a', 1), ('d', 1), ('l', 1)).

The precondition for the *subtract* method is that the occurrence list *y* is a subset of the occurrence list *x* – if the list *y* has some character then the frequency of that character in *x* must be greater or equal than the frequency of that character in *y*. When implementing *subtract* you can assume that *y* is a subset of *x*.

Hint: you can use *foldLeft*, and *-*, *apply* and *updated* operations on *Map*.

Now we can finally implement our *sentenceAnagrams* method for sequences.

```
def sentenceAnagrams(sentence: Sentence): List[Sentence]
```

Note that the anagram of the empty sentence is the empty sentence itself.

Hint: First of all, think about the recursive structure of the problem: what is the base case, and how should the result of a recursive invocation be integrated in each iteration? Also, using for-comprehensions helps in finding an elegant implementation for this method.

Test the *sentenceAnagrams* method on short sentences, no more than 10 characters. The combinations space gets huge very quickly as your sentence gets longer, so the program may run for a very long time. However for sentences such as `Linux rulez`, `I love you` or `Mickey Mouse` the program should end fairly quickly.

Step 6:

Create a readme file which includes your name and the course number (CSCI 4900 or CSCI 6900) you enrolled in. You could also put any necessary information for grading in this file too.

Put **Readme**, **Main.scala**, **Recursion.scala**, **RemoteKeypad.scala** and **Anagrams.scala** into one

folder lastname_lab06 and submit the folder to cs4900a through Nike(submit cs4900a lastname_lab06).

Grading Rubric (CSCI 4900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	95		
step 1	10		
step 2	10		
step 3-1	25		
step 3-2	30		
step 4	20		
Bonus	50		
step 3-3	10		
step 5	40		

Grading Rubric (CSCI 6900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	95		
step 1	5		
step 2	5		
step 3-1	5		
step 3-2	10		
step 3-3	20		
step 4	10		
step 5 (unit test 1-11)	40		
Bonus	20		
step 5 (unit test 12-13)	20		

CSCI 4900/6900 Lab/Project Specification (Week 7)

Lab #7

Programming in Python

Goals

In this course, we will explore using the Python Coroutine model to program concurrency. However, the Python programming language is not a major teaching topic in the course schedule. Instead, you are expected to learn Python by reading the textbook, discussing with your classmates and finishing this lab. After completing this lab, you should gain knowledge of Python programming, its syntax and basic programming conventions as preparation for using

the Coroutine model and its corresponding language features to program concurrency. For students enrolled in CSCI 6900, you should finish all the exercises. For students enrolled in CSCI 4900, you should finish only those exercises that are not marked with “CSCI 6900 Only”.

Due Date

Feb. 27, 2013 (Wednesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab, but you are welcome to post any questions or start a discussion on piazza. Please refer to the course policy for questions that are allowed. You may also discuss general strategies with your classmates. However, you must complete coding on your own. **No copy of or detailed line by line cooperation on code is allowed.**

You are encouraged to look at the Python library documentation while solving this exercise, which can be found here:

<http://docs.python.org/2/library/> (python 2.x) <http://docs.python.org/3/library/> (python 3.x)

Comprehensive tutorial is also available from Python official site:

<http://docs.python.org/2/tutorial/> (python 2.x) <http://docs.python.org/3/tutorial/> (python 3.x)

Lab Description

Step 1: Project Set Up

Please create a Python project, name “lab07”, with Eclipse. Create a Python Module “Main” and “Configurator” in the default package, then copy and paste the codes given in [main.py](#) and [configurator.py](#), which sets up the input and output for functions and classes defined in step 2 and step 4.

The main function defined in [main.py](#) uses a Python ConfigParser to write and read configuration data (test inputs) to and from a configure file and then evaluate with `literal_eval()` function in `ast` package. The actual `write_conf()` and `read_conf()` methods are defined in module [configurator.py](#). You are not encouraged to modify [main.py](#) or [configurator.py](#). However, if you are confident in manipulating ConfigParser and configure files, you could modify it to test on different inputs.

Information about ConfigParser could be found at following documentation:

<http://docs.python.org/2/library/configparser.html> .

Information about `ast` package could be found at following documentation:

<http://docs.python.org/2/library/ast.html> .

A sample output with the given configurations is:

```
myPrint
*
***
*****
*****
*****
```



```

String: both_ends
spring -> spng
hello -> helo
google -> gole
a ->
->

String: fix_start
babble -> ba**le
aardvark -> a*rdv*rk
google -> goo*le
donut -> donut

String: mix_up
('mix', 'pod') -> pox mid
('dog', 'dinner') -> dig donner
('gnash', 'sport') -> spash gnort
('pezzy', 'firm') -> fizzy perm

String: not_bad
This movie is not so bad -> This movie is good
This dinner is not that bad! -> This dinner is good!
This tea is not hot -> This tea is not hot
It is bad yet not -> It is bad yet not

String: font_back
('abcd', 'xy') -> abxcdy
('abcde', 'xyz') -> abcxydez
('Kitten', 'Donut') -> KitDontenut

List: match_ends
['aba', 'xyz', 'aa', 'x', 'bbb'] -> 3
['', 'x', 'xy', 'yx', 'xx'] -> 2
['aaa', 'be', 'abc', 'hello'] -> 1

List: front_x
['bbb', 'ccc', 'axx', 'xzz', 'xaa'] -> ['xaa', 'xzz', 'axx', 'bbb', 'ccc']
['ccc', 'bbb', 'aaa', 'xcc', 'xaa'] -> ['xaa', 'xcc', 'aaa', 'bbb', 'ccc']
['mix', 'xyz', 'apple', 'xanadu', 'aardvark'] -> ['xanadu', 'xyz', 'aardvark', 'apple', 'mix']

List: sort_last
[(1, 3), (3, 2), (2, 1)] -> [(2, 1), (3, 2), (1, 3)]
[(2, 3), (1, 2), (3, 1)] -> [(3, 1), (1, 2), (2, 3)]
[(1, 7), (1, 3), (3, 4, 5), (2, 2)] -> [(2, 2), (1, 3), (3, 4, 5), (1, 7)]

List: remove_adjacent
[1, 2, 2, 3] -> [1, 2, 3]
[2, 2, 3, 3, 3] -> [2, 3]
[] -> []

List: linear_merge
(['aa', 'xx', 'zz'], ['bb', 'cc']) -> ['aa', 'bb', 'cc', 'xx', 'zz']
(['aa', 'xx'], ['bb', 'cc', 'zz']) -> ['aa', 'bb', 'cc', 'xx', 'zz']
(['aa', 'aa'], ['aa', 'bb', 'bb']) -> ['aa', 'aa', 'aa', 'bb', 'bb']

Remote:

```

```

right down enter
up right right right enter
right down enter
enter
left left left down enter

right right down enter
left left up enter
right right right enter
up enter
left left up enter

left left down down down down enter
enter
enter

```

Step 2: A Simple Print Function

In the Main module defined in step 1, define another method `myPrint()` which prints out the following output:

```

      *
     ***
    *****
   *********
  ***********
 
```

Step 3: Strings, Lists, Dictionary and Files

In this step, you will program to manipulate Python strings, lists, dictionary data structure and files. You should complete the code skeleton in [mystring.py](#), [mylist.py](#), [mimic.py](#) and [wordcount.py](#) (CSCI 6900 only)

3.1 Strings in Python ([mystring.py](#))

Create a new module [mystring.py](#) in the same default package as the Main module. Finish the definition of following functions in [mystring.py](#).

1. `both_ends(s)`

Given a string `s`, return a string made of the first 2 and the last 2 chars of the original string, so 'spring' yields 'spng'. However, if the string length is less than 2, return the empty string instead.

2. `fix_start(s)`

Given a string `s`, return a string where all occurrences of its first char have been changed to '*', except do not change the first char itself. e.g. 'babble' yields 'ba**le'. Assume that the string is length 1 or more.

Hint: `s.replace(stra, strb)` returns a version of string `s` where all instances of `stra` have been replaced by `strb`.

3. `mix_up(a, b)`

Given strings `a` and `b`, return a single string with `a` and `b` separated by a space '<a> ', except swap the first 2 chars of each string.

e.g.

```
'mix', 'pod' -> 'pox mid'
'dog', 'dinner' -> 'dig donner'
```

Assume a and b are strings of length 2 or more.

4. `not_bad(s)`

Given a string, find the first appearance of the substring 'not' and 'bad'. If the 'bad' follows the 'not', replace the whole 'not'...'bad' substring with 'good'. Return the resulting string. So 'This dinner is not that bad!' yields 'This dinner is good!'

5. `front_back(a, b)`

Consider dividing a string into two halves. If the length is even, the front and back halves are the same length. If the length is odd, we'll say that the extra char goes in the front half. e.g. 'abcde', the front half is 'abc', the back half 'de'.

Given 2 strings, a and b, return a string of the form: a-front + b-front + a-back + b-back

e.g.

```
front_back('abcd', 'xy') -> 'abxcdy'
front_back('abcde', 'xyz') -> 'abcxydez'
front_back('Kitten', 'Donut') -> 'KitDontenut'
```

3.2 Lists in Python (*mylist.py*)

Create a new module **mylist.py** in the same default package as the Main module. Finish the definition of following functions in **mylist.py**.

1. `match_ends(words)`

Given a list of strings, return the count of the number of strings where the string length is 2 or more and the first and last chars of the string are the same.

Note: python does not have a ++ operator, but += works.

2. `front_x(words)`

Given a list of strings, return a list with the strings in sorted order, except group all the strings that begin with 'x' first.

e.g.

```
['mix', 'xyz', 'apple', 'xanadu', 'aardvark']
->
['xanadu', 'xyz', 'aardvark', 'apple', 'mix']
```

Hint: this can be done by making 2 lists and sorting each of them before combining them.

3. `sort_last(tuples)`

Given a list of non-empty tuples, return a list sorted in increasing order by the last element in each tuple.

e.g.

```
[(1, 7), (1, 3), (3, 4, 5), (2, 2)]
->
[(2, 2), (1, 3), (3, 4, 5), (1, 7)]
```

Hint: define an extra function and then use a *custom key= function* to extract the last element from each tuple. For tuple data structure, please refer to following documentation:

<http://docs.python.org/2/tutorial/datastructures.html#tuples-and-sequences>

4. `remove_adjacent(nums)`

Given a list of numbers, return a list where all adjacent equal elements have been reduced to a single element.

e.g.

`[1, 2, 2, 3] -> [1, 2, 3]`.

You may create a new list or modify the passed in list.

5. `linear_merge(list1, list2)`

Given two lists sorted in increasing order, create and return a merged list of all the elements in sorted order. You may modify the passed in lists. Ideally, the solution should work in "linear" time, making a single pass of both lists.

3.3 Dictionary and Files in Python

In this section, you will practice using python dictionary and file utilities. The given file `small.txt` is a small test case you could use to feed your program while debugging while `alice.txt` is a relative large test case.

3.3.1 Mimic Exercise

Create a new module `mimic.py` in the same default package as the Main module. Finish the definition of following functions in `mimic.py`.

1. `mimic_dict(filename)`

Returns mimic dict mapping each word to a list of words which follow it.

Read in the file specified on the command line. Do a simple `split()` on whitespace to obtain all the words in the file. Rather than read the file line by line, it's easier to read it into one giant string and split it once.

Build a "mimic" dict that maps each word that appears in the file to a list of all the words that immediately follow that word in the file. The list of words can be in any order and should include duplicates. So for example the key "and" might have the list `["then", "best", "then", "after", ...]` listing all the words which came after "and" in the text. We'll say that the empty string is what comes before the first word in the file.

Hint: after getting the file object, use `f.read()` method to get the file as a whole big string.

2. `print_mimic(mimic_dict, word, n)`

Given mimic dict and start word, prints n random words.

With the mimic dict, it's fairly easy to emit random text that mimics the original. Print a word, then look up what words might come next and pick one at random as the next word. Use the empty string as the first word to prime things. If we ever get stuck with a word that is not in the dict, go back to the empty string to keep things moving.

For fun, feed your program to itself as input.

Hint: the standard python module 'random' includes a `random.choice(list)` method which picks a random element from a non-empty list.

Note: `mimic.py` has its own main function and could run directly without the "Main" module defined in step 1.

3.3.2 Word Count Exercise (CSCI 6900 only)

Create a new module **wordcount.py** in the same default package as the Main module. Finish the definition of following functions in **wordcount.py**.

The `main()` function is already defined and complete. It calls `print_words` and `print_top` functions which you will write to complete.

1. `print_words(filename)`

For the `--count` flag, implement a `print_words(filename)` function that counts how often each word appears in the text and prints:

```
word1 count1
word2 count2
...
```

Print the above list in order sorted by word (python will sort punctuation to come before letters and that's fine). Store all the words as lowercase, so 'The' and 'the' count as the same word.

2. `print_top(filename, n)`

For the `--topcount` flag, implement a `print_top(filename, n)` which is similar to `print_words` but which prints just the top `n` most common words sorted. So the most common word is first, then the next most common, and so on.

Hint: Use `str.split()` (no arguments) to split on all whitespace. Define a helper function to avoid code duplication inside `print_words` and `print_top`.

Note: **wordcount.py** has its own main function and could run directly without the "Main" module defined in step 1.

Step 4: Define a Remote Keypad Class

Implement a Remote Keypad Class that has the same properties and functionality as described in Step4 of Lab05 by completing the code skeleton in **remotekeypad.py**. You are not supposed to modify any function signatures, i.e. the names, parameters and return value of functions.

Hint: `print 'some string',` (with a comma at the end of print statement) will only print "some string" without a newline.

Step 5: Regular Expressions & Utilities (CSCI 6900 Only)

For the Log Puzzle exercise, you'll use Python code to solve two puzzles. This exercise uses the `urllib` (url library) module and `re` (regular expression) module. The files for this exercise are in the "logpuzzle" zip. Add your code to the **logpuzzle.py** file.

An image of an animal has been broken into many narrow vertical stripe images. The stripe images are on the internet somewhere, each with its own url. The urls are hidden in a web server log file. Your mission is to find the urls and download all image stripes to re-create the original image.

The slice urls are hidden inside apache log file: **animal_code.google**. The log file encodes what server it comes from like this: the log file `animal_code.google.com` is from the `code.google.com` server (formally, we'll say that the server name is whatever follows the first underbar). The `animal_code.google.com` log file contains the data for the "animal" puzzle image. Although the data in the log files has the syntax of a real apache web server, the data beyond what's needed for the puzzle is randomized data from a real log file.

Here is what a single line from the log file looks like (this really is what apache log files look like):

```
10.254.254.28 - - [06/Aug/2007:00:14:08 -0700] "GET /foo/talks/ HTTP/1.1"
200 5910 "-" "Mozilla/5.0 (X11; U; Linux i686 (x86_64); en-US; rv:1.8.1.4) Gecko/20070515 Firefo
```

The first few numbers are the address of the requesting browser. The most interesting part is the "GET *path* HTTP" showing the path of a web request received by the server. The path itself never contains spaces, and is separated from the GET and HTTP by spaces (regex suggestion: `\S` (upper case S) matches any non-space char). Find the lines in the log where the string "puzzle" appears inside the path, ignoring the many other lines in the log.

5.1 Log File to Urls

Complete the `read_urls(filename)` function that extracts the puzzle urls from inside a logfile. Find all the "puzzle" path urls in the logfile. Combine the path from each url with the server name from the filename to form a full url.

e.g. "http://www.example.com/path/puzzle/from/inside/file".

Screen out urls that appear more than once. The `read_urls()` function should return the list of full urls, sorted into alphabetical order and **without duplicates**. Taking the urls in alphabetical order will yield the image slices in the correct left-to-right order to re-create the original animal image. In the simplest case, `main()` should just print the urls, one per line.

```
$ ./logpuzzle.py animal_code.google.com
http://code.google.com/something/puzzle-animal-baaa.jpg
http://code.google.com/something/puzzle-animal-baab.jpg
...
```

5.2 Download Images Puzzle

Complete the `download_images()` function which takes a sorted list of urls and a directory. Download the image from each url into the given directory, creating the directory first if necessary (see the "os" module to create a directory, and "urllib.urlretrieve()" for downloading a url).

Name the local image files with a simple scheme like "img0", "img1", "img2", and so on. You may wish to print a little "Retrieving..." status output line while downloading each image since it can be slow and is nice to have some indication that the program is working.

Here's what it should look like when you can download the animal puzzle:

```
$ ./logpuzzle.py --todir animaldir animal_code.google.com
$ ls animaldir
img0 img1 img2 img3 img4 img5 img6 img7 img8 img9
index.html
```

Each image is a little vertical slice from the original. To put the slices together to re-create the original, you could create a little html file: **index.html** by `download_images()` function in the directory with an `*img*` tag to show each local image file. The img tags should all be on one line together without separation. In this way, the browser displays all the slices together seamlessly. You do not need knowledge of HTML to do this; just create an index.html file that looks like this:

```
<verbatim>
<html>
<body>



...
</body>
</html>
```

5.3 Find out the Animal

After creating the [index.html](#) file, you could open it in any browser and then report the animal you find in your readme file.

Step 6:

Create a readme file which includes your name and the course number (CSCI 4900 or CSCI 6900) you enrolled in. For CSCI 6900 students, you should also report the animal you find in step 5. You could also put any necessary information for grading in this file too.

Put **Readme**, **main.py**, **mystring.py**, **mylist.py**, **remotekeypad.py**, **mimic.py**, **wordcount.py** (CSCI 6900) and **logpuzzle.py** (CSCI 6900) into one folder lastname_lab07 and submit the folder to cs4900a through Nike(submit cs4900a lastname_lab07).

Grading Rubric (CSCI 4900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	95		
step 1	5		
step 2	5		
step 3-1	25		
step 3-2	25		
step 3-3	20		
step 4	20		
Bonus	20		
step 5	20		

Grading Rubric (CSCI 6900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	95		
step 1	5		
step 2	5		
step 3-1	15		
step 3-2	15		
step 3-3	25		
step 4	15		
step 5	15		

CSCI 4900/6900 Lab/Project Specification (Week 8)

Lab #8

Party Matching as Shared Memory System

Goals

In lab 08, we will practice programming concurrency problems with Java threads for shared memory systems, with Scala actors for message passing systems and with Python coroutines for cooperative multi-tasking systems. In this lab, you are going to program the party matching problem as a shared memory system with Java threads. The goal of the lab is to gain hands-on experience in programming Java threads and using thread models.

Due Date

Mar 6, 2013 (Wednesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab, but you are welcome to post any questions or start a discussion on piazza. Please refer to the course policy for questions that are allowed. You may also discuss general strategies with your classmates. However, you must complete coding on your own. **No copying of or detailed line by line cooperation on code is allowed.**

You are encouraged to look at the documentation and tutorials related to Java Threads while solving this exercise, which can be found in the course reading materials. Of particular interest:

<http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html>

<http://docs.oracle.com/javase/tutorial/essential/concurrency/>

Lab Description

Step 1: The Party Matching Problem

The problem is to simulate a party where boys and girls come arrive but leave in pairs consisting of one boy and one girl.

A simple version of this simulation is to let any boy or girl leave the party when there's a partner available. One design alternative for this version is to have threads represent boys and girls and a class of shared data that records the number of boy and girl threads. The threads consult the shared data and decide whether they can exit (leave the party).

A more sophisticated and realistic version of this simulation is to allow boys and girls have the choice of whom they would like to leave with. The design of this version requires more complicated classes that represent boys, girls and the party.

Step 2: Implement the Simple Version of Design

Define three Java classes, **ParticipatorGenerator.java**, **Party.java** and **Main.java**.

ParticipatorGenerator.java should implement the `run()` method in Java. It represents a thread that generates either boys or girls. The gender should be decided randomly when it is initialized. Two member functions should be called in the `run()` method repeatedly. One is to check into the party and the other is to leave the party.

Party.java should record all shared data required for this simulation. The member functions defined in **ParticipatorGenerator.java** should access and modify data defined inside this class.

Main.java is the starting place of the simulation. The party object and 10 participator generator objects should be initialized and start here.

No specific output (print out) is required. Please keep your code as concise as possible.

Step 3: Design of the Realistic Version of Simulation

Suppose boys and girls still come to the party individually. But in this design, pairs of boys and girls may only exit when a boy invites an available girl (girl who's already at the party) to leave. If the girl agrees, then they both leave the party. If not, the boy may invite any other available girl.

3.1 Draft a UML Class Diagram

Use a UML class diagram to express one design you envision as feasible for this realistic version of simulation.

Hint:

1. a more complicated data structure may be necessary for the party class
2. you might want to separate the functionality of boy/girl threads from the ParticipatorGenerator
3. synchronized methods might be necessary for boy and/or girl threads

3.2 Write Pseudocode (for CSCI 6900 only)

Write out the pseudocode with your design.

Note: You may use only the shared memory pseudocode syntax since the design is to simulate the problem as a shared memory system.

Step 4: Implement the Realistic Version of Simulation (optional)

Implement your design on the realistic version of the simulation with Java threads.

Step 5:

Create a readme file that includes your name and the course number (CSCI 4900 or CSCI 6900) you enrolled in.

Put **Readme**, **Main.java**, **ParticipatorGenerator.java**, and **Party.java**, for the simple version of the simulation into one folder lastname_lab08.

Put your UML class diagram design and pseudocode (for CSCI 6900 only) into one file **design.pdf** and put it also into the folder lastname_lab08.

If you implement the realistic version of simulation, please put all files related to that into a folder lab08_real, and then copy the whole folder into lastname_lab08.

Submit the folder to cs4900a through nike(submit cs4900a lastname_lab08).

Grading Rubric (CSCI 4900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	35		
step 2	35		
Designs	10		
step 3-1	10		
Bonus	20		
step 3-2	5		
step 4	15		

Grading Rubric (CSCI 6900)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	25		
step 2	25		
Designs	20		
step 3-1	5		
step 3-2	15		
Bonus	20		
step 4	20		

CSCI 4900/6900 Lab/Project Specification (Week 9)

Lab #9

Party Matching as Message Passing System

Goals

In previous lab, we practiced programming party matching simulation with Java threads model as a shared memory system. In this lab, we will program the same problem but as a message passing system with Scala actors. The goal of this lab is to gain hands-on experience in programming Scala and using Actor models.

Due Date

Mar 19, 2013 (Tuesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab, but you are welcome to post any questions or start a discussion on piazza. Please refer to the course policy for questions that are allowed. You may also discuss general strategies with your classmates. However, you must complete coding on your own. **No copying of or detailed line by line cooperation on code is allowed.**

You are encouraged to look at the documentation and tutorials related to Scala Actor while solving this exercise, which can be found in the course reading materials. Of particular interest:

<http://www.scala-lang.org/node/242>

<http://www.scala-lang.org/api/current/index.html#scala.actors.Actor>

Lab Description

Step 1: Implement a Simplest Version

Follow the design below and implement a simplest version of the party matching simulation:

Messages:

```

Enter(gender)
Exit(gender, participantGenerator)
SucceedExit()
FailExit()

```

Behaviors:

Party

When receives an *Enter(gender)* message:

Update the correspondent status variable

When receives an *Exit(gender, participantGenerator)* message:

If condition is satisfying, send *SucceedExit()* to the participantGenerator
 Otherwise, send *FailExit()* to the participantGenerator

ParticipantGenerator

Send out an *Enter(gender)* message to the party

Send out an *Exit(gender, self)* message to the party

When receives a *SucceedExit()* message:

Send out another *Enter(gender)* message to the party

Send out another *Exit(gender, self)* message to the party

When receives a *FailExit()* message:

Send out another *Exit(gender, self)* message to the party

Note: All codes should be put into file **match1.scala**

Step 2: Implement another Version of Design

The design of a message passing system has a lot of alternatives. In step 1, the party participants keep trying to exit the party. In this alternative below, the party notifies the participants to exit. Implement this design with Scala Actors.

Messages:

```

Arrive(gender, participantGenerator)
Leave()

```

Behaviors:

Party

When receives an *Arrive(gender, participantGenerator)* message:

If a counter-gender participant exists, send *Leave()* message to both of them

Otherwise, record the participantGenerator

ParticipantGenerator

Send *Arrive(gender, self)* message to the party

When receives a *Leave()* message

Send out another *Arrive(gender, self)* message to the party

Note: All codes should be put into file **match2.scala**

Hint: To record all participants arrived, two lists variables might be necessary. You could conform to a first-come-first-leave rule when selecting counter-gender participant to leave.

Step 3: Design of the Realistic Version of Simulation

Suppose boys and girls still come to the party individually. But in this design, pairs of boys and girls may only exit when a boy invites an available girl (girl who's already at the party) to leave. If the girl agrees, then they both leave the party. If not, the boy may invite any other available girl.

3.1 Write out the Protocol of Design

Specify out what messages will be used for communication and the corresponding behaviors of each entity in the system. Accompany any diagrams as needed for clarification or illustration purpose.

Hint:

1. a more complicated message and behavior set is necessary
2. you might want to separate the functionality of boy/girl with different class definitions

3.2 Write Pseudocode

Write out the pseudocode with your design.

Note: You may use only the message passing pseudocode syntax since the design is to simulate the problem as a message passing system.

Step 4: Implement the Realistic Version of Simulation (optional)

Implement your design on the realistic version of the simulation with Scala Actors.

Step 5:

Create a readme file that includes your name and the course number (CSCI 4900 or CSCI 6900) you enrolled in.

Put **Readme**, **match1.scala**, and **match2.scala**, for the two simple versions of simulation into one folder lastname_lab09.

Put your design and pseudocode into one file **design.pdf** and put it also into the folder lastname_lab09.

If you implement the realistic version of simulation, please put all files related to that into a folder lab09_real, and then copy the whole folder into lastname_lab09.

Submit the folder to cs4900a through nike(submit cs4900a lastname_lab09).

Grading Rubric (Total: 100 + 20)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	55		
step 1	20		
step 2	35		
Designs	40		
step 3-1	15		
step 3-2	25		
Bonus	20		
step 4	20		

CSCI 4900 Homework Specification (Week 10)

Homework #4

Completing Python Codes for Cooperative Dining Philosophers

Goals

In this homework, you need to finish the given code skeleton with Python for two different designs of a dining philosopher problem. The goal of this homework is to practice using the Python generators for multitasking concurrent system.

Due Date

Mar 22, 2013 (Friday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual homework. You have to finish this homework on your own.

No exchange or copy of code is allowed!

Homework Description

Task 1: Completing code skeleton in dining.py

dining.py is shown below. The portions you are going to complete are marked with green comments. Other comments are for your understanding. By completing the code, you achieve following design:

- asymmetric fork acquisition
- Philosophers with even id acquire left fork first
- Philosophers with odd id acquire right fork first

```
#dining.py
import random
import time

class forks:
    def __init__(self, num):
        self.num = num #number of forks
        self.usage = [False] * self.num

    def acquire_left(self, name):
        #finish this method

    def acquire_right(self, name):
        #finish this method

    def release_left(self, name):
        #finish this method

    def release_right(self, name):
        #finish this method

def philosopher(name, forks):
    while True:
        if name % 2 == 0:
            #finish the if branch
            #acquire left fork first, then right fork
        else:
```

```

        #finish the else branch
        #acquire right fork first, then left fork
        yield #give others a chance to proceed while eating
        forks.release_left(name)
        forks.release_right(name)
        yield #give others a chance to proceed while thinking

def main():
    f = forks(5)
    phils = []

    for i in xrange(5):
        phil = philosopher(i, f)
        phils.append(phil)

    while True:
        random.choice(phils).next()
        time.sleep(0.1)

# Standard boilerplate to call the main() function.
if __name__ == '__main__':
    main()

```

dining.py

Task 2: Completing code skeleton in dining2.py

dining2.py is shown below. The portions you are going to complete are marked with green comments. Other comments are for your understanding. By completing the code, you achieve the design of a smarter fork acquisition mechanism:

- When more than 1 fork is available, grant the requested fork if it is available
- When only 1 fork is available, grant it to a philosopher with one fork in hand already

```

#dinning2.py

import random, time

class forks:
    def __init__(self,num):
        self.num = num #total number of forks
        self.hasFork = [] #philosophers with at least a fork
        self.usage = [False] * self.num
        self.availabe = num #number of forks not in use

    def acquire_left(self, name):
        #finish this method

    def acquire_right(self, name):
        #finish this method

    def release_left(self, name):
        #finish this method

    def release_right(self, name):
        #finish this method

def philosopher(name, forks):
    while True:
        #finish code here to acquire left fork then right fork
        print 'Philosopher %s is eating...' %(name)
        yield #give others a chance to proceed while eating
        forks.release_left(name)
        forks.release_right(name)

```

```

        yield #give others a chance to proceed while thinking

def main():
    f = forks(5)
    phils = []

    for i in xrange(5):
        phil = philosopher(i, f)
        phils.append(phil)

    while True:
        random.choice(phils).next()
        time.sleep(0.1)

# Standard boilerplate to call the main() function.
if __name__ == '__main__':
    main()

```

dining2.py

Submission:

Use the `mkdir` command to create a new folder (named “lastname_hw04”) in your local Nike account. Use the `cp` command to copy **dining.py** and **dining2.py** to the newly created folder and submit the folder to cs4900. (submit cs4900a lastname_hw04)

Grading Rubric

Deliverables	Total	Points	Comments
Subjective Survey	5		
Answers	45		
task 1	20		
task 2	25		

CSCI 4900/6900 Lab/Project Specification (Week 10)

Lab #10

Party Matching as Cooperative System

Goals

With knowledge of cooperative multi-tasking as well as the coroutine model in Python, we will practice to program the same party matching problem as a cooperative system. Please read slides and corresponding supplementary materials on Python coroutine before working on codes. The goal of this lab is to gain hands-on experience in programming concurrency with generators and coroutines in Python.

Due Date

Mar 27, 2013 (Wednesday) 11:59 pm.

Late Penalty

As described in the course policies.

Collaboration Policy

This is an individual lab, but you are welcome to post any questions or start a discussion on piazza. Please refer to the course policy for questions that are allowed. You may also discuss general strategies with your classmates. However, you must complete coding on your own. **No copying of or detailed line by line cooperation on code is allowed.**

You are encouraged to look at the documentation and tutorials related to Python Coroutine while solving this exercise, which can be found in the course reading materials. Of particular interest:

<http://www.python.org/dev/peps/pep-0342/>

Lab Description

Step 1: Implement a Simplest Version

Define a Python classes, **party** and a generator function *participant_generator(name, gender, party)*.

The function *participant_generator* should generate 10 party participants in total (randomly all girls or all boys). Whenever a participant is generated, it should be sent to party and try to find a partner to leave. If it is not able for it to leave, the generator object should yield for others to proceed. After all generated participants leave the party, this generator object should stop.

The class **party** should record the number of boys and girls and the number of pairs. It should support at least 4 methods:

```
boy_check_in -- check in a boy
girl_check_in -- check in a girl
boy_check_out -- check out a boy
girl_check_out -- check out a girl
```

Use the *main()* function defined in [figure 1](#) to test your program. This function terminates all generator objects after 1 seconds (therefore, your program should not run forever). It also checks whether the total number of pairs equals the minimum of number of boys or girls. If your program is incorrectly implemented, you will read error output message as shown in [figure 2](#). Otherwise, your program should just exist with the print outs of number of boys, girls and pairs.

```
def main():
    p = party()

    num_boys = 0;
    num_girls = 0;

    participants = []
    for i in xrange(10):
        if random.randint(0,1) == 0:
            pg = participant_generator(i, True, p)
            num_boys += 10
        else:
            pg = participant_generator(i, False, p)
            num_girls += 10
        participants.append(pg)

    t0 = time.time()
    while len(participants) > 0 and time.time() - t0 < 1:
        task = random.choice(participants)
        try:
            task.next()
```



```

except StopIteration:
    participants.remove(task)

print 'boy enter: %s' %num_boys
print 'girl enter: %s' %num_girls
print 'pair: %s' %int(p.pair)

assert min(num_boys, num_girls) == int(p.pair)

if __name__ == '__main__':
    main()

```

figure 1: main() function

```

boy enter: 70
girl enter: 30
pair: 40
Traceback (most recent call last):
  File "C:\Users\Jane\workspace\Concurrency Python\concurrency\match.py", line 77, in <module>
    main()
  File "C:\Users\Jane\workspace\Concurrency Python\concurrency\match.py", line 74, in main
    assert min(num_boys, num_girls) == 0#int(p.pair)
AssertionError

```

figure 2: Error Message

Note: All codes should be put into file match1.py

Step 2: Implement the Realistic Version

Add necessary data structures in **party** class and modify the *boy_check_out* and *girl_check_out* methods to implement the realistic version of party matching.

Use the same main() function as defined in figure 1 to test your program. The same testing logic and error message in figure 2 applies.

Hint: The added data structure should be able to record all girls in the party.

Note: All codes should be put into file match2.py

Step 3: Make Use of send() Function (optional)

In this step, you are still working with the basic party matching problem setting (not the realistic one).

Instead of define a party class, this time you will define party also as a function generator. The different function generators (participant_generator functions and a party function) coordinate their execution through yield, send and next so that the problem is simulated with all coroutines.

You could use the main() function defined in figure 3 to test your program or write your own. If you write your own main() function, please make sure the final output of the program is the total number of boys, the total number of girls and the total number of pairs in a row (e.g. 50 50 50 or 30 70 30).

```

def main():
    p = party()
    p.next() #start the party generator
    p.send(10)

    tasks = [p]
    num_boys = 0
    num_girls = 0
    for i in range(10):
        if random.randint(0,1) == 0:

```

```

    pg = participant_generator(i, True, p, num_boys)
    num_boys += 10
else:
    pg = participant_generator(i, False, p, num_girls)
    num_girls += 10
p.send((i, pg)) #make party aware of other coroutines
tasks.append(pg)

t = time.time()
while len(tasks) > 1 and time.time() - t < 5:
    task = random.choice(tasks)
    try:
        task.next()
    except StopIteration:
        tasks.remove(task)
        time.sleep(0.01)

print num_boys , num_girls , pair
assert min(num_boys, num_girls) == pair

if __name__ == '__main__':
    main()

```

figure 3: main() function

Hint: Since send function could only carry one argument, you could define a class type to carry multiple pieces of information in one send.

Note: All codes should be put into file **match3.py**

Step 4:

Create a readme file that includes your name and the course number (CSCI 4900 or CSCI 6900) you enrolled in.

Put **match1.py**, **match2.py** and **match3.py** (if available) into one folder lastname_lab10.

Submit the folder to cs4900a through nike(submit cs4900a lastname_lab010).

Grading Rubric (Total: 70 + 30)

Deliverables	Total	Points	Comments
Subjective Survey	5		
Programs	65		
step 1	30		
step 2	35		
Bonus	30		
step 3	30		

CSCI 4900/6900 Lab/Project Specification (Week 11-13)

Lab #11

Sleeping Barber Simulation

Goals

In this lab you will use your prior knowledge of Java Threads, Scala Actors and Python Coroutines and your prior hands-on experience implementing the party matching simulation with these three

models to program a sleeping barber simulation using concurrency constructs from the three different models. You are required to understand the simulation setting, synthesize pieces of knowledge with different programming models and realize the system in Java, Scala and Python.

Due Date

On each of the following date, you should submit one deliverable part of the lab (details described in submission portion of this document at the end).

1. Apr 9th, 2013 (Tuesday) 11:59 pm.
2. Apr 16th, 2013 (Tuesday) 11:59 pm.
3. Apr 19th, 2013 (Friday) 11:59 pm.

Late Penalty

For each deliverable part, the late penalty described in the course policies applied.

Collaboration Policy

This is an individual lab, but you are welcome to post any questions or start a discussion on piazza. Please refer to the course policy for questions that are allowed. You may also discuss general strategies with your classmates. However, you must complete coding on your own. **No copying of or detailed line by line cooperation on code is allowed.**

Lab Description

1. The Simulation Problem – Sleeping Barbers

A barbers shop has a particular number of barbers working in it. Different customers come to the shop to have different kinds of barbering services that take different amounts of time to finish. When there's no customer in the shop, the barbers just rest. If a customer comes and there are available barbers (barbers are resting), one of the barbers should provide the customer the barbering service he wants. When all the barbers are serving customers, a newly arriving customer should wait in the shop. However, if the waiting space is already full, that customer will leave directly without having any service. Barbers keep a record of the total time of service they provide and do not serve anymore customers when that time reaches/exceeds their maximum working time.

Here is a list of key points in the simulation:

- Barbers have a limited amount of total work time to provide service
- Barbers rest when there's no customer to serve
- Barbers cannot rest when there are waiting customers
- Customers go to the barber's shop to get a certain service that takes a particular amount of time to finish
- Customers in the waiting area should be served on a first-come-first-served base.
- Customers should stay in the waiting area if it is not full. Otherwise, customers should leave the shop immediately.
- Customers do not pick barbers. They will be served by the first available barber.

2. The Simulation Settings

To set up a simulation, the following arguments should be specified:

- the capacity of the waiting area in the barber's shop
- the number of barbers working in the barber's shop
- the maximum total service time a barber can provide over all their customers (in milliseconds, same for all barbers,)
- the number of customer generators that send customers to the barber's shop
- the number of customers each generator will send (same for all generators)
 - ❖ the total number of customers sent to the barber's shop is decided by above two arguments
- the minimum amount of time a barbering service may take (in milliseconds, same for all generators)
- the maximum amount of time a barbering service may take (in milliseconds, same for all generators)
 - ❖ for any particular customer, pick a random number between the minimum and maximum as the time of the service that the customer requires to have
- the minimum interval between generation of customers (in milliseconds, same for all generators)
- the maximum interval between generation of customers (in milliseconds, same for all generators)
 - ❖ after sending a customer to the barber's shop, the generator should pause for a random time between this minimum and maximum before sending the next customer

3. Output Requirements

The simulation should print out the events that happen in the barber's shop according to their order of occurrence. To be specific, the following and only the following events should be reported:

1. A customer starts waiting
2. A customer leaves the barber's shop immediately
3. A barber starts to serve a customer
4. A barber finishes serving a customer
5. A barber completes his total working time

Here is a table of the corresponding output format of the above events:

*contents in square brackets are program variables

Event No.	Output Format
1	Customer [<i>customer_id</i>] starts waiting at position [<i>position</i>]
2	Customer [<i>customer_id</i>] leaves without service
3	Barber [<i>barber_id</i>] starts serving [<i>customer_id</i>]
4	Barber [<i>barber_id</i>] finishes serving [<i>customer_id</i>]
5	Barber [<i>barber_id</i>] quits

Also, at the end of simulation (no more eligible events as described above could happen), the program should print out the following pieces of information:

1. total number of customers sent to the barber's shop
2. total number of customers that left without having service

3. total number of customers that are still waiting in the barber's shop
4. total number of customers that are served

Here is a table of corresponding output format of above information:

*contents in square brackets are program variables

Info No.	Output Format
1	Total number of customers: [total]
2	Number of customers that left: [left]
3	Number of customers waiting: [wait]
4	Number of customers served: [served]

Here is one sample simulation output from a demo program that describes the events that happened in the barber's shop and gathers the final records:

*10 waiting seats, 5 barbers and 30 customers

```

Customer 29 starts waiting at position 1
Barber 3 starts serving 29
Customer 26 starts waiting at position 1
Barber 4 starts serving 26
Customer 24 starts waiting at position 1
Barber 1 starts serving 24
Customer 20 starts waiting at position 1
Barber 0 starts serving 20
Customer 27 starts waiting at position 1
Barber 2 starts serving 27
Customer 23 starts waiting at position 1
Customer 28 starts waiting at position 2
Customer 25 starts waiting at position 3
Customer 22 starts waiting at position 4
Customer 21 starts waiting at position 5
Customer 14 starts waiting at position 6
Customer 10 starts waiting at position 7
Customer 17 starts waiting at position 8
Customer 13 starts waiting at position 9
Customer 15 starts waiting at position 10
Customer 18 leaves without service
Customer 19 leaves without service
Customer 11 leaves without service
Customer 12 leaves without service
Customer 16 leaves without service
Customer 5 leaves without service
Customer 7 leaves without service
Customer 4 leaves without service
Customer 3 leaves without service
Customer 1 leaves without service
Customer 8 leaves without service
Customer 2 leaves without service
Customer 6 leaves without service
Customer 9 leaves without service
Customer 0 leaves without service
Barber 0 finishes serving 20
Barber 0 starts serving 23
Barber 3 finishes serving 29
Barber 3 starts serving 28
Barber 1 finishes serving 24
Barber 1 starts serving 25
Barber 2 finishes serving 27
Barber 2 starts serving 22
Barber 4 finishes serving 26
Barber 4 starts serving 21
Barber 0 finishes serving 23
Barber 0 starts serving 14
Barber 2 finishes serving 22
Barber 2 starts serving 10
Barber 3 finishes serving 28

```

```

Barber 3 starts serving 17
Barber 1 finishes serving 25
Barber 1 starts serving 13
Barber 4 finishes serving 21
Barber 4 starts serving 15
Barber 0 finishes serving 14
Barber 2 finishes serving 10
Barber 1 finishes serving 13
Barber 4 finishes serving 15
Barber 3 finishes serving 17
Total number of customer: 30
Number of customer that left: 15
Number of customer waiting: 0
Number of customer served: 15

```

4. Deliverable Portions

This lab has three different deliverable portions. You could choose any order to work on these portions. But you should submit one deliverable portion on each of the three dates specified out in beginning of this document titled **Due Dates**.

Portion in Java Threads Model

One deliverable portion of this lab is to implement the simulation with Java Threads. You may use the given code skeleton in **skeleton/java** folder or create your own. If you decide to create your own program, it should accept setting arguments in the order as described in [The Simulation Settings](#) and produce output as described in [Output Requirements](#).

The given code skeleton has a full implementation of a **Main** class and a **CustomerGenerator** class. The **CustomerGenerator** class send customers to the barber shop and the **Main** class sets up the simulation with setting arguments described in [The Simulation Settings](#) above and finishes with a simple assertion. The assertion checks whether the total number of customers sent to the barber shop is equal to the sum of the different customers (served, left without service and those still waiting in the shop) as recorded by the shop.

The given code skeleton has a partial implementation of **Barber**, **Customer** and **Shop** classes. To work with the **Main** and **CustomerGenerator** classes, you should not change the number, type or order of parameters taken by the constructors of these classes. However, you may rename them according to your preferences. The **Barber** and **Customer** classes implement the **Runnable** interface and you have to define the corresponding **run()** methods for them. The **shop** class records the shared data of the barber shop and it is a good practice to utilize synchronization mechanisms there. The method **isInService()** is used by the **Main** class to determine the timing of the assertion. It takes no arguments and returns a Boolean value. Only after the shop is not in service, i.e. all waiting customers are served or all barbers have quit, will the main function in the **Main** class evaluate the assertion.

Note that all source code should be put into a **barber** package.

Portion in Scala Actors Model

One deliverable portion of this lab is to implement the simulation with Scala Actors. You may use the given code skeleton in the **skeleton/scala** folder or create your own. If you decide to create your own program, it should accept setting arguments in the order as described in [The Simulation Settings](#) and produce output as described in [Output Requirements](#).

The given code skeleton has a full implementation of a **SleepingBarbers** object and **CustomerGenerator** class. The **SleepingBarbers** object sets up the simulation with setting arguments described in [The Simulation Settings](#) above and finishes with a simple assertion. The assertion checks whether the total number of customers sent to the barber shop is equal to the sum of different

customers (served, left without service and those still waiting in shop) as recorded by the shop. Messages used for assertion (`assertReq`, requesting assertion data and `assertMsg`, carry assertion data) as well as notification (`nomore`, sends by a generator to indicate that all customers it is supposed to generate have been all sent to the shop) are also provided.

The given code skeleton has partial implementations of the **Barber**, **Customer** and **Shop** classes. To work with the **SleepingBarbers** object and the **CustomerGenerator** class, you should not change the number, type or order of parameters taken by these classes' declarations. However, you may rename them according to your preferences. All these class are inherited from the Actor class and you have to define the corresponding `act()` methods for them. The **shop** class implements an `isInService()` methods to determine the timing of the assertion. It takes no arguments and returns a Boolean value. Only after the shop is not in service, i.e. all waiting customers are served or all barbers have quit, will the **shop** reply to the **SleepingBarbers** object with arguments for assertion in the `assertMsg` message.

Note that all source code should be put into a **barber** package.

Portion in Python Coroutine Model

One deliverable portion of this lab is to implement the simulation with Python Coroutines. You may use the given code skeleton in the **skeleton/python** folder or create your own. If you decide to create your own program, it should accept setting arguments in the order as described in [The Simulation Settings](#) and provide output as described in [Output Requirements](#).

The given code skeleton has full implementation of a **main()** function and a **customer_generator()** function. The **main()** function sets up the simulation with setting arguments described in [The Simulation Settings](#) above and finishes with a simple assertion. The assertion checks whether the total number of customers sent to the barbers shop is equal to the sum of different customers (served, left without service and those still wait in shop) as recorded by the shop. A global variable (`num_active_generator`) used for recording the progress of the **customer_generator()** function is also provided (an active customer generator still has customers that have not been sent to the barber shop yet).

The given code skeleton has a partial implementation of the **shop** class and the **barber()** and **customer()** functions. To work with the **main()** and **customer_generator()** functions, you should not change the number or order of parameters taken by **barber()** and **customer()** functions. However, you may rename them according to your preferences. Function **barber()** and **customer()** are generator functions in which you should utilize `yield`. The **shop** class records the shared data of the barber shop. The method `is_in_service()` defined in the **shop** class is used by the **main()** function to determine the timing of the assertion. It takes no arguments and returns a Boolean value. Only after the shop is not in service, i.e. all waiting customers have been served or all barbers have quit, will the **main()** function evaluate the assertion.

5. Submissions

This lab contains three deliverable portions (as described in [Deliverable Portions](#) above). You should submit **exactly one complete** portion on each of the following due date with the order of your own choice:

- Due Date 1: Apr 9th, 2013 (Tuesday) 11:59 pm.
- Due Date 2: Apr 16th, 2013 (Tuesday) 11:59 pm.
- Due Date 3: Apr 19th, 2013 (Friday) 11:59 pm.

To submit a portion, put all your source code for that portion (do not include compiled binaries since they are large) into a folder `lastname_lab11_X` (X should be either `java`, `scala` or `python` that describes the source code you are submitting) and submit the folder to `cs4900a` through `nike` (submit

cs4900a lastname_lab11_X)

6. Grading Rubrics

Each delivery portion will be tested against 10 different combinations of simulation settings (details of setting arguments are described in 2.) For each of these 10 settings, the outputs will be checked against the following rubrics:

1. Total number of customers is equal to the specification of setting parameters
2. Total number of customers is equal to the sum of different types of customers (left, wait, served)
3. Number of different types of customers is equal to the corresponding event outputs (e.g. number of served customers equals the number of lines printed as “Barber ... finishes serving ...”; number of customers that left equals number of lines printed as “Customer ... leaves without service”)
4. Event output sequence is reasonable (e.g. start serving happened before finish serving, waiting happened before start serving)

Any one violation of above rubrics in an output causes deduction of 2 points until the deductions accumulates to 10, which is the total points for that output.

So, the total of this lab is:

10 points per running output * 10 running settings per deliverable portion * 3 deliverable portions = 300 points

FIGURE 38 LAB MATERIALS OF CSCI4900 FOR SPRING 2013 STUDY

CSCI 4900/6900 Midterm Exam I

Single-Lane Bridge Problem as Shared Memory System

A single-lane bridge is wide enough to permit only a single lane of traffic. That is, the bridge permits only one-way traffic at any time and cars exit the bridge according to their order of entering the bridge. To simplify the problem, we will define the cars that move from left to right as red cars and those that move from right to left as blue cars.

Figure 1 contains the pseudocode implementation of a single-lane bridge simulation as a shared memory concurrent system. Read the code and answer the following questions based on the above problem description.

Make use of the sequence diagram templates to help you think about the questions.

1. Consider the following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
```

```
PARA
    redCarA.run()
    redCarB.run()
END_PARA
```

Suppose that **redCarA** has just returned from the *redEnter()* method on line 9 and **redCarB** starts its *run()* method.

Decide if each of the scenarios below (a-c) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

Example:

- (a) **redCarB** calls the *redEnter()* method, then returns..

☒ YES NO

Explanation:

Since **redCarA** has already returned from the *redEnter()* method, it no longer holds the lock. When **redCarB** executes the *redEnter()* method, it could get the lock, pass the conditional check since **redCarA** is on the bridge and then returns from *redEnter()* method.

- (a) **redCarB** calls the *redEnter()* method, then blocks on the EXC_ACC marker on line 10.

YES NO

Explanation:

- (b) **redCarB** calls the *redEnter()* method, returns, then calls *redExit()* on line 19, then returns also.

YES NO

Explanation:

- (c) **redCarB** calls *redEnter()* but a context switch occurs before the call returns, and **redCarA** calls *redExit()* and blocks on EXC_ACC marker on line 20.

YES NO

Explanation:

2. Consider the following code.

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
```

```
PARA
    redCarA.run()
    redCarB.run()
END_PARA
```

Now, suppose **redCarA** has called the *redEnter()* method on line 9 but has **not** returned and **redCarB** starts its *run()* method.

Decide if each of the scenarios below (d-g) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

- (d) **redCarB** calls *redEnter()*, then returns.

YES NO

Explanation:

- (e) **redCarB** calls *redEnter()*, then blocks on the EXC_ACC marker on line 10.

YES NO

Explanation:

- (f) **redCarB** calls *redEnter()*, returns, and then calls *redExit()* on line 19, and also returns.

YES NO

Explanation:

- (g) **redCarB** calls *redEnter()* and a context switch occurs. Then **redCarA** blocks on `EXC_ACC` marker on line 10.

YES NO

Explanation:

3. Consider the following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)
```

```
PARA
    redCarA.run()
    redCarB.run()
    blueCarA.run()
END_PARA
```

Suppose **redCarA** has just returned from the *redEnter()* method on line 9. **blueCarA** starts its *run()* method, calls *blueEnter()* on line 29 and has not yet returned.

Decide if each of the scenarios below (a-c) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

- (a) The *blueEnter()* method returns, and **blueCarA** calls the *blueExit()* method on line 39.

YES NO

Explanation:

- (b) **redCarB** starts its *run()* method, calls the *redEnter()* method on line 9 and then returns, then calls the *redExit()* method on line 19, and blocks on the `EXC_ACC` marker on line 20.

YES NO

Explanation:

- (c) **redCarB** starts its *run()* method, calls the *redEnter()* method on line 9 and then blocks on `EXC_ACC` marker on line 10.

YES NO

Explanation:

4. Consider the following code:

```

bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)

```

PARA

```

    redCarA.run()
    redCarB.run()
    blueCarA.run()

```

END_PARA

Suppose **redCarA** and **redCarB** have both returned from the *redEnter()* method on line 9. Then **blueCarA** starts its *run()* method, calls the *blueEnter()* method on line 29 and starts execution of the WAIT statement on line 33.

Decide if each of the scenarios below (e-j) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(e) **redCarB** calls the *redExit()* method on line 19 and then returns.

YES NO

Explanation:

(f) **redCarB** calls the *redExit()* method on line 19 and then blocks on the EXC_ACC marker on line 20.

YES NO

Explanation:

(g) **redCarA** calls the *redExit()* method on line 19 and starts execution of the WAIT statement on line 22.

YES NO

Explanation:

(h) **redCarA** calls the *redExit()* method on line 19 and returns. **blueCarA** returns from the WAIT statement on line 33, and returns from the *blueEnter()* method on line 29.

YES NO

Explanation:

(i) **redCarA** calls the *redExit()* method on line 19 and returns. **redCarB** calls the *redExit()* method on line 19 and returns. **blueCarA** returns from the WAIT statement on line 33 and then returns from the *blueEnter()* method on line 29.

YES NO

Explanation:

(j) **redCarB** calls the *redExit()* method on line 19 but has not yet returned. **blueCarA** finishes execution of the WAIT statement on line 33, but blocks on EXC_ACC marker on line 31.

YES NO

Explanation:

5. Consider the following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)
```

PARA

```
    redCarA.run()
    redCarB.run()
    blueCarA.run()
```

END_PARA

Suppose **redCarA** has called the *redEnter()* method on line 9 but has not returned. Then **redCarB** starts its *run()* method and called the *redEnter()* method but also has not returned.

Decide if each of the scenarios below (k-t) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(k) **redCarB** returns from the *redEnter()* method, then calls the *redExit()* method on line 19 and returns.

YES NO

Explanation:

(l) **redCarB** returns from the *redEnter()* method, then calls the *redExit()* method on line 19 and starts execution of WAIT statement on line 22.

YES NO

Explanation:

(m) **redCarB** returns from the *redEnter()* method, then calls *redExit()* method on line 19 and blocks on the EXC_ACC marker on line 20.

YES NO

Explanation:

- (n) **redCarA** returns from the *redEnter()* method, then calls the *redExit()* method on line 19 and blocks on the `EXC_ACC` marker on line 20.

YES NO

Explanation:

- (o) **redCarA** returns from the *redEnter()* method, then calls the *redExit()* method on line 19 and starts execution of `WAIT` statement on line 22.

YES NO

Explanation:

- (p) **blueCarA** starts its *run()* method, calls the *blueEnter()* method and returns.

YES NO

Explanation:

- (q) **blueCarA** starts its *run()* method, calls the *blueEnter()* method and blocks on the `EXC_ACC` marker on line 30.

YES NO

Explanation:

- (r) **blueCarA** starts its *run()* method, calls the *blueEnter()* method on line 29, and then starts execution of `WAIT` statement on line 33.

YES NO

Explanation:

- (s) **blueCarA** starts its *run()* method, calls the *blueEnter()* method on line 29, returns, and then calls the *blueExit()* method on line 39, and blocks on the `EXC_ACC` marker on line 40.

YES NO

Explanation:

- (t) **blueCarA** starts its *run()* method, calls the *blueEnter()* method on line 29, returns, and then calls *blueExit()* method on line 39, and start execution of `WAIT` statement on line 43.

YES NO

Explanation:

6. Consider following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)
```

```
PARA
    redCarA.run()
    redCarB.run()
    blueCarA.run()
END_PARA
```

Which of the following outputs are possible? Circle YES if the output is possible; otherwise, circle NO and indicate the impossible output line with an arrow on its right.

Example:

(a) YES

NO

red 2 ←

red 1 ←

(a) YES NO

red 1

red 2

red 3

(b) YES NO

red 1

blue 1

red 2

(c) YES NO

red 1

blue 2

red 2

(d) YES NO

blue 1

blue 2

red 1

(e) YES NO

red 1
blue 1
blue 2
red 2

(f) YES NO

blue 1
blue 2
red 1
red 2
blue 3

7. Use pseudocode to modify the definition of the Bridge class such that red and blue cars may use the bridge in turns. Therefore, only the following two output sequences are possible.

Output possibility 1:

red 1
blue 1
red 2
blue 2
red 3
blue 3
red 4
...

Output possibility 2:

blue 1
red 1
blue 2
red 2
blue 3
red 3
blue 4
...

Please write down your new definition of Bridge class in pseudocode. (**Hint:** the minimum modification only needs one more class variable and the modification of *redEnter()* and *blueEnter()* methods.) You can make the modification directly on Figure 1.

CSCI 4900/6900 Midterm Exam II

Single-Lane Bridge Problem as Message Passing System

A single-lane bridge is wide enough to permit only a single lane of traffic. That is, the bridge permits only one-way traffic at any time and cars exit the bridge according to their order of entering the bridge. To simplify the problem, we will define the cars that move from left to right as red cars and those that move from right to left as blue cars.

Figure 1 contains the pseudocode implementation of a single-lane bridge simulation as a message passing concurrent system. Read the code and answer the following questions based on the above problem description.

Make use of the sequence diagram templates to help you think about the questions.

1. Consider the following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)

PARA
  bridge.start()
  redCarA.start()
  redCarB.start()
END_PARA
```

Suppose that **redCarA** has just finished execution of line 56 and **redCarB** starts its *start()* method.

Decide if each of the scenarios below (a-c) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

Example:

(a) **redCarB** sends a *redEnter* message and receives a *succeedEnter* message.

☒ YES NO

Explanation:

Since **redCarA** has already received the *succeedEnter* message but not yet sent the *redExit* message, when the *redEnter* message sent by **redCarB** is received by **bridge**, the bridge will pass the conditional check without any blue cars and **redCarB** may receive a *succeedExit* message to enter the bridge.

(a) **redCarB** sends a *redEnter* message and then waits to receive a message of any type.

YES NO

Explanation:

(b) **redCarB** sends a *redEnter* message, receives a *succeedEnter* message, sends a *redExit* message, and receives a *succeedExit* message.

YES NO

Explanation:

- (c) **redCarB** sends a *redEnter* message, but before it receives a *succeedEnter* message, **redCarA** sends a *redExit* message which is received by the bridge.

YES NO

Explanation:

2. Consider the following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
```

```
PARA
    bridge.start()
    redCarA.start()
    redCarB.start()
END_PARA
```

Suppose **redCarA** has sent a *redEnter* message but has not received any messages yet. **redCarB** calls its *start()* method at this time.

Decide if each of the scenarios below (d-g) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

- (d) **redCarB** sends *redEnter* message and receives *succeedEnter* message.

YES NO

Explanation:

- (e) The bridge receives the *redEnter* message sent by **redCarA**. **redCarB** sends a *redEnter* message., then waits to receive a message of any type.

YES NO

Explanation:

- (f) **redCarB** sends *redEnter* message, receives *succeedEnter* message, sends *redExit* message and also receives *succeedExit* message.

YES NO

Explanation:

- (g) **redCarB** sends *redEnter* message, but before it receives *succeedEnter* message, **redCarA** receives `MESSAGE.succeedEnter(2)`.

YES NO

Explanation:

3. Consider the following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)
```

```
PARA
    bridge.start()
    redCarA.start()
    redCarB.start()
    blueCarA.start()
END_PARA
```

Suppose **redCarA** has sent a *redEnter* message and received `MESSAGE.succeedEnter(1)`. **blueCarA** starts its *start()* method, and sends a *blueEnter* message but has not received any messages.

Decide if each of the scenarios below (a-c) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(a) **blueCarA** receives a *succeedEnter* message and then sends *blueExit* message.

YES NO

Explanation:

(b) **redCarB** starts its *start()* method, sends *redEnter* message, receives *succeedEnter* message, then sends *redExit* message and then **blueCarA** receives a *failEnter* message.

YES NO

Explanation:

(c) **redCarB** calls its *start()* method, sends *redEnter* message. The bridge receives the *redEnter* message sent by **redCarB** and then the *blueEnter* message.

YES NO

Explanation:

4. Consider following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)
```

```

PARA
    bridge.start()
    redCarA.start()
    redCarB.start()
    blueCarA.start()
END_PARA

```

Suppose **redCarA** and **redCarB** have both sent *redEnter* messages and received *succeedEnter* messages. Then **blueCarA** calls its *start()* method, sends *blueEnter* message.

Decide if each of the scenarios below (e-j) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(e) **redCarB** sends *redExit* message and receives *succeedExit* message.

YES NO

Explanation:

(f) **redCarB** sends *redExit* message. The bridge sends a *failEnter* message to **blueCarA** and then receives the *redExit* message from **redCarB**.

YES NO

Explanation:

(g) **redCarB** sends *redExit* message and receives *failExit* message.

YES NO

Explanation:

(h) **redCarA** sends *redExit* message and receives *succeedExit* message. **blueCarA** receives *failEnter* message, sends *blueEnter* message again, and receives *succeedEnter* message.

YES NO

Explanation:

(i) **redCarA** sends *redExit* message and receives *succeedExit* message. **redCarB** sends *redExit* message and receives *succeedExit* message. **blueCarA** receives *failEnter* messages, sends *blueEnter* message again, and receives *successEnter* message.

YES NO

Explanations:

(j) **redCarB** sends *redExit* message. The bridge receives the *redExit* message and then receives the *blueEnter* message.

YES NO

Explanations:

5. Consider following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)
```

PARA

```
    bridge.start()
    redCarA.start()
    redCarB.start()
    blueCarA.start()
```

END_PARA

Suppose **redCarA** has sent *redEnter* message but has not received any message yet. Then **redCarB** starts its *start()* method, sends *redEnter* message but does not receive any message yet.

Decide if each of the scenarios below (k-t) could happen immediately after the above. Circle YES if the sequence is possible; otherwise, circle NO. Then please provide a brief explanation of your reasoning.

(k) **redCarB** receives *succeedEnter* message, then sends *redExit* message and receives *succeedExit* message.

YES NO

Explanation:

(l) **redCarB** receives *succeedEnter* message, then sends *redExit* message and receives *failExit* message.

YES NO

Explanation:

(m) **redCarB** receives *succeedEnter* message, then sends *redExit* message and receives `MESSAGE.succeedExit(2)`.

YES NO

Explanation:

(n) **redCarA** receives *succeedEnter* message, then sends *redExit* message and receives `MESSAGE.succeedExit(2)`.

YES NO

Explanation:

- (o) **redCarA** receives *succeedEnter* message, then sends *redExit* message and receives *failExit* message.

YES NO

Explanation:

- (p) **blueCarA** calls its *start()* method, sends *blueEnter* message and receives *succeedEnter* message.

YES NO

Explanation:

- (q) **blueCarA** calls its *start()* method, sends *blueEnter* message. The bridge receives *blueEnter* message first and then the *redEnter* message sent by **redCarA**.

YES NO

Explanation:

- (r) **blueCarA** starts its *start()* method, sends *blueEnter* message and receives *failEnter* message.

YES NO

Explanation:

- (s) **blueCarA** starts its *start()* method, sends *blueEnter* message, receives *succeedEnter* message, and then sends *blueExit* message. The bridge send *failEnter* to both **redCarA** and **redCarB** and then receives the *blueExit* message.

YES NO

Explanation:

- (t) **blueCarA** starts its *start()* method, sends *blueEnter* message, receives *succeedEnter* message, and then sends *blueExit* message, and receives *failExit* message.

YES NO

Explanation:

6. Consider following code:

```
bridge = new Bridge()
redCarA = new RedCar(bridge)
redCarB = new RedCar(bridge)
blueCarA = new BlueCar(bridge)
```

PARA

```

bridge.start()
redCarA.start()
redCarB.start()
blueCarA.start()
END_PARA

```

Which of the following outputs are possible? Circle YES if the output is possible; otherwise, circle NO and indicate the impossible output line with an arrow on its right.

Example:

(a) YES ☒ NO

red 2 ←
red 1 ←

(a) YES NO

red 1
red 2
red 3

(b) YES NO

red 1
blue 1
red 2

(c) YES NO

red 1
blue 2
red 2

(d) YES NO

blue 1
blue 2
red 1

(e) YES NO

red 1
blue 1
blue 2
red 2

(f) YES NO

```

blue 1
blue 2
red 1
red 2
blue 3

```

7. Modify the definition of Bridge class with pseudocode such that red and blue cars could use the bridge in turns. Therefore, only the following two output sequences are possible.

Output possibility 1:

```

red 1
blue 1
red 2
blue 2
red 3
blue 3
red 4
...

```

Output possibility 2:

```

blue 1
red 1
blue 2
red 2
blue 3
red 3
blue 4
...

```

Please write down your new definition of Bridge class in pseudocode. (**Hint:** the minimum modification only needs one more class variable and the modification of actions taken on receiving *redEnter* and *blueEnter* messages.) You can make the modifications directly on Figure 1.

FIGURE 39 MIDTERM EXAM OF CSCI4900 FOR SPRING 2013 STUDY

Scenario

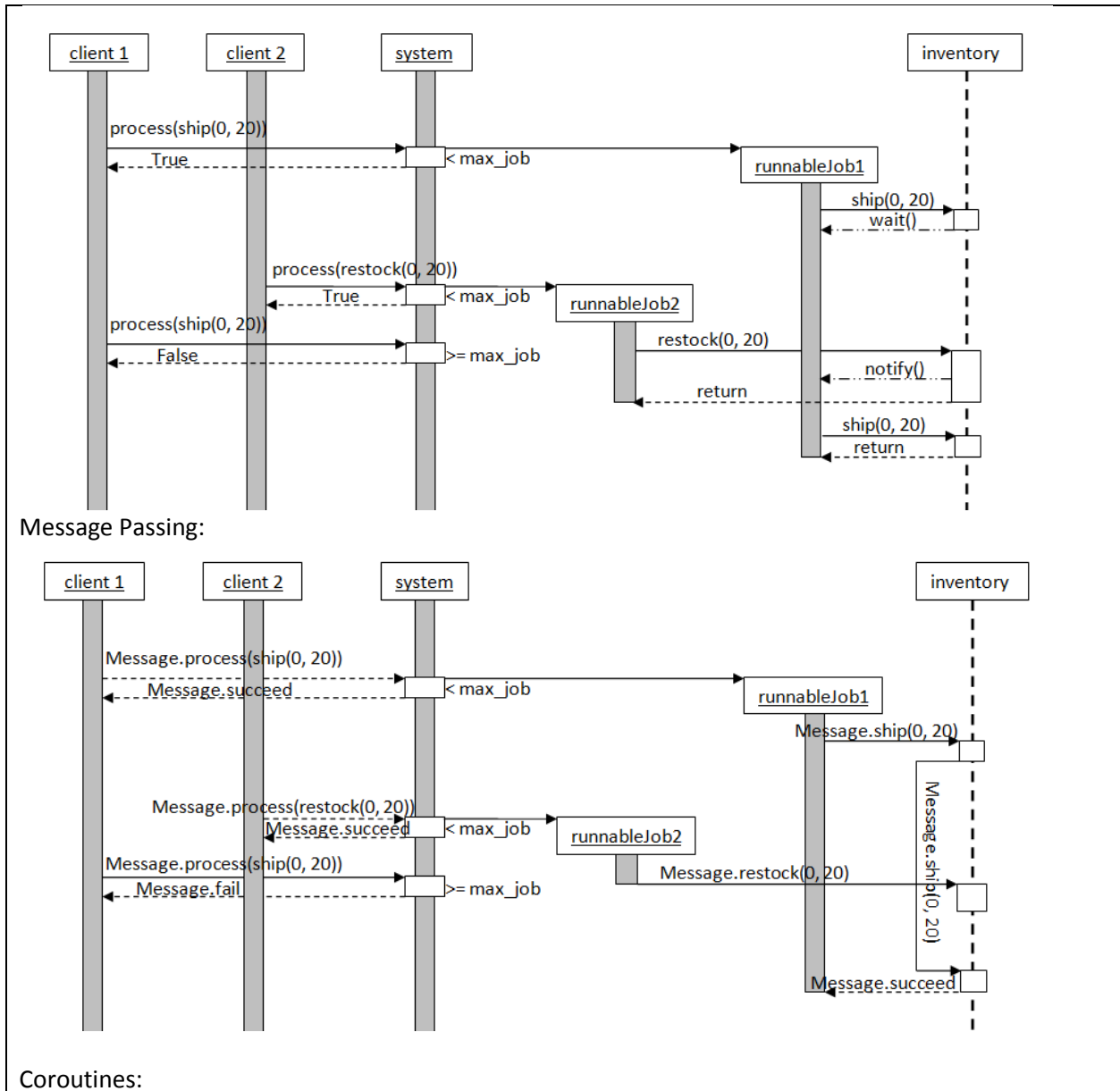
FBD (fictional book dealer) is the largest book vendor company in the (fictional) world. The company works with two kinds of clients. One is book publisher, who provides book to the company. The other is book retailer, who requests book from the company. To better meet their increasing number of clients (both book publisher and book retailer) and business, the company is eager to have a new generation of warehouse system for the automated management of their world wide book inventory. After a long term discussion and consultant with different departments in the company, the following (fictionally simplified) system requirements are discerned:

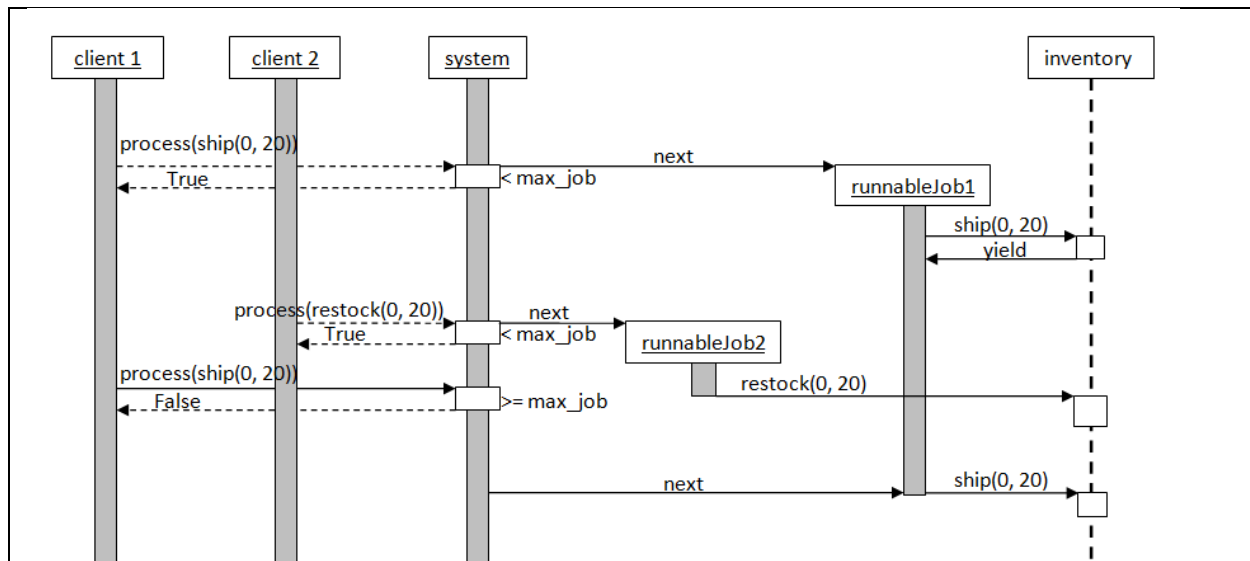
1. The system should process two types of jobs, shipping and restocking. It has a limitation on the number of jobs of each type being processed at any moment so that it will not use up the resource of the computing facility it resides on. The system should contain at least one book inventory to manage storage.
2. Each job contains multiple works and a work is one increment or decrement of a certain book. A shipping job contains only work of decreasing a book's stock amount and a restocking job contains only work of increasing a book's stock amount. Both shipping and restocking jobs should be viewed as transactions, which means that a job is fulfilled either completely or not at all. When the stock of any one kind of book in the job is insufficient, a shipping job should automatically retry that decreasing work. All increasing work in restocking job, however, should always be fulfilled on the first attempt.
3. To maximize the throughput (number of jobs being processed in a certain amount of time), we would like the jobs to be processed concurrently in the system.
4. Client programs are not part of the system development goal. But to test the system, mock client programs are created to guarantee that:
 - Multiple clients (publishers and retailers) will be able to connect concurrently to the system and request their jobs to be processed.
 - The clients could keep making job requirements without waiting for the previous job to complete.
 - If the system reaches its limitation on the number of jobs of a certain type, the request of processing that type of job by a client at the moment will fail (and the client will be responsible to re-request that job to be processed later.)

Models

Above system requirements are finally being analyzed by IT stuff members of the company and they came up with the following designs in three different models (shared memory, message passing, and coroutines). The behavior of the system is illustrated with UML sequence diagram below. All designs use the same concept of runnable jobs and support automatic retries of failed job.

Shared Memory:





Mockups

To further evaluate and decide which model to use, the IT stuff members in the company want to develop mockups first. These mockup systems eliminate all tedious utilities and settings of the real problem, but focus on the core functionalities. The mockups have these further simplified assumptions:

1. A book in the mockup system is only differentiated from other book by its name. Further, only 5 books are defined in the mockup system. They are:
 - Gone with the wind
 - How to kill a mock bird
 - Programming Concurrency
 - Operating Systems
 - Introductory Robotics
2. A client in the mockup system makes requests of both shipping and restocking jobs randomly and alternatively.
3. The mockup system only contains one simplest book inventory that records the amount of different books. This inventory supports both shipping and restocking of a certain kind of book as defined in the mockup systems.

The input of mockup systems are four program arguments:

1. num_of_clients: the number of clients that concurrently making requests to the system
2. client_life_time: the milliseconds for a client to live and make requests to the system (so that the simulation ends when all clients quit)
3. max_num_of_job_for_each_type: the maximum number of jobs of each type running in the system at any time
4. max_job_length: the maximum number of works contained in a job

The output of mockup systems are print outs of critical events. Two events are identified by IT stuff members in the company:

1. A work is added to a job in client program
 - Output Format:
 - A [job type] work([book name], [amount to change]) will be performed by job id = [job id]
 - Example Output:

- A shipping work(Introductory Robotics , 11) will be performed by job id = 4378

2. A work is performed by a job that modifies inventory records

- **Output Format:**

- A [job type] work([book name], [old quantity] -> [new quantity]) is performed by job id = [job id] and seqNum = [sequence number]

- **Example Output**

- A restocking work(Operating Systems , 1103 -> 1112) is performed by job id = 4356 and seqNum = 1

A sequence number (seqNum) is the number assigned to a job before it starts in the system. This number indicates how many other jobs (of same type) are running when it starts. Since the system has a limitation on the total number of jobs running simultaneously, this sequence number should never reaches or exceeds the limitation.

Testing

A simple testing script is written before developing the mockups. This script checks against mockup outputs for following violations:

1. Work execution violation

A job execution violation is discerned as the appearance of a work performed before it is added as shown in following example:

no output reading: A restocking work(Gone with the wind, 17) will be performed by job id = 246

A restocking work(Gone with the wind , 838 -> 855) is performed by job id = 246 and seqNum = 1

=>

unscheduled restocking work with job id = 2466 (Gone with the wind, 838 -> 855)

2. Job sequence number violation

A job sequence number violation is discerned as the appearance of a sequence number that is equal to or greater than the maximum number of jobs allowed as shown in following example:

(mockup setting specifies maximum number of job as 3)

A restocking work(Gone with the wind , 838 -> 855) is performed by job id = 246 and seqNum = **3**

=>

invalid job sequence number 3 that equals or exceeds maximum 3

3. Negative book quantity violation

A negative book quantity violation is discerned as a negative quantity of a book in a work performed on it as shown in following example:

A shipping work(Operating Systems , **-2** -> 7) is performed by job id = 121 and seqNum = 1

=>

negative stock quantity -2 -> 7

4. Quantity update discrepancy violation

A quantity update discrepancy violation is discerned as an inconsistency of quantity of a book between two continuous works performed on it as shown in following example:

A restocking work(Operating Systems , 1103 -> **1112**) is performed by job id = 4356 and seqNum = 1

...works on other books...

A shipping work(Operating Systems , **231** -> 220) is performed by job id = 4378 and seqNum = 3

=>

quantity update discrepancy: original amount of "Operating System" is **1112** not **231**

5. Concurrency warning violation

A concurrency warning violation is discerned as low throughput of the system as shown in following example:

(only three jobs are performed according to print outs)

=>

concurrency warning: total number of jobs (3) gets run in server is small.

This could probably due to that the mock client programs not generating many jobs or that potential deadlocks exist in the system.

The Contest

The IT stuff members in the company programmed out three mockup systems in Java, Scala and Python respectively. However, after testing, they see various violations reported for all three models and it becomes a headache for them.

So the fictional company decided to give some real dollars to programmers for the debugging work. The first who successfully debug one mockup (either in Java, Scala or Python) will get \$20 and a total of \$60 is there for you to win.

The buggy codes are available online through course calendar and ELC of UGA's 4900 Programming with Concurrency class. Two utility files are also available. A makefile can be used to compile and run three mockup systems towards different combination of input settings. A script (book_checker.pyc) can be used to check program outputs.

Following rules of the contest must be followed:

1. For each model, put all codes in a folder lastname_contest_model and submit it to cs4900a through nike. (model should be either java, scala or python)
2. Only last submitted version will be reviewed in the contest. So do not take chance and spam the nike server. Only submit a version when you are confident.
3. The decision of speed is based on timestamp of a submitted folder:
 - lastname_contest_java_Wed_Apr_24_23_34_26_2013 => 04/24/2013, 23:34:26 Mr. lastname submit code in java
 - If two submissions have exactly the same timestamp and both of them successfully find and correct all bugs, the two participants will share the money award.

You could refer to any resources (internet, book, etc) during the contest.

FIGURE 40 DEBUGGING CONTEST OF CSCI4900 FOR SPRING 2013 STUDY

CSCI 4900 Final Specification

Final Exam

Single Lane Bridge Simulation

Exam Policy

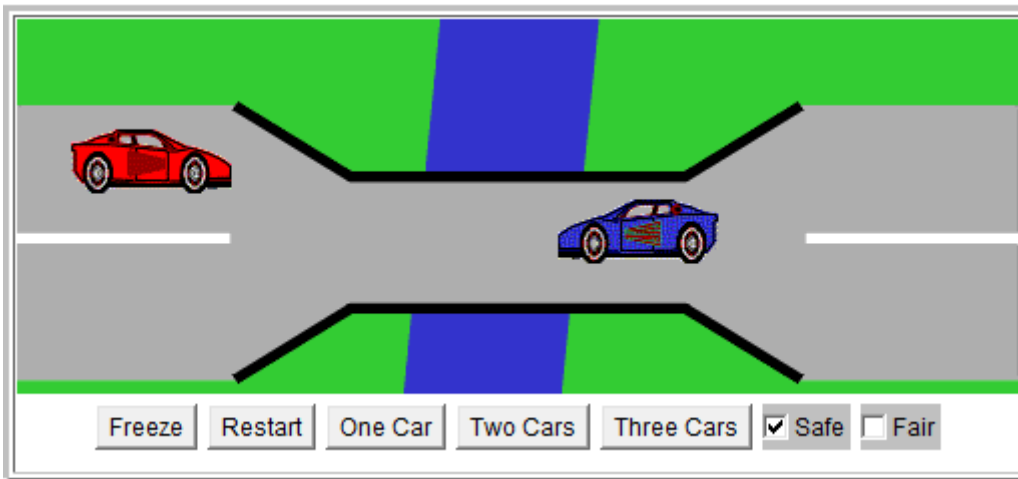
This is an open-book, open-resource exam. You may access the Internet (with the exception of solutions to the single lane bridge problem), your textbooks or any other resources you would like to use. No requests for outside assistance are permitted (i.e., no phone calls, no emails, no message board posts, etc.). The total exam time is 3 hours. Submit by the end of the 3 hour period; o late submissions will be graded. You may have multiple submissions to the department server and assignment dropbox, but only the last submission will be graded. Make sure you save your work frequently.

Exam Description

1. The Simulation Problem – Single Lane Bridge

A single-lane bridge is wide enough to permit only a single lane of traffic. That is, the bridge permits only one-way traffic at any time and cars exit the bridge according to the order in which they enter the bridge. To simplify the problem, we will define the cars that move from left to right as **red** cars and those that move from right to left as **blue** cars. The bridge should guarantee safety and fairness, i.e. no crashes on the bridge and cars of both colors have a roughly equal chance to use the bridge.

Here is a graphical illustration of the single lane bridge problem (click to view the applet online):



Here is a list of key points in the simulation:

- Cars of different colors cannot exist on the bridge at same time
- A car that enters the bridge later cannot exit before cars that entered the bridge earlier
- Cars from both directions should have a roughly equal chance to use the bridge

2. Simulation Settings

To set up a simulation, the following arguments should be specified:

- the maximum usage difference of the bridge, i.e. the maximum allowed difference in the number of cars that have entered bridge from each side, at any time.
 - ❖ For example, if this number is 3, at any time, the total number of red cars that have entered bridge should not be more than 3 more than the total number of blue cars that have entered bridge and vice versa
- the number of car generators for each direction that send (red/blue) cars to the single-lane bridge
- the number of cars each generator will send (same for all generators)
 - ❖ the total number of each type of cars sent to the single-lane bridge is the same and is decided by the above two arguments
- the minimum interval between generation of cars (in milliseconds, same for all generators)
- the maximum interval between generation of cars (in milliseconds, same for all generators)
 - ❖ after sending a car to the single-lane bridge, the generator should pause for a random time between this minimum and maximum before sending the next car
- status check frequency, i.e. the number of reported events in between each status check
 - ❖ For example, if this number is 10, after reporting 10 events, a status check should be executed (event reporting and fairness check are addressed in detail in [Output Requirements](#))

3. Output Requirements

The simulation should print out the events that happen at the bridge according to their order of occurrence. To be specific, the following and only the following events should be reported: (no graphical interfaces, simply command-line outputs)

6. A car arrives at the bridge
7. A car enters the bridge
8. A car exits the bridge

Here is a table of the corresponding output format of the above events:

*contents in square brackets are program variables

*a car's id is its order among same color cars that arrive at bridge

Event No.	Output Format
1	[<i>car_color</i>] car [<i>car_id</i>] arrives at waiting position [<i>position</i>]
2	[<i>car_color</i>] car [<i>car_id</i>] enters bridge
3	[<i>car_color</i>] car [<i>car_id</i>] exits bridge

Also, the program should do a status check (printing out the following information) at the specified frequency (details in [Simulation Settings](#)):

5. number of cars of each color that have used the bridge
6. number of cars of each color that are on the bridge

Here is a table with the corresponding output format of the above information:

*contents in square brackets are program variables

Info No.	Output Format
1	Status Check of Usage: [<i>num_red_cars</i>] [<i>num_blue_cars</i>]
2	Status Check of Bridge: [<i>num_red_cars</i>] [<i>num_blue_cars</i>]

Here is one sample simulation output from a demo program that describes the events that happened at the bridge and with status checks:

*maximum usage difference = 3, 10 cars of each color, status check frequency = 10

```

Red car 1 arrives at waiting position 1
Red car 2 arrives at waiting position 2
Red car 3 arrives at waiting position 3
Blue car 1 arrives at waiting position 1
Blue car 1 enters bridge
Blue car 2 arrives at waiting position 1
Blue car 2 enters bridge
Red car 4 arrives at waiting position 4
Blue car 3 arrives at waiting position 1
Blue car 3 enters bridge
Status Check of Usage: 0 3
Status Check of Bridge: 0 3
Red car 5 arrives at waiting position 5
Blue car 4 arrives at waiting position 1
Red car 6 arrives at waiting position 6
Red car 7 arrives at waiting position 7
Red car 8 arrives at waiting position 8
Blue car 5 arrives at waiting position 2
Blue car 6 arrives at waiting position 3
Red car 9 arrives at waiting position 9
Red car 10 arrives at waiting position 10
Blue car 7 arrives at waiting position 4
Status Check of Usage: 0 3
Status Check of Bridge: 0 3
End Status Check
Blue car 8 arrives at waiting position 5
Blue car 1 exits bridge
Blue car 2 exits bridge
Blue car 3 exits bridge
Red car 1 enters bridge
Red car 2 enters bridge
Red car 3 enters bridge
Red car 4 enters bridge
Red car 5 enters bridge
Red car 6 enters bridge
Status Check of Usage: 6 3
Status Check of Bridge: 6 0
Red car 1 exits bridge
Red car 2 exits bridge
Red car 3 exits bridge
Red car 4 exits bridge
Red car 5 exits bridge
Red car 6 exits bridge
Blue car 4 enters bridge
Blue car 5 enters bridge
Blue car 6 enters bridge
Blue car 7 enters bridge
Status Check of Usage: 6 7
Status Check of Bridge: 0 4
Blue car 8 enters bridge
Blue car 4 exits bridge
Blue car 5 exits bridge
Blue car 6 exits bridge
Blue car 7 exits bridge
Blue car 8 exits bridge
Blue car 9 arrives at waiting position 1
Blue car 9 enters bridge
Blue car 9 exits bridge
Red car 7 enters bridge
Status Check of Usage: 7 9

```



```
Status Check of Bridge: 1 0
Red car 8 enters bridge
Red car 9 enters bridge
Red car 10 enters bridge
Red car 7 exits bridge
Red car 8 exits bridge
Red car 9 exits bridge
Red car 10 exits bridge
Blue car 10 arrives at waiting position 1
Blue car 10 enters bridge
Blue car 10 exits bridge
Status Check of Usage: 10 10
Status Check of Bridge: 0 0
```

4. Deliverables You should choose one of the three models described below to implement and submit by the end of the exam period. If you have time, you may implement and submit a second model (or even a third). In this case we'll grade both (or all three) and use the higher (highest) grade.

Java Threads Model

One choice is to implement the simulation with Java Threads. You may use the given code skeleton in the **skeleton/java** folder or create your own. If you decide to create your own program, it should accept setting arguments in the order described in [Simulation Settings](#) and produce output as described in [Output Requirements](#).

The given code skeleton has a full implementation of a **Main** class, a **RedCarGenerator** and a **BlueCarGenerator** class. The **RedCarGenerator** class and the **BlueCarGenerator** class send red cars and blue cars to the single-lane bridge and the **Main** class sets up the simulation with setting arguments described in [Simulation Settings](#) above and terminates the whole program when all work is done.

The given code skeleton has a partial implementation of the **Bridge**, **RedCar** and **BlueCar** classes. To work with the **Main**, **RedCarGenerator** and **BlueCarGenerator** classes, you should not change the number, type or order of parameters taken by the constructors of these classes. However, you may rename them according to your preference. The **RedCar** and **BlueCar** classes implement the **Runnable** interface and you have to define the corresponding *run()* methods for them. The **Bridge** class records the shared data of the single-lane bridge and it is a good practice to utilize synchronization mechanisms there. The synchronized method *finish()* is used by the **main** thread to notify the bridge that all cars have been sent. The synchronized method *allExit(int total)* is used by the **main** thread to predict whether all car threads have finished their executions so that the program only exits after all work has been done. The parameter *total* it is passed is of **int** type which indicates the total number of cars from each side (i.e. the product of program setting parameter *number of generators of each color car* and *number of cars each generator send to bridge*). The private method *checkStatus()* may be used by the **Bridge** class to determine the timing of printing status information.

Note that all source code should be put into a **bridge** package.

Scala Actors Model

Another choice is to implement the simulation with Scala Actors. You may use the given code skeleton in the **skeleton/scala** folder or create your own. If you decide to create your own program, it should accept setting arguments in the order described in [Simulation Settings](#) and produce output as described in [Output Requirements](#).

The given code skeleton has a full implementation of a **SingleLaneBridge** object, a **RedCarGenerator** and a **BlueCarGenerator** class. The **RedCarGenerator** class and **BlueCarGenerator**

class send red cars and blue cars to the single-lane bridge and the **SingleLaneBridge** object sets up the simulation with setting arguments described in [Simulation Settings](#) above and terminates the whole program when all work is done. Messages used to determine the status of actors (finish, message sends by *main()* to notify all generators have finished; done, message sends by bridge actor to indicate all work is done) as well as notification (nomore, message sends by a generator to indicate that all cars it is supposed to generate have been sent to the bridge) are also provided.

The given code skeleton has a partial implementation of the **Bridge**, **RedCar** and **BlueCar** classes. To work with the **SingleLaneBridge** object, **RedCarGenerator** and **BlueCarGenerator** classes, you should not change the number, type or order of parameters taken by the constructors of these classes. However, you may rename them according to your preference. All these class are inherited from the Actor class and you have to define the corresponding *act()* methods for them. The **Bridge** class implements an *allExit(total: Int)* method to determine the timing of notifying program exit. The parameter total it is passed is of type Int, which indicates the total number of cars from each side (i.e. the product of program setting parameter number of generators of each color car and number of cars each generator send to bridge). It takes no arguments and returns a Boolean value. Only after the bridge has all work done, i.e. all cars have used the bridge and exited, will the **bridge** reply to the **SingleLaneBridge** object with a done message. The private method *checkStatus()* may be used by the **Bridge** class to determine the timing of printing status information.

Note that all source code should be put into a **bridge** package.

Python Coroutine Model

A third choice is to implement the simulation with Python Coroutines. You may use the given code skeleton in the **skeleton/python** folder or create your own. If you decide to create your own program, it should accept setting arguments in the order described in [The Simulation Settings](#) and provide output as described in [Output Requirements](#).

The given code skeleton has full implementations of a **main()** function and **red_car_generator()** and **blue_car_generator()** functions. The **main()** function sets up the simulation with setting arguments described in [The Simulation Settings](#) above and terminates the whole program when all work is done. A global variable (num_active_generator) used for recording the progress of the **red_car_generator()** and **blue_car_generator()** functions is also provided (an active car generator still has cars that have not been sent to the bridge yet).

The given code skeleton has a partial implementation of the **bridge** class and the **red_car()** and **blue_car()** functions. To work with the **main()**, **red_car_generator()** and **blue_car_generator()** functions, you should not change the number or order of parameters taken by **red_car()** and **blue_car()** functions. However, you may rename them according to your preference. Functions **red_car()** and **blue_car()** are generator functions in which you should utilize yield. The **bridge** class records the shared data of the single-lane bridge. The method *finish()* is used by the **main()** function to notify the bridge that all cars have been sent. The method *allExit(total)* is used by the **main()** function to predict whether all cars have finished their executions so that the program only exits after all work is done. The parameter total it passed in indicates the total number of cars from each side (i.e. the product of program setting parameter number of generators of each color car and number of cars each generator send to bridge). The method *checkStatus()* may be used by the **Bridge** class to determine the timing of printing status information.

5. Submissions

This test contains three potential deliverables, as described above. You must submit one; you may submit solutions for additional models. For each model solution, put all your source code for that portion (do not include compiled binaries since they are large) into a folder lastname_final_X (X

should be either java, scala or python that describes the source code you are submitting) and submit the folder on nuke (submit cs4900a lastname_final_X).

6. Grading

Each submission will be tested against 20 different combinations of simulation settings (details of setting arguments are described in [The Simulation Settings](#)). For each of these 20 settings, the outputs will be checked against the following rubrics:

5. In all status checks, the difference between the number of red cars and number of blue cars that have entered bridge does not exceed the set maximum
6. In all status checks, no more than one type of car is occupying the bridge
7. The status check output frequency is correct
8. Event output sequence is reasonable (e.g. entering happened before exiting, car that enters first exits first)
9. The total number of cars exited is equal to the specification of setting parameters

Any one violation of the above rubrics in an output causes the deduction of 1 point until the deductions accumulates to 5, which is the total points for that output. The total of this test is:

5 points per running output
× 20 running settings per deliverable portion
= 100 points

FIGURE 41 FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY

```

package bridge;

public class Bridge {

    private boolean blueTurn, fin;
    private int redEntered, redExited, blueEntered, blueExited, redWaited,
blueWaited;

    private int useDiff, checkFreq, eventNumSinceLastCheck;

    public Bridge(int useDiff, int checkFreq) {
        blueTurn = true;
        fin = false;
        redEntered = redExited = blueEntered = blueExited = redWaited = blueWaited =
0;
        this.useDiff = useDiff;
        this.checkFreq = checkFreq;
        eventNumSinceLastCheck = 0;
    }

    public synchronized boolean isAllExit(int total) {
        fin = true;
        notifyAll();
        return redWaited == 0 && blueWaited == 0 &&
            (redEntered == total) && (redExited == total) &&
            (blueEntered == total) && (blueExited == total);
    }

    private synchronized void checkStatus() {
        if ((++eventNumSinceLastCheck) >= checkFreq) {
            System.out.println("Status Check of Usage: " + redEntered + " " +
blueEntered);
            System.out.println("Status Check of Bridge: " + (redEntered-redExited) +
" " + (blueEntered-blueExited));
            eventNumSinceLastCheck = 0;
        }
    }

    private synchronized void shiftTurn() {
        if (blueEntered - redEntered >= useDiff) blueTurn = false;
        if (redEntered - blueEntered >= useDiff) blueTurn = true;
    }

    public synchronized int redEnter() throws InterruptedException {
        ++redWaited;
        System.out.println("Red car " + (redEntered+redWaited) + " arrives at
waiting position " + redWaited);
        checkStatus();
        while (blueEntered != blueExited || ((!fin || blueWaited != 0) && blueTurn))
wait();
        --redWaited;
        ++redEntered;
        shiftTurn();
        System.out.println("Red car " + redEntered + " enters bridge");
        checkStatus();
        return redEntered;
    }

    public synchronized void redExit(int order) throws InterruptedException {
        while (order != redExited + 1) wait();
        ++redExited;
        System.out.println("Red car " + order + " exits bridge");
        checkStatus();
    }
}

```

```

        notifyAll();
    }

    public synchronized int blueEnter() throws InterruptedException {
        ++blueWaited;
        System.out.println("Blue car " + (blueEntered+blueWaited) + " arrives at
waiting position " + blueWaited);
        checkStatus();
        while (redEntered != redExited || ((!fin || redWaited != 0) && !blueTurn))
wait();
        --blueWaited;
        ++blueEntered;
        shiftTurn();
        System.out.println("Blue car " + blueEntered + " enters bridge");
        checkStatus();
        return blueEntered;
    }

    public synchronized void blueExit(int order) throws InterruptedException {
        while (order != blueExited + 1) wait();
        ++blueExited;
        System.out.println("Blue car " + order + " exits bridge");
        checkStatus();
        notifyAll();
    }
}

package bridge;

public class RedCar implements Runnable {
    final Bridge bridge;

    public RedCar(Bridge bridge) {
        this.bridge = bridge;
    }

    public void run() {
        try {
            int order = bridge.redEnter();
            bridge.redExit(order);
        } catch (InterruptedException e) {}
    }
}

package bridge;

public class BlueCar implements Runnable {
    final Bridge bridge;

    public BlueCar(Bridge bridge) {
        this.bridge = bridge;
    }

    public void run() {
        try {
            int order = bridge.blueEnter();
            bridge.blueExit(order);
        } catch (InterruptedException e) {}
    }
}

```

FIGURE 42 JAVA THREADS SOLUTION TO FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY

```

package bridge

import scala.actors._
import Actor._
import scala.util.Random
import scala.collection.mutable.Queue

case class redCarEnter(red:Actor, initial:Boolean)
case class redCarExit(red:Actor, order:Int)
case class blueCarEnter(blue:Actor, initial:Boolean)
case class blueCarExit(blue:Actor, order:Int)
case class succeedEnter(order:Int)
case class succeedExit()
case class nomore()
case class allExit(main:Actor, total:Int)
case class done()

class Bridge(useDiff:Int, checkFreq:Int) extends Actor {
  var redEntered, redExited, blueEntered, blueExited, redWaited, blueWaited,
  eventNumSinceLastCheck = 0;
  var blueTurn = true;
  var fin = false;
  val reds:Queue[Actor] = Queue()
  val blues:Queue[Actor] = Queue()

  def isAllExit(total:Int) : Boolean = {
    fin = true
    return redWaited == 0 && blueWaited == 0 &&
      (redEntered == total) && (redExited == total) &&
      (blueEntered == total) && (blueExited == total)
  }

  def checkStatus() {
    eventNumSinceLastCheck += 1
    if (eventNumSinceLastCheck >= checkFreq) {
      println("Status Check of Usage: " + redEntered + " " + blueEntered)
      println("Status Check of Bridge: " + (redEntered-redExited) + " " +
(blueEntered-blueExited))
      eventNumSinceLastCheck = 0
    }
  }

  def shiftTurn() {
    if (blueEntered - redEntered >= useDiff) blueTurn = false
    if (redEntered - blueEntered >= useDiff) blueTurn = true
  }

  def redEnter(red:Actor, initial:Boolean) {
    if (initial) {
      redWaited += 1
      println("Red car " + (redEntered+redWaited) + " arrives at waiting position "
+ redWaited)
      checkStatus()
    }
    if (blueEntered == blueExited && ((fin && blueWaited == 0) || (!blueTurn))) {
      redWaited -= 1
      redEntered += 1
      shiftTurn()
      println("Red car " + redEntered + " enters bridge")
      checkStatus()
      red ! succeedEnter(redEntered)
    } else {
      self ! redCarEnter(red, false)
    }
  }
}

```

```

    }
  }

  def redExit(red:Actor, order:Int) {
    if (order != redExited + 1) self ! redCarExit(red, order)
    else {
      redExited += 1
      println("Red car " + order + " exits bridge")
      checkStatus()
      red ! succeedExit()
    }
  }

  def blueEnter(blue:Actor, initial:Boolean) {
    if (initial) {
      blueWaited += 1
      println("Blue car " + (blueEntered+blueWaited) + " arrives at waiting position"
" + blueWaited)
      checkStatus()
    }
    if (redEntered == redExited && ((fin && redWaited == 0) || blueTurn)) {
      blueWaited -= 1
      blueEntered += 1
      shiftTurn()
      println("Blue car " + blueEntered + " enters bridge")
      checkStatus()
      blue ! succeedEnter(blueEntered)
    } else {
      self ! blueCarEnter(blue, false)
    }
  }

  def blueExit(blue:Actor, order:Int) {
    if (order != blueExited + 1) self ! blueCarExit(blue, order)
    else {
      blueExited += 1
      println("Blue car " + order + " exits bridge")
      checkStatus()
      blue ! succeedExit()
    }
  }

  def act() {
    loop {
      react {
        case redCarEnter(red:Actor, initial:Boolean) =>
          redEnter(red, initial)

        case redCarExit(red:Actor, order:Int) =>
          redExit(red, order)

        case blueCarEnter(blue:Actor, initial:Boolean) =>
          blueEnter(blue, initial)

        case blueCarExit(blue:Actor, order:Int) =>
          blueExit(blue, order)

        case allExit(main:Actor, total:Int) =>
          if (mailboxSize != 0 || !isAllExit(total)) {
            self ! allExit(main, total)
            fin = true
          }
          else {

```

```

        main ! done()
    }
}

}

}

}

class RedCar(bridge:Actor) extends Actor {
    def enter() {
        bridge ! redCarEnter(self, true)
    }

    def leave(order:Int) {
        bridge ! redCarExit(self, order)
    }

    def act() {
        enter()
        loop {
            react {
                case succeedEnter(order:Int) =>
                    leave(order)

                case succeedExit() =>
                    exit()
            }
        }
    }
}

class BlueCar(bridge:Actor) extends Actor {
    def enter() {
        bridge ! blueCarEnter(self, true)
    }

    def leave(order:Int) {
        bridge ! blueCarExit(self, order)
    }

    def act() {
        enter()
        loop {
            react {
                case succeedEnter(order:Int) =>
                    leave(order)

                case succeedExit() =>
                    exit()
            }
        }
    }
}

class RedCarGenerator(numToGenerate:Int, minInterval:Int, maxInterval:Int,
bridge:Bridge, main:Actor) extends Actor {
    def act() {
        for (i <- 1 to numToGenerate toList) {
            val car:RedCar = new RedCar(bridge)
            car.start()
            Thread.sleep((new Random()).nextInt(maxInterval-minInterval)+minInterval)
        }
        main ! nomore()
    }
}

```



```

        exit()
    }
}

class BlueCarGenerator(numToGenerate:Int, minInterval:Int, maxInterval:Int,
bridge:Bridge, main:Actor) extends Actor {
    def act() {
        for (i <- 1 to numToGenerate toList) {
            val car:BlueCar = new BlueCar(bridge)
            car.start()
            Thread.sleep((new Random()).nextInt(maxInterval-minInterval)+minInterval)
        }
        main ! nomore()
        exit()
    }
}

object SingleLaneBridge {
    val usage = """Usage: [maximum waiting line difference]
                    [number of car generators for each color]
                    [number of cars generated by each generator]
                    [minimal time of interval between two generations of cars]
                    [maximal time of interval between two generations of cars]
                    [status check frequency]"""

    def main(args:Array[String]) {
        val mainActor:Actor = actor {
            if (args.length < 6) {
                println(usage)
                System.exit(1)
            }

            val waitDiff:Int = args(0).toInt
            val numGen:Int = args(1).toInt
            val numCarPerGenerator:Int = args(2).toInt
            val minInterval:Int = args(3).toInt
            val maxInterval:Int = args(4).toInt
            val checkFreq:Int = args(5).toInt

            val bridge:Bridge = new Bridge(waitDiff, checkFreq)
            bridge.start()

            for (i <- 1 to numGen toList) {
                val generator:RedCarGenerator = new RedCarGenerator(numCarPerGenerator,
minInterval, maxInterval, bridge, self)
                generator.start()
            }

            for (i <- 1 to numGen toList) {
                val generator:BlueCarGenerator = new BlueCarGenerator(numCarPerGenerator,
minInterval, maxInterval, bridge, self)
                generator.start()
            }

            loop {
                react {
                    case nomore() =>
                        if ((mailboxSize + 1) == (numGen*2)) {
                            bridge ! allExit(self, numGen*numCarPerGenerator)
                        } else {
                            self ! nomore()
                        }
                }

                case done() =>

```

```
        System.exit(0)
    }
}
}
```

FIGURE 43 SCALA ACTORS SOLUTION TO FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY

```

import random, sys, time

num_active_generator = 0

class bridge:
    def __init__(self, useDiff, checkFreq):
        self.redEntered = self.redExited = self.redWaited = self.blueEntered =
self.blueExited = self.blueWaited = 0
        self.fin = False
        self.blueTurn = True
        self.useDiff = useDiff
        self.checkFreq = checkFreq
        self.evenSinceLastStatusCheck = 0

    def is_all_exit(self, total):
        self.fin = True
        return self.redWaited == 0 and self.blueWaited == 0 and (self.redEntered ==
self.redExited == total == self.blueEntered == self.blueExited)

    def shift_turn(self):
        if self.redEntered - self.blueEntered >= self.useDiff:
            self.blueTurn = True
        if self.blueEntered - self.redEntered >= self.useDiff:
            self.blueTurn = False

    def check_status(self):
        self.evenSinceLastStatusCheck += 1
        if self.evenSinceLastStatusCheck >= self.checkFreq:
            print "Status Check of Usage: %s %s" %(self.redEntered, self.blueEntered)
            print "Status Check of Bridge: %s %s" %(self.redEntered - self.redExited,
self.blueEntered - self.blueExited)
            self.evenSinceLastStatusCheck = 0

    def red_arrive(self):
        self.redWaited += 1
        print "Red Car %s arrives at waiting position %s"
%(self.redEntered+self.redWaited, self.redWaited)
        self.check_status()

    def red_enter(self):
        if self.blueEntered == self.blueExited and ((self.fin and self.blueWaited == 0)
or not self.blueTurn):
            self.redWaited -= 1
            self.redEntered += 1
            self.shift_turn()
            print "Red Car %s enters bridge" %(self.redEntered)
            self.check_status()
            return self.redEntered
        else:
            return -1

    def red_exit(self, order):
        if self.redExited + 1 == order:
            self.redExited += 1
            print "Red Car %s exits bridge" %(order)
            self.check_status()
            return True
        else:
            return False

    def blue_arrive(self):
        self.blueWaited += 1
        print "Blue Car %s arrives at waiting position %s"

```

```

%(self.blueEntered+self.blueWaited, self.blueWaited)
    self.check_status()

    def blue_enter(self):
        if self.redEntered == self.redExited and ((self.fin and self.redWaited == 0) or
self.blueTurn):
            self.blueWaited -= 1
            self.blueEntered += 1
            self.shift_turn()
            print "Blue Car %s enters bridge" %(self.blueEntered)
            self.check_status()
            return self.blueEntered
        else:
            return -1

    def blue_exit(self, order):
        if self.blueExited + 1 == order:
            self.blueExited += 1
            print "Blue Car %s exits bridge" %(order)
            self.check_status()
            return True
        else:
            return False

def red_car(bridge):
    bridge.red_arrive()
    order = bridge.red_enter()
    while order == -1:
        yield
    order = bridge.red_enter()
    while not bridge.red_exit(order):
        yield

def blue_car(bridge):
    bridge.blue_arrive()
    order = bridge.blue_enter()
    while order == -1:
        yield
    order = bridge.blue_enter()
    while not bridge.blue_exit(order):
        yield

def red_generator(name, num_car, interval_min, interval_max, bridge):
    global num_active_generator
    cars = []
    for i in xrange(num_car):
        cars.append(red_car(bridge))
        t0 = time.time()
        interval = random.randint(interval_min, interval_max)
        while (time.time() - t0) * 1000 < interval: #trasfer sec to msec
            yield
    num_active_generator -= 1
    while len(cars) > 0:
        try:
            task = random.choice(cars)
            task.next()
            yield
        except StopIteration:
            cars.remove(task)

def blue_generator(name, num_car, interval_min, interval_max, bridge):
    global num_active_generator
    cars = []

```

```

for i in xrange(num_car):
    cars.append(blue_car(bridge))
    t0 = time.time()
    interval = random.randint(interval_min, interval_max)
    while (time.time() - t0) * 1000 < interval: #trasfer sec to msec
        yield
num_active_generator -= 1
while len(cars) > 0:
    try:
        task = random.choice(cars)
        task.next()
        yield
    except StopIteration:
        cars.remove(task)

def main():
    usage = """SingleLaneBridge: [maximum usage difference]
    [number of car generators for each color]
    [number of cars generated by each generator]
    [minimal time of interval between two generations of cars]
    [maximal time of interval between two generations of cars]
    [status check frequency]"""

    if len(sys.argv) < 6:
        print '%s' %usage
        exit()

    max_diff = int(sys.argv[1])
    num_gen = int(sys.argv[2])
    num_car_per_gen = int(sys.argv[3])
    min_interval = int(sys.argv[4])
    max_interval = int(sys.argv[5])
    check_freq = int(sys.argv[6])

    global num_active_generator
    num_active_generator = num_gen*2

    b = bridge(max_diff, check_freq)

    tasks = []
    for i in range(num_gen):
        rg = red_generator(i, num_car_per_gen, min_interval, max_interval, b)
        bg = blue_generator(i, num_car_per_gen, min_interval, max_interval, b)
        tasks.append(rg)
        tasks.append(bg)

    while len(tasks) > 0:
        if num_active_generator == 0:
            b.is_all_exit(int(sys.argv[2])*int(sys.argv[3]))
        try:
            task = random.choice(tasks)
            task.next()
        except StopIteration:
            tasks.remove(task)
        #time.sleep(0.1)

if __name__ == '__main__':
    main()

```

FIGURE 44 PYTHON COROUTINES SOLUTION TO FINAL EXAM OF CSCI4900 FOR SPRING 2013 STUDY

Jan 23, 2013 Lab 01 Survey Name_____

1. How much time did you spend to complete lab 01?

_____ hour (s) _____minutes (s)

Do you feel any time pressure to finish it?

_____Yes _____No

2. Which part of the lab do you think is most helpful?

3. Which part of the lab do you think is most difficult?

4. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

5. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade_____

6. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Jan 29, 2013 Hwk 02 Survey Name_____

7. How much time did you spend to complete homework 02?

_____ hour (s) _____minutes (s)

Do you feel any time pressure to finish it?

_____Yes _____No

8. What do you refer to when writing pseudo codes?

_____slides

_____pseudo code guide

_____other, please specify_____

9. From 0 to 10 (0 as easiest and 10 as most difficult), how hard do you feel to write the pseudo codes?

Grade _____

10. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the homework?

Grade _____

11. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

12. Please write down any other thoughts, ideas, comments you have regarding this homework if any.

Thanks!

Feb 5, 2013 Lab 02 Survey Name _____

13. How much time did you spend to complete lab 02?

_____ hour (s) _____ minutes (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

14. Which part of the lab do you think is most difficult?

_____ Understand the system requirements
 _____ Design the appropriate architecture to meet system requirements
 _____ Draw UML diagrams to reflect my design
 _____ Write pseudo codes to reflect my design
 _____ Other, please specify:

15. Did you see the following relations between the lab project and in-class examples?

_____ Clients/Job Lists and executor/book stock are analogical to worker/sum.
 _____ Restock and shipping is analogical to deposit and withdraw in bank account system.
 _____ Unprogressive shipping job use up all executor resource causes deadlock.

16. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

17. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

18. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Feb 5, 2013 Hwk 03 Survey Name _____

19. How much time did you spend to complete homework 03?

_____ hour (s) _____ minutes (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

20. What do you refer to when writing pseudo codes?

_____ slides _____ pseudo code guide
 _____ homework 2 (shared memory) _____ group designs
 _____ other, please specify:

21. From 0 to 10 (0 as easiest and 10 as most difficult), how hard do you feel to write the pseudo codes?

Grade _____

22. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the homework?

Grade _____

23. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

24. Which one do you think is harder to write?

_____ Pseudocode for shared memory
 _____ Pseudocode for message passing
 _____ Both are very easy
 _____ Both are very hard

25. Please write down any other thoughts, ideas, comments you have regarding this homework if any.

Thanks!

Feb 13, 2013 Lab 05 Survey Name _____

26. How much time did you spend to complete lab 05 (Basic Practice in Java)?

_____ hour (s) _____ minutes (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

27. Which part of the lab do you think is more helpful for you to practice Java programming?

_____ Implement backtracking algorithm for eight queens problem

_____ Implement the remote keypad class and its methods

_____ Design the data structure (for CSCI6900 students only)

_____ Other, please specify:

28. Which part of the lab do you think is most difficult?

_____ Implement a backtracking algorithm for eight queens problem

_____ Implement the remote keypad class and its methods

_____ Design the data structure (for CSCI6900 students only)

_____ Other, please specify:

29. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

30. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

31. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Feb 13, 2013 Lab 03 Survey Name _____

32. How much time did you spend to complete lab 03 (Message Passing Book Inventory System)?

_____ hour (s) _____ minutes (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

33. Which part of the lab do you think is most difficult?

- _____ Understand the system requirements
- _____ Design the message passing protocols and behaviors
- _____ Draw UML diagrams to reflect my design
- _____ Write pseudo codes to reflect my design
- _____ Other, please specify:

34. Compared to design the same system in shared memory form, which do you think is harder?

- _____ Shared Memory
- _____ Message Passing
- _____ Both are hard
- _____ Both are easy
- _____ Design message passing becomes easy by knowing the design of shared memory

35. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

36. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

37. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Feb 19, 2013 Test 1 Survey Name _____

38. Which part do you think is more difficult?

_____ Shared Memory _____ Message Passing

39. We will just count one part of the exam towards 15% of your final score. Which part would you like to be counted?

- _____ Shared Memory
- _____ Message Passing

40. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Shared Memory Grade _____

Message Passing Grade _____

Thanks!

Feb 21, 2013 Lab 06 Survey Name _____

41. How much time did you spend to complete lab 06 (Basic Practice in Scala)?

_____ hour (s) _____ minutes (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

42. Which part(s) of the lab do you think is helpful for you to practice Scala programming?

_____ Implement recursive algorithms for Pascal Triangle, etc.

_____ Implement the remote keypad class and its methods

_____ Implement anagram solver (for CSCI6900 students only)

_____ Other, please specify:

43. Which part of the lab do you think is most difficult?

_____ Implement recursive algorithm for Pascal Triangle

_____ Implement recursive algorithm for Counting Changes

_____ Implement recursive algorithm for Parenthesis Balancing (for CSCI 6900 students only)

_____ Implement the remote keypad class and its methods

_____ Implement anagram solver (for CSCI6900 students only)

_____ Other, please specify:

44. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

45. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

46. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Mar 5, 2013 Lab 07 Survey Name _____

47. How much time did you spend to complete lab 07 (Basic Practice in Python)?

_____ hour (s) _____ minutes (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

48. Which part(s) of the lab do you think is helpful for you to practice Python programming?

- _____ Implement string manipulations in Python (mystrings.py)
- _____ Implement list manipulations in Python (mylists.py)
- _____ Implement dictionary and files manipulations in Python (mimic.py, wordcount.py)
- _____ Implement the remote keypad class and its methods (remotekeypad.py)
- _____ Implement with url library and regular expression packages (logpuzzle.py, 6900 only)
- _____ Other, please specify:

49. Which part of the lab do you think is most difficult?

- _____ Implement string manipulations in Python (mystrings.py)
- _____ Implement list manipulations in Python (mylists.py)
- _____ Implement dictionary and files manipulations in Python (mimic.py, wordcount.py)
- _____ Implement the remote keypad class and its methods (remotekeypad.py)
- _____ Implement with url library and regular expression packages (logpuzzle.py, 6900 only)
- _____ Other, please specify:

50. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

51. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

52. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Mar 19, 2013 Lab 08 Survey Name _____

53. How much time did you spend *in total* to complete lab 08 (Party Matching with Java Threads)?

_____ hour (s) _____ minute (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

54. How much time did you spend to read online tutorials and supplementary readings?

_____ hour(s) _____ minute(s)

From 0 to 10 (0 as most unhelpful and 10 as most helpful), how do you think these materials help you finish the lab?

Grade _____

55. Which part(s) of the lab do you think are difficult?

_____ Understanding the problem setting

_____ Implement a simple version of design

_____ Design a more realistic simulation

_____ Implement your design of a more realistic simulation (optional)

_____ Other, please specify:

56. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

57. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

58. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Mar 21, 2013 Lab 09 Survey Name _____

59. How much time did you spend **in total** to complete lab 09 (Party Matching with Scala Actors)?

_____ hour (s) _____ minute (s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

60. How much time did you spend to read online tutorials and supplementary readings?

_____ hour(s) _____ minute(s)

From 0 to 10 (0 as most unhelpful and 10 as most helpful), how do you think the following materials help you finish the lab? (use a -1 if you did not refer to a specific material)

Grade _____ Textbook

Grade _____ Scala Actor tutorial and API on official Scala websites

Grade _____ Online videos

61. Which part(s) of the lab do you think are difficult?

- _____ Implement first design (match1.scala)
- _____ Implement second design (match2.scala)
- _____ Write out protocol (design) of a more realistic version
- _____ Write out pseudocode of a more realistic version
- _____ Implement the realistic version (optional)
- _____ Other, please specify:

62. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?
Grade _____

63. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?
Grade _____

64. Which do you think is more complicated and difficult for you to handle?
 _____ Implement Party Matching with Java Threads
 _____ Implement Party Matching with Scala Actors

65. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Mar 26, 2013 Hwk 04 Survey Name _____

66. How much time did you spend to complete homework 04 (complete dining.py and dining2.py)?
 _____ hour (s) _____ minutes (s)
 Do you feel any time pressure to finish it?
 _____ Yes _____ No

67. What do you refer to when writing pseudo codes?
 _____ other python codes covered in slides _____ python api documents
 _____ codes found online _____ group design picture/document
 _____ other, please specify:

68. Which of the following implementation alternatives did you see when completing dining2.py?
 _____ All conditional predictions go into Fork class with the usage of *hasFork* list
 _____ All conditional predictions go into Fork class without the usage of *hasFork* list
 _____ Only predictions about fork usage go into Fork class and prediction of available fork go in philosopher generator
 _____ All conditional predictions go into philosopher generator
 _____ Other implementation alternative, please specify:

69. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the

homework?

Grade _____

70. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

71. Please write down any other thoughts, ideas, comments you have regarding this homework if any.

Thanks!

Mar 28, 2013 Lab 10 Survey Name _____

72. How much time did you spend *in total* to complete lab 10 (Party Matching with Python)?

_____ hour(s) _____ minute(s)

Do you feel any time pressure to finish it?

_____ Yes _____ No

73. How much time did you spend to read online tutorials and supplementary readings?

_____ hour(s) _____ minute(s)

From 0 to 10 (0 as most unhelpful and 10 as most helpful), how do you think the following materials help you finish the lab? (use a -1 if you did not refer to a specific material)

Grade _____ Online PDF tutorial

Grade _____ Python Coroutine document on official Scala websites

Grade _____ Online videos

74. Which part(s) of the lab do you think are difficult?

_____ Implement the simple version (match1.py)

_____ Implement the realistic version (match2.py)

_____ Implement the simple version with send() function (optional)

_____ Other, please specify:

75. From 0 to 10 (0 as most dissatisfied and 10 as most satisfied), how would you grade the lab?

Grade _____

76. From 0 to 10 (0 as least successful and 10 as most successful), how would you grade your performance?

Grade _____

77. Please write down any other thoughts, ideas, comments you have regarding this lab if any.

Thanks!

Lab 11 Survey Name _____

Deliverable Portion I

78. How much time did you spend **in total** to complete the 1st portion of Lab 11 (sleeping barber)
 _____ hour (s) _____ minute (s)

79. What is the language that you used to finish the 1st portion? _____

80. What do you like most and dislike most about this language and/or its concurrency constructs?
 How do you think these language features decrease or increase difficulties to implementation?

Lab 11 Survey Name _____

Deliverable Portion II

81. How much time did you spend **in total** to complete the 2nd portion of Lab 11 (sleeping barber)
 _____ hour (s) _____ minute (s)

82. What is the language that you used to finish the 2nd portion? _____

83. What do you like most and dislike most about this language and/or its concurrency constructs?
 How do you think these language features decrease or increase difficulties to implementation?

Lab 11 Survey Name _____

Deliverable Portion II

84. How much time did you spend **in total** to complete the 3rd portion of Lab 11 (sleeping barber)
 _____ hour (s) _____ minute (s)

85. What is the language that you used to finish the 3rd portion? _____

86. What do you like most and dislike most about this language and/or its concurrency constructs?
 How do you think these language features decrease or increase difficulties to implementation?

Final Survey Name _____

87. Assume that you are given a program listing for a complete concurrent program.

For which concurrency model do you think it will be easiest for you to understand the program listing?

_____ Shared memory _____ Message Passing _____ Coroutine

For which concurrency model do you think it will be the most difficult for you to understand this program listing?

_____ Shared memory _____ Message Passing _____ Coroutine

88. Assume that you are given a specification and are asked to write a complete concurrent program.

Which concurrency construct do you think is the easiest to use?

_____ Java threads _____ Scala actors _____ Python coroutines

Which concurrency construct do you think is the most difficult to use?

_____ Java threads _____ Scala actors _____ Python coroutines

89. How did you perceive your familiarity with these concurrency constructs before this course?

	No Knowledge	Novice		Intermediate		Expert
Java threads	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Scala actors	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Python coroutines	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

How do you perceive your familiarity with these concurrency constructs now?

	No Knowledge	Novice		Intermediate		Expert
Java threads	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Scala actors	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Python coroutines	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

90. How did you perceive your general programming expertise before this course?

No Knowledge	Novice		Intermediate		Expert
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

How do you perceive your general programming expertise now?

No Knowledge	Novice		Intermediate		Expert
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

91. How would you describe your learning outcome on concurrency concepts in this course?

I learned nothing at all	I learned the basics of concurrency concepts		I gained a moderate knowledge of concurrency concepts		I became an expert on concurrency concepts
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

How would you describe your learning outcome on gaining programming capabilities in this course?

I learned nothing at all	I learned the basics of programming techniques		I gained a moderate knowledge of programming techniques		I became an expert on programming techniques
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

92. How did you perceive your expertise in different languages before this course?

	No Knowledge	Novice		Intermediate		Expert
--	--------------	--------	--	--------------	--	--------

Java	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Scala	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Python	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other _____	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Other _____	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

How do you perceive your expertise in different languages now?

	No Knowledge	Novice		Intermediate		Expert
Java	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Scala	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Python	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

93. Which of the following statements best describes your procedure for debugging a particular model of the book inventory system in the contest?

_____ I run the checker repeatedly and try to fix each violation/exception reported one by one.

_____ I read the program codes to understand it first, and then run checker repeatedly and try to fix each violation/exception reported one by one.

_____ I read the program codes and make some changes directly first, and then run the checker to find out other violations/exceptions. Each time after running the checker, I make changes to fix multiple violations/exceptions before running the checker again.

_____ I repeat the following process until all violations/exceptions are solved: running the checker to find violations, making guess on possible bugs that might cause these violations, making changes in codes to fix one or more potential bugs and running checker again to validate my guess.

_____ Other, please specify:

94. Do you think any previous course or any content you learned from a previous course or courses helped you perform better in this course? Explain?

95. Which elements of the course were most useful and should be included in future versions of this course? Explain.

96. Which elements of the course should be changed or eliminated in future versions of this course? Explain.

97. Some content of this course was presented in class and other elements were presented online (readings, etc.)

Of the topics presented in-class, should any be moved to on-line? Which ones? Explain.

Of the topics presented online, should any be moved to in-class? Which ones? Explain.

98. Is there anything you would like to tell us about how to make this class better in the future?

Thanks!

FIGURE 45 SURVEY MATERIALS OF CSCI4900 FOR SPRING 2013 STUDY

Online Course Plan

CSCI 4900E Programming in Concurrency

PART I: KNOWLEDGE AND CONCEPTS

*Discussion, homework and project assignments have corresponding grading rubrics

*Wiki is graded according to discussion's rubric and students are required to contribute to wiki

*Help is not graded, but students are encouraged to ask questions and provide answers

*Peer critique is to critique at least 3 and at most 5 of other student's work that have no more than 3 critiques yet.

*One week for each topic and the two exams takes 1 week time together.

Topic 0: Syllabus and Course Introduction

Reading1: Course Syllabus

Reading2: Instructions on Textbook Reading

Reading3: Library resource with GALILEO password

Discussion: Getting to Know You

A semi-formal self-introduction and chance to socialize

Preparation: Run live classroom setup wizard and Install Skype

Topic 1: Introduction to Parallelism and Concurrency

Reading1: *Introduction to Parallel Computing*, Chapter 1-4

Reading2: *Parallel Computer Architecture*, Flynn's Taxonomy

Reading3: *Parallel Computer Architecture*, Memory Organizations

Reading4: *Parallel Computer Architecture*, Caches and Memory Hierarchy

Live Class: Parallel and Distributed Processing

Quiz 01: Parallel and Distributed Processing

Discussion: Super Computers

Topic 2: Concurrency

Reading1: *Parallel Programming*, Thread Level Parallelism

Reading2: *Multicore processors and Systems*, General-Purpose Multi-core Processors

Reading3: *Parallel Programming*, Interconnection Networks

Reading4: *Parallel Programming*, Routing and Switching

Live Class: Concurrency

Quiz 02: Concurrency

Homework1: Observing Multicore Architecture

Topic 3: UML and Concurrent System Design

Reading1: UML Tutorial

Live Class: Use UML Diagrams for Concurrent System Design on Wimba through D2L

Discussion: Inferring the behavior of a readers-writers system through UML notations (Question Driven)

Topic 4: Shared Memory Systems

Reading1: *Introduction to Parallel Computing*, Chapter 5.1-5.3

Reading2: *Parallel Programming*, Parallel Programming Patterns

Reading3: *Parallel Programming*, Synchronization Mechanisms

Reading4: *Mutexes and Semaphores*, Part I – III (online blog)

Livelock (online blog)

Live Class: Shared Memory Concurrent Systems

- Race Condition
 - Sum & Worker Example in Java, C++, Pseudo Code
 - Ornamental Garden Videos
- Pseudo Code System
- Conditional Synchronization
 - Bank Account Example in Pseudo Code
 - Group Design Activity: Bounded-Buffer System (Activity 1)
- Deadlock & Livelock
 - Large Printing Job Example in Pseudo Code
 - Four necessary condition for deadlock
 - Group Design Activity: Dining Philosopher System (Activity 2)
- Fairness Issue
 - Readers/Writers Example in Pseudo Code

Homework 02: Practice Pseudocode with Shared Memory Systems

Project 01: Design Book Inventory as Shared Memory System

Topic 5: Message Passing Systems

Reading1: *Introduction to Parallel Computing*, Chapter 5.4

Reading2: *Parallel Programming*, Message Passing Programming

Live Class: Message Passing Concurrent Systems

- Non-deterministic Order of Messages
 - Sum & Worker Example in Scala, Pseudo Code
 - Ornamental Garden White Board Illustration
- Pseudo Code System
- Conditional Synchronization
 - Bank Account Example in Pseudo Code
 - Group Design Activity: Bounded-Buffer System (Activity 3)
- Deadlock & Livelock
 - Large Printing Job Example in Pseudo Code
 - Group Design Activity: Dining Philosopher System (Activity 4)
- Fairness Issue
 - Readers/Writers Example in Pseudo Code

Homework 03: Practice Pseudocode with Message Passing System

Project 02: Design Book Inventory as Message Passing System

Topic 6: Cooperative Multi-tasking Systems

Live Class: Cooperative Multi-tasking Concurrent Systems

Quiz 03: Cooperative Bounded Buffer on ELC

Homework 04: Practice Python with Cooperative Multi-tasking System

Midterm Exam I: Comprehensive Test on Topics 2-5

PART II: PROGRAMMING PROJECTS

Topic 7: IDE Installation and Basic Programming Skills in Java

Project 03: Installing IDE

Discussion: Post any questions/difficulties encountered during IDE installation

Project 04: Practice Java Basics

- Basic Function Definition
- Backtracking Algorithm in Java (Eight Queen Problem)
- Basic Class Definition (Remote Keypad)
- Data Structure Definition and Java Templates [bonus]

Wiki: Project 04, Practice Java Basics

Help: Project 04, Practice Java Basics

Live Office: Live office-hour times to tackle technical barriers for students installing IDE.

Live office-hour times to tackle coding barriers for students programming Java.

Topic 8: Basic Programming Skills in Scala

Reading1: *Programming in Scala*, Chapter 1-5, 7, 13-17

Project 05: Practice Scala Basics

- Basic Application and Function Definition
- Recursion in Scala (Parenthesis Balancing)
- Class and Objects (Remote Keypad)
- Collections in Scala (Anagram) [bonus]

Wiki: Project 05, Practice Scala Basics

Help: Project 05, Practice Scala Basics

Reading2: Scala Cheat Sheet

Live Office: Live office-hour times to tackle coding barriers for students programming Scala.

Topic 9: Basic Programming Skills in Python

Reading1: *Learning Python*, Chapter 1-13, 16-18, 25-27

Video: Google Python Class Part 1- 7

Project 06: Python Basics (Project 5)

- Basic Module and Function Definition
- Python Strings (String Manipulation)

- Python Lists and Sorting (List Manipulation)
- Python Dictionary and Files (Word Count)
- Class and Objects (Remote Keypad)
- Python Regular Expressions (Log Puzzle) [bonus]

Wiki: Project 06, Practice Python Basics

Help: Project 06, Practice Python Basics

Live Office: Live office-hour times to tackle coding barriers for students programming Python

Reading2: Python Quick Guide

Topic 10: Design and Program Party Matching Problem in Java

Reading1: Java Concurrency Tutorial

Quiz 04: Java Concurrency

Discussion: Java Concurrency Tutorial (Question Driven)

Coding Salon: Java Threads and Shared Memory Systems

Project 07: Party Matching with Java Threads

Wiki: Project 07, Party Matching with Java Threads

Help: Project 07, Party Matching with Java Threads

Live Office: Live office-hour times to tackle coding barriers for students programming Java Threads.

Discussion: Project 07 Design (Products Driven, Build Further Upon Project 07)

Topic 11: Design and Program Party Matching Problem in Scala

Reading1: *Programming in Scala*, Chapter 32

Video: Scala Actors (on Parley's)

Quiz 05: Scala Concurrency

Discussion: Scala Concurrency (Question Driven)

Coding Salon: Scala Actors and Message Passing Systems

Project 08: Party Matching with Scala Actors

Live Office: Live office-hour times to tackle coding barriers for students programming Scala Actors.

Discussion: Project 08 Design (Products Driven, Build Further Upon Project 08)

Topic 12: Design and Program Party Matching Problem in Python

Reading1: A Comprehensive Tutorial on Python Coroutines (online resource)

Video: An Outsider's Look at Coroutine

Video: Using Coroutines to Create Efficient, High-concurrency Web Application

Video: Coroutines, Event Loops and the History of Python Generator

Quiz 06: Python Concurrency

Discussion: Python Concurrency (Question Driven)

Coding Salon: Python Coroutines and Cooperative Systems

Project 09: Party Matching with Python Coroutines

Wiki: Project 09, Party Matching with Python Coroutines

Help: Project 09, Party Matching with Python Coroutines

Live Office: Live office-hour times to tackle coding barriers for students programming Python Coroutine

Discussion: Project 09 Design (Products Driven, Build Further Upon Project 09)

Topic 13: Getting Everything Together: Sleeping Barber Simulation

Project 10: Sleeping Barber Simulation (in Java, Scala and Python)

Wiki: Project 10, Sleeping Barber Simulation

Help: Project 10, Sleeping Barber Simulation

Live Office: Live office-hour times to tackle coding barriers for students

Topic 14: Debugging Contest

Project 11: Debugging Contest on Book Inventory System (in Java, Scala and Python)

Final Exam: Programming Test on Topic 7-14.

PART III: COURSE WORK ON RESEARCH AND PRESENTATION

This part runs in parallel with PART II from week 6 (after the first midterm exam) to week 15.

Task 1: Paper Selection (Week 6)

Students read a paper list (with paper title and its planned presentation time) provided by instructor and post the number id of the paper they would like to present (first come first serve).

Task 2: Paper Presentation (Week 7 – Week 15)

During these weeks, students are required to participate in a 50-minute live class each week to listen to 3 student's paper presentation. After that, students are required to write three summaries of the three presentations in live class. Each summary should be less than 300 words that briefly cover the topic of the paper, the organization of the presentation and a critique on the presenter's performance.

PART IV: LOGISTICS AND RETROSPECT

Logistics issues are included as a separate module in the course content. It provides Questions & Help, some quick links and useful information to manage online environments. A final Prezi presentation provide a retrospect and conclusion for the course.

FIGURE 46 COURSE PLAN FOR ONLINE CSCI4900