THE IMPACT OF PRESENTING CONCURRENCY IN THE INTRODUCTORY COMPUTER SCIENCE COURSE

By

Chester Benford Lund, Jr.

B.S. in Electrical Engineering, May 1974, George Washington University
M.S. in Computer Science, May 1977, George Washington University

A Dissertation submitted to

The Faculty of

The School of Engineering and Applied Science
of The George Washington University in partial satisfaction
of the requirements for the degree of Doctor of Science

August 31, 1999

Dissertation directed by

Michael Bliss Feldman
Professor of Engineering and Applied Science

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

ABSTRACT

The teaching of introductory computer science today is focused on sequential algorithms and processing. Students are taught from day one to think and design in a sequential manner. By the time students are introduced to concurrent and parallel processing, the students are imprinted with sequential algorithmic thought processes.

It is well documented that students who are taught introductory computer science using the imperative programming paradigm have a difficult time transitioning to the object-oriented programming paradigm. Students have to unlearn one programming paradigm in order to learn and use another programming paradigm. The teaching of introductory computer science using multi-paradigm programming methods has been documented as successful.

A controlled experiment was devised to teach introductory computer science students both sequential and concurrency programming to students in their first programming language course. The experiment was created to potentially answer the following questions about introducing concurrency and parallelism into a first programming language course:

- Should it be done?
- How could it be done?

The experiment was conducted by actually teaching a course that introduced both sequential and concurrent concepts together to novice programming students. The first question should be answered with a simple yes or no. The second question may have many possible answers; a target of this experiment was to provide an answer (given that the answer to the first question is yes).

ACKNOWLEDGEMENTS

The author gratefully acknowledges all of the following:

- My wife, Katherine, for her unceasing patience and loving support
- Professor Michael Feldman, my advocate and advisor, for patient wisdom and for *always* being there when needed
- Dr. Bruce Bachus, for his guidance, driving force, and pushing me to continue
- Professor John McCormick, for understanding and good council
- Professors Diane Martin and Robert Harrington, for their guidance and support
- Eun-Joung Ko ("EJ") for her faithfulness and patience as the teaching assistant for the three semesters of this experiment

Thanks and praise to the Lord, Jesus Christ, by whose mercy and gifts, this work could be done. Amen.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

1.1 Problem Statement

Concurrency and parallelism exist today in programming practice from motion picture animation to financial and banking applications. Concurrency and parallelism are an intrinsic part of parallel and distributed computing. Parallel and distributed computing will be shown to pervade many areas of the undergraduate computer science curriculum. Today novice programming students learn to think of computers and computation as a sequential engine for one or more courses before concurrency is introduced. The earliest introductions of concurrent material documented (found in the literature search) are fragmentary introductions of the topic into the computer science course one (CS1) curriculum. The earliest non-fragmentary introduction of the concurrent materials documented (found in the literature search) is in the computer science course two (CS2) curriculum [Hurley, 1994].

The significance of this experiment is that it:

- demonstrates that teaching concurrency to novice programming students can be done without significant loss of performance in other materials taught;
- demonstrates a working model of a computer science course one (CS1) curriculum that incorporates concurrency;
- provides empirical data on the side-effects of introducing concurrency early in the CS1 curriculum; and
- provides sufficient course materials for the introduction of concurrency in CS1 that the study can be replicated.

The introduction of sequential and concurrent programming in the same course should minimize transition problems between sequential and concurrent programming. This is similar to Decker's and Hirshfield's observation [Decker, 1994, page 53] that students who are taught object-oriented programming first simply avoid transition problems between imperative programming and object-oriented programming.

1.2 Novice Programming Student

For the purpose of this experiment, a novice programming student is defined as follows:

- Student entering his or her first computer science programming language course
- Student has no or inconsequential exposure on any of the following levels:
  1. Collegiate programming
  2. Professional programming
  3. Tool creation programming

Collegiate programming is the design, coding, and implementation of programs as part of a college curriculum. Professional programming is the design, coding, and implementation of software systems and programs as an occupation. Tool creation programming is the design, coding, and implementation of programs used as tools as part of a different occupation. An example of tool creation programming is an electrical engineer who creates programs that simulate circuits. No exposure is self-explanatory. Inconsequential exposure is defined as exposure that has minimal impact on student performance in learning to program (the student has learned and retained little or nothing). A common example of inconsequential exposure results from a student taking a one semester computer applications overview class where the student learns about word processing, spreadsheets, presentations, electronic mail, and some programming (such as Basic, "C", or Pascal).

1.3 Overview

The programming language used in the experiment is Ada 95. Ada 95 was selected over several other programming languages for the following reasons (see Section 4 for more details):

- Ada 95 contains concurrent programming language constructs;
- Ada 95 concurrent programming language constructs have minimal differences in syntactic structure between sequential procedures and concurrent tasks;

- Ada 95 concurrent programming language constructs are imbedded into the design of the language and are not extensions or layered macros to a base sequential language; and
- Ada 95 is the programming language used in the George Washington University (GWU) CS1 course CSci 51 -- Introduction to Computing.

Several other languages were given serious consideration for the experiment.

The following are presumed to be true by definition of a novice programming student:

- No or inconsequential exposure to a multi-user computer
- No or inconsequential exposure to a high-level language
- No or inconsequential exposure to a multi-user operating system
- No or inconsequential exposure to a programming editor (for the purpose of this experiment a word processing program is not considered a programming editor)
- No or inconsequential exposure to computer architecture

Thus, all the above items are new to the entering CS1 student.

The programming environment to be used by the CS1 student is as follows:

- Unix operating system -- Solaris 2.5 (Unix 5.5)
- A multiprocessor server architecture computer (similar to the SUN SPARCserver)
- Ada 95 programming language
- VI programming editor

1.4 Research Questions

This research project examines three questions:

Question 1 -- Does teaching concurrency to novice programmers reduce their test performance on sequential material?

> Hypothesis 1: Students who have had concurrency training will have significantly lower sequential test question scores than those who have not had concurrency training.

> Null Hypothesis 1: There is no difference in the sequential test question scores between a student who has had concurrency training and a student who has not had concurrency training.

Question 2 -- After instruction in concurrency, are novice programmers more / less able to solve concurrent questions than sequential questions?

> Hypothesis 2: Students, with both sequential and concurrency instruction, will score significantly different on concurrency test questions than on sequential test questions.
>
> Null Hypothesis 2: There is no difference in the concurrency test question scores and sequential test question scores.

Question 3 -- Are novice programmers less able to use concurrent methods than sequential methods on "large" projects?

> Hypothesis: Students who use concurrent methods on a "large" project will have significantly lower project scores than those who use sequential methods.
>
> Null Hypothesis: There is no difference in "large" project scores between a student using concurrent methods and sequential methods.

In addition, the study examines differences in compile error profiles of students using concurrent and sequential methods on "large" projects. The profile examined is the distinct error ratio (DER). DER is the ratio of total occurrences of compilation errors to number of distinct errors.

1.5 Methodology Overview

1.5.1 Subject Groups

The control and treatment groups in the experiment were as follows:

- The CS1 course in the Fall of 1997 was taught using traditional methods and materials (the control group)
- The CS1 course in the Spring of 1998 was taught using the revised methods and materials for concurrency and parallelism (the first treatment group)

- The CS1 course in the Fall of 1998 was taught using the revised methods and materials for concurrency and parallelism (the second treatment group)

1.5.2 Periods of Comparison

There were two treatment groups used to demonstrate that the experimental results are reproducible and consistent within the two treatment groups. The CSci 51 class was divided into two time periods:

- Period one -- the first eight weeks of the semester
- Period two -- the second eight weeks of the semester

During period one both the control group and the treatment groups received the same instructional materials (all sequential material). During period two, the control group continued to receive the traditional lectures and materials (all sequential materials), while the treatment groups also received both traditional (sequential) and experimental (concurrency) lectures and materials.

1.5.3 Observation, Collection of Data

Student progress and work were collected in a non-intrusive manner. Grade information collected included projects, mid-term exams, final exams, and compilation data. Information collected during period one allowed for a comparison of the similarity of the three subject groups; project scores and mid-term scores for the groups were compared. Information collected during period two allowed for a comparison of the control and treatment groups; project scores and final exam question scores were collected to allow the testing of the hypotheses.

In addition, student compilation data was collected during the entire course. Each compilation and each program link was recorded using Unix shell scripts:

- The shell script named “gcompile” encapsulates the traditional shell script of the same name. The source listing and compiler error and warning messages for each compilation are recorded by the student.
- The shell script named “glink” encapsulates the traditional shell script of the same name. The input data from each execution is recorded.

Students were told at the beginning of the course that their compilation and program executions would be recorded. However, the shell scripts used in the recording process were transparent to the student. In addition, the students' mid-term and final exam papers were copied and analyzed.

1.6 Dissertation Outline

The dissertation is organized into six chapters. The chapters are as follows:

1. *Introduction* -- presents an overview of the experiment and its significance
2. *Literature Search* -- presents the literature search results for the following: introductory concurrency and parallelism materials, documented research in the introductory concurrency and parallelism, and teaching introductory courses at the undergraduate level
3. *Experimental Method* -- presents the methodology of how the experiment was conducted and how the experimental data were be recorded
4. *Course Content Overview* -- presents the current introductory courses materials and the replacement course materials
5. *Results* -- presents the experimental results and observations of the experiment
6. *Conclusions and Future Work* -- presents conclusions and topics of future research that directly follow the completion of this work

Appendices -- presents the following items: definitions, course syllabus, homework project assignments, student demographic data, raw test scores, raw compilation error data, and statistical data

2 LITERATURE SEARCH

This section of the document presents the literature search on concurrency and how concurrency is taught today. This section discusses the following materials in the order listed:

- Concurrency as a paradigm
- Computer science curriculum
- Teaching concurrency
- Unique approach to concurrency -- Sisal
- Concurrency as a paradigm and teaching concurrency today
- Common introductory concurrency materials and languages
- Teaching CS1 students

Section Four of this document presents the current introductory course materials, the replacement course materials, and the schedule of material presentation. As such, the overview of computer science curriculum is presented in that section.

2.1 Concurrency As A Paradigm

Concurrency is a programming paradigm. King [King, 1992] presented a paper on the evolution of programming languages. In this paper King cited the language paradigms covered in texts about programming languages. Wegner [King, 1992] used the concept of a language paradigm a method of classifying programming languages. The language paradigms cited in King's paper were:

- Concurrent
- Database
- Functional
- Imperative (also called procedural)
- Logic
- Object-oriented

Of the 15 textbooks referenced in the paper, twelve (12) texts presented at least some discussion of concurrency as a language paradigm. The textbooks covered in the article have dates from 1981 to 1991.

Before entering into a discussion on concurrency as a paradigm, it is important to define concurrency (or concurrent) and distinguish concurrency from parallelism. For the purpose of this document the following definitions [Paralogic, 1996] are given:

- Concurrent -- "two or more things can be done independently, but not necessarily on different processors [one or more processors]"
- Parallel -- "two or more things can be done independently at the same time on different processors"

By these definitions, several users on one workstation constitute concurrent operation. Ben-Ari [Ben-Ari, 1990] defines a concurrent program as "a set of ordinary sequential programs which are executed in abstract parallelism". The Paralogic definition matches with Ben-Ari's definition in that the number of processors is not implied. Paralogic's definition of parallelism requires two or more processors. Ben-Ari's definition does not imply the number of processors (multiple processes on a single processor fits the definition). Using these definitions, the following distinction between concurrency and parallelism is true:

- Parallelism implies concurrency
- Concurrency does not imply parallelism

This distinction is important -- concurrency is an abstraction in creating software, while parallelism [Paralogic's view] implies a division of activity across processors. Although concurrent processes can be modeled, as if they are executing on multiple processors, the processes can share execution on a single processor. Implementation is separate from processing modeling. For example, using the Concurrent Design Approach for Real-Time Systems method (CODARTS, pronounced "code arts") [Gomaa, 1993], a process designer does not need to distinguish between concurrency and parallelism. Hence, a student studying concurrency does not need to differentiate between the concurrency and parallelism. CODARTS is a relatively new methodology that supports modeling processes and objects naturally in a concurrent manner.

Concurrency is a natural consequence of modeling specific problems. There are many classes of problems whose formulation and subsequent algorithms are inherently concurrent [Burns, 1995]:

- Process control
- Air traffic control
- Avionics systems
- Industrial robots
- Engine controllers
- Domestic appliances
- Environmental monitors

All these applications in their implementation have the characteristics of being time-dependent (real-time), embedded, and may be complicated. However, all these applications have simplistic components that are easily modeled.

There exist problems that are easier to model using concurrency. One type of problem that is easier to model using concurrency is independent objects that have co-dependencies. Edsger Dijkstra's famous "dining philosophers" problem, published in 1971, is classic example of this type of problem. Five philosophers do only two activities -- thinking and eating [Ben-Ari, 1990]. Dijkstra's rules for the correct behavior of the philosophers are as follows:

- Philosopher must have two forks to eat
- Two philosophers cannot share a fork
- All philosophers cannot hold a single fork
- A philosopher cannot starve
- Philosophers behave efficiently (in the absence of contention)

The form of the concurrent solution given by Ben-Ari is given below:

```
task body philosopher is
begin
    loop
        think;
        pre_protocol;
        eat;
        post_protocol;
    end loop;
end philosopher;
```

The pre_protocol and post_protocol can be implemented as semaphores; such that, the philosopher tasks follow Dijkstra's rules. This example is both childish and simple. However, it demonstrates the ease with which certain classes of problems are modeled and solved (algorithm specified) by concurrency. Concurrency is a paradigm for modeling certain classes of real world problems. Some of these problems are sufficiently simple to be understood by entry-level computer science students.

Ben-Ari [Ben-Ari, 1990] asserted the need for concurrency as a paradigm by specifying the following:

- "Important techniques used in creating software abstractions are encapsulation and concurrency"
- "Concurrent programming is an abstraction that is designed to make it possible to reason about the dynamic behavior of programs"

The concurrent paradigm creates design possibilities that are difficult to model in a sequential model. Many of these possibilities come from having multiple processes within a single program:

- Processes allow a program to do more than one thing at a time. For example, a database application has a database file writer process, query processes, and a log process running simultaneously [Oracle, 1992].
- Processes can be modeled after real objects. This may be the easiest way to model and program a real world object, because this real object can have independent and autonomous actions [Gomaa, 1998].
- Processes augment object-oriented behavior. Processes allow for asynchronous messaging; such that, when object "a" sends a message to object "b", object "a" can continue processing without waiting for object "b" to respond.
- Processes can be suspended, resumed, and stopped independently; this is difficult to model and implement in a sequential program.

Traditional CS1 and CS2 courses use the sequential computational model, the classical von Neumann model (this conclusion can be drawn from the information presented in the next subsection). This classical model does not easily, or at all, include many classes of problems because of their inherently concurrent or parallel nature. Introducing the concurrent paradigm to entry-level computer science allows these students to understand and recognize a larger scope of problems (applications).

There is a second reason for introducing concurrency and parallelism to students. There are applications whose computer resource requirements are not meet by available or existing computers. Characteristics of these problems are as follows:

- Excessive execution time (the need for faster computation)
- The application is near the limits of existing single processor computers
- An aspect of the application has insufficient resources (the problem is too large)

General examples of applications requiring high performance computing include the following:

- Weather programs
- Database programs
- Air traffic control programs
- Avionics programs

Examples of problems found in large database applications include search times, access times, and data mining. These applications areas require tremendously fast computational capability. Students should be able to recognize that "one purpose of parallel processing is to perform computations faster than can be done by a single processor by using a number of processors concurrently" [Jaja, 1992]. Jaja defines three factors that contribute to the development of concurrent and parallel processing:

- Hardware costs are dropping -- multiprocessor computers are available at reasonable cost
- Integrated circuit technology has advanced tremendously -- even PC type computers are available with multiple processors

- "The fastest cycle time of a von Neumann-type processor seems to be approaching fundamental physical limitations beyond which no improvement is possible"

Concurrency and parallelism can no longer be claimed to be an academic exercise. Entry-level computer science students should be given the opportunity to understand that concurrency is a paradigm and a technique for modeling real world problems and as Toll [Toll, 1997] states it -- that concurrent and parallel programming is "different" and "needs to be understood".

2.2 Computer Science Curriculum

This subsection of the document presents the Association of Computing Machinery (ACM) guidelines for computer science curriculum and illustrates the numerous topics where concurrency and parallelism occur in the curriculum.

The current computer science curriculum guidelines are summarized by Turner, et al, in the summary of "Curriculum 91" [Turner, et al, 1991]. These curriculum guidelines are the latest in a series of curriculum guidelines starting in 1968. The ACM published "Curriculum 68," its first recommendation for undergraduate computer science. A second set of recommendations was published in 1979, called "Curriculum 1978." Curriculum 1978 specified eight core courses in computer science. In the early 1980s informal discussions occurred between members of the Institute of Electrical and Electronic Engineers (IEEE) Computer Society and the ACM Education Board about their working together in the curriculum area. A joint task force of computer scientists was created in 1985 that included both the ACM and the IEEE Computer Society membership. This task force, chaired by Peter Denning, created a second task group to produce recommendations for the entire undergraduate curriculum; the undergraduate task group's report was presented to the main task force in February 1988. In 1989, "Computing as a Discipline" specified a breadth-first approach to undergraduate computer science education and defined nine subject areas. The Joint ACM/IEEE-CS Curriculum Task Force report was published in March 1991. The report contains "a collection of subject matter modules called *knowledge units* that comprise the common requirements for all undergraduate programs in the field of computing." These knowledge units are organized into subject areas. The joint report extends the subject areas to ten to include social, ethical, and professional issues.

A summary of the joint task force report was presented in the Communications of the ACM in June 1991 [Turner, et al, 1991]. Turner stated in the summary report that "[t]he report recognizes that there are many effective ways to organize a curriculum, even for a particular set of goals and objectives: it [the report] emphasizes the specification of a minimal set of subject matter that should be included in all programs along with guidelines for organizing the subject matter into courses and incorporating additional material and pedagogy to complete a curriculum." The subject areas are as follows:

- AL: Algorithms and Data Structures (approximately 47 lecture hours)
- AR: Architecture (approximately 59 lecture hours)
- AI: Artificial Intelligence and Robotics (approximately nine lecture hours)
- DB: Database and Information Retrieval (approximately nine lecture hours)
- HU: Human-Computer Communication (approximately eight lecture hours)
- NU: Numerical and Symbolic Computation (approximately 7 lecture hours)
- OS: Operating Systems (approximately 31 lecture hours)
- PL: Programming Languages (approximately 46 lecture hours)
- SE: Software Methodology and Engineering (approximately 44 lecture hours)
- SP: Social, Ethical, and Professional Issues (approximately 11 lecture hours)

Numerous knowledge units within many of these subject areas have explicit references to concurrency and parallelism; they are as follows:

- AL9: Parallel and Distributed Algorithms
- OS2: Tasking and Processing
- OS3: Process Coordination and Synchronization
- OS4: Scheduling and Dispatch
- OS10: Distributed and Real-time Systems
- PL12: Distributed and Parallel Programming Constructs

Further, numerous knowledge units within these subject areas have implicit references to concurrency and parallelism; they include the following:

- AL3: Recursive Algorithms
- AL4: Complexity Analysis
- AL6: Sorting and Searching
- AR5: Memory System Organization
- AR6: Interfacing and Communication
- AR7: Alternative Architectures
- OS9: Communications and Networking
- PL4: Sequence Control

The programming languages subject area includes the following knowledge units:

- PL1: History and Overview of Programming Languages
- PL2: Virtual Machines
- PL3: Representation of Data Types
- PL4: Sequence Control
- PL5: Data Control, Sharing, and Type Checking
- PL6: Run-time Storage Management
- PL7: Finite State Automata and Regular Expressions
- PL8: Context-Free Grammars and Pushdown Automata
- PL9: Language Translation Systems
- PL10: Programming language Semantics
- PL11: Programming Paradigms
- PL12: Distributed and Parallel Programming Constructs

Many of the programming language knowledge units contain direct inferences to concurrency and parallelism. The following items provide examples of these inferences:

- Virtual machines includes the subset of parallel virtual machines
- Sequence control includes by definition sequence control operations that control parallel operations
- Data control, sharing, and type checking includes shared memory which is one method of interprocess communication

- Run-time storage management includes interprocess communication by message passing is a specialized form of storage management
- Programming paradigms includes paradigms that share very similar behavior -- object orientation and concurrency and parallelism share message passing
- Distributed and parallel programming constructs includes explicitly concurrent and parallel constructs (by definition)

In summary, the computer science curriculum specified in Curriculum 91 contains many references to concurrency and parallelism. In addition, as the application of concurrency and parallelism becomes more ubiquitous, many subject areas and knowledge units in the of the curriculum are sufficiently broad to incorporate more inference to concurrency and parallelism.

2.3 Teaching Concurrency

This subsection is organized around the timing of the initial presentation of concurrency to students:

- Concurrency introduced within a CS1 course
- Concurrency introduced to freshman within a CS2 course
- Concurrency introduced within the second year of a computer science curriculum
- Concurrency introduced within the third and fourth years of a computer science curriculum

The CS1 course is a freshman (first year student) course.

2.3.1 Teaching Concurrency In CS1

Only two references were found to teaching concurrency as part of a CS1 curriculum:

1. VanScoy [VanScoy, 1994] at West Virginia University has prepared three lectures on Ada 83 tasks that are given at the end of the course. Students learn to modify existing tasking materials.

2. Allen [Allen, 1996] at Duke University has inserted one lab assignment on "parallel computing". The lab assignment is for students to familiarize themselves with the parallel computing environment and to run a prepared problem. (This work is presented in the next section, since teaching parallelism is reserved for the freshman CS2 course.)

VanScoy's materials were found at the "Asset" web site under the title "Power Point Documents In Support Of An Ada-Based CS 1 Course". VanScoy working at West Virginia University (at Morgantown) started teaching CS1 courses in Ada in the fall of 1989. The course materials are in Ada 83. The Department of Statistics and Computer Science chose Ada for several reasons, among the reasons were the following:

- Support for data abstraction
- Support for generics
- Affordability (inexpensive software available)
- Students should graduate with marketable skills

The course was designed to present 25 lectures over 30 class periods (at a pace of two classes per week). The philosophy used in preparing the course was for the students to use "units" provided by the instructor. VanScoy organized the course around six "units" (general topics):

- Using packages -- 5 lectures
- Writing subprograms -- 5 lectures
- Designing packages -- 3 lectures
- Designing and implementing using types -- 5 lectures
- Implementing packages -- 4 lectures
- Using concurrency -- 3 lectures

The "using concurrency" unit contained the following lectures:

- Observing tasks
- Using tasks
- Writing tasks

Important aspects about VanScoy's approach to teaching concurrency are as follows:

- Information on tasks was held to the end of the course (similar to a "last chapter" in the book approach included for completeness)
- Information on tasks was not integrated into the other units of the course
- Example based approach -- students learn to modify existing tasking materials (not how to write concurrent logic or code)

Unit 5 was described by the author as a "catch all" unit. This statement adds credence to the first bullet in the above list. Finally, the course materials were laid out in a curious manner; for example, generics are introduced in the first unit titled "using packages". The Feldman text [Feldman, 1996] does not address generics until Chapter 11, which is outside the designated range of chapters for a CS1 course.

The course materials provided were produced under DARPA contract. The course materials include the following items:

- Ada source code -- 25 files
- Power point briefing materials -- 25 files
- Overview

The Power point briefing materials for the concurrency lectures are the Ada source code placed into slides.

In a controlled experimental setting outside the classroom, Bachus performed an experiment in the middle 1990s to demonstrate that concurrency could be taught to "novice" students [Bachus, 1996]. Bachus's definition of a novice included students who had completed one and possibly more computer science classes in college. Bachus's work demonstrated that "novice" students did learn the concurrency material presented. Bachus's work is the precursor to this experiment.

2.3.2 Teaching Concurrency to Freshman within a CS2 course

The Department of Computer Science at the University of North Carolina at Charlotte (UNCC) received a two-year grant from the National Science Foundation (NSF) in 1996 to integrate parallel programming in to the freshman computer science curriculum [Allen, 1996]. UNCC computer science students take CS1 in the fall semester and CS2 in the spring semester of the same college year. A summary of the CS1 parallel computing activities at UNCC is as follows:

- "Nothing is covered in the lecture about the theory of parallel computing in [the] first semester" [Allen, 1996].
- The students read Web-based information on PVM and MPI.
- "Parallel computing is introduced ... through a lab assignment". The object of the lab assignment is to familiarize the student with the parallel programming environment and to run a demo program on a multi-workstation environment.

A summary of the CS2 parallel computing activities is as follows:

- During the lecture discussion of algorithms, two perspectives are presented:
  - Sequential computation approach
  - "Large number of workstations" approach
- Students develop, code, and run a parallel solution to a "divide and conquer" type problem

Further reading of Allen's paper also discusses other aspects of the NSF funded project; such as, presenting information by "teleclass" and presentation of expert guest speakers. UNCC has also developed a "comprehensive parallel programming course". This class is an elective for seniors.

The important aspects about UNCC's approach to teaching concurrency to freshman are as follows:

- CS1 students are not taught concurrency and parallelism material, rather, they are given the opportunity to learn about the parallel processing environment in lab.
- CS2 students are to compare algorithms that are sequential and algorithms that take advantage of a "large number of workstations"
- In both cases the teaching of concurrency and parallelism material is not integrated into the curriculum.

Kotz [Kotz, 1995] at Dartmouth College implemented a C++ class library named DAPPLE (for Data-Parallel Programming Library For Education). The DAPPLE library is "designed to provide the illusion of a data-parallel programming language on a conventional hardware and with conventional compilers". The DAPPLE library provides the following:

- Vector classes, such as intVector and floatVector
- Matrix classes, such as intMatrix and floatMatrix
- Parallel IF statement, ifp()
- Matrix slices, parallel operation on slices (defined by [_])

```
// compute C =  A * B
for (int r = 0; r < P; r++)
  for (int c = 0; c < R, c++)
    C [r][c] = inner(A[r][_], B[_][c]);
```

Kotz wanted students to be able to "experiment with parallel computing concepts without being distracted by the *mechanics* of parallel programming." Kotz also looked at other language implementations that were more task-parallel (such as programming language COOL [Chandra, 1994]) than data-parallel and rejected them due to complexity. DAPPLE was originally conceived as an add-on to the Dartmouth CS2 curriculum in 1995. Today, DAPPLE is part of a coordinated CS1, CS2 curriculum [Dartmouth, 1998].

The important aspects about Kotz's approach to teaching parallelism are as follows:

- Data-parallelism is taught, message passing and shared memory concepts are not taught
- DAPPLE models only a single thread of control like a sequential language
- DAPPLE performs only selected operations can be applied to vectors of data "simultaneously"
- DAPPLE is modeling data-parallelism on a "sequential processor budget"

Hence, DAPPLE covers only a small fraction of introductory concurrency and parallelism concepts and material.

The Washington University in Saint Louis has a two-semester CS1, CS2 sequence that teaches computer science fundamentals. Both classes are offered fall and spring semesters. The classes "are taught using the object-oriented paradigm using the Java programming language" [Washington, 1998, instructor is Goldman]. In the CS2 class approximately six consecutive hours of instruction are spent on threads and concurrency in Java [Washington, 1998, instructor is Kraemer] over a two and one-half week period. The Java programs are executed on Sun SPARCstation-class computers. The CS2 class schedule is on the Web at http://www.classes.cec.wustl.edu/~cs102/Spring98/calendar.html.

The important aspects about the Washington University approach to teaching concurrency are as follows:

- Threads are taught as tools; prior and recent sequences of new information in the CS2 course include graphical user interfaces and event handling
- The course descriptions do not include references to message passing, data-parallel, or shared memory concepts
- Information on threads and concurrency was held to the end of the course
- Information on threads and concurrency does not appear to be integrated into other units of the course

Hence, the Washington University CS2 course covers only part of introductory concurrency and parallelism concepts and material.

2.3.3 Teaching Concurrency To Second-Year Students

Teaching concurrency in the second year of undergraduate school implies one or more of the following:

- The student has completed at least a CS1 course and perhaps a CS2 course
- Given that a student has completed a CS2 course -- the student may have been exposed to the following:
  - Data structures
  - Object orientation
  - Second programming language

This directly implies that curriculum designers have placed concurrency and parallelism as a topic that follows these subjects. Concurrency and parallelism is taught to second-year students at the following universities:

- University of Wales, Cardiff [Hurley, 1994]
- University of Reading, England [Reading, 1997]
- University of Liverpool, England [Jackson, 1991]
- Queen's University, Belfast [Bustard, 1990]
- Carnegie Mellon University [Fisher, 1991]
- Wake Forest University [John, 1992, 1994]
- University of Manchester [Manchester, 1998]

Several of these classes are discussed in the following paragraphs.

The University of Wales at Cardiff has two parallel processing courseware modules for computer science students:

- An introductory course module for second year students
- An advanced course module for third year students

The utility of solving problems based on concurrent capabilities is demonstrated for computer science, engineering, electronics, and other disciplines. The computer used for these modules is a nCUBE2 32-processor computer. The course work is done using the "C" language with a dialect of function calls. The introductory module addresses the following topics (the topic titles are Hurley's):

- The Need For High Performance Computers
- Classification of Parallel Machines
- Fundamentals of Inter-processor Communication
- Shared Memory and Message Passing
- Interconnection Networks
- Parallel Algorithm Construction
- Pipelined Algorithms / Algorithmic Parallelism
- Geometric Parallelism / Partitioned Algorithms
- Asynchronous / Relaxed Parallelism
- Factors That Limit Speedup

The course materials are available on the Web (http://www.cs.cf.ac.uk/Parallel).

The University of Reading uses different computer languages and computer platforms for instruction of first-year and second-year students. First-year students are exposed to the following programming courses:

- Logic of Computer Science
- Event Driven Programming
- Introductory Computing

These courses are taught using "Delphi, a visual software environment from Borland using a Pascal-like programming language (Object Pascal) originally invented by Professor Niklaus Wirth of ETH Zurich." The Borland product runs on PCs. Prior to October 1997 the introductory computer language was Modula-2. The second-year computer science students are exposed to the following courses that have programming associated with them:

- Operating Systems
- Computer Architecture
- Software Engineering

The architecture course programming is done on a Motorola MC68000 processor. In the other courses programming is done on a Unix based SUN computer. The concurrency and parallelism topics introduced in the Operating Systems course include:

- Synchronization of processes
- Interprocess communication by message passing
- Multi-tasking systems

The University of Reading school year includes three terms: autumn, lent, and summer. Although the summer term is technically part of the first year, it is considered second-year for the purposes of this document. Students attend nine "terms" in college to obtain a Bachelor of Science degree over three years.

The University of Reading is also a mirror site for Designing and Building Parallel Programs (Online) [Foster, 1997]. The materials at this Web site are used by the students for undergraduate and graduate work [see http://www.cs.rdg.ac.uk/dbpp/text/node1.html for a table of contents for the text].

Jackson at the University of Liverpool [Jackson, 1991] addressed teaching concurrency to undergraduates by creating a mini-course which was a "self-contained unit". The material was presented along with a second-year unit on "systems software -- the major portion of which is a detailed study of operating systems." The language Jackson chose for the mini-course was Ada. Jackson's mini-course contained five lectures:

- Process concepts -- process definition, process execution, context switch
- Concurrency -- Ada tasks, synchronization
- Problems of parallelism -- indeterminacy, mutual exclusion, producer-consumer
- Advanced programming -- like protected variables, bounded buffer
- Task termination and deadlock

Jackson described the ease with which students could understand simple problems and then described that it was "surprising how many students have difficulty in getting the synchronization correct."

Queen's University of Belfast [Queens, 1998] has a complete course titled "Parallel Programming Systems". This course is available to second year students. The course content for this course is as follows:

- Parallel program design
- High level representation of parallelism
- Software for parallel system
- Cooperative and communicating processes

Bustard is credited as the source of this information. CAR Hoare was the Computer Science Department Head from 1968 to 1977; his presence assisted "the Department [in acquiring] an enviable international reputation, particularly in the areas of programming methodology and programming language design and implementation." The Introductory Computer Programming I course is taught in Modula-2.

Carnegie Mellon University has a course titled "Programming Languages Design and Processing" that presents several paradigms:

- Imperative
- Functional
- Logic
- Concurrent programming

The course is setup to show how "different design goals can lead to radically different languages and models of computation". The course is open to second year students and up.

2.3.4 Teaching Concurrency To Third-Year and Fourth-Year Students

Several universities first present concurrency and parallelism to their third and fourth-year students. Many universities introduce concurrency and parallelism as part of the operating systems course; some of these universities include the following:

- University of Indiana [Indiana, 1998]
- Villanova University [Villanova, 1998]
- Rensselaer Polytechnic Institute [Rensselaer, 1998]
- Brown University [Brown, 1998]
- George Mason University [Mason, 1998]
- Dartmouth [Dartmouth, 1998]

Lewis [Lewis, 1997] at Villanova recently completed an introductory programming text using the language Java. Villanova started using the Lewis text for its CS1 course in October 1997. The text includes material on threads and synchronization. However, this information is under the topic "Advanced Flow of Control" (Chapter 14) and is not present in the CS1 class. Chapter 14 is one of the closing chapters of the text.

Numerous other university offer concurrency and parallelism course work to third and fourth-year students (not directly associated with an operating systems course):

- Blackburn College [Meredith, 1992]
- Illinois State University [Hartman, 1991]
- Florida International University [Berk, 1996]

- University of Houston -- Clear Lake [Yue, 1994]
- University of Pennsylvania [Penn, 1998]

There is a separate class on the subject titled "Introduction to Parallel Processing" at the University of Pennsylvania. Also, the Carnegie Mellon University course titled "Programming Languages Design and Processing" is available to third and fourth-year students.

Some universities offer concurrency and real-time processing course work to fourth-year and graduate students (not directly associated with an operating system course):

- George Mason University -- Course CS621, Software Architecture / Design
- McMaster University -- Course CS-730, Real-Time Systems [McMaster, 1999]
- Millersville University -- Course CS360, Real-Time Systems Engineering [Millersville, 1999]
- Washington University -- Course CS520A, Intelligent Real-Time Systems [Washington, 1999]

2.4 Unique Approach to Concurrency -- Sisal

The Computer Research Group at Lawrence Livermore National Laboratory developed and supports a programming language called Sisal [Sisal, 1996]. The Sisal language is based on mathematical foundations. The Sisal language supports functional parallel programming using data-parallelism. The language was designed to use mathematical semantic rules to **guarantee** the following:

"Sisal programs are determinate, regardless of platform or environment."

The language was designed, such that, "race-conditions" and time-dependent behavior are detected by the compiler and prevented. The syntax of the Sisal language has a distinctive mathematical structure. The language manual contains the following acknowledgement: "if you have any experience programming in languages like Fortran, Cobol, C, or Pascal, there is a change in mind-set necessary in learning to use it." There is a paradigm shift because the Sisal language deviates from the imperative programming paradigm. An example of the distinctive syntax of the language is given below:

```
for I in 1, num_counters
   new_count := counters[I] + increment[I]
returns value of sum counters(I)
        value of sum new_count(I)
        array of new_count
end for
```

This example demonstrates aggregation (can be done in parallel) and reduction (must be done sequentially) in the same loop.

Lawrence Livermore National Laboratory develops and maintains software, in particular concurrent software. The concept behind Sisal is to develop a programming language that supports parallelism and removes the details of parallelism from "the shoulders of the programmer". This implies that the Lawrence Livermore National Laboratory approach to parallel programming is to require the programmer to not learn the concurrent paradigm. As a result, Sisal has limited applicability within the concurrent paradigm.

2.5 Concurrency as a Paradigm and Teaching Concurrency Today

There is no common answer across universities as to the when and how to introduce concurrency into the undergraduate curriculum. There is evidence that some universities are introducing concurrency into the curriculum earlier in the sequence of undergraduate computer science courses. The NSF is funding research into how to introduce concurrency earlier in the computer science curriculum.

Toll [Toll, 1995] in his paper "Decision Points in the Introduction of Parallel Processing Into the Undergraduate Curriculum" raised numerous curriculum level questions. A subset of Toll's questions is as follows:

- "How early should parallel processing be introduced?"
- "Should parallel processing be taught in one course or in several?"
- "Should the languages used for programming be extensions of known languages or new languages?"

In order to answer some of Toll questions, some basic goals need to be set forth. Introductory computer science students should be taught to reach these goals at the earliest possible time:

- Learn to think in the classical Von Neumann (sequential) paradigm
- Learn to think in the concurrent paradigm (Toll states this as "learn to think in parallel")
- Be able to recognize a simple sequential algorithms
- Be able to recognize a simple concurrent algorithms (Tolls states this as "understand some standard parallel algorithms")
- Learn to recognize that sequential (as a paradigm) is a subset of the concurrent paradigm
- Learn certain basic concepts within the concurrent paradigm (Toll states this as "understand different models of parallelism")

Examples of some of the basic concurrent paradigm concepts are as follows:

- Shared memory (Toll planned to include in a future CS2 class)
- Message passing (Toll planned to include in a future CS2 class)
- Non-determinism
- Aliveness
- Deadlock
- Synchronous activity
- Asynchronous activity
- Simple machine architecture (Toll planned to teach SIMD)

The amount of new information presented to students in this experiment on concurrency can be integrated into the existing CS1 class over seven weeks. One objective of this experiment was to demonstrate that simple, concurrent concepts and programming can be integrated into an existing CS1 course. The results of this experiment will demonstrate that the answer to Toll's first question is that parallel processing concepts can be, and should be, introduced in CS1.

The answer to Toll's third question is curriculum dependent. The CS1 and CS2 courses at George Washington University are taught using the programming language Ada. Since Ada includes concurrency using tasks; there is no need for either extensions of existing programming languages or new programming languages to introduce concurrency into the beginning of the computer science curriculum.

The answer to Toll's second question would be to integrate concurrency and parallelism across the computer science curriculum. By integrating only introductory concurrent paradigm material into the CS1 course, there will be additional concurrency information not presented in the course. This additional concurrency information should be presented in a later course or courses.

Toll [Toll, 1997] raises a new set of questions in his paper "Parallel Processing Integration in the Computer Science Curriculum: A Question of Balance". Toll is the principle investigator for a NSF grant research project at Taylor University for "Integrating Parallel Processing as a Tool Throughout the Undergraduate Computer Science Curriculum". Toll's questions are posed as alternatives about teaching concurrency and parallelism; a subset of his questions is:

- "Concept versus Programming"
- "Study Parallel Processing versus Use as a Tool"
- "Simulators versus Hardware"

This experiment introduces students to concurrency concepts and includes programming assignments. The answers to these questions are as follows:

- Both concepts and programming are included in teaching the concurrency material
- Parallel processing concepts are taught. Concurrency is a way to model and solve problems. Concurrency is not taught as a tool to solve some other type of problem.
- Neither -- students are taught concurrency concepts; the concepts are not tied to a specific hardware or simulator configuration

2.6 Common Introductory Concurrency Materials

This subsection presents commonly used introductory concurrency materials. The materials are not presented in the order that they would be in a class. Material presentation order is addressed in Section 4.

2.6.1 Concepts in Use

Many concurrency courses include Flynn's models of computation. Flynn defines two information streams [Hurley, 1994]:

- Stream of **instructions** (is the algorithm) that tells the processor what to do
- Stream of **data** (is the input) which are processed by the stream of instructions

Four classes of computer architectures that Flynn defined are as follows:

- SISD -- Single Instruction Stream, Single Data Stream
- MISD -- Multiple Instruction Stream, Single Data Stream
- SIMD -- Single Instruction Stream, Multiple Data Stream
- MIMD -- Multiple Instruction Stream, Multiple Data Stream

In addition to the four architectures above, Flynn included a "pseudo-machine" architecture: SPMD -- Single Program Multiple Data. Flynn's taxonomy was referenced numerous times; some of the references included Jackson, 1991; Fisher, 1991; Hartman, 1991; Olszewski, 1993; Hurley, 1994; Duckworth, 1994; Harlan, 1995; Kotz, 1995; Schaller, 1995; Toll, 1995; and Manchester, 1998.

In the SIMD architecture all processors are performing the same instruction at any given instant of time. SIMD is an example of synchronous parallelism. SPMD is asynchronous parallelism where each independent processor is running the same program. The SPMD concept can be implemented on a single MIMD computer or on a collection of SISD computers. SPMD is not a hardware architecture.

Many concurrency courses addressed one or more of the four concurrent programming models [from Fisher, 1991]:

- Message passing (MP)
- Shared memory (SM)
- Data parallel (DP)
- Extraction of parallelism (XP)

Shared memory and message passing are commonly discussed regarding interprocessor communications. One or more of these concurrent programming models were referenced numerous times; some of the references included Hartman, 1991; Fisher, 1991; McDonald, 1992; Kitchen, 1992; Langan, 1993; Hartman, 1993; Kotz, 1995; Anrow, 1995; Toll, 1995; Elenbogen, 1996; Berk, 1996; Ben-Ari, 1996; and Ashton, 1997.

In addition to shared memory, there were many references to the four subclasses of shared memory access (from Jaja, 1992, page 11) in the parallel random-access machine (PRAM) model:

- EREW -- Exclusive read, exclusive write shared memory access
- CREW -- Concurrent read, exclusive write shared memory access
- ERCW -- Exclusive read, concurrent write shared memory access
- CRCW -- Concurrent read, concurrent write shared memory access

Some of the references included Aki, 1989; Cormen, 1990; Fisher, 1991; Jaja, 1992; and Hurley, 1994.

In the paragraphs above, the underlying concept is that there is a continuum of parallelism from control-only to data-only parallelism [Hummel, 1997]:

- Control-only parallelism -- This is concurrency involving heterogeneous (or dissimilar) "heavy-weight" processes (each process is executing different executables)
- Data-only parallelism -- This is parallelism (possibly in a numeric application) where homogenous "light-weight" threads are co-operating to solve a single problem

Fox and Furmanski put forth a new classification of the various forms of concurrency based on a Web and Java language view of concurrent and parallel programming [Fox, 1997, page 415]:

- Data parallelism -- "parallelism over what is large in the problem", examples included the following:
    - "Natural parallelism over the particles in a [molecular] computation
    - Parallelism over several possible trails in the Sieve algorithm as the 'data' for data parallelism
    - Data parallelism tends to be 'massive' because computations are time consuming over what is 'large' in the problem"
- Functional parallelism -- typical thread parallelism, examples included the following:
    - The overlap of computation and communication
    - Multiple computation tasks executing concurrently
    - Units of concurrency are modest grain size (larger than a few instructions scheduled by a compiler and smaller than an application)
- Object parallelism -- "the type of problem solved by discrete event simulators"
    - "This is quite natural for C++ and Java where the latter can use the applet mechanism to portably represent objects"
- Metaproblems -- functional concurrency with large-grain size components

Fox characterizes metaproblems as problems (applications) with modest amounts of large-grain concurrency. Each concurrency grain in metaproblems is much more self-contained than in functional parallelism. "[M]eta problems are naturally implemented in a distributed (Web) environment."

2.6.2 Languages in Use

Miller [Miller, 1994] reports that courses with parallel processing content are beginning to appear at the undergraduate level. Of the 63 colleges and universities that responded to Miller's request for information and provided a specific course count (at the undergraduate level), there are 73 courses with parallel processing content offered (1.16

course per institution). Since an undergraduate student will be exposed to about one course, the selection of tools and language for that course will indicate information about what topic areas in parallel processing are taught.

Brilliant [Brilliant, 1996] documents that according to "Dick Reid at Michigan State" that the programming language Ada is the most popular alternative introductory programming language to Pascal. Further Brilliant states that "[p]robably the single language that best meets the pedagogical needs of computer science education is Ada".

Ben-Ari [Ben-Ari, 1996] describes in the paper "Using Inheritance To Implement Concurrency" how to simulate the concurrent primitive of other concurrent languages (such as occam, Joyce, and Linda). Ben-Ari uses Ada to teach "concurrency primitives that are not directly supported by the language" by simulation.

Yue [Yue, 1994] at the University of Houston - Clear Lake teaches an undergraduate course in concurrent programming using Ada that is oriented to seniors. The course uses a software orientation and does not address parallel hardware architectures directly. The course also emphasizes the language Linda because "it is based on a asynchronous communication model, as opposed to the synchronous communication model in Ada."

Many universities have chosen a base language for their computer science curriculum that does not support concurrency and parallelism. The programming languages Pascal, C, C++, and Fortran do not support concurrency and parallelism. These universities need to "add" or "extend" concurrency to their existing programming language. These universities concurrency classes are taught using extended languages that "add" concurrency to an already existing language. Examples of these extended languages include the following:

- Message passing libraries:
    - Parallel Virtual Machine (PVM)
    - Message Passing Interface Standard (MPI)
- High Performance Fortran (HPF)

Message passing is one of the most effective and widely used communication paradigms in parallel computing.

For universities not using Ada as the base language, "instructors of courses in parallel computing necessarily discussed the use and development of parallel software in the context of a vendor's proprietary software or software being developed by a research group, or, even worse, simply left the problem of using and developing parallel software as 'exercises' for the students" [Pacheco, 1977]. The Message Passing Interface Standard provides instructors with a standardized library of subprograms that can be called from the languages C, C++, and Fortran 77 (and later).

The core of the MPI concept is a pair of send and receive routines that allow processes to communicate data. The C language calls to the MPI routines are as follows:

- `Int MPI_Send (void* message, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm);`
- `Int MPI_Recv (void* message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status);`

The tag parameter is an integer value used to uniquely identify a message. The tag concept allows a sending process to repeatedly send the same message; each message is differentiated by a unique tag number. The source and destination parameters are integer values used to identify a process by its number (similar to the Unix process ID number, "pid"). MPI supports user-defined datatypes; these datatypes are simply the combination of an address and displacement (or sequence of displacements). This allows multiple pieces of information to be included in a single message. MPI was developed in an open forum. MPI has the following advantages:

- Teaches message passing
- Uses the existing compiler and a library extension (for MPI)
- MPI libraries can be recompiled (hardware independence)
- MPI libraries exist for the languages C and Fortran

MPI is referenced by the following authors: Ercal, 1996; Allen, 1997; and Pacheco, 1997.

High Performance Fortran (HPF) provides the following parallel processing support [NPAC, 1994]:

- Fine-grained data parallelism, using DO loops and other constructs
- Critical sections (within loops)

- Multiple processes that communicate using the following mechanisms:
    - Shared data abstraction (shared memory controlled by a monitor mechanism)
    - Virtual channels / files
    - Message passing (using MPI)

HPF was originally written for the purpose of creating data-parallel programs. The three items under the last bullet were added after HPF was first written.

HPF has compiler directive words (pragmas) that affect the compiler's interpretation of the DO loop:

- Independent -- Fortran statements following "independent" can be execute in parallel
- Reduction -- Fortran statements following "reduction" must be executed sequentially

The directive word independent represents a fan-out directive. The directive word reduction represents a fan-in directive.

HPF's virtual file mechanism supports user defined relationships between "reader" and "writer" processes. Each of the following parameter lists is valid for open:

- mode=virtual, writers=single, readers=single
- mode=virtual, writers=single, readers=multiple
- mode=virtual, writers=multiple, readers=single
- mode=virtual, writers=multiple, readers=multiple

Read operations associated with virtual files normally block the reading process(es) until data is available.

Other universities chose specialty languages that highlight concurrency and parallelism. In this case the base language for their computer science curriculum need not support concurrency and parallelism. Examples of these specialty languages include the following:

- Joyce and Linda
- Cooperating Sequential Processes (CSP)

The Joyce and Linda programming language [McDonald, 1992] has its origins in two programming languages:

- Linda parallel programming language
- Joyce parallel programming language

The Linda programming language was developed by Gelernter in the 1980s as a method of inserting a parallel programming capability into a sequential language such as C, Modula-II, and Pascal. Linda uses a concept called a tuple-space that is shared among all processes. Tuples provide two functions:

- Hold elements of a data structure
- Synchronize access to the tuple-space across processes

The arity of a tuple defines the number of fields in the tuple. Tuples are strictly typed objects. The Joyce programming language supports concurrent processes called agents. Agents are not allowed to share variables. Joyce uses a Pascal-like syntax. The Joyce/Linda programming language includes the concurrent processes support from Joyce and the tuple-space concept from Linda. In Joyce/Linda the tuple-space is the only means of processes communicating. The Joyce/Linda language is used at the University of Western Australia.

The University of Western Australia [McDonald, 1997] is currently working with PVM. PVM provides a "simple environment offering interprocess communication between heterogeneous systems". McDonald and Kazemi have developed PVM library modifications to support multiple students using a single virtual machine.

The University of Michigan at Dearborn uses the Joyce/Linda language for computer science lab assignments that explore "various parallel and distributed architectures and paradigms" [Elenbogen, 1996]. The University of Michigan at Dearborn uses the Joyce/Linda language in four undergraduate courses: algorithms, data structures, computer networks, and operating systems.

Although the Communicating Sequential Processes (CSP) language was developed in the 1970s, the language is still used and studied today. Olszewski [Olszewski, 1993] describes a "CSP Laboratory" at the University of New South Wales for the study of concepts pertaining to "a formal basis of parallel programming". The

CSP language constructs are translated into the programming language Miranda. Olsson [Olsson, 1995] describes a CSP language preprocessor that converts CSP notation into the equivalent program written in the SR programming language. The CSP language preprocessor was written at the University of California, Davis. These CSP language tools are used in advance undergraduate and graduate courses in operating systems and concurrency programming courses.

2.7 Teaching CS1 Students

Section 2 of this document is constrained by dissertation protocol to be a literature search only. As such, this subsection presents the literature on teaching CS1; the implementation of the items presented here is in Section 3 as additional information about the experiment.

Reid [Reid, 1994] at Michigan State University has developed a set of window based objects. One of these objects is a droid with 16 programmable control points. The control points are representative of joints in the human body. Students write a program that send messages to the droid object that controls a sequence of droid movements. Two of the goals of the program are to demonstrate the following:

- Use of loops and nested loops
- Existence of starting conditions and ending conditions

The setting of this exercise is shifted to the classroom in this experiment. The object of the exercise is to allow the students to apply their knowledge of loops to a new and concrete situation [Howard, 1996].

Liu [Liu, 1996] recommends that instructors "consider a remedial introductory course for incoming students who are less prepared". This is proposed as one of several ways to help retain female computer science students. The remedial instruction can be given at any time during the course; however, student confidence is a factor. Liu stated that "resources to assist students who do not do well in large class settings" were needed. The remedial instruction should have a "mentoring" style.

As an aid in organizing and learning the material, students were taught the basic elements of David Ausubel's learning theory [Ausubel, 1970]. Students are introduced to Ausubel's advanced organizer concept and encouraged to build their own organizer.

Ausubel's theory of learning fits into the traditional pedagogy of instruction. Ausubel's theory of learning addresses the following areas:

- Focus on inputs
  - Inputs are important to the learning process
  - Emphasis is on well organized material
- Mature learners (such as adults)
- Focus on cognitive demand (organization of material to reduce cognitive demand)
- Meaningfulness of new ideas (to the learner)

Ausubel's advanced organizer concept is a schema (a mental structure) by which a student can organize newly acquired information. This schema is instilled into the student with advance preparation and repetition. Intrinsic to Ausubel's theory is that well organized material leads to better reception by the student. According to Ausubel, "the principal function of the organizer is to bridge the gap between what the learner already knows and what he needs to know before he can successfully learn the task at hand" [Ausubel, 1970]. Using an organizer is a technique in facilitating the student's "differential analysis" between the information the student already knows and the information to be assimilated. Thus, the schema is the framework for the organization and recall of information. The schema provides the framework by which students accommodate and categorize new information (new ideas are hooked to existing knowledge), and recall and capture existing knowledge. The advance organizer technique is useful as a schema for storing, ordering, referencing, and retrieving technical information [Barsalou, 1992].

3 EXPERIMENTAL METHOD

This section presents an overview of the methodology used in the experiment and additional information. This section discusses the following materials in the order listed:

- Overview of the experiment and subject groups
- Choices not available in the experiment
- Constants in the experiment
- Types of information collected
- Experimental design
- Validity threats and their prevention
- Analysis tools
- Additional information

The last subsection is included to provide continuity with Section 2.7, Teaching CS1 Students. The last subsection documents additional items provided to all classes.

3.1 Overview Of The Experiment

This subsection of the document presents an overview of the experiment and describes how the class schedules are organized.

The programming language used by the CSci 51 students in the class is Ada 95. Ada 95 is the programming language used in the George Washington University (GWU) CS1 course CSci 51, Introduction to Computing. There are three groups of subjects in this experiment; they are as follows:

- Control group -- the students in the Fall 1997 CSci 51 class. These students received the same materials as taught by Feldman. In addition, these students received supplemental information to enhance the materials as taught by Feldman.
- Treatment group one -- the students in the Spring 1998 CSci 51 class. These students received both the traditional and experimental (concurrency and parallelism) materials.

- Treatment group two -- the students in the Fall 1998 CSci 51 class. These students received both the traditional and experimental (concurrency and parallelism) materials.

The CSci 51 class is divided into two time periods:

- Period one -- the first eight weeks of the semester
- Period two -- the second eight weeks of the semester

During period one both the control group and the treatment groups received the same instructional materials. A list of the many items that remained constant during period one of the experiment are presented later in this section. The planned similarity of instruction and materials during period one allows a comparison of the three groups. Thus, the performance of the groups (control and both treatment) during the first period is compared. For example, common folklore, as observed by the teaching assistant, says that the fall classes in CSci 51 do better (on average) than the spring classes in CSci 51. The fall classes, as observed by Feldman, are approximately one-half the size of the spring classes. By having a period of direct comparison of the three groups on the same material, differences between the three groups can be identified and normalized as necessary; such that, differences in the three groups are compensated in comparing student performance during period two.

During period two, the control group continued to receive the traditional lectures and materials, while the treatment groups also received the experimental lectures and materials (concurrency and parallelism). The treatment groups materials differed from the control group in the following areas:

- Different instructional materials were presented during the second eight week period -- the treatment groups received instruction in concurrency and parallelism
- Different lecture schedules were maintained
- Different projects were assigned -- the treatment group projects were varied to contain concurrency and parallelism programming exercises

- Different final exam reviews were presented
- Different final examinations were given (however, six very similar sequential questions were given to each group)

3.2 Choices Not Available In The Experiment

The following items could not be controlled in the experiment (they resulted due to outside factors):

- Number of students in each group (class size) -- students elect to register for the class based on their own schedules
- Time of day for lecture -- lecture times were chosen by Mr. Rowan, administrator for Electrical Engineering / Computer Science (EECS)
- Time of day for recitation -- recitation times were chosen by Mr. Rowan
- Computer facilities used -- the engineering school computer facilities were predetermined to be the Sun SparcServer Unix computer, called "Felix"
- Classroom location and size -- these were chosen by the university administration
- Use of Ada 95 language -- the language Ada 95 is the introductory programming language in the engineering school for computer science and computer engineering students
- Programming experience, or lack thereof, in the students' background

3.3 Constants In The Experiment

The following items are maintained constant throughout the experiment:

- Same instructor employed
- Same teaching assistant employed
- Same grading standard used
- Same computer facilities used
- Same opportunity for electronic mail interaction with the instructor
- Same opportunity for electronic mail interaction with the teaching assistant

In addition, during period one (the first eight weeks) many more items were maintained constant for both the control group and the treatment groups:

- Same lectures were presented
- Same instructional material was presented
- Same lecture schedule was maintained
- Same projects were assigned (different geometric shapes and numerical values were used)
- Same mid-term review was presented (different numerical values were used)
- Similar midterm examinations were given (questions were in different order and numerical values were changed)

3.4 Types Of Information Collected

The purpose of this subsection is to associate the data collected with the hypotheses and findings. The following types of information were collected from all groups (control and treatment) for the experiment:

- Mid-term exams -- a copy of each exam was kept
- Final exams -- the original of each final exam was kept
- Grades of each individual problem given on a mid-term or final exam
- Selected homework projects (originals)
- Grades of every homework project including total points earned, penalty for lateness, and actual score
- All electronic mail correspondence -- both messages received and replies
- Student surveys collected at the beginning of each class
- A complete record of every compilation and link done by a student including all source code and time of day information

The first question to be answered is as follows -- are there significant differences in the three groups? All the groups received the same instruction during period one of the semester (first eight weeks). This question is answered in Section 5, Results, of the document. The answer is based on the two separate comparisons: the classroom part of the mid-term exam scores and project scores (Projects 2, 3, 4, and 5 were used).

The subject area of the first hypothesis is as follows -- does teaching concurrency to novice programmers reduce their test performance on sequential material? All the groups received six final exam questions that were sequential in nature and were very similar or nearly identical. The answer is based on the comparison of the final exam scores for these six sequential questions given to all three groups.

The subject area of the second hypothesis is as follows -- after instruction in concurrency, are novice programmers more / less able to solve concurrent questions than sequential questions? The answer is based on a comparison of the treatment groups' concurrency final exam scores versus the treatment groups' sequential final exam scores. Since the point value of the sequential questions is different than the point value of the concurrency final exam scores, the total points of the sequential questions (57 points) is normalized to the total points of the concurrency questions (41 points).

The subject area of the third hypothesis is as follows -- are novice programmers less able to use concurrent methods than sequential methods on "large" projects? The last project in the course, Project 9, is the large project. The control group students used sequential methods to do the project. The treatment group students used concurrency methods to do the project. The answer is based on a comparison of the control group and treatment group scores on Project 9.

In addition, the compilation data collected is used as a "sanity check" on the level of effort for Project 9, as measured by the number of compiles. During period one of the semester, all groups performed five projects (these projects are very similar or nearly identical from group to group). The following ratio is then computed on a per student basis:

$$\text{Ratio} = \frac{\text{Number of compiles for Project 9}}{\text{Number of compiles for Projects 2, 3, 4, and 5}}$$

A comparison of the control group's ratio to each treatment group's ratio was made. The comparison of the groups' ratios provided a measure of the level of effort required of the students in completing the last project, Project 9. Given that significant differences in the groups' ratios are not found, then Project 9 experimental results are less subject to attack due to the relative level of effort, as measured by the number of compiles.

The subject area of the "findings" research is as follows -- an examination of differences in compile error profiles of students using concurrent and sequential methods on "large" projects. The profile examined is the distinct error ratio (DER). DER is the ratio of total occurrences of compilation errors to number of distinct errors. The DER is examined for the last project, Project 9. As previously stated, compilation and link data were collected. Compressing all the variations in error messages into a smaller number of distinct error messages is not an exact science. The method used to reduce the total variation of error and warning messages is as follows:

- Replace variable names with a representative symbol (such as "token")
- Replace numbers with a representative symbol (such as the value "99")

Thus, the error message "*"class_sum" is undefined*" is translated to "*"token" is undefined*". The error message "*"=" should be ":="*" remains unchanged. Approximately 450 distinct error messages (both sequential and concurrent) were found. Approximately 20 distinct warning messages were found.

Student surveys were collected from the students in each group. A few students refused to provide survey information and were excluded from the experiment -- background information on the student is not available. The data in the surveys along with other information (refer to Section 5, Results) were used to determine which students were novices, as defined by the experiment. Also, survey data were used as input to correlation studies. For example, the correlation between college year and concurrency question test scores was examined.

Although not necessary to the experiment, the following additional information can be derived from the compilation and link data collected:

- Number of compiles to successful compilation
- Number of successful compiles to project completion
- Number of compiles necessary to avoid a specific error (by syntax error type)

Program executions were not collected due to the certainty of infinite loop output being recorded to disk and crashing the engineering school computer.

The students in both the control and treatment groups were told the following about the recording of every compilation and link done by the students:

- The instructor was collecting information to make changes in the course
- The compilation and link information was not used for grading at any time
- Information collected would be used to make changes in the information presented
- Students programs were automatically backed-up every time they compiled, a benefit used on occasion by students

3.5 Experimental Design

The design methodology for this experiment is called "static-group comparison design". This design methodology is part of a major classification of experimental designs called pseudoexperimental designs. Pseudoexperimental designs are often used because of the experimental resources available. The essence of the static-group comparison design is as follows:

- There are two or more separate and distinct groups of subjects
- The subjects in each group are self selecting -- in this experiment each student has chosen to enroll in the course
- Random selection of subjects is not feasible -- again the student chose to enroll during a given semester

Traditional discussions of this design methodology imply that both the control and treatments groups are studied at the same time. In this experiment, the groups are studied sequentially (one group at a time). The engineering school enrollment supports only a single CS1 class per semester. There are several validity threats to an experiment of this design. They are presented in the next subsection.

In a static-group comparison design experiment the treatment group is exposed to the treatment and the control group is not exposed. Thus, the experimental variable is the exposure or non-exposure to the treatment (the treatment is the independent variable). The observed results are the dependent variables (for example, the sequential question final exam scores).

The statistical tests used in the experiment are described in Appendix M, Statistical Tests. However, some general background on how the statistical tests are applied is appropriate to present here. The size of the control group was 24 students. The sizes of the treatment groups were 52 and 21 students, respectively. At these group sizes normality test results are inaccurate [Hintze, 1999]. Thus, non-parametric statistical tests are appropriate (normality is not an assumption of these statistical tests).

Many of the statistical tests used compare the distribution of three groups; the three group distribution comparisons done were as follows:

- Mid-term exam scores
- Project 2, 3, 4, and 5 scores together
- Sequential question final exam scores
- Project 9 scores
- Compilation ratios (Project 9 compilations divided by Project 2, 3, 4, and 5 compilations)
- Distinct Error Ratio

The Kruskal-Wallis One-Way ANOVA on Ranks test [Hintze, 1999] is an appropriate non-parametric test for comparing three groups for significant differences.

The analysis associated with the second hypothesis compares an individual treatment group's concurrency final exam scores versus the treatment group's sequential final exam scores. This comparison requires the analysis of two distributions for significant differences. The Mann-Whitney U test [Huck, 1974] [Hintze, 1999] is an appropriate non-parametric test for comparing two distributions for significant differences.

The Kruskal-Wallis One-Way ANOVA on Ranks test and the Mann-Whitney U test include the following assumption: the data groups have equal variances. The Modified-Levene Equal-Variance test [Hintze, 1999] is an appropriate non-parametric test for comparing the variance of two or three distributions. In the event that the data groups do not pass an equality of variance test (Modified-Levene), the non-parametric Kolmogorov-Smirov test is used. This test is used to detect a significant difference in two samples or distributions. Restated, the Kolmogorov-Smirov test computes the probability that two samples are from the same distribution.

3.6 Validity Threats and Their Prevention

There are several validity threats to a static-group comparison design experiment [Huck, 1974]. This design methodology is called pseudoexperimental because the methodology lacks built-in controls to deal with threats. These validity threats include the following:

- Self selection of subjects
- Mortality
- Maturation
- History

The self selection of subjects is the greatest threat. This is because each subject group will be different, possibly very different. It is a given in this experiment that each group is going to be different. However, there are techniques that minimize the impact of these differences. One of the simplest and most effective techniques is to calibrate each group against the others by providing identical treatments and observing the result. This is exactly what was done during the first eight weeks of the semester. All groups received the same material during the first eight weeks of the course. The relative performance of each group can be calibrated against the same measure. Thus, each group's performance during the period when different treatments are applied, the second eight weeks in the course, could be adjusted for group differences.

Mortality deals with the loss of subjects from any of the groups. The experiment can not be adjusted to defend against mortality -- students leaving the group (either by withdrawal or change of status, such as audit). Enough students remained in each class for a statistically valid sample size to exist at the end of the semester. Students are required to take and successfully complete CSci 51 to advance to the other computer science courses. This alone "appears" to provide sufficient motivation for students to complete the course (and provide a statistically valid sample size). However, it is important to determine the underlying causes of mortality. Using the control group as an example, five students changed their status. The students' rationales for their status changes were as follows:

- Student A -- this student was told by her father that her major in college was to be computer science and that she would attend an American university. This student expressed interest in art (particularly drawing) and was not willing to spend the time she felt was necessary. The student withdrew as soon as her father gave permission.
- Student B -- this student became ill in the sixth week of class and could not continue.
- Student C -- this student was an employee of the university and newly married. The student realized that the course included homework and chose not to continue the course.
- Student D -- this student has a bachelor degree from another university and was an employee of the university. The student realized she was not willing to spend the time necessary to complete many of the homework assignments on time and changed her status to audit.
- Student E -- this student received a low mid-term grade, "D", and chose not to risk an average grade for the class. The student withdrew. The student's major was computer science. To attend the graduate school of his choice, the student claimed that all his grades must be "A" or "B".

Maturation deals with the subjects growing maturity (both psychological and biological). The impact of this threat is that any one group may be more mature than the another group. This could impact the experiment's results. One of the simplest and most effective techniques is to calibrate each group against the others by providing identical treatments and observing the result. This was done as part of the experiment.

History deals with external non-experimental events affecting a subject's performance over the period of the experiment. A significant emotional event will impact a subject's performance. For example, during the control group phase, the grandmother of a student died. This student was excused from the course for three class periods. The student's makeup work was not graded. The overall work was not affected. The important aspect of history threats to an experiment is that if a threat goes unobserved, a subject's performance is affected by an influence outside the experiment and may be attributed to the experimental treatment.

In addition to the above external validity threats, another threat that can be controlled within the experiment is instrumentation. Instrumentation deals with the problems evaluating dependent variables. The instruments in this experiment include the instructor and the graders. For example, differences between the control group and the experimental groups could be attributed to the instructor and graders unless mechanisms were carefully built-in to the conduct of the experiment. The mechanisms are presented in Section 5, Results.

3.7 Analysis Tools

There were two tools developed for this experiment:

- Compilation-recording tool
- Compilation analysis tool

The compilation-recording tool is a Unix shell script. The script records the compilation listing file (such as "*.lsb" files for Ada 95) along with date and time information. The information is recorded on a "userid" (student account) basis. Thus, a complete history of each student's compilation information is recorded. An entry level computer science student does not possess the skills necessary to avoid or bypass the recording process. Although student compilation information is recorded, students do not have read access to the directories that contain their compilation histories. Students are required to submit all project work using the SEAS computer Felix. This means that students must compile, link, execute, and print their project work on the computer named Felix in order to submit their projects. If a student chooses to develop part of the project homework on another computer, the student is required to provide very simple progress reports by electronic mail. Only two students (one in the control group and one in the Fall 1998 treatment group) chose to do this; and terminated the activity within a week.

This tool also generates electronic mail to the instructor when a new student starts using the computer and when recording direciory problems occur. The recorded information is captured weekly and archived monthly.

In addition to the compilation-recording tool, a linker recording script is also operating. The linker recording script is a Unix shell script. This script records the object name, date and time when the linker is invoked.

The compilation analysis tool is designed to aggregate compilation error message information into a format that is easily analyzed for the identification of statistical distributions.

Originally, the probability of a given error type was to be computed using a form of time series analysis. This analysis was futile and was not done, because many students chose to solve only one error type at a time. Thus, for these students, the probability of a given error being repeated is primarily dependent upon the presence of other errors. The distinct error ratio (DER) analysis (see Section 5, Results) and Project 9 level of effort result are sufficient indicators of novice programmer behavior.

Processing the recorded compilation data was done in three phases:

1. Generate a list of distinct errors
2. Generate a table of information about each event
3. Generate a table of information about the error message data associated with each event

In phase one, a table of all known distinct errors was generated. This was done by a series of Unix utilities. These Unix utilities included "sed" (for string recognition), sort (to order the error messages), and "unique" (to create a list of distinct error messages). The result of this phase was that each distinct error message was assigned a unique identification number. Two examples of distinct messages are given below:

- 1044: "token" is undefined
- 2009: warning: Program_Error may be raised at runtime

Message numbers beginning with the number one were error messages. Message numbers beginning with two were warning messages. The distinct messages were placed into a file.

In phase two a table of distinct and non-repeating event data was generated for each compile. An event was one compile of a specified source code file. A program was written to do the data transformation, the compilation analysis tool. The result of this tool was one record per compile. The format of the record is given in the table below. The collection of all these records was then stored in a database table. The format of the date is not year 2000 ("Y2K") date format compliant; the experiment was started in 1997 and completed in 1999. Date format compliance was never an issue in the experiment.

Column Name	**Description**
event_id	Unique identification number for each compile
userid	Student's userid (such as bjones)
date	Six digit number, format YYMMDD, the date the source code was compiled
time	Six digit number, format HHMMSS, the time the source compilation began
project_id	Two digit project number, valid project numbers are 1 through 9, the value 0 means the compile is not directly project related
lines	Compiler reported, number of program lines
package	Package identifier (1 means program, 2 means package)
clean_compile	1 means clean compile, 2 means warning or error messages
post_link	Program was linked (1 means yes, 0 means no)
program_name	File name of the program (such as proj_03.adb)

In phase three, a file of compilation error occurrences was created. Another result of the data transformation program was the creation of the compilation error occurrence record. The record's format was given in the table below. The phase two and phase three processing were combined into a single program; such that, both phases were done together. The collection of all these records was then stored in a database table.

Column Name	Description
event_id	Unique identification number for each compile
line_number	Program line number where error reported
message_id	Error identification number

Once all files were created, the data was loaded into a Microsoft Access 97 database (as tables) for processing.

3.8 Additional Information

The purpose of this subsection is to document additional items done by the instructor as part of teaching the classes. These items tie back to Section 2.7, Teaching CS1 Students.

Reid [Reid, 1994] developed a set of window based objects. One of these objects is a droid with 16 programmable control points. This exercise was changed from a programming assignment into a classroom exercise. The goals of the class exercise were to demonstrate the following:

- Use loops and use nested loops
- Have starting conditions and ending conditions
- Keep the exercise simple enough that it could be "programmed" by the class

The set of motions chosen was dancing -- in particular, the bunny hop. Several students were invited to the front of the classroom. The students were divided into two groups -- programmers and performers. The programmers were required to create a "program" that included "N" number of iterations of the bunny hop set. FOR LOOPs were used.

The results were as follows -- students demonstrated the following:

- Knowledge of FOR LOOPs
- Comprehension of how to use the FOR LOOPs

After the exercise students had to explain how they would apply this knowledge to the upcoming homework project. Later students performed a class exercise in drawing analogies between the number of dancers and processor architectures. Computer architectures analyzed include SISD, SIMD, and MIMD [Hurley, 1994].

Another technique used was to appoint a student to represent a programming language construct. The student became the symbol or icon representing the programming language construct. For example, one student was appointed to be the "loop lady". In many (but not all) classroom exercises on loops, the instructor started the exercise and the loop lady completed the exercise. The "loop lady" was associated with the following loop syntax:

- FOR LOOP
- Loop increment variable
- END LOOP

The purpose of the "loop lady" symbol was to get students to think of all three items above when they remember information on loops.

Liu [Liu, 1996] recommends that instructors "consider a remedial introductory course for incoming students who are less prepared". This is proposed as one of several ways help retain female computer science students. Additional drill and practice instruction was available to students during weeks 10 through 15 of the course. These sessions were offered at a time chosen by the students -- normally late Friday afternoon (one-two hours long). The primary purpose of these classes was to teach students how to study for, and take, an examination. These additional sessions were recommended for students whose grade average was below "C". However, any student regardless of grade average could attend. Attendance to these drill and practice sessions was optional. These extra sessions had the following structure:

- 20 minutes on the current homework project
- 25 minutes drill and practice on one pseudo-final exam question
- 25 minutes drill and practice on another pseudo-final exam question
- 10 minutes of after class questions

The topic was drawn from the course material. For example, in the lecture on functions and procedures, the students were taught the following items:

- Identify the components of a function
- Identify the components of a procedure
- Write a simple function

- Write a simple procedure
- Rewrite a function as a procedure
- Rewrite a procedure as a function (if possible)
- Create a one-page outline of the material presented and reference the textbook by page number

As an aid in organizing and learning the material, students were taught something about David Ausubel's learning theory. Students were introduced to Ausubel's advanced organizer concept and encouraged to build their own organizer [Ausubel, 1970].

The organizer used was a hypothetical program named "bob"; the program was organized like a ladder where students place, "hang", information. For example, when students reviewed functions, the students were asked to place the functions learned in the organizer and then write procedures that performed the same work as the functions. At the next extra session, students were reminded about the procedures and asked to mentally retrieve the functions. Another example, when students reviewed records and arrays, the students were asked to place the arrays and records in the organizer and then compose arrays of records and records that contain arrays. Again at the next extra session, students were reminded about arrays of records and asked to mentally retrieve basic array and record information. The program name bob was intentional. After the organizer concept is implanted in the students, lecture material was often structured around routines named bob and array and record definitions named bob. Students in the extra sessions have been taught to associate the name "bob" with storing information away. After many weeks of practice, the "program bob" also contained other reference information, like textbook page numbers. During the final exam review, students were asked again to retrieve information.

4 COURSE CONTENT OVERVIEW

This section of the document presents the control group course materials, the treatment group replacement course materials, and the schedule of material presentation. In addition, this section discusses the motivation for the replacement course materials. This section discusses the following materials in the order listed:

- CS1 course outline without the concurrency and parallelism paradigm
- CS1 course outline as taught in the Fall of 1997
- Additional materials introduced in the Fall of 1997
- Introductory concurrency and parallelism materials to be included in a CS1 course
- Introductory concurrency and parallelism materials to be excluded from a CS1 course
- CS1 course outline with concurrency and parallelism introduced

4.1 Current CS1 Course Outline

The class named Introduction to Computing, Computer Science (CSci) 51, is the introductory computer science class for computer science majors and computer engineering majors at George Washington University. It is the CS1 class. The class is taught two days per week for 16 weeks. Two of the 16 weeks are used for review and examination; therefore, only 14 weeks of class time are available for instruction. One possible weekly class schedule, as prepared by Feldman [Feldman, 1997], is given below (the item numbers represent the week):

1. Introduction (from Chapter 1)
2. Introduction to Programming with Ada 95 (from Chapter 2)
3. Introduction to Design, Enumeration Types, the Spider (from Chapter 3)
4. Using Packages (from Chapter 3)
5. Decision Statements (from Chapter 4)
6. Writing Functions and Packages (from Chapter 4)
7. Review for Mid-term Exam and Exam
8. Counting Loops, Introduction to External Files (Chapter 5)

9. General Loops, Exception Handling (Chapter 6)
10. Writing Procedures, Parameter Modes, Robust Input (Chapter 6)
11. Case Statements, Math Library, Random Numbers (Chapter 7)
12. Composite Types: Records (Chapter 8)
13. Composite Types: Arrays (Chapter 8)
14. Strings and Files, Part 1 (Chapter 9)
15. Strings and Files, Part 2 (Chapter 9)
16. Review for Final Exam and Exam

The chapter numbers are from the Feldman and Koffman text. In addition to the two weekly class lectures, the class includes a one hour lab per week. Class lecture and lab attendance is required. Both the lecturer and lab instructor provide office hours. Office hours are operated on a first come, first served, bases; appointments are not required.

This paragraph contains a general listing of the topics covered in the CS1 class. The information is organized on a week-by-week basis (as presented by Feldman). The topics covered in week one (1) are as follows:

- Syllabus
- Computer account setup and use of electronic mail (Email)
- Background on Ada
- Components of a computer
- Syntax and semantics, a definition

The topics covered in week two (2) are as follows:

- Syntax and semantics, a definition
- First program compilation and program execution
- Execution exception, its general appearance
- The "look" and general form of Ada
- Numerical input and output
- Declaration of variables and constants
- Assignment statements
- Input / output statements

- Beginning data types and expressions
- Software development and project requirements, as defined by a case study

The topics covered in week three (3) are as follows:

- Building programs from existing information
- Extending a problem solution
- Introduction to packages (WITH)
- Package building blocks (methods called functions and procedures)
- Using a screen control package
- Introduction to the Spider
- Enumeration types (operations and input / output)

Most of the topics presented in week three are related to the rationale for, and the use of, packages. The topics covered in week four are an extension of week three; these topics include the following:

- Importance of packages
- Using the Ada package calendar
- Using packages
- Using the Ada Text_IO package -- several data types are output using put
- Subprograms in packages -- concept of a method
- Scope of a variable declared in a PACKAGE
- Using a constant value declared in a PACKAGE

The topics covered in week five (5) include the following items about decisions:

- IF ... THEN statement
- IF ... THEN ... ELSE statement
- IF ... THEN ... ELSIF ... ... ELSE statement
- Logical operators <, <=, >, >=, =, /=
- Nested IF statements and nesting level

The topics covered in week six (6) include the following about functions and packages:

- FUNCTION Statement -- specification
- FUNCTION BODY Statement
- Subprogram parameters -- all "IN" for FUNCTIONS

- RETURN statement
- Scope of a variable in declared in a FUNCTION
- PACKAGE specification
- PACKAGE BODY
- More on WITH
- Creating a simple PACKAGE
- Using a simple PACKAGE

The topics covered in week seven (7) include the review for the mid-term exam and the exam. The topics covered in week eight (8) include the following:

- FOR LOOP
- Nested FOR LOOP
- Algorithm development using LOOPs
- External file declaration
- External file input -- open, get, close
- SUBTYPE ... IS ... RANGE ... ...;
- Logical operators IN, NOT IN
- Overloading concept
- More on Spider using LOOPs

The topics covered in week nine (9) are as follows:

- WHILE statement
- LOOP design -- zero iteration, flag control, increment up or down
- LOOP statement and EXIT WHEN statement
- Exception handling: data_error, constraint_error
- EXCEPTION WHEN ... statements WHEN ... statements END
- Introduction to robust input using exceptions

The topics covered in week 10, involving procedures and robust input, are as follows:

- Continuation of robust input using exceptions
- PROCEDURE declaration
- PROCEDURE BODY
- Parameter modes (IN, OUT, IN OUT)

- Parameter lists
- Parameter association (named / position)
- Case study: sum_and_factorial (robust input examples)
- Continuation of Spider (more on procedures)

The topics covered in week 11 are as follows:

- More on WITH and USE
- Data types: review of integer, natural, float
- Data types: boolean, character
- Explicit type conversion
- Ada.Numerics package: more on USE, using SQRT and SIN routines
- CASE statement
- Continuation of Spider (random walk)

The topics covered in week 12 include the following about records:

- Records provide organization of data
- TYPE ... IS RECORD
- END RECORD;
- record_name.field_name
- Record operations are store, retrieve, assignment, equality ("=" and "/=")
- Record aggregate assignment
- Records and packages
- Record hierarchies (x.y.z concept)
- Read files and write records concept (added)
- Introduction to birthdays database (a case study)

The topics covered in week 13 include the following about arrays:

- Arrays provide organization of data
- TYPE ... IS ARRAY (<index_range>) of <data_type>
- Array subscript (or array index) concept
- Array operations are store, retrieve, assignment, equality ("=" and "/=")
- Array aggregate assignment
- Using arrays and subscripts as expressions

- FOR LOOPs and arrays
- Access to arrays (random or sequential)
- Copying and array
- Arrays with non-integer subscripts
- Simple searching and sorting
- Continuation of birthdays database (a case study)

The topics covered in week 14 include the following about strings and files:

- String data type
- String data type is not character data type
- String assignment, comparison, output
- String concatenation
- String input
- Character and string package Ada.Text_IO
- End_of_Line and End_of_File concepts

The topics covered in week 15 are a continuation of the strings and files discussion presented the previous week:

- Redirection of input and output
- Command line parameters
- File open
- File close
- File input using strings

In week 16 of the class includes the review for the final exam and the exam.

4.2 Revised CS1 Course Outline, Fall 1997 Semester

The Fall 1997 Introduction to Computing class, CSci 51, was the control group for this experiment. The class schedule as taught is given in the table below. The table includes both the week and the day of the week that the material was covered. Day 1 is Tuesday; day 2 is Thursday.

Week	Days	Text Chapter	Material Covered
1	1 & 2	1	Introduction
2	1 & 2	2	Introduction to Programming with Ada 95
3	1 & 2	3	Introduction to Design, Enumeration Types, the Spider
4	1	3	Using Packages
	2	4	Decision Statements
5	1 & 2	4	Writing Functions and Packages
6	1 & 2	5	Counting Loops; Introduction to External Files
7	1	--	Additional Material
	2	--	Mid-term Review
8	1	6	Exception Handling (Start), General Loops
	2	--	Mid-term Exam -- Covers Chapters 1 through 5
9	1 & 2	6	Exception handling; Writing Procedures; Parameter Modes
10	1 & 2	7	Case Statements; Math Library; Random Numbers, More Data Types
11	1	8	Composite Types: Records
	2		Composite Types: Arrays
12	1	8 & 9	Composite Types: Arrays (concluded); Systematic View of Strings and Files
	2	--	Last Project, Number 9
13	1	--	Last Project, Number 9
	2	9	Systematic View of Strings and Files (continued)
14	1	9	Systematic View of Strings and Files (concluded)
	2	--	Thanksgiving Day Holiday
15	1	--	Introduction to Computer Architecture
	2	--	Final Exam Review
16	1	--	(No Class)
	*	--	Final Exam (one week later)

The syllabus for the Fall 1997 class, see Appendix B, has a slower schedule. This syllabus was based on the instructional pace of the prior instructor. For example, the syllabus states that "decision statements" were to be started in class week five; however, "decision statements" were started at the end of class week four. This acceleration of the pace was most noticeable for the topic of arrays. In the syllabus arrays were scheduled for class week 13; however, the array lectures were completed at the beginning of class week 12 (this represented a three lecture acceleration of the pace). A comparison of the control group and treatment groups class schedules is presented later in this section.

4.3 Additional Materials Introduced to the Fall 1997 Class, the Control Group

The following materials were added to all classes:

- Basic object-oriented principles
- Introductory computer architecture
- Order of N notation
- Two-dimensional arrays
- Unix commands

Instruction of these materials is spread out throughout the course, and introductory computer architecture was repeated as a single class period. This information is included here to document all materials covered (for repeatability of the experiment).

Students were taught basic object-oriented principles. These principles were a tool, a framework, to assist in teaching students about functions, procedures, and (and later in the treatment group) tasks. These principles were taught using the word "item", rather than object, because students were not taught a rugged definition of "object". The principles taught were explained as presented by Forsythe and Mavrovouniotis [Forsythe, 1996]; the principles taught were:

- Abstraction -- identifying similarities between "items" as a core of object-orientation
- Encapsulation -- the combination of both data and/or methods in an "item"

- Polymorphism -- methods that are used to perform operations are stored piecewise in the "items"; the compiler identifies the correct method using a hierarchy defined by the "items" and/or the method's parameters

Introductory computer architecture was taught as a framework for presenting computer operation and the division of operation within the computer. Computer architecture topics included:

- Modern PC and its components -- processors, memory, disk drive(s), controller card, monitor, video card, keyboard, CD ROM, and additional selected components (such as a zip drive) as part of a single user system
- Workstation and its components (the computer Felix) -- multiple processors, multiple memory, multiple disk drives, multiple controllers, and additional selected components (such as communications processor) as part of a multi-user system
- Single Instruction, Single Data (SISD) model of computation
- Sharing concepts -- multi-processing, multi-programming, multi-computing, time-slicing, and time-sharing

Order of N notation was introduced as a way of understanding the relative number of iterations necessary to produce a result. Order of N notation was associated with the generation of geometric shapes. These shapes included lines, triangles, squares, rhombuses, and rectangles. Students were taught to associate the order of N necessary to produce these geometric shapes. Order of N notation instruction was highly coupled to instruction of loops and nested loops.

One-dimensional arrays were part of the introductory programming course (prior to the control group class). Two-dimensional arrays were introduced for their utility in applications (such as matrices and determinants).

Unix commands were taught to the students to enable them manipulate, protect, and share files. The following Unix commands were taught to all students in prior CSci 51 classes (prior to the control group class):

- cat -- concatenate, display a file (or files)
- cd -- change directory

- cp -- copy a file (or files)
- ls -l -- list directory contents
- mkdir -- make directory
- more -- incremental display of a file
- pwd -- display current directory path
- rmdir -- remove a directory
- users -- display current users on the computer
- "<" -- redirected input from a file
- ">" -- redirected output from a file

The following additional Unix commands were also taught to all CSci 51 students during the experiment:

- chmod -- change mode
- elm -- electronic mail, alternative to pine
- grep -- display lines that contain specified character sequence
- head -- display the first lines of a file
- script -- record terminal output to a file
- telnet -- establish session on another Unix computer
- "|" -- pipe, concatenate a series of commands

Most Unix commands are taught to the students during the first half of the class.

4.4 Introductory Concurrency and Parallelism Materials to be Included

The concurrency and parallelism material was organized around eight units (one unit per week). Each unit was designed to require no more than one class period (approximately 80 minutes). Some units required only 15 to 30 minutes presentation time (refer to units one and eight). Other units require the entire period (refer to unit four). Unit two contained both procedures (sequential behavior) and tasks (concurrent behavior); procedures and tasks were introduced together. The unit topics included:

1. Flynn's models of computation
2. Introduction to procedures and tasks
3. Beginning concurrent concepts

4. Modeling a problem concurrently and programming constructs
5. Message passing and programming
6. More concepts of concurrent programming
7. Shared memory and programming
8. Efficiency and Amdahl's law

Two homework assignments commingled sequential and task behavior.

The outline of these units is given in paragraphs below. The outline of these units includes both the unit content and order of presentation. Other lecture material (such as records and arrays) is interspersed with the concurrent units.

Unit one was scheduled for course week seven (7). Flynn's models of computation are discussed as part of a computer architecture lecture. The models of computation were:

- Single Instruction stream, Single Data stream (SISD)
- Single Instruction stream, Multiple Data stream (SIMD)
- Multiple Instruction stream, Single Data stream (MISD)
- Multiple Instruction stream, Multiple Data stream (MIMD)
- Single Program Multiple Data (SPMD)

SISD model of computation was also taught to the control group.

Unit two was scheduled for course week nine (9). Procedures and tasks were introduced in the same time period (course week nine). The Ada language material presented was as follows:

- Prototype for a PROCEDURE (a declaration)
- PROCEDURE BODY
- Parameter modes (IN, OUT, IN OUT)
- Parameter lists and association (named / position)
- TASK TYPE declaration including ENTRY statements
- TASK BODY
- ACCEPT statement (a simple wait)
- DELAY statement

Students were taught to think of tasks for concurrent work. Concurrent concepts introduced were as follows:

- Tasks are scheduled
- Simplified Ada task model
- Non-determinism
- Messages are sent to tasks (tasks can wait for a message)
- Tasks start at beginning of the instantiation of an enclosing frame
- Synchronization of tasks
- Tasks writing to a screen

Students were asked as a classroom exercise to visualize "adding up" ten thousand numbers by hand. The students were also asked to organize a work detail of two or more students to "add up" all the numbers.

Unit three was scheduled for course week 10. This unit covered the following concepts (some concepts were repeated from the prior week):

- Non-determinism
- Message passing

```
task_two.go;  -- Statement in task one
ACCEPT go;  -- Statement in task two
```

- Deadlock
- Interleaving
- Liveness
- Race condition
- Waiting (ACCEPT statement)
- Synchronizing tasks (modeling dependent activities)

Students in the extra class session were asked to crumple paper and pass messages.

Unit four was scheduled for course week 11. The emphasis of this unit was modeling, more basic concepts and Ada syntax. The modeling part of the unit included the following applications:

- Sequential applications that are better served by concurrent processing [Burns, 1995, pages 30-33]
- Dining philosophers, a concurrent application

The basic concepts and Ada syntax included the following (some repeated) items:

- Tasks are scheduled
- Starting and stopping a task
- Passing a "message" to a task
- How tasks are started (tasks are not called routines)
- Basic concepts
  - Atomicity
  - Contention
  - Communication
  - Interleaving
- Simple wait (ACCEPT statement)
- Semaphore
- Other types of waits (using SELECT in combination with "OR" or "ELSE") [Cohen, 1986, page 728-729]
  - Busy waits
  - Conditional waits
  - Selective waits
  - Timed waits

Unit five was scheduled for course week 12. The emphasis of this unit was message passing. The unit contained the following information:

- Message passing tied to object-oriented methodology
- Rendezvous (more on synchronization)
- Message passing characterization
  - Tasks (processes) exchange messages to synchronize activities
  - Tasks (processes) exchange messages to pass data
  - Each task has a unique identifier (or tag)
  - Messages are sent to a single process (task)
  - Messages are sent to all processes, called a broadcast

  - Message passing works on computers with distributed memory
  - Message passing works on distributed computers on a network
- Knowledge within a message (Ada task model):
  - Source task must know target task (destination address)
  - Target task may not need to know source task (source address)
- Pattern of communication is determined by the programmer

Unit six was scheduled for course week 13. The emphasis of this unit was refined definitions (using Bustard's curriculum module "Concepts of Concurrent Programming" [Bustard, 1990]) and completeness. The unit contained the following information:

- More on task scheduling
- Ada task model (simple Ada task model is expanded)
- Atomic instruction concept
- Refined and new definitions:
  - Concurrent program
  - Parallel program
  - Non-determinism
  - Process (task) interaction
  - Synchronous communication
  - Asynchronous communication
  - Critical regions
  - Mutual exclusion
  - Lockout
  - Safety
  - Multi-tasking

Unit seven was scheduled for week 14. The emphasis of this unit was shared memory. The unit contained the following information:

- Shared variables
- Relatively simple to program
- Multiple processors access a central memory
- Review of multi-tasking techniques (like semaphores)
- All processors share the same main memory (like SUN SPARCstation 20 or SUN Enterprise 3000 or Felix)

- Memory is available to every processor except for small sections of memory that may be exclusively used by one processor for short time periods

Unit eight was scheduled for week 15. The emphasis of this unit was factors that limit efficiency and Amdahl's law. This small unit contained the following information:

- speedup = SN = Ts / Tp
- efficiency = EN = SN / N
- Software overhead
- Load balancing
- Communication overhead
- Amdahl's Law

Due to classroom time constraints, coverage of unit eight materials was limited.

4.5 Introductory Concurrency and Parallelism Materials to be Excluded

The concurrency and parallelism material excluded from the class resulted from time constraints; some of these topics included the following:

- Selected fundamentals of interprocessor communication

  Exclusive Read, Exclusive Write (EREW)
  Concurrent Read, Exclusive Write (CREW)
  Exclusive Read, Concurrent Write (ERCW)
  Concurrent Read, Concurrent Write (CRCW)

- Methods of obtaining parallelism (data parallelism vs. domain decomposition)
- Concurrent programming paradigms (specialist parallelism, agenda parallelism, and result parallelism) and data structures associated with them
- Bi-directional message passing communications (Ada supports both IN and OUT parameters, OUT parameters were not addressed)
- Pipelining techniques
- Monitors
- Protected types
- Interconnection Networks (such as fully connected, mesh or torus, ring, hypercube, and shuffle exchange)

- Metrics for interconnection networks (such as connectivity, diameter, and narrowness)
- Parallel algorithm construction
- Global parallelism (coarse-grained) versus local parallelism (fine-grained)
- Implicit concurrency (fine-grained) versus explicit concurrency (coarse-grained)

4.6 Topics With Removed, Reduced, Or Changed Coverage

This subsection addresses difference in topic coverage between the control group and the treatment groups. As previously stated in Section 4.2, the syllabus for the Fall 1997 class, refer to Appendix B, had a slower schedule. The Fall 1997 syllabus was based on the instructional pace of the prior instructor. The instructional pace used in the experiment was faster; eight classes under the old pace required almost seven classes under the instructional pace of the experiment. The change in pace was discussed and accepted by the course director, Professor Feldman. During the Fall 1997 semester, this provided additional classroom time at the end of the semester. Additional time was available and spent on Project 9 and the "Systematic View of Strings ands Files" topic during the Fall 1997 semester. For example, one class period was spent on Project 9 homework. The faster instructional pace decreased the need for sequential topics to be removed from the treatment group's instruction.

The following topics were removed from the sequential material taught to the treatment groups:

- Random numbers
- Case studies based on the birthday's database
- Case studies based on the Spider during the second half of the semester

The following topics received reduced class time instruction to the treatment groups:

- Robust input and output
- Math library
- Case statements

All topics removed and reduced were discussed with the course director, prior to the topic being removed or reduced.

The presentation of procedures and tasks were commingled for the treatment groups. The identical overhead transparencies on procedures were presented to the control and treatment groups. Again, this was discussed with the course director.

During the Fall 1997 semester, the week seven class finished early. Thus, the computer architecture briefing was moved from week 15 for the control group to week 7 for the treatment groups. All instruction given to the control group during the first eight weeks of the course was repeated with the two treatment groups.

The class textbook presents records before arrays in Chapter 8. The control group was taught records before arrays. The treatment groups were taught arrays before records in order for coverage of arrays to coincide with project work, Project 7.

4.7 Revised CS1 Course Outline, Fall 1998 Semester

The Fall 1998 Introduction to Computing class, CSci 51, was the second treatment group for this experiment. The class schedule as taught is given in the following table. The table includes both the week and the day of the week that material was covered. Day 1 is Tuesday; day 2 is Thursday. Schedule differences between the Spring 1998 treatment group and the Fall 1998 treatment group were as follows:

- Period one (weeks one through eight) -- none
- Period two (weeks nine though 16) -- at most one class

The "*" in the column named material covered indicates concurrency and parallelism material presented. The only topic present in the control group and not present in the treatment group is random numbers and their generation.

Week	Days	Text Chapter	Material Covered
1	1 & 2	1	Introduction
2	1 & 2	2	Introduction to Programming with Ada 95
3	1 & 2	3	Introduction to Design, Enumeration Types, the Spider
4	1	3	Using Packages
	2	4	Decision Statements
5	1 & 2	4	Writing Functions and Packages
6	1 & 2	5	Counting Loops; Introduction to External Files
7	1	--	Additional Material -- Computer Architecture with Models of Computation *
	2	--	Mid-term Review
8	1	6	Exception Handling (start), General Loops
	2	--	Mid-term Exam -- Covers Chapters 1 through 5
9	1 & 2	6	Exception Handing (concluded), Parameter Modes; Introduction to Procedures and Tasks *
10	1 & 2	8	Composite Types: Arrays, Beginning Concurrent Concepts *
11	1 & 2	8	Composite Types: Records, Concurrent Modeling and Constructs *
12	1 & 2	7	Case Statements; Math Library, and Message Passing *
13	1 & 2	--	More on Data Types, Last Project, Number 9, Concepts of Concurrent Programming *
14	1	9	Systematic View of Strings and Files (started); Shared Memory *
	2	9	Thanksgiving Day Holiday
15	1 & 2	9	Systematic View of Strings and Files (concluded) Efficiency and Amdahl's law *
16	1	--	Final Exam Review
	*	--	FINAL EXAM (week after review)

5 RESULTS

This section of the document presents the results and findings of the experiment. This section of the document is organized into the following subsections:

1. Student population
2. Comparability of the groups
3. Assessment of concurrency instruction on learning sequential material
4. Concurrency materials and sequential materials performance comparison
5. Comparison of student performance on a large project
6. Compilation of concurrency and sequential programs findings
7. Correlation of student characteristics to student performance
8. Validity and sensitivity check
9. Hypothesis and findings

In addition to the above items, other observations made during the experiment that are not directly related to an area of hypothesis or findings can be found in the appendices.

5.1 Student Populations

Students were placed into one of four categories. The four categories are as follows:

- Novice -- students included in the experiment
- Experienced -- students excluded from the experiment due to completing one or more college level classes in computers and / or computer languages
- Removed -- students excluded from the experiment for cause
- Withdrawal -- students withdrawing from the course at any time

Only students in the novice category were included in the experiment. The student populations for the three semesters of the experiment are given in the table below.

Category	Control Group	Treatment Groups		Totals
Semester	*Fall 1997*	*Spring 1998*	*Fall 1998*	
Novice	24	52	21	97
Experienced	4	5	1	10
Removed	2	5	5	12
Withdrawal	5	14	11	30
Initial Class Enrollment	35	76	38	149

Students were assigned to the removed category for one or more of the following reasons:

- Repeating the course due to a prior failing grade or a prior late withdrawal
- Failing to provide background information in the student survey
- Failing to do most of the homework
- Failing to try to succeed on an examination
- Having a physical handicap that impaired performance

Some examples of actual students removed from the experiment illustrate the point:

- One student refused to do any more than the first of nine project assignments.
- Another student decided not to study for the final exam and sat in the final examination for two hours and did only one of eleven problems.
- One disabled student refused special treatment until late in the semester and experienced difficulty in taking exams at Disabled Student Services (DSS).

Students who withdraw from the course have not taken the final examination and have not completed the last project. These two items are important parts of the experiment. Therefore, students withdrawing from the class were not included in the experiment. The withdrawal rate for the Fall 1998 semester was unusually high. This was partially due to student illness.

The Fall and Spring semester classes in the experiment were different in the following characteristics:

- Spring semester starting enrollment was 75 to 80 students (double the Fall enrollment)
- Spring semester ending enrollment was 62, while Fall semester ending enrollments were 30 and 27, respectively
- Spring semester students were younger
- Spring semester students were mostly freshmen while the Fall semester students were a mix of mostly freshmen and sophomores

All classes in the experiment shared the following characteristics:

- There were more males than females in the class
- Most students were required to take the class
- Entry skill levels were very similar
- The number of foreign students was low

The novice students were placed into several categories of prior computer programming learning experience. The category selected was based on student survey responses and student responses to individual questions. These categories were as follows:

- None -- the student claimed zero computer-related experience (this includes high school computer literacy courses)
- HTML -- the student had had formal training in HTML in a college course or in a business setting
- Lost -- the student had had college level programming experience in the past and was unable to use or recall the experience (for example, a 43 year old student had a college level programming class over 20 years ago)
- Some -- the student had some minor experience (typical examples were as follows: a student took six weeks of Java at another university and withdrew from the class, another student wrote one Perl script at the office)
- Self taught -- two students in the Spring 1998 class had previously purchased compilers for their home computers and had started learning to program

- HS-1 -- the student claimed to have completed one high school level class in computers (a typical class included computer literacy and introductory programming with Pascal)
- HS-2 -- the student claimed to have completed two high school level classes in computers (the second class is typically a Pascal programming class)

The number of novice students in each of the programming experience categories is given in the table below. Each of the categories in the table is assigned a factor that is representative of the programming experience associated with the category (for example, the factor for category None is 0). Categories where a student's experience may be beneficial to the student learning to program are assigned a factor of 1. A factor of 2 is assigned only to the HS-2 category (the experience should have been beneficial to the student).

Category	**Control Group**	**Treatment Groups**		**Factor Assigned**
Semester	*Fall 1997*	*Spring 1998*	*Fall 1998*	
None	14	31	12	0
HTML	1	3	2	1
Lost	3	1	2	1
Some	1	1	3	1
Self Taught	0	2	0	1
HS-1	3	11	1	1
HS-2	2	3	1	2
Novice Total	24	52	21	

As previously stated, students who had completed one or more college level courses in computers were excluded from the experiment (experienced student category). Thus, categorization of college level student experience is omitted from the table above.

A summary of the general characteristics of the novice student population in the experiment is given in the table below. The averages are expressed to three significant digits. The numerical codings used in the table are as follows:

- Ratio of Males (female is designated by 0 and male is designated by 1)
- Years in College

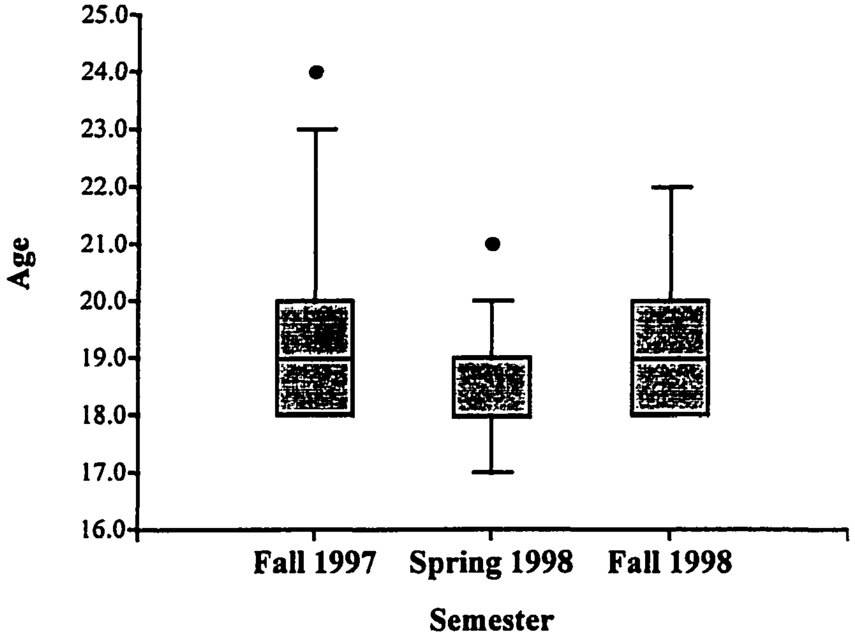
  Freshman is 1
  Sophomore is 2
  Junior is 3
  Senior is 4
  Graduate is 5

- Required Class (Required is designated by 1, otherwise 0)
- Skill Level (explained above)
- Foreign Student (Foreign is designated by 1, otherwise 0)

Statistic	**Control Group**	**Treatment Groups**	
Semester	*Fall 1997*	*Spring 1998*	*Fall 1998*
Ratio of Males			
Average	0.542	0.673	0.667
Median	1	1	1
Age (see Figure below)			
Average	19.5	18.75	21.1
Median	19	18.5	19
Year in College			
Average	1.75	1.08	1.90
Median	1	1	2
Required Class			
Average	0.833	0.923	0.857
Median	1	1	1
Skill Level			
Average	0.500	0.462	0.476
Median	0	0	0
Foreign Student			
Average	0.125	0.038	0.143
Median	0	0	0

A box plot of the student's ages is given in Figure 5.1. The box plot is a simple diagram for assessing symmetry, general equality of location, and equality of variation among the three groups. The age distribution of the Spring 1998 class is different.

Figure 5.1. Novice Student Age By Semester



The figure shows that the age distribution of the spring class is younger than the age distributions of either fall class. The youngest students were in the spring class (age 17). The box plot shows the following age characteristics of the Spring 1998 students:

- Top line -- age 20, normally the 100th percentile (unless outliers are present)
- Box top -- age 19, the 75th percentile
- Box bottom -- age 18, the 25th percentile
- Bottom line -- age 17, normally the 0th percentile (unless outliers are present)

When the 50th percentile (the median) is distinct from the box top or box bottom, the box diagram includes a center line for the median. When the median is the same as box top or box bottom, the median does not show in the box plot. The "height" measure of the box is called the interquartile range (IQR). When outliers are present, represented by little circles, the nearest line to the little circle(s) represents:

- Top line -- 75th percentile plus 1.5 IQR
- Bottom line -- 25th percentile minus 1.5 IQR

There were two students in the Fall 1998 semester whose ages are outliers outside the range of the box plot (there ages are 32 and 43). The average age of the Fall 1998 class was 19.4 years old after these outliers are removed from the average.

Students in the novice category are also categorized based on their homework participation, either full or partial. Full participation means that the student must have completed almost all of the computer projects. Some students completed insufficient homework projects to be included in *all* program compilation statistics; these students are placed into the partial participation subcategory (because one or more projects included in the experiment were not done). Thus, statistics derived from program compilations may not include the partial participation students. The novice student subcategory populations for the three semesters of the experiment are given in the table below. The full participation counts are an upper bound. For example, one full participation student had over two thousand compilation errors in a single project, this observation is an outlier.

Category	**Subcategory**	**Control Group**	**Treatment Groups**		**Total**
Semester		*Fall 1997*	*Spring 1998*	*Fall 1998*	
Novice	Full	18	44	18	80
	Partial	6	8	3	17
Total		24	52	21	97

5.2 Comparability of the Groups

The first eight weeks of each class contained the same sequential material. During this eight week period the students attended lectures, performed five projects, and took the mid-term exam. The mid-term exam was given at the end of the eight week period. The control group and two treatment group classes were compared in this subsection on their performance during the first eight weeks of class.

5.2.1 Mid-Term Exam

The mid-term exam was given in two-parts. Part one was a six question test, given in the classroom. The second part of exam was a single mini-project to be completed in lab during mid-term week. Only part one of the mid-term exam was included in the experiment. Appendix F, Mid-Term Examination, contains the part one mid-term exam. From this point forward in the document, any reference to mid-term exam refers solely to the part one, six question, exam.

The overall statistics for the mid-term exam are given in the table below.

	Control Group	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	59.88	55.79	54.62
Median	65	58	58
Standard Deviation	12.862	12.670	16.660
Variance	165.419	160.523	277.548

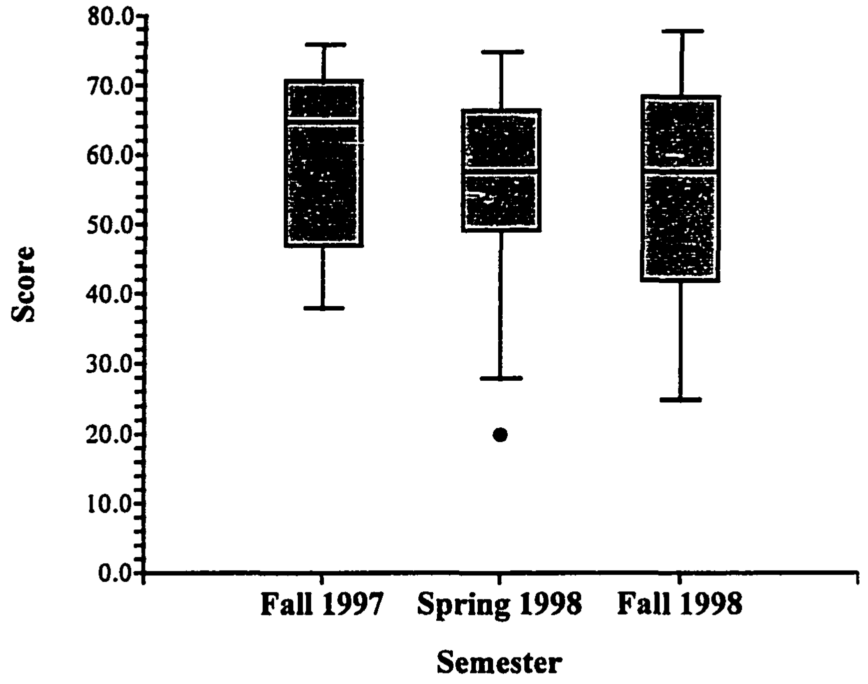
In all three groups the median score is higher than the average score. The maximum score is 79. The greatest variance in scores is in the Fall 1998 class. Both treatment group classes are consistent in both average and medians. The control group class has out performed the treatment group classes in both average and median score.

The box plot for the mid-term exam score is given below in Figure 5.2. The figure shows the following about the percentile ranks of the classes:

- 100th percentile scores are almost the same
- 75th percentile score for the Fall 1997 class is higher than the others
- 50th percentile score for the Fall 1997 class is higher than the others
- 25th percentile for the Fall 1997 class is between the other two classes.
- 0th percentile score for the Fall 1997 class is higher than the others

The lowest score in the Spring 1998 class is statistically an outlier. In summary, the 75th, 50th (median) and 0th percentiles are higher for the Fall 1997 class, the control group.

Figure 5.2. Novice Student Mid-Term Scores



The first item to determine is whether the medians of the three classes are significantly different. The table below shows the results of normality tests (for all three groups together) and equal-variance test on the combined groups.

Assumption	Test Value	Probability	Decision (0.05)
Skewness Normality of Residuals	-2.5731	0.010078	Reject
Kurtosis Normality of Residuals	-0.7914	0.428726	Accept
Omnibus Normality of Residuals	7.2473	0.026685	Reject
Modified-Levene Equal-Variance Test	1.1602	0.317881	Accept

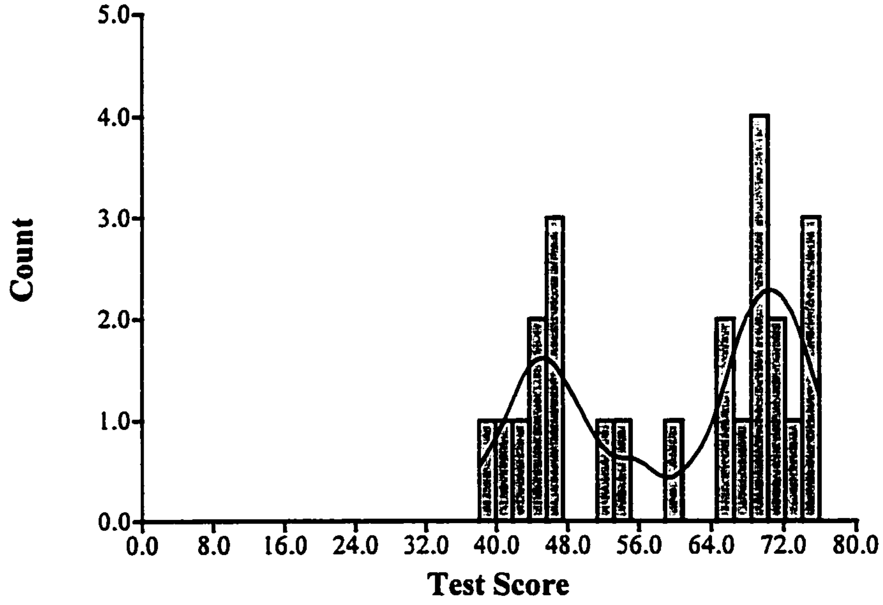
The skewness, kurtosis, and omnibus tests are tests of normality. The skewness test measures the direction and degree of asymmetry of a distribution. The kurtosis measures the heaviness of the tails of a distribution. The omnibus test combines skewness and kurtosis in a single measure of the overall normality of a distribution. The Modified-Levene test is an equal-variance test. The omnibus test result indicates that one or more of the three groups are not normal. Therefore, non-parametric statistical tests are required. The Modified-Levene test indicates that there is sufficient equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks. The results of the Kruskal-Wallis One-Way ANOVA on Ranks test are given in the table below. See the appendix section "Statistical Tests" for a discussion on "ties".

Method	Chi-Square (H)	Probability	Decision (0.05)
Corrected for Ties	1.961352	0.375058	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.**

The histogram of the Fall 1997 control group mid-term scores is shown below in Figure 5.3. The distribution is bimodal. There is only one student with a test score below 40. The scores have clustered in two distinct groups.

Figure 5.3. Novice Students, Control Group, Mid-Term Scores



The histograms of the two treatment groups are shown below. The histogram of the Spring 1998 treatment group mid-term scores is shown in Figure 5.4. The histogram of the Fall 1998 treatment group mid-term scores is shown in Figure 5.5.

Figure 5.4. Novice Students, Spring 1998 Treatment Group, Mid-Term Scores

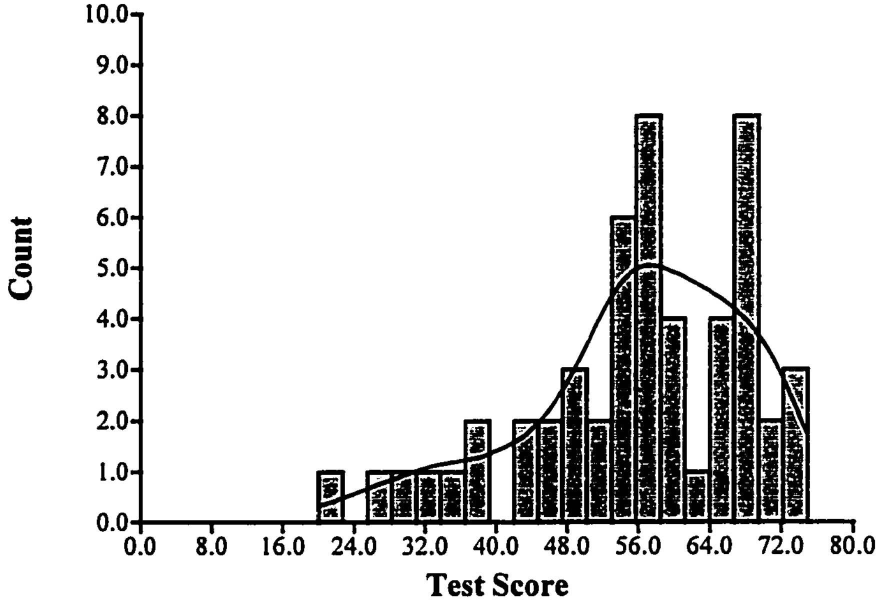
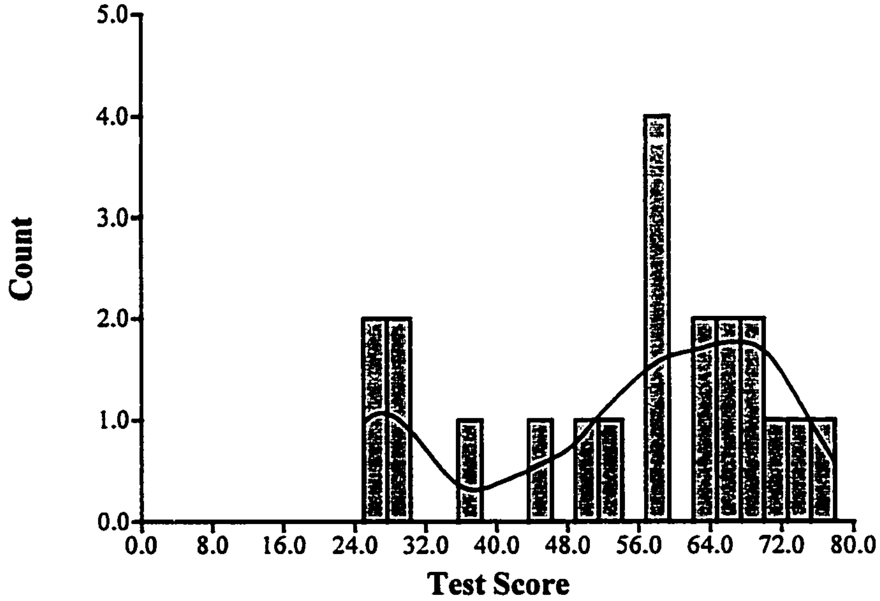


Figure 5.5. Novice Students, Fall 1998 Treatment Group, Mid-Term Scores



The Spring 1998 treatment group distribution is not bimodal. There are seven students with test scores below 40. The histogram bars are almost contiguous. The distribution has two peaks of eight students each.

The Fall 1998 treatment group distribution is not bimodal. The distribution is fairly flat, except for the four test score at about 56. There are five students with scores below 40. The control group has the least students with scores below 40.

5.2.2 Pre-Concurrency Projects

There were five projects given before concurrency was introduced. The first project was very simple, correct some syntax errors and change an existing program. The first project did not include original programming and was not included in the experiment. Projects two, three, four, and five were included in the experiment. These projects provided simple problems for the students to solve.

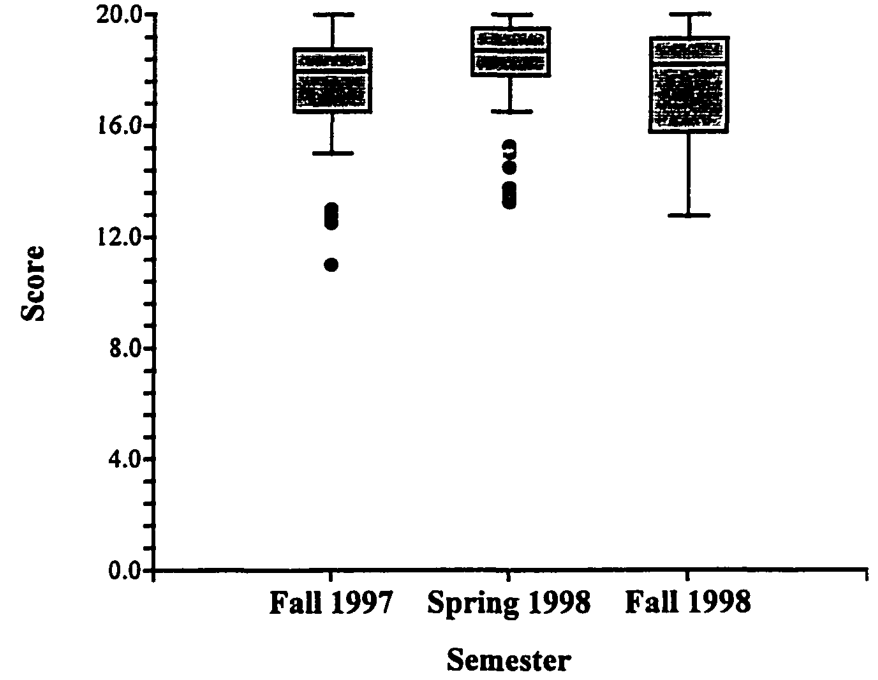
Students were graded on the best seven project scores from the first eight projects. In addition, students had to complete the last and large project, Project 9, as part of their grade. This grading method was a given in teaching the Introduction To Concurrency class. Therefore, some students chose to skip one of the first five projects. While, other students chose not to skip any of the first five projects and reserve "skipping a project" for later. Both situations are examined in this subsection.

Pre-Concurrency Projects With Zero Scores Included. The overall statistics for projects two through five (including zero scores) are given in the table below. Each of these projects are scored on a 20 point scale. The average of the four project scores was computed for each student.

Includes Zero Scores	**Control Group**	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	17.16	18.11	17.63
Median	18	18.75	18.25
Standard Deviation	2.480	1.966	1.981
Variance	6.151	3.866	3.923
Student Count	24	52	21

In all three groups the median score is higher than the average score. The higher variance in the control group results from five of the 24 students electing to skip Project 5. All three groups are consistent in both average and medians. The box plot for project two through five scores is given below in Figure 5.6.

Figure 5.6. Project Scores (Pre-Concurrency), With Zero Project Scores



The figure shows the following about the percentile ranks of the project scores:

- 100th percentile scores are the same for all groups
- 75th percentile score for the Spring 1998 class is higher than the others
- 50th percentile score for the Spring 1998 class is higher than the others
- 25th percentile score for the Spring 1998 class is higher than the others
- 0th percentile score for the Spring 1998 class is higher than the others

The Spring 1998 treatment group has slightly outperformed the two fall groups. Many of the outliers in the Fall 1997 group result from five of the 24 students electing to skip Project 5. The outliers in the Spring 1998 group result from students skipping assorted projects (only five of 52 students did not do one of the projects). There are no outliers in the Fall 1998 group (only two of 21 student did not do one of the projects).

The table below shows the results of the normality tests (for all three groups together) and the equality of variance test on the combined groups.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-4.3146	0.000016	Reject
Kurtosis Normality of Residuals	1.1091	0.267397	Accept
Omnibus Normality of Residuals	19.8457	0.000049	Reject
Modified-Levene Equal-Variance Test	0.3759	0.687731	Accept

The omnibus test result indicates that one or more of the three groups are not normal. The Modified-Levene test indicates that there is sufficiently equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks. The results of the Kruskal-Wallis One-Way ANOVA on Ranks test are given in the table below.

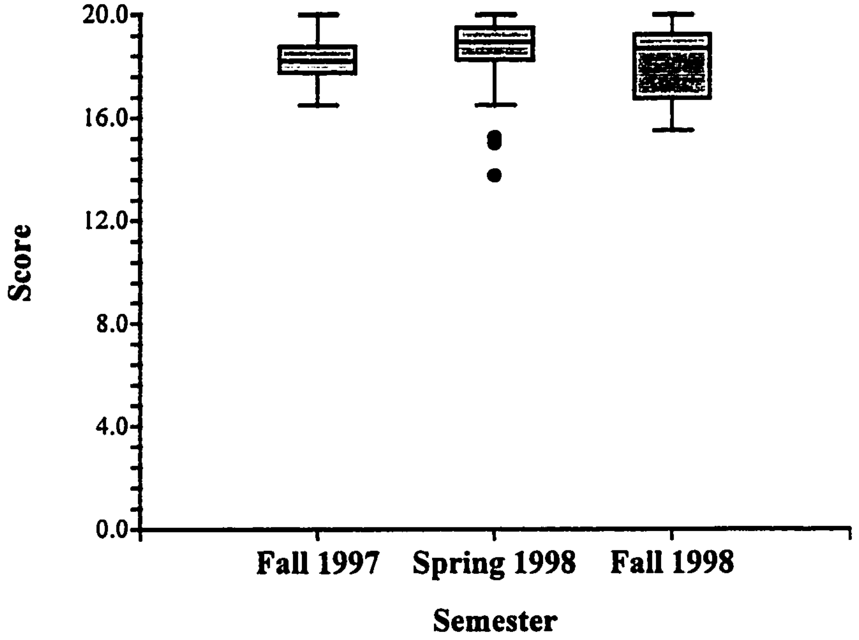
Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Corrected for Ties	5.634519	0.059770	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.**

Pre-Concurrency Projects With All Projects Completed. The overall statistics for projects two through five, where all four projects have been completed, are given in the table below. The average of the four project scores was computed for each student. In the control group the average and median are almost equal. In the treatment groups the median scores are still higher than the average score. The variances are reduced in all groups as compared to the project scores with zeros included. The box plot for project two through five scores is given below in Figure 5.7.

	Control Group	Treatment Groups	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	18.29	18.54	18.05
Median	18.25	19	18.75
Standard Deviation	0.947	1.507	1.517
Variance	0.898	2.270	2.303
Student Count	19	47	19

Figure 5.7. Project Scores All Projects Completed (Pre-Concurrency)



The Spring 1998 treatment group slightly outperformed the two fall groups. The only remaining outliers are in the Spring 1998 group.

The table below shows the results of the normality tests (for all three groups together) and the equality of variance test on the combined groups.

Assumption	Test Value	Probability	Decision (0.05)
Skewness Normality of Residuals	-4.6789	0.000003	Reject
Kurtosis Normality of Residuals	2.8611	0.004221	Reject
Omnibus Normality of Residuals	30.0778	0.000000	Reject
Modified-Levene Equal-Variance Test	0.8754	0.420538	Accept

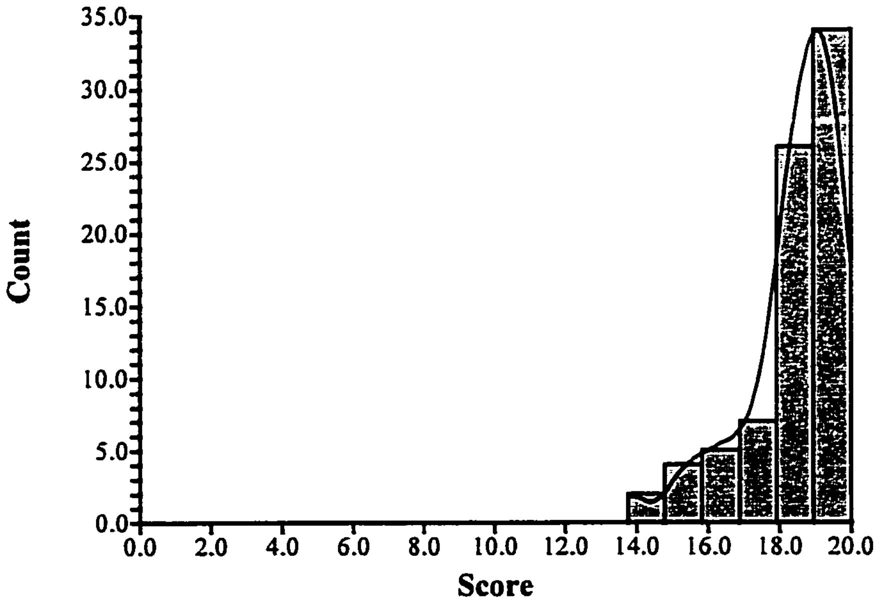
The omnibus test result indicates that one or more of the three groups are not normal. The Modified-Levene test indicates that there is sufficiently equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks. The test results are given in the following table.

Method	Chi-Square (H)	Probability	Decision (0.05)
Corrected for Ties	4.313945	0.115675	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.**

Figure 5.8 is the histogram of the combined project scores for all three groups. After eight weeks in the class all groups are doing well on projects two through five (*when the students chose to do the projects*).

Figure 5.8. Project Scores, All Projects Completed (Pre-Concurrency) (All Groups Together)



Conclusion. Significant differences in the mid-term scores and the project scores (projects two through five) were not found among any of the groups. Therefore, significant differences between the three groups were not found. The control group and two treatment groups are comparable for the purposes of this experiment. Differences in age and maturity (by college year standing) are not a cause of significant differences among the control group and the two treatment groups at eight weeks into the semester.

5.3 Assessment Of Concurrency Instruction On Learning Sequential Material

An assessment of the impact of concurrency instruction on learning sequential material was done as part of the final exam. The final exam in all three classes had eleven questions. The layout of the final exam for the treatment groups was as follows:

- Questions based on sequential material were 1, 3, 5, 9, 10, and 11 (57 points total)
- Questions based on concurrent material were 2, 4, 7, and 8 (41 points total)
- Extra complexity sequential question (like extra credit) was question 6 (12 points)

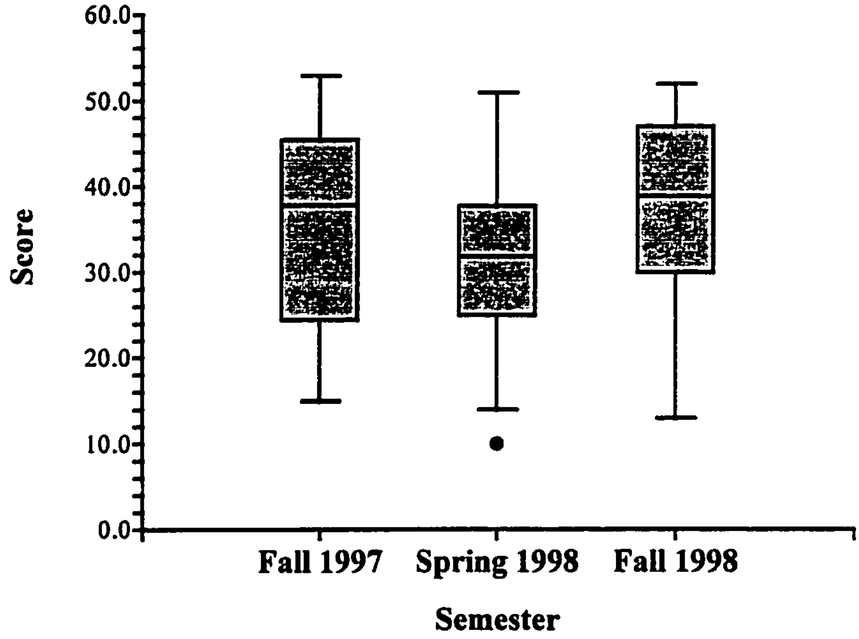
The control group also had an eleven question final exam. Questions 1, 3, 5, 9, 10, and 11 remained almost the same for both the control group and the two treatment groups. There is one exception that is covered later in this subsection. The test scores for each group were compared as an assessment of concurrency instruction on learning sequential material. The control group students were not taught concurrency material. Question 6 (extra complexity sequential question) was not part of the study.

Six Common Sequential Questions. The overall statistics for the six common sequential final exam questions are given in the table below.

	Control Group	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	35.54	31.40	36.95
Median	38	32	39
Standard Deviation	11.662	8.997	11.469
Variance	135.998	80.951	131.548

In all three groups the median score is above the average score. The maximum score possible is 55. Two of 57 points are removed from the inter-group comparison. The Fall 1998 treatment group has the highest scores. The Spring 1998 treatment group has the lowest scores. The Fall 1997 control group and the Fall 1998 treatment group have very close average and median scores. The difference between the minimum and maximum average is about 5.5. The difference between the minimum and maximum medians is 7. The fall classes did better than the spring class. The box plot for the six sequential questions is given in Figure 5.9 below.

Figure 5.9. Final Exam Six Sequential Question Scores



The figure shows the following about percentile ranks of the final exam six sequential question scores:

- 100th percentile scores are in the same range (51 to 53)
- 75th percentile score for the Spring 1998 treatment group is about the same as the 50th percentile scores (medians) of both the Fall 1997 control group and the Fall 1998 treatment group.

- 25th percentile score for the Fall 1997 control group and the Spring 1998 treatment group are about the same and both are less than the Fall 1998 treatment group.
- 0th percentile scores are in the same range (13 to 15)

The top 50 percent of the Spring 1998 treatment group under performed the two other groups. Also, the outlier subject is in the Spring 1998 treatment group. The outlier subject is also the lowest score of all three groups. The outlier bound in this figure is set for 1.1 IQR. As stated earlier in the document, the interquartile range (IQR) is the "height" measure of the box (the 75th percentile to the 25th percentile).

The table below shows the results of the normality tests (for all three groups together) and the equal-variance test on the combined groups.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-1.4576	0.144960	Accept
Kurtosis Normality of Residuals	-1.2665	0.205320	Accept
Omnibus Normality of Residuals	3.7286	0.155003	Accept
Modified-Levene Equal-Variance Test	1.6534	0.196922	Accept

The omnibus test result indicates that all three groups pass the normality test. The Modified-Levene test indicates that there is sufficient equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks. The results of the Kruskal-Wallis One-Way ANOVA on Ranks test are given in the table below.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Corrected for Ties	6.020551	0.049278	Reject Ho

The "Reject Ho" decision means that the Spring 1998 treatment group score is significantly different than the Fall 1997 control group and the Fall 1998 treatment group, when tested around their medians. Since only one treatment group is the source of the rejection (the Spring 1998 treatment group) and the outlier is in Spring 1998 group, the test was rerun with the outlier removed. The results of rerunning the test with the outlier removed are given in the table below.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Corrected for Ties	5.420731	0.066512	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.** The effect of this one outlier illustrates the impact of outliers on small sample populations.

In the table below are the average (on the left) and median (on the right) for each of the six test questions. Test questions are changed slightly from semester to semester. The results for question five in the Spring 1998 treatment group are in bold. The Spring 1998 treatment group was asked to solve a question that was similar to the question five given to the other two groups. The difference was that the Spring 1998 group question included an "integer divide" and the other groups' question did not. This clearly impacted the test results for question five.

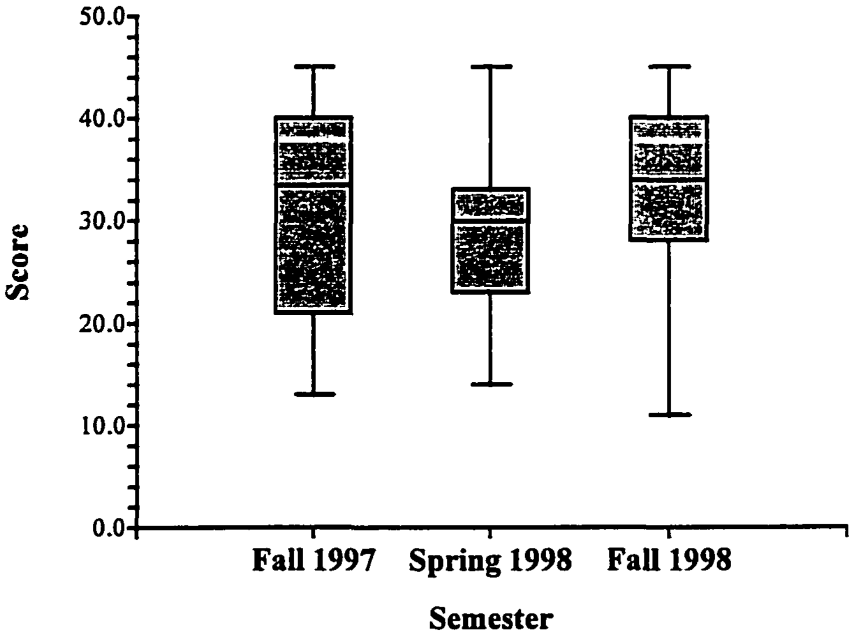
	Control Group	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Question 1	5.13 5	4.56 5	5.14 5
Question 3	5.29 6	4.50 5	5.33 6
Question 5	4.13 4	**3.27 2.5**	4.43 5
Question 9	7.08 8	5.92 6.5	6.24 6
Question 10	7.25 8	6.87 8	8.38 9
Question 11	6.67 6	6.29 6	7.43 8
Total	35.54 38	31.40 32	36.95 39

Five Common Sequential Questions. The overall statistics for the five common sequential final exam questions (with question five removed) are given in the table below.

	Control Group	Treatment Groups	
	Fall 1997	*Spring 1998*	*Faii 1998*
Average	31.42	28.14	32.52
Median	33.5	30	34
Standard Deviation	9.873	7.731	9.548
Variance	97.471	59.765	91.162

In all three groups, the median score is above the average score. The maximum score possible is 47. The Fall 1998 class group, a treatment group, has the highest scores. The Spring 1998 treatment group has the lowest scores. The Fall 1997 control group and the Fall 1998 treatment group have very close average and median scores. The difference between the minimum and maximum average is about 4.5. The difference between the minimum and maximum medians is 4. The fall classes did better than the spring class. The removal of question five from the scores decreased the difference between the minimum and maximum medians from 7 to 4. The box plot for the five sequential question score is given in Figure 5.10 below.

Figure 5.10. Final Exam Five Sequential Question Scores



The figure shows the following about percentile ranks of the final exam five sequential question scores:

- 100th percentile scores are in same (45)
- 75th percentile score for the Spring 1998 treatment group is about the same as the 50th percentile scores (medians) of both the Fall 1997 control group and the Fall 1998 treatment group.
- 25th percentile score for the Spring 1998 treatment group is slightly higher the 25th percentile score for the Fall 1997 control group.
- 0th percentile scores are in the same range (11 to 14)

Again, the top 50 percent of the Spring 1998 treatment group under performed the two other groups. The outlier subject in Spring 1998 treatment group data has been removed. The outlier bound in this figure is set for 1.5 IQR.

The table below shows the results of the normality tests (for all three groups together) and the equal-variance test on the combined groups for the five sequential questions. The omnibus test result indicates that all three groups pass the normality test. The Modified-Levene test indicates that there is sufficiently equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-1.5925	0.111275	Accept
Kurtosis Normality of Residuals	-0.7483	0.454264	Accept
Omnibus Normality of Residuals	3.0960	0.212672	Accept
Modified-Levene Equal-Variance Test	1.5547	0.216688	Accept

The results of the Kruskal-Wallis One-Way ANOVA on Ranks test are given in the table below.

Method	Chi-Square (H)	Probability	Decision (0.05)
Corrected for Ties	5.329297	0.069624	Accept Ho

The "Accept Ho" decision means the following **the three groups (both control and treatment) are not significantly different when tested around their medians.**

The histograms for each group are presented in the figures below. The shape of the Spring 1998 treatment group density trace line is the most unimodal. The shape of the Fall 1997 control group density trace line is the most bimodal.

Figure 5.11. Final Exam Five Sequential Question Scores (Novice Students, Fall 1997 Control Group)

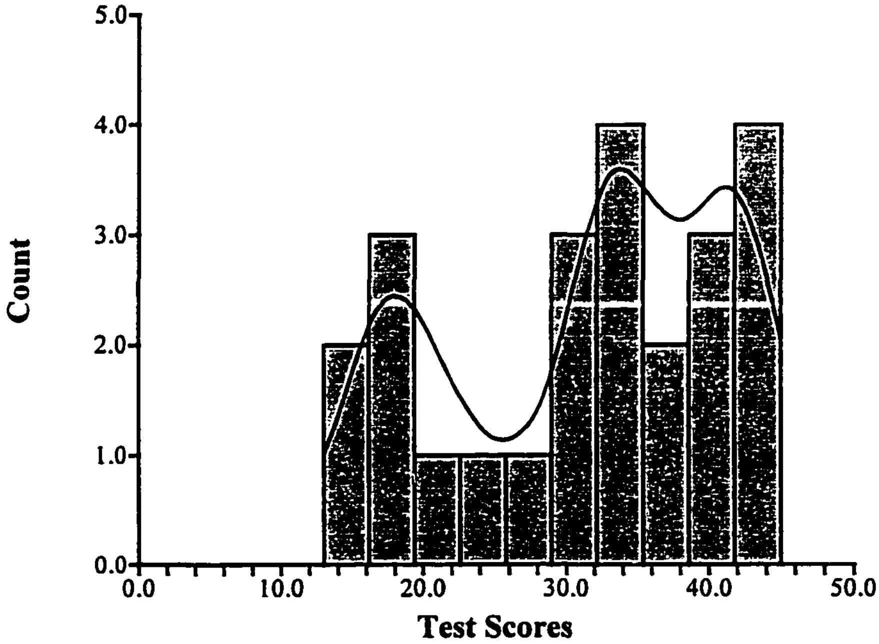


Figure 5.12. Final Exam Five Sequential Question Scores (Novice Students, Spring 1998 Treatment Group)

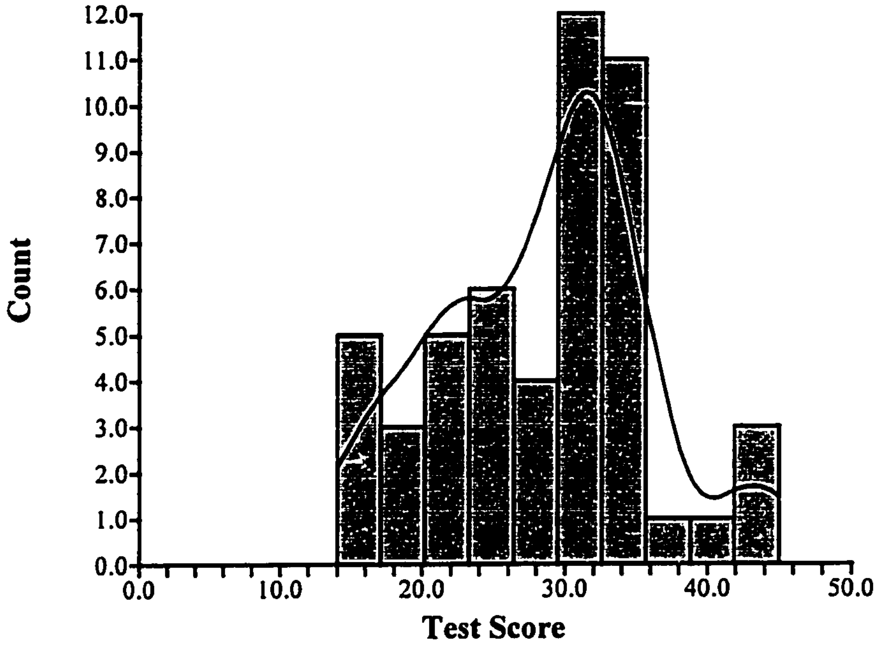
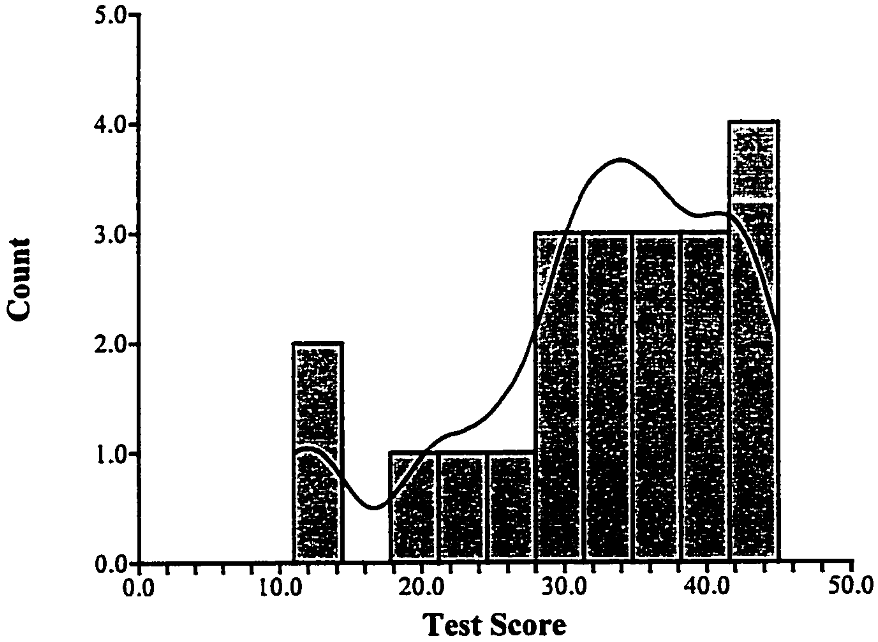


Figure 5.13. Final Exam Five Sequential Question Scores (Novice Students, Fall 1998 Treatment Group)



Conclusion. Significant differences in the final exam sequential question scores (either the six question set or the five question set) were not found among any of the three groups. Therefore, significant differences between the control group and the treatment groups were not found. Although, differences in age and maturity (by college year standing) may have impacted the Spring 1998 treatment group at 16 weeks into the semester; significant differences in the groups performance on sequential questions in the final exam were not found. Subsection 5.7 of this document presents the correlation of maturity to student performance. Subsection 5.8.3 of this document presents different grouping selections that increase the probability results for the six question set (from the 0.06 range to the 0.30 range of probability). Subsection 5.8.3 also presents a comparison of the Fall 1997 control group to the Fall 1998 treatment group (with the probability in the range of 0.69).

5.4 Concurrency Materials And Sequential Materials Performance Comparison

A comparison of the students' ability to solve sequential questions and concurrency questions was done as part of the final exam. Four final exam questions were given to the treatment groups on concurrency material. The students selected the order in which the questions were answered. The students also chose which questions, if any, to skip.

In order for the sequential question scores to be compared to the concurrency question scores, the number of sequential question points must be normalized to the number of concurrency question points (41 points). For the purpose of this document, the term "sequential index" refers to the sequential questions score normalized over the total concurrency questions score (41 points). Scaling the six sequential test questions score on the final exam creates the sequential index. First, a factor is created that equals 41 / 55, where 55 of the 57 total points from the six sequential final exam questions are used. Second, each student's sequential score is scaled by this factor (the value is 0.7454545).

A second sequential index is created for the first concurrency question on the final exam (question 2); another factor is created that equals 10 / 55 (the value is 0.1818182). Again, each student's sequential test questions score is multiplied by the factor to create the sequential index.

The sequential index is a simple tool for comparing between a group's concurrency scores and their sequential scores on the final exam.

Four Concurrency Questions. The overall statistics for the four concurrency final exam questions given to the two treatment groups are in the table below.

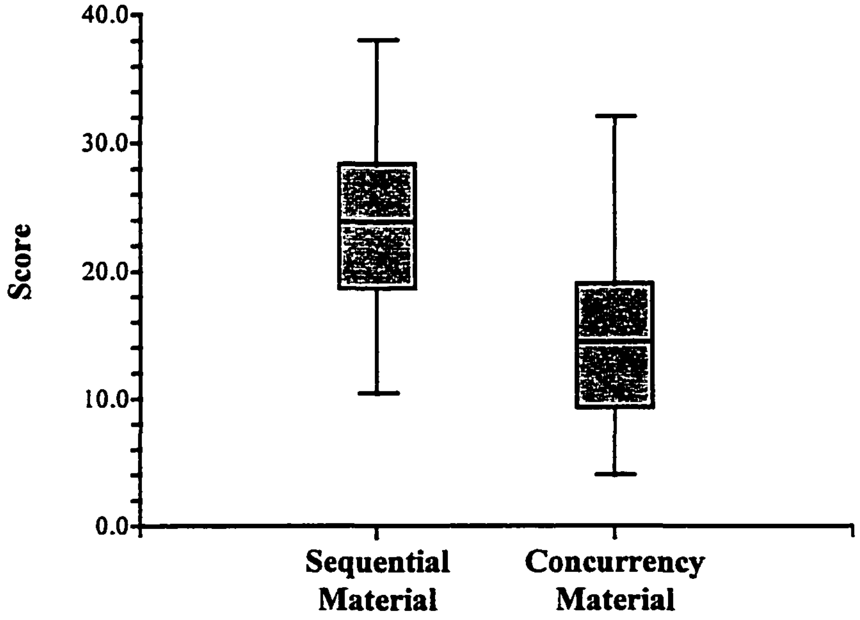
	Treatment Groups			
	Spring 1998		*Fall 1998*	
	Sequential *	Concurrency	Sequential *	Concurrency
Average	23.72	15.02	27.55	17.76
Median	23.9	14.5	29.1	18
Standard Deviation	6.380	7.334	8.558	6.602
Variance	40.707	53.784	73.244	43.590

The "*" indicates the sequential score is a created sequential index. In both treatment groups the students do better on the sequential questions than on the concurrency questions. The Fall 1998 treatment group did better overall than the Spring 1998 treatment group. The difference between the minimum and maximum concurrency average is about 2.75. The difference between the minimum and maximum concurrency median is 3.5.

There is a difference in the instructional materials of the two treatment groups. The Spring 1998 treatment group received their concurrency instructional materials and supplemental sequential materials on a week-by-week basis. The Fall 1998 treatment group received their concurrency instructional materials and supplemental sequential materials at the beginning of week nine of the class. In addition, the Fall 1998 treatment group received some of their concurrency instructional material as a 14 page web-formatted document. All information given to both treatment groups was identical. The differences were in the timing of the presentation of the printed material and the format of the printed material.

The box plot with the comparison of the sequential and concurrency question scores for the Spring 1998 group is given in Figure 5.14 below.

Figure 5.14. Sequential / Concurrency Material Comparison, Spring 1998 Group

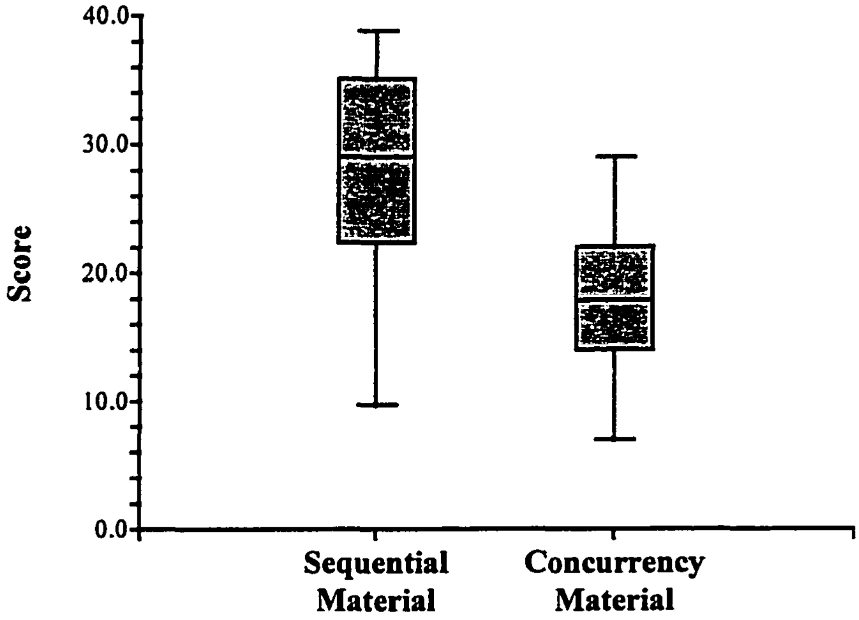


The Figure 5.14 shows the following about the percentile ranks of the different material question scores:

- 100th percentile score for the concurrency material is between the 75th and 100th percentile score for the sequential material
- 75th percentile score for the concurrency material is approximately at the 25th percentile score for the sequential material
- 25th percentile score for the concurrency material is just below the percentile score for the sequential score

The Spring 1998 treatment group under performed in concurrency material as compared to the sequential material. The box plot with the comparison of the sequential and concurrency question scores for the Fall 1998 group is given in Figure 5.15 below.

Figure 5.15. Sequential / Concurrency Material Comparison, Fall 1998 Group



The Figure 5.15 shows the following about the percentile ranks of the different material question scores:

- 100th percentile score for the concurrency material is at the median score for the sequential material
- 75th percentile score for the concurrency material is approximately at the 25th percentile score for the sequential material

The Fall 1998 treatment group under performed in concurrency material as compared to the sequential material.

The table below shows the results of the normality test and the equal-variance test for the Spring 1998 treatment group data. The omnibus test result indicates that the two distributions (sequential index and concurrency scores) pass the normality test.

Assumption (Spring 1998 Data)	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	1.7490	0.080284	Accept
Kurtosis Normality of Residuals	-0.2696	0.787490	Accept
Omnibus Normality of Residuals	3.1318	0.208900	Accept
Modified-Levene Equal-Variance Test	0.3956	0.530794	Accept

The Modified-Levene test indicates that there is more than sufficient equality of variance in the Spring 1998 for a non-parametric test with an equality of variance assumption to be used. The test selected is the Mann-Whitney U test. The results of the Mann-Whitney U test are given in the table below.

Alternative Hypothesis	**Test Value**	**Probability**	**Decision (0.05)**
D(S) <> D(C)	5.4078	0.000000	Reject Ho
D(S) > D(C)	5.4144	1.000000	Accept Ho

Two alternative hypotheses are presented. The "D(S) <> D(C)" hypothesis is the null hypothesis (the probability that there is no significant difference between the two groups). The second hypothesis shown is the difference hypothesis with the highest probability (in this case the sequential distribution, "D(S)", is greater than concurrency distribution, "D(C)" with a probability of 1.00000). Clearly, the sequential index score is greater than the concurrency material scores for the Spring 1998 treatment group.

The table below shows the results of the normality test and the equal-variance test for the Fall 1998 treatment group data. The omnibus test result indicates that the two distributions (sequential index and concurrency scores) pass the normality test.

Assumption (Fall 1998 Data)	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-1.3694	0.170874	Accept
Kurtosis Normality of Residuals	-0.1451	0.884671	Accept
Omnibus Normality of Residuals	1.8963	0.387458	Accept
Modified-Levene Equal-Variance Test	1.1146	0.297413	Accept

The Modified-Levene test indicates that there is more than sufficient equality of variance in the Fall 1998 data for a non-parametric test with an equality of variance assumption to be used. The test selected is the Mann-Whitney U test. The results of the Mann-Whitney U test are given in the table below.

Alternative Hypothesis	**Test Value**	**Probability**	**Decision (0.05)**
D(S) <> D(C)	3.5994	0.000319	Reject Ho
D(S) > D(C)	-3.6245	0.999855	Accept Ho

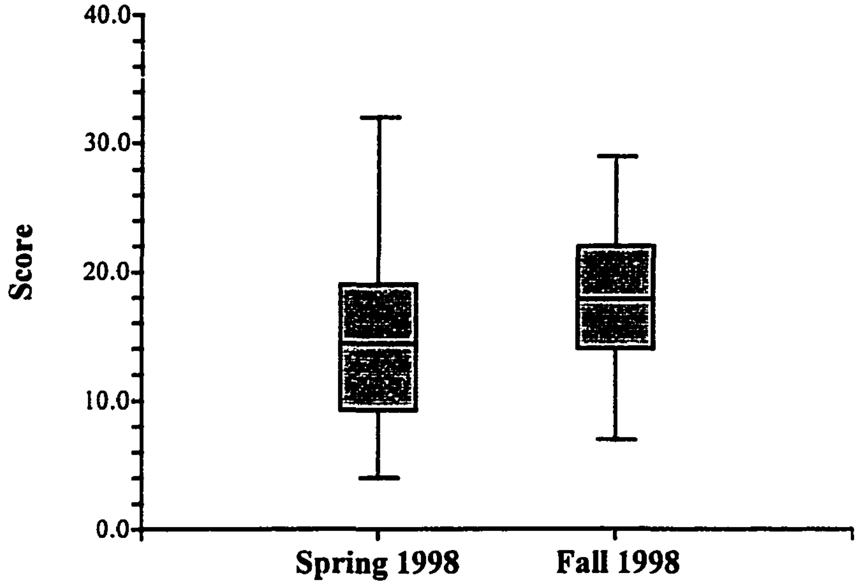
Again, the sequential index score is greater than the concurrency material scores for the Fall 1998 treatment group.

A comparison of the concurrency question scores for the two treatment groups is given below in Figure 5.16. The figure shows the following about the percentile ranks of the two treatment groups on the concurrency material:

- 100th percentile score for the Spring 1998 group is higher
- Each other percentile scores for the Fall 1998 group are higher than the corresponding percentile score for the Spring 1998

The Spring 1998 treatment group under performed in concurrency material as compared to the Fall 1998 treatment group.

Figure 5.16. Comparison of Treatment Concurrency Material Performance



In the following paragraphs the results of each concurrency material question are examined separately. For three of the questions a non-rigorous approach is used. That is the results of these questions are not subjected again to statistical tests. The relationship between sequential and concurrency question scores has already been presented. In the first concurrency question the students in both treatment classes did well. Therefore, a rigorous statistical test approach is used for the next question.

Semaphores (Question 2). The question tested the student on the ability to create and understand semaphores. Students were asked to compose two tiny tasks that act as semaphores. The question was worth 10 points. The statistics for the question given to the two treatment groups are in the table below.

	Treatment Groups			
	Spring 1998		*Fall 1998*	
	Sequential *	Question 2	Sequential *	Question 2
Average	5.79	6.10	6.72	7.29
Median	5.8	6.5	7.1	8
Standard Deviation	1.567	2.475	2.095	2.552
Variance	2.455	6.128	4.391	6.514

The "*" indicates the sequential score is a created sequential index. In both treatment groups the average for the semaphore question is higher than the sequential index average. In both treatment groups the median for the semaphore question is higher than the sequential index median. The box plot with the comparison of the sequential index score and the semaphore question score for the Spring 1998 treatment group is given in Figure 5.17 below.

Figure 5.17. Question 2 -- Semaphores, Spring 1998 Group (Sequential Index To Question 2 Score Comparison)

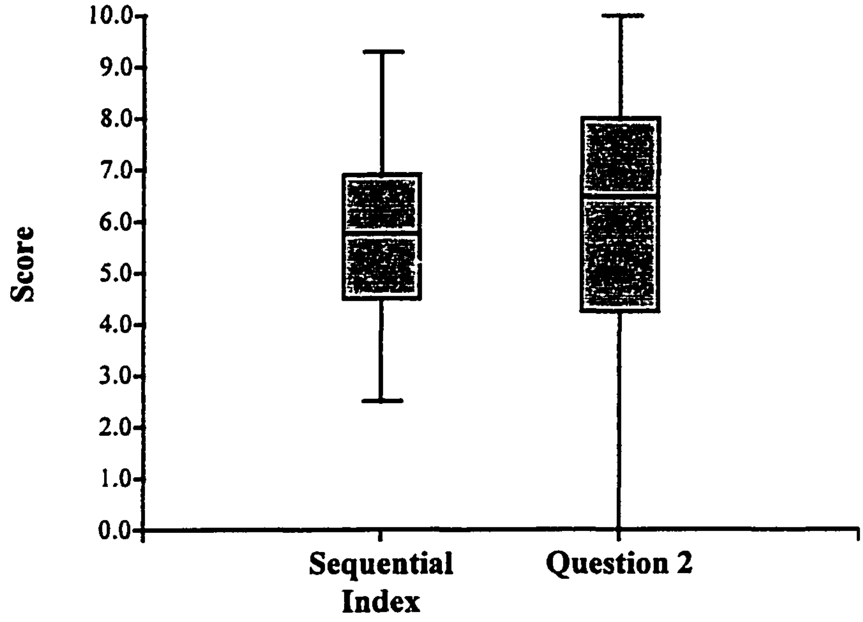
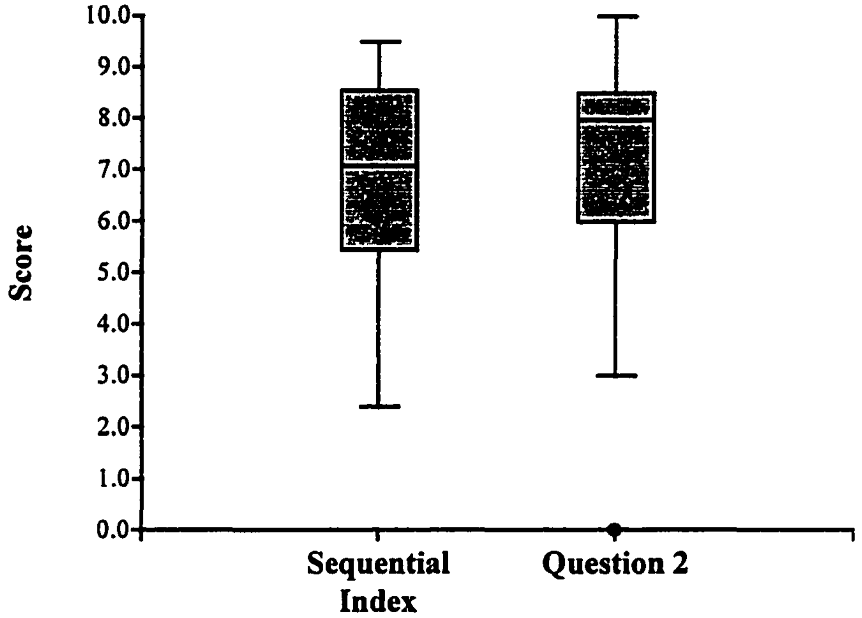


Figure 5.17 shows the following about the percentile ranks of the scores:

- 100th, 75th, and 50th percentile scores for the question are higher than for the sequential index
- 25Th percentile scores are about the same
- 0th percentile score for the question is lower than the index because some student chose to skip this concurrency material question

The Spring 1998 treatment group performed better on this question than in the sequential index. The box plot with the comparison of the sequential index score and the semaphore question score for the Fall 1998 treatment group is given in Figure 5.18 below. Other than the 75 percentile score, the percentile score is higher for Question 2 than the corresponding percentile score in the sequential index.

Figure 5.18. Question 2 – Semaphores, Fall 1998 Group (Sequential Index To Question 2 Score Comparison)



The table below shows the results of the normality test and the equal-variance test for the Spring 1998 treatment group data for the semaphore question. The omnibus test result indicates that the two distributions (sequential index and concurrency scores) pass the normality test.

Assumption (Spring 1998 Data)	Test Value	Probability	Decision (0.05)
Skewness Normality of Residuals	-1.6959	0.089910	Accept
Kurtosis Normality of Residuals	0.9281	0.353366	Accept
Omnibus Normality of Residuals	3.7373	0.154330	Accept
Modified-Levene Equal-Variance Test	8.3977	0.004608	Reject

The Modified-Levene test indicates that sufficient equality of variance does not exist in the Spring 1998 data. The test selected is the non-parametric Kolmogorov-Smirnov test. The results of the Kolmogorov-Smirnov test are given in the table below. The test results indicate that the two distributions are not the same. Therefore, the conclusion is that the students performed better on the semaphore question than on the index.

Hypothesis	Criterion Value	Probability	Decision (0.05)
D(S) <> D(2)	0.264706	0.0391	Reject Ho

The table below shows the results of the normality test and the equal-variance test for the Fall 1998 treatment group data for the semaphore question. The omnibus test result indicates that one of the two distributions (sequential index and concurrency scores) fails the normality test.

Assumption (Fall 1998 Data)	Test Value	Probability	Decision (0.05)
Skewness Normality of Residuals	-2.8677	0.004135	Reject
Kurtosis Normality of Residuals	1.5853	0.112899	Accept
Omnibus Normality of Residuals	10.7369	0.004661	Reject
Modified-Levene Equal-Variance Test	0.0130	0.909831	Accept

The Modified-Levene test indicates that there is more than sufficient equality of variance in the Fall 1998 data for a non-parametric test with an equality of variance assumption to be used. The test selected is the Mann-Whitney U test. The results of the Mann-Whitney U test are given in the table below. The test result shows that there is no significant difference in the two distributions and that the semaphore question distribution is significantly higher than sequential index score.

Alternative Hypothesis	Test Value	Probability	Decision (0.05)
D(S) <> D(2)	1.2919	0.196389	Accept Ho
D(S) < D(2)	1.3172	0.906121	Accept Ho

Reading Code and Determining Output (Question 4). The question tested the students' ability to read code and show the results of the code. The question was worth 12 point. However, the question was sufficiently difficult that it was improbable to get full credit. The question was in three parts:

- Read the code and show the output (6 points)
- Given the code is changed, what is the output (4 points)
- A variable is outside the task, state what the variable is called (2 points)

The score break out for the question is given in the table. A score of 4 and up indicates either (1) the student has recognized a shared memory variable and can predict some of the output or (2) can show most of the output. A score of 7 and up indicates the ability to read code and show the output.

Score	Spring 1998	Fall 1998
Average	3.27	3.76
Median	2.5	4
7 and up	6	3
4, 5, and 6	17	8
3 or below	19	8
Zero score	10	2
7 and up percentage	14.39	15.79
4 and up percentage	54.76	57.89

The histogram of the Spring 1998 treatment group scores on the reading code question is given in Figure 5.19 below. The histogram of the Fall 1998 treatment group scores on the question is given in Figure 5.20 below.

Figure 5.19. Histogram of Reading Code Question Scores For Spring 1998 Treatment Group

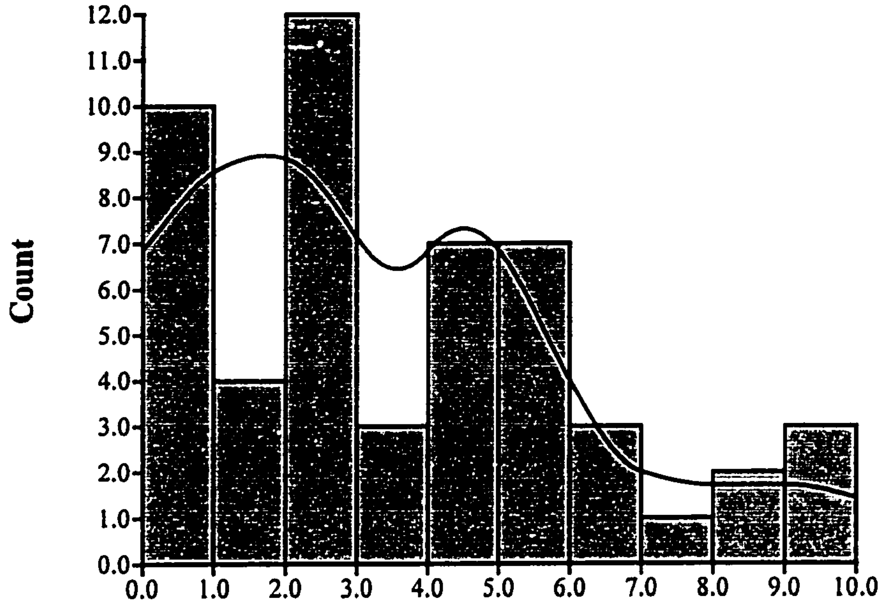
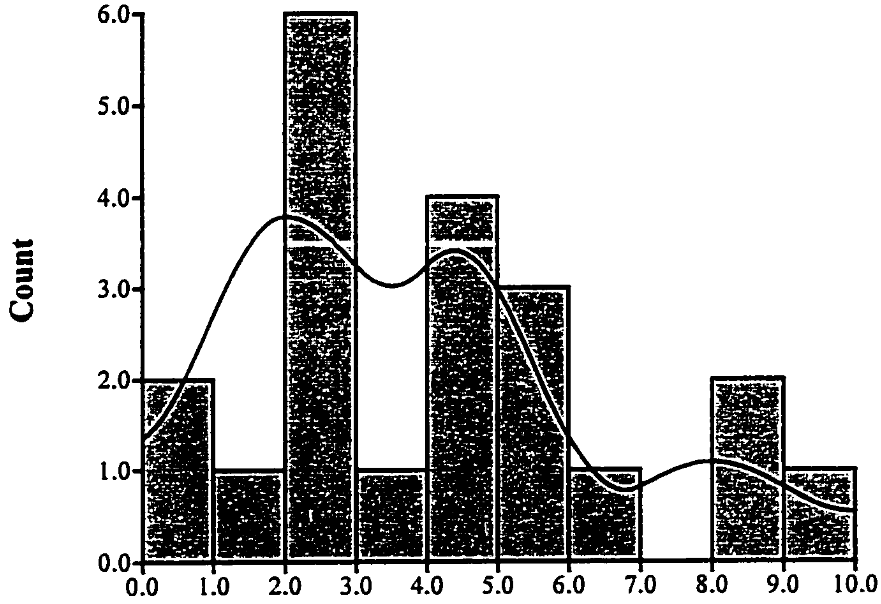


Figure 5.20. Histogram of Read Code Question Scores For Fall 1998 Treatment Group



Message Passing (Question 7). The question tests the student on the ability to compose two tiny tasks -- one task to send a message and one task to receive a message. The question is worth 10 points. A score of 4 points or less indicates that the student had insufficient skills to write the syntax. Scores above 5 indicate at least some comprehension of solving the question. The score break out for the question is given in the table. The "5 and up percentage" shows the percentage of students attempting the problem (non-zero score) that attained a score of 5 and above.

Score	**Spring 1998**	**Fall 1998**
Average	2.98	2.52
Median	3	2
7 and up	5	1
5 and 6	4	2
4 or below	34	16
Zero score	9	2
5 and up percentage	20.93	15.79

The histogram of the Spring 1998 treatment group scores on the message passing question is given in Figure 5.21 below. The histogram of the Fall 1998 treatment group scores is given in Figure 5.22 below.

Figure 5.21. Histogram of Message Passing Question Scores For Spring 1998 Treatment Group

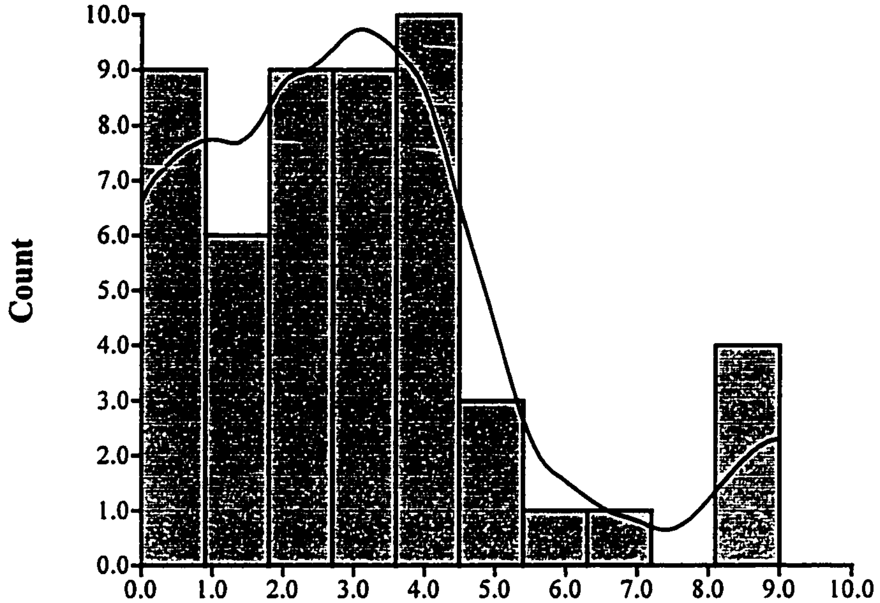
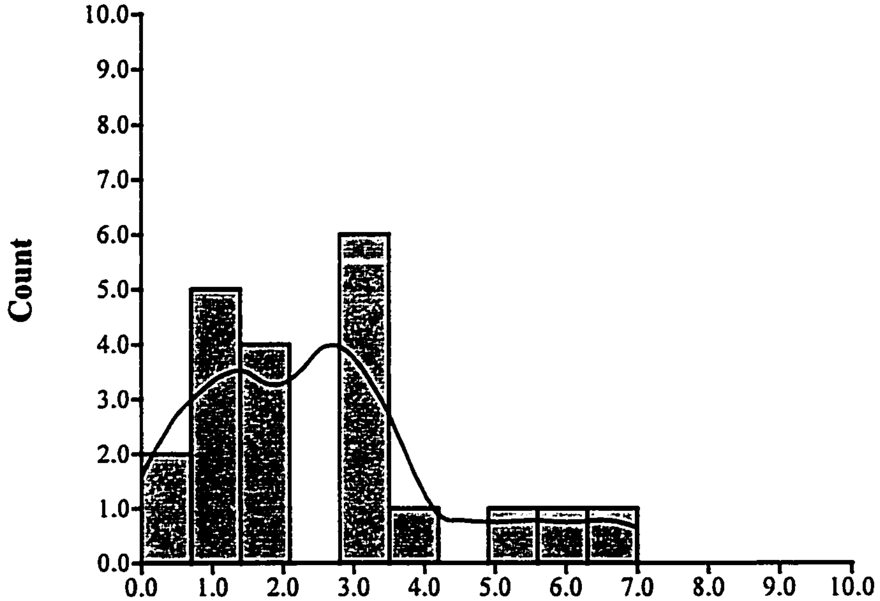


Figure 5.22. Histogram of Message Passing Question Scores For Fall 1998 Treatment Group



Shared Memory (Question 8). The question tests the student on the ability to recall information about shared memory (4 points) and the ability to compose one tiny task using shared memory (5 points). The question was worth 9 points. A score of 3 and 4 points indicated that a student could discuss the subject. A score of 5 and 6 indicates additional knowledge of coding syntax. A score of 7 and up indicates subject knowledge. The score break out for the question is given in the table below. The discussion percentage shows the percentage of those students attempting the question (non-zero score) who can discuss shared memory. The "7 and up percentage" show the percentage of students attempting the question that attained a score of 7 and up.

Score	Spring 1998	Fall 1998
Average	2.67	4.19
Median	2.5	4
7 and up	2	5
5 and 6	8	5
3 and 4	16	5
1 and 2	13	2
Zero score	13	4
Discussion percentage (3 and up)	66.67	88.24
7 and up percentage	5.13	29.41

The histogram of the Spring 1998 treatment group scores on the shared memory question is given in Figure 5.23 below. The histogram of the Fall 1998 treatment group scores on the question is given in Figure 5.24 below.

Conclusion. Among the same students, significant differences were found between final exam scores on sequential material and final exam scores on concurrency material. Students do significantly better on the sequential material. Students are learning concurrent material (some material was learned very well); however, the concurrency test scores are lower.

Figure 5.23. Histogram of Shared Memory Question Scores For Spring 1998 Treatment Group

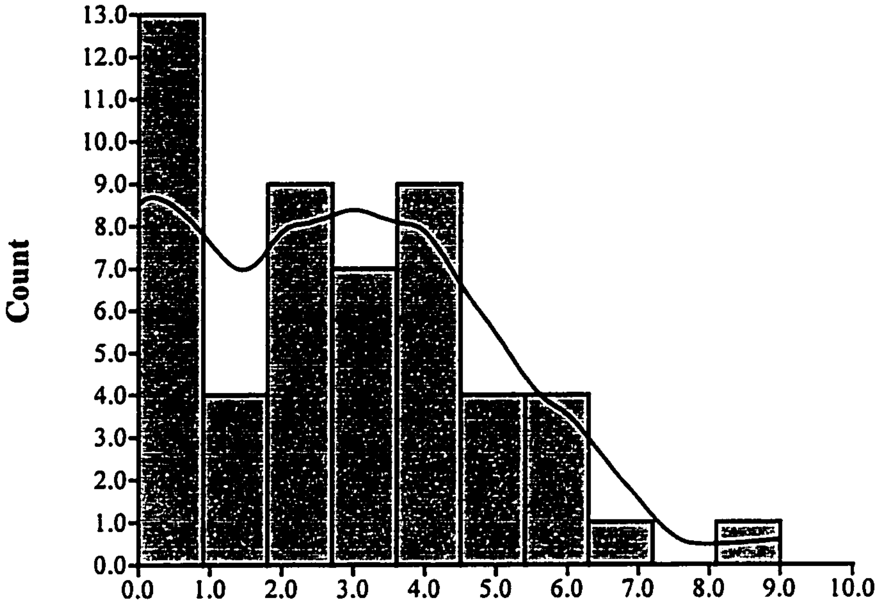
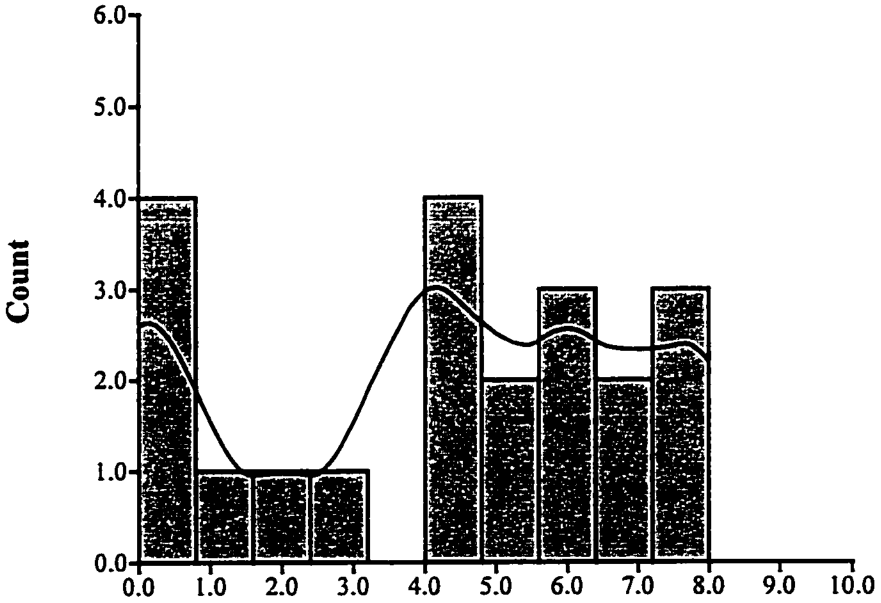


Figure 5.24. Histogram of Shared Memory Question Scores For Fall 1998 Treatment Group



5.5 Comparison of Student Performance on a Large Project

The large project in the Introduction To Computing course was the last project, Project 9. In this project the students were asked to create a small database and to provide a rudimentary set of capabilities. All groups were given a crude character based menu interface to modify. The control group class did the project using all sequential methodology. The treatment group classes did the project using concurrent methodology including an independent database writer, rollback and commit of data, plus semaphores to control shared memory access. The students were given three weeks to complete the project, plus one week extra time, if necessary (with a penalty assessed).

The overall statistics for the Project 9 scores are given below in the table. These Project 9 scores include zero grades. The median is above the average in all three groups.

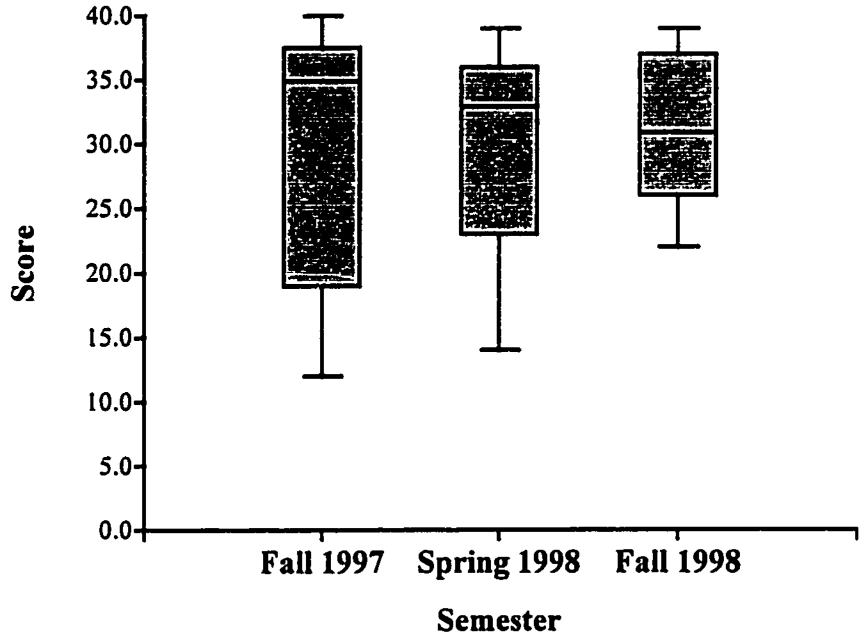
Zero Scores Included	**Control Group**	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	25.83	29.02	30.76
Median	32	32.5	31
Standard Deviation	13.519	8.519	6.024
Variance	182.754	72.568	36.290
Zero Scores	3	1	0

In the Fall 1997 group and in the Spring 1998 group, the averages are lower in part due to zero scores (students did not turn in the project). The overall statistics for the Project 9 scores with zero scores excluded are given below in the next table.

Non-Zero Statistics	**Control Group**	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	29.52	29.59	30.76
Median	35	33	31
Standard Deviation	9.786	7.540	6.024
Variance	95.762	56.847	36.290

Again, the median is above the average in all three groups. The average scores are much closer together than the median scores; the difference between the minimum and maximum average score is under 1.3. The highest median is in the control group. The variance is decreasing with each succeeding class. The box plot of the Project 9 scores is given in Figure 5.25.

Figure 5.25. Project Nine Scores



The box plot graphically shows the decrease in variance with each succeeding class. The figure shows the following about the percentile ranks of the Project 9 scores:

- 100th percentile scores are in the same range (39 to 40)
- 75th percentile scores are in the same range (36 to 37)
- Median scores decrease over a 4 point range
- 25th percentile scores increase with each succeeding class
- 0th percentile scores increase with each succeeding class

The greatest change in scores is in the 25th and 0th percentiles. The scores of the lesser performing students increase with each succeeding class.

The table below shows the results of the normality tests (for all three groups together) and the equal-variance test on the combined groups. The omnibus test result indicates that two or more of the three groups are not normal. The Modified-Levene test indicates that there is sufficient equality uf variance for a non-parametric test with an equality of variance assumption to be used (even with the largest variance 2.6 times thc smallest variance).

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-2.4218	0.015445	Reject
Kurtosis Normality of Residuals	-2.5610	0.010436	Reject
Omnibus Normality of Residuals	12.4239	0.002005	Reject
Modified-Levene Equal-Variance Test	0.9819	0.378593	Accept

The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks. The test results are given in the following table.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Corrected for Ties	0.4230784	0.809338	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.**

The histograms of the Project 9 scores are given in the next three figures: Figure 5.26 for the Fall 1997 class, Figure 5.27 for the Spring 1998 class, and Figure 5.28 for the Fall 1998 class. Zero scores are included in the histograms.

Figure 5.26. Histogram of Project Nine Scores, Fall 1997 Control Group

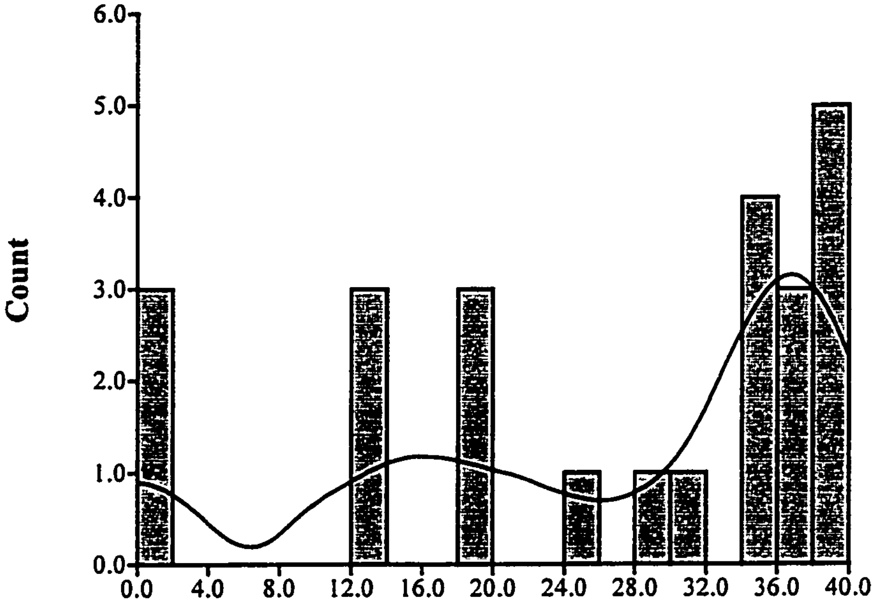


Figure 5.27. Histogram of Project Nine Scores, Spring 1998 Treatment Group

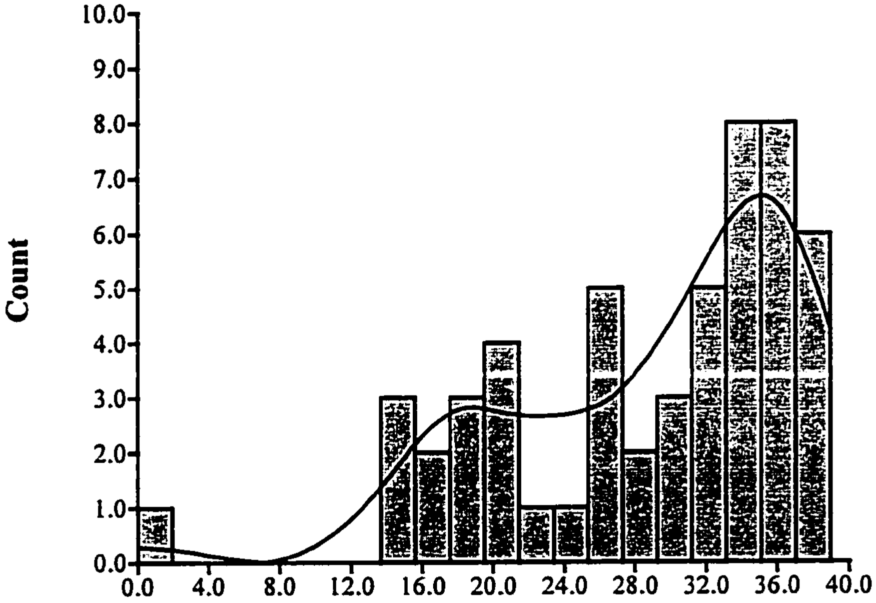
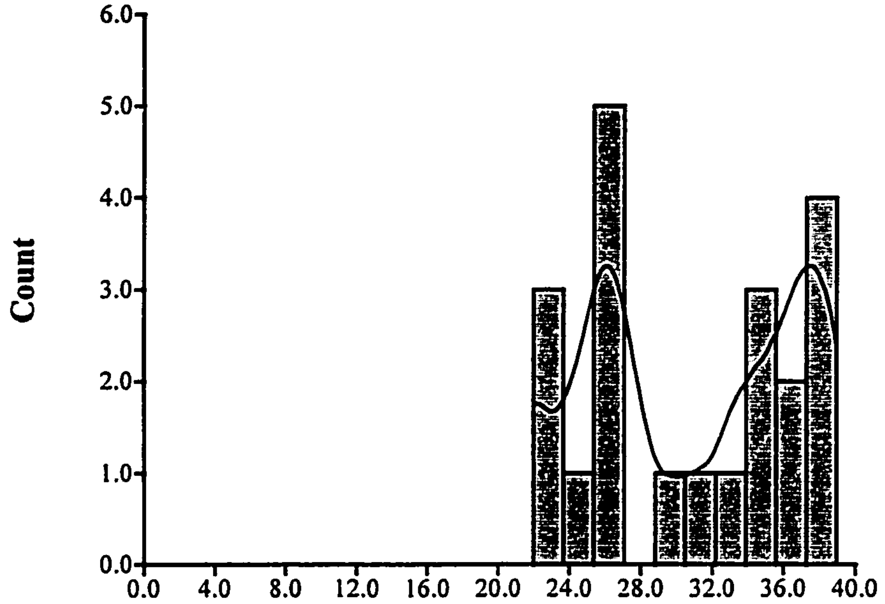


Figure 5.28. Histogram of Project Nine Scores, Fall 1998 Treatment Group



The Fall 1997 group and the Fall 1998 group distributions appear to be bimodal.

Conclusion. Significant differences in the large project scores were not found (using either sequential or concurrency methodologies). Significant differences between the groups were not found. The treatment groups had a smaller variance of scores than the control group. The smaller variance resulted from lowerr performing students earning higher project scores.

5.6 Findings for the Compilation of Concurrency and Sequential Programs

Compilations. All program compilations were recorded. This included both sequential and concurrency programs. Program compilations had various outcomes; these outcomes included the following:

1. Typographical error(s) in the compilation command line -- the program to be compiled is not found
2. Program compiles with errors -- the program contains one or syntax errors and warning messages
3. Program compiles with out error -- the program contains zero syntactic errors and warnings detected by the compiler (a clean compile)

Although type 1 compilations are recorded, these compiles were not included in the statistics. The compiler did not access the program for the type 1 compilations. Students often compile programs during a project that are unrelated to the project, except that the student is searching for some information in the compiled program. These compilations are not included in the statistics presented in this document.

One of the measures of the work required to perform a project was the number of compilations. Different students had different methods of solving compilation errors and warnings. One type of student behavior was to solve one or two compilation errors, and then recompile the program. Another type of student behavior was to solve all or most compilation errors, and then recompile the program. Thus, compilation counts alone were not a good measure of work. However, the ratio of the number of compilations between two projects would be a more consistent measure of work. For example, project five required twice the compilations of project three is a measure of relative work.

As previously stated in Subsection 5.2.2, there were five projects given before concurrency was introduced. The first project was very simple, correct some syntax errors and change an existing program. The first project did not include original programming and was not included in the experiment. Projects two, three, four, and five were included in the experiment. The large project was the last project, Project 9. The measure of relative work studied (by compilations) was as follows:

$$\text{Ratio} = \frac{\text{Compilations for project nine (P9)}}{\text{Compilations for projects two through five (P2-5)}}, \text{ per student}$$

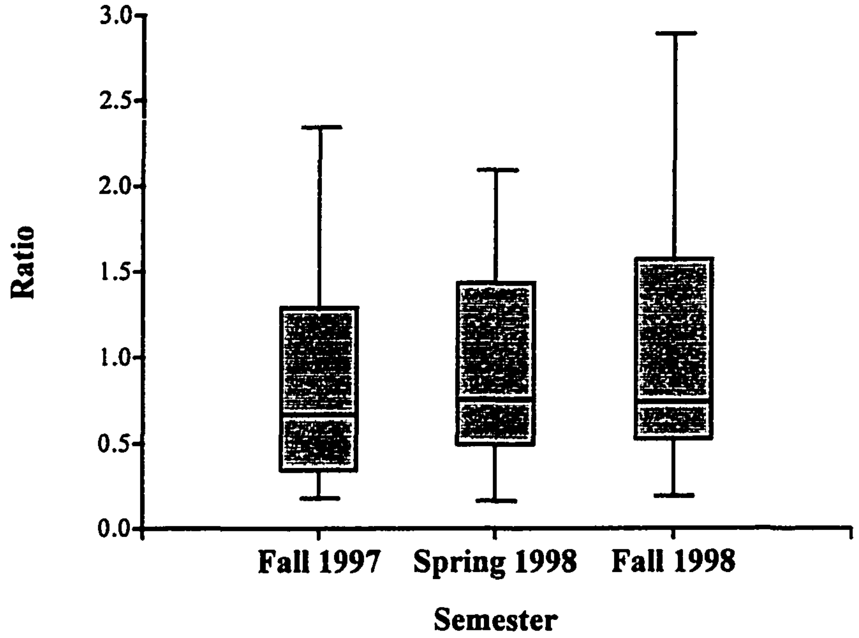
Projects two through five were performed over a five week period. The period was longer if project five was completed late. Project 9 was performed over a three week period. Thus, the "ratio" studied was derived from eight weeks of student compilation activity.

The overall statistics for the compilation "ratio", (P9 / P2-5) are given below in the table. Students had to perform these five projects to be included in this comparison.

	Control Group	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	0.84	0.99	1.07
Median	0.68	0.76	0.75
Standard Deviation	0.603	0.540	0.807
Variance	0.363	0.291	0.651
Count	18	43	18

The average ratio is above the median for all groups. The averages for the treatment groups are above the average for the control group. The medians for the treatment groups are above the median for the control group. The box plot for the compilation "ratio" is given in Figure 5.29. The box plot graphically shows an increase in the compilation "ratio" for the treatment groups in the 75th, 50th, and 25th percentile ratios over the control group. The 0th percentile compilation "ratios" for both the control and treatment groups are about the same.

Figure 5.29. Compilation Ratio for Projects (P9 / P2-5)



The table below shows the results of the normality tests (for all three groups together) and the equal-variance test on the combined groups. The omnibus test result indicates that one or more of the three groups are not normal. The Modified-Levene test indicates that there is more than sufficient equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	3.2950	0.000984	Reject
Kurtosis Normality of Residuals	0.7032	0.481923	Accept
Omnibus Normality of Residuals	11.3513	0.003428	Reject
Modified-Levene Equal-Variance Test	0.3475	0.707582	Accept

The results of the Kruskal-Wallis One-Way ANOVA on Ranks test are given below.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Not Corrected for Ties	1.593634	0.450761	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.** Duplicate values do not exist within any single group; thus, the statistic is not corrected for ties. Significant differences in the compilation "ratio" are not found, even though the treatment group students used concurrency methodology in Project 9.

Compilation Errors. Compressing all the variations in error messages into a smaller number of distinct error messages is not an exact science. The method used to reduce the total variation of error and warning messages is as follows:

- Replace variable names with a representative symbol (such as "token")
- Replace numbers with a representative symbol (such as the value "99")

Thus, the error message "*"class_sum" is undefined*" is translated to "*"token" is undefined*". The error message "*"=" should be ":="*" remains unchanged.

Replacing only variable names and numbers with representative symbols results in some error messages being classed as distinct even though there is very little variation in the error message. For example, there are 16 distinct variations of the error message "*invalid operand types for operator*". Each operator is classed as a distinct error message. Examples include the following:

invalid operand types for operator "<="

invalid operand types for operator "="

invalid operand types for operator ">="

Another example, there are numerous distinct variations of the error message "*incorrect spelling of keyword*". There is one variation of the message for each keyword. Examples include the following:

incorrect spelling of keyword "FUNCTION"

incorrect spelling of keyword "PACKAGE"

incorrect spelling of keyword "PROCEDURE"

In these examples distinct error messages are clearly not unique.

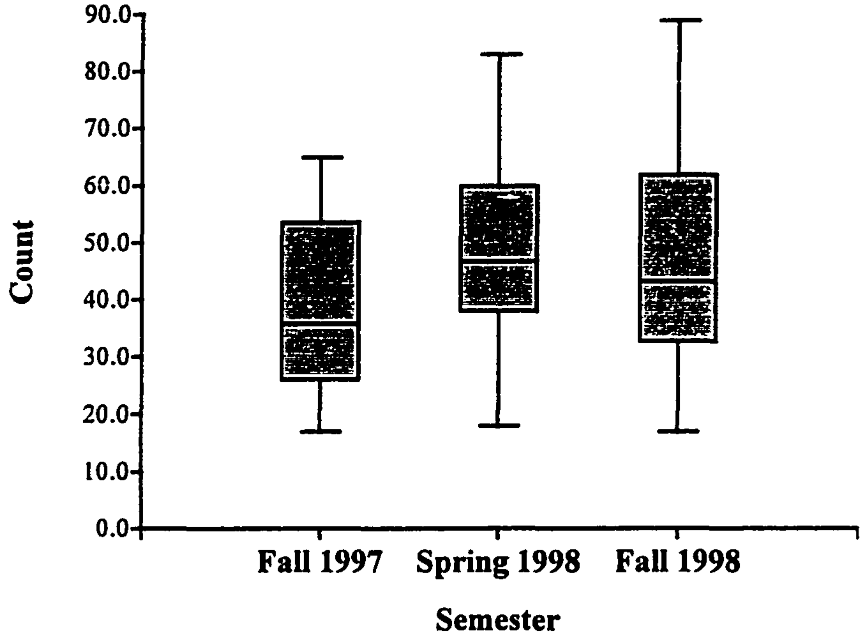
Further, the software that translates the variable names and numbers is not perfect. For example, the error message "*functions can only have "IN" parameters*" should not have been translated to "*functions can only have "token" parameters*". However, for the purpose of creating distinct errors and tracking their occurrences, this is not a problem. Further, when in doubt, error messages that could be compressed into a single message are not. For example, the error message of the type "*incorrect use of "Cars'"*" can apply to more than just packages in Ada. The seven variations of this error message are not compressed into a single distinct error.

Distinct Error Messages. The statistics for the number of distinct error messages in Project 9 for all three groups is given below in the table.

	Control Group	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	39.25	48.74	46.89
Median	36	47	43.5
Standard Deviation	15.256	15.471	20.370
Variance	232.733	239.338	414.928
Count	16	43	18

The average and median for the control group are lower than the average and median for the two treatment groups. The box plot for the distinct error messages is given below in Figure 5.30. The increase in the average and median for the treatment groups is the addition of distinct error messages dealing solely with concurrent syntax constructs in Ada.

Figure 5.30. Distinct Error Messages



On average, the treatment groups saw seven to nine more syntax errors and warnings due to using concurrency. For the median, the treatment groups saw seven to 11 more syntax errors and warnings due to using concurrency. The box plot graphically shows an increase in the distinct error messages in the treatment groups in the 100th, 75th, 50th, and 25th percentile ratios over the control group. The 0th percentile for both the control and treatment groups are about the same.

The table below shows the results of the normality tests (for all three groups together) and the equal-variance test on the combined groups. The omnibus test result indicates that all three groups may be normal. The Modified-Levene test indicates that there is more than sufficient equality of variance for a non-parametric test with an equality of variance assumption to be used.

Assumption	Test Value	Probability	Decision (0.05)
Skewness Normality of Residuals	0.9225	0.356256	Accept
Kurtosis Normality of Residuals	-0.8294	0.406876	Accept
Omnibus Normality of Residuals	1.5390	0.463254	Accept
Modified-Levene Equal-Variance Test	0.6643	0.517662	Accept

The non-parametric statistical test selected is, again, the Kruskal-Wallis One-Way ANOVA on Ranks. The test results are given in the following table.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Corrected for Ties	3.741726	0.153991	Accept Ho

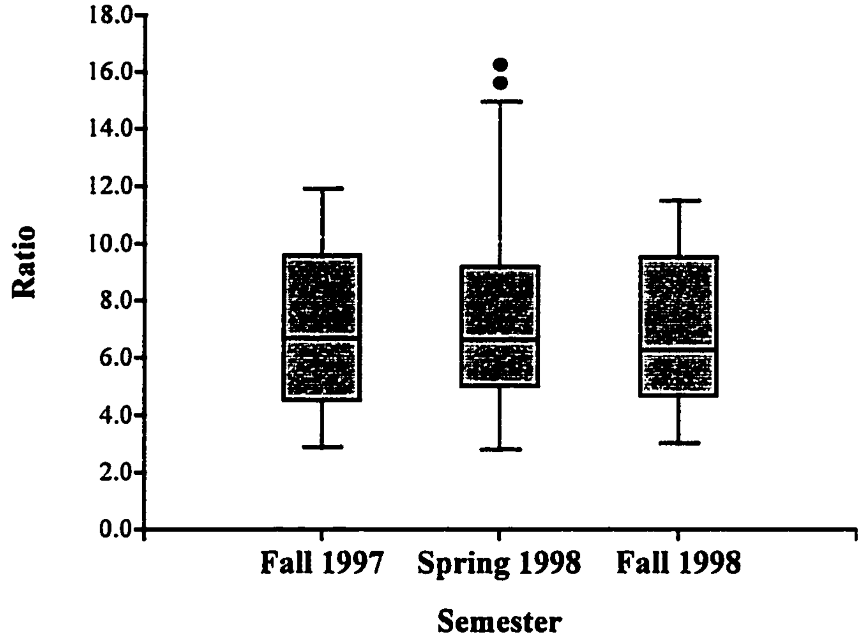
The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.** Even with the known increase in distinct error messages due to the use of concurrency in the treatment groups, significant differences in the control and treatment groups are not found.

Error Ratio (Total Errors / Distinct Errors). Another way to examine compilation errors is to compare the ratio of total errors to distinct errors. This ratio is measure of the number of times the "average" error message occurs. The statistics for the ratio of total errors to distinct errors in Project 9 for all three groups is given below in the table. The statistics for the Fall 1998 treatment group are skewed by a single outlier.

	Control Group	**Treatment Groups**	
	Fall 1997	*Spring 1998*	*Fall 1998*
Average	7.02	7.53	9.18
Median	6.74	6.69	7.36
Standard Deviation	2.754	3.377	6.683
Variance	7.584	11.404	44.664
Count	16	43	18

The averages for the control group and the Spring 1998 treatment group are within 0.5. The medians for the control group and the Spring 1998 treatment group are very close. The box plot for the three groups is given in Figure 5.31. One outlier in the Fall 1998 treatment group is *off* the scale at 28. Both treatment groups have outliers.

Figure 5.31. Error Ratio (Total Errors / Distinct Errors)



The 75th, 50th, 25th, and 0th percentiles are similar in all three groups (both control and treatment). Differences in the three groups appear in the 100th percentile error ratio.

The table below shows the results of the normality tests (for all three groups together) and the equal-variance test on the error ratio for the combined groups. The omnibus test result indicates that two or more groups are not normal. The Modified-Levene test indicates that it is doubtful that sufficient equality of variance exists. However, a severe outlier exists in the Fall 1998 treatment group, and two outliers exist in the Spring 1998 treatment group. Outliers lower Modified-Levene test values -- ignoring outliers means that a higher equality of variance is present. This can be seen in the box plot; the 75th, 50th, 25th, and 0th percentiles are similar in all three groups. Therefore, sufficient equality of variance is assumed, and a non-parametric test with an equality of variance assumption to be used.

Assumption	Test Value	Probability	Decision (0.05)
Skewness Normality of Residuals	5.2824	0.000000	Reject
Kurtosis Normality of Residuais	4.1335	0.000036	Reject
Omnibus Normality of Residuals	44.9899	0.000000	Reject
Modified-Levene Equal-Variance Test	2.2280	0.114925	Accept

The non-parametric statistical test selected is the Kruskal-Wallis One-Way ANOVA on Ranks. The test results are given in the following table.

Method	Chi-Square (H)	Probability	Decision (0.05)
Not Corrected for Ties	0.3133233	0.854993	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.** The very high probability is significant, it implies that compilation error resolution is not affected by concurrency. The ratio of total errors to distinct errors appears constant among all three distributions at and below the 75th percentile.

The error ratio distribution histograms for each group are given in Figures 5.32 through Figure 5.34. The Fall 1998 treatment group has the flattest distribution. The Spring 1998 treatment has the distribution most resembling a normal distribution. With the exception of the outliers (between 14.0 and 16.0) in the Spring 1998 treatment group, all three groups have very similar lower bounds (around 3.0), modes (around 6.0) and upper bounds (between 11.0 and 13.0).

Figure 5.32. Histogram of Error Ratio (Total Errors / Distinct Errors) Fall 1997 Control Group

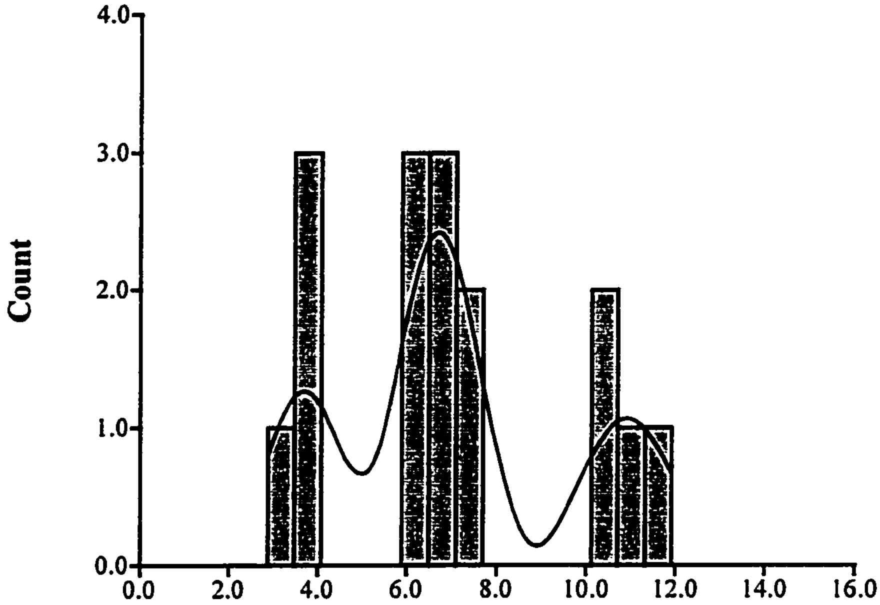


Figure 5.33. Histogram of Error Ratio (Total Errors / Distinct Errors) Spring 1998 Treatment Group

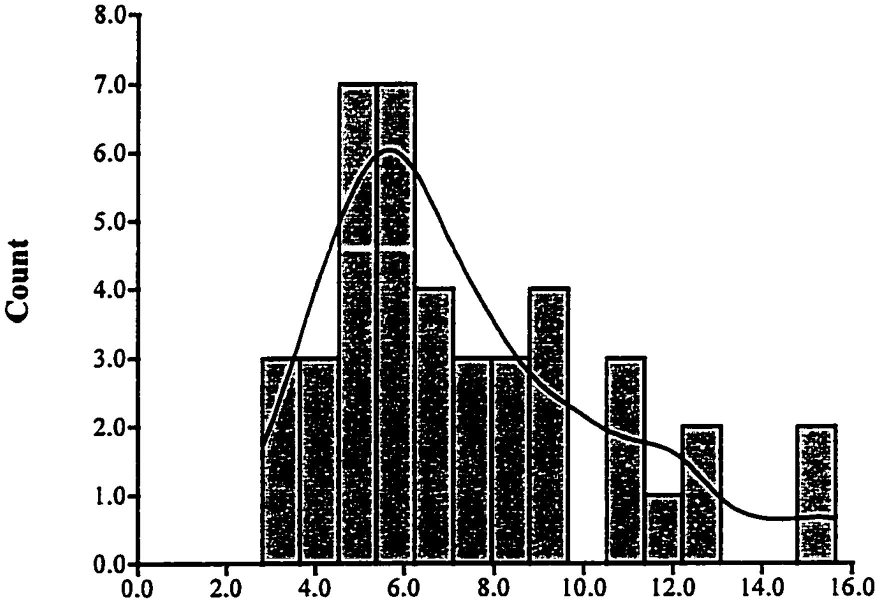
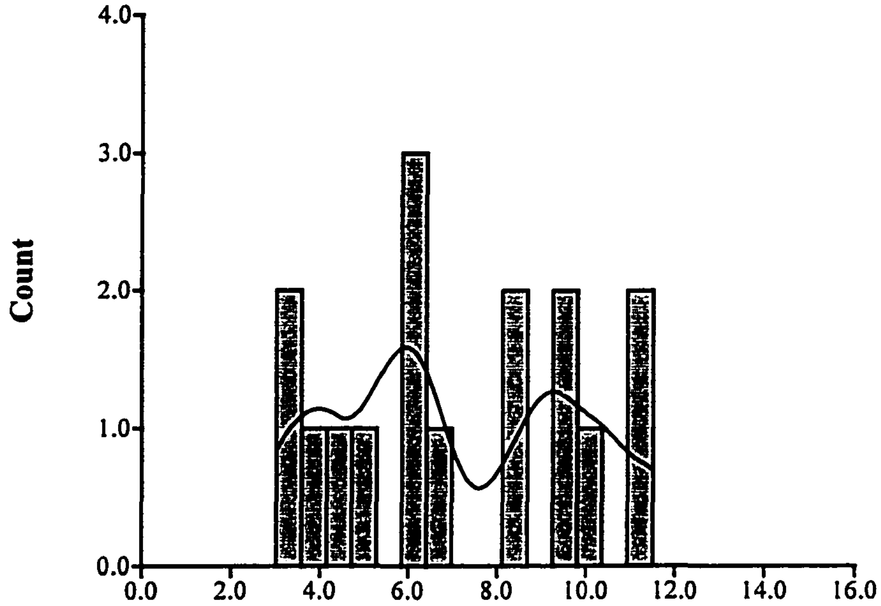


Figure 5.34. Histogram of Error Ratio (Total Errors / Distinct Errors) Fall 1998 Treatment Group



Conclusions. Significant differences in the compilation "ratio" were not found, even though the treatment group students used concurrency methodology in Project 9. Therefore, significant differences in comparing the relative amount of work for the sequential and concurrent version of the large project were not found.

Using concurrency increased the number of distinct error messages. However, the ratio of total errors to distinct errors did not change significantly with the introduction of concurrency compilation errors and warnings. The ratio of total errors to distinct errors appears constant among all three distributions at and below the 75th percentile.

5.7 Correlation of Student Characteristics to Student Performance

The correlations between several aspects of the experiment are presented in the table below. The sequential material in the correlation table is the six sequential questions given on the final exam to all three groups. The concurrency material in the correlation table is the four concurrency questions given to the two treatment groups. The "*" means concurrency material only. The concurrency material column contains the correlation for the treatment groups only. The control group did not receive concurrency material. The "Project 9" column contains mixed comparisons -- some include all three groups (like age) and concurrent material includes the two treatment groups.

	Male Female	**Age**	**Year**	**Sequent. Material**	**Concur. Material ***	**Project 9**
Male / Female	1.000000	0.125079	0.039078	0.030688	-0.106056	-0.094576
Age		1.000000	0.539450	-0.000125	0.093666	0.156081
Year			1.000000	0.119484	0.153045	0.104913
Sequential Material				1.000000	0.682322	0.362781
Concurrent Material					1.000000	0.396738 *
Project 9						1.000000 *

The male / female, age, year, and sequential material correlations contain data from all three groups (both control and treatment).

The correlations are computed using the Spearman's rho (row-wise deletion) technique. The Spearman's rho correlation coefficient measures the monotonic association between two variables in terms of ranks. The Spearman's rho is a non-parametric technique. The technique works well with data that has the following characteristics:

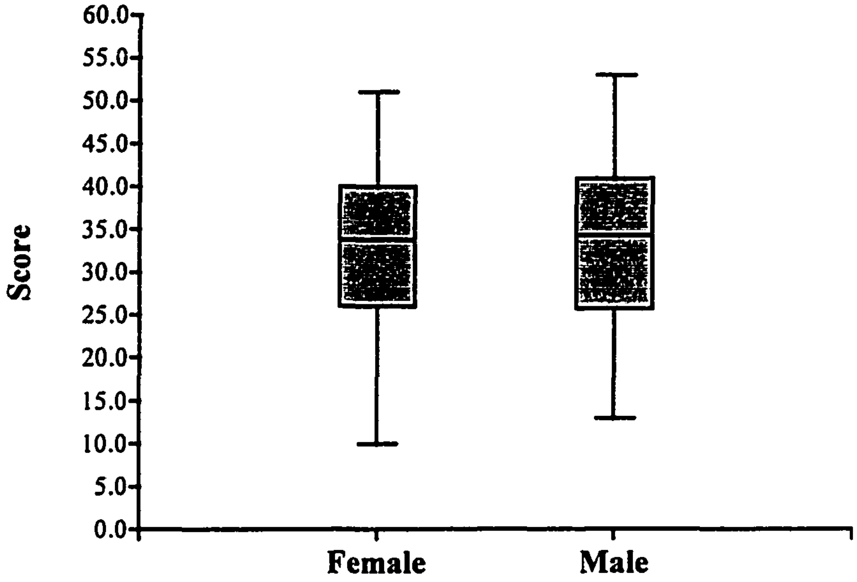
- Non-normal
- Non-linear relationships
- Unequal variance between groups compared

Male / Female Comparison. The negative signs in the male / female row indicate that female students did better than male students. This is true for the concurrency material (for the treatment groups). It is also true for project nine (for all three groups). These correlations are low. The final exam sequential material scores for females and males are given in the table below. The sequential material averages for both sexes are very close. The medians for both sexes are very close. The standard deviations for the two groups are also very close. There are almost twice as many males as females in the comparison.

Sequential Material	Female (0)	Male (1)
Average	33.29	33.82
Median	34	34.5
Standard Deviation	10.363	10.564
Variance	107.387	111.591
Count	35	62

The box plot in Figure 5.35 shows the final exam sequential material scores. The 100th and 0th percentile scores for females are a few points less. Differences in the two sexes performance on final exam sequential material are small.

Figure 5.35. Final Exam Questions – Sequential Material, Male / Female Comparison

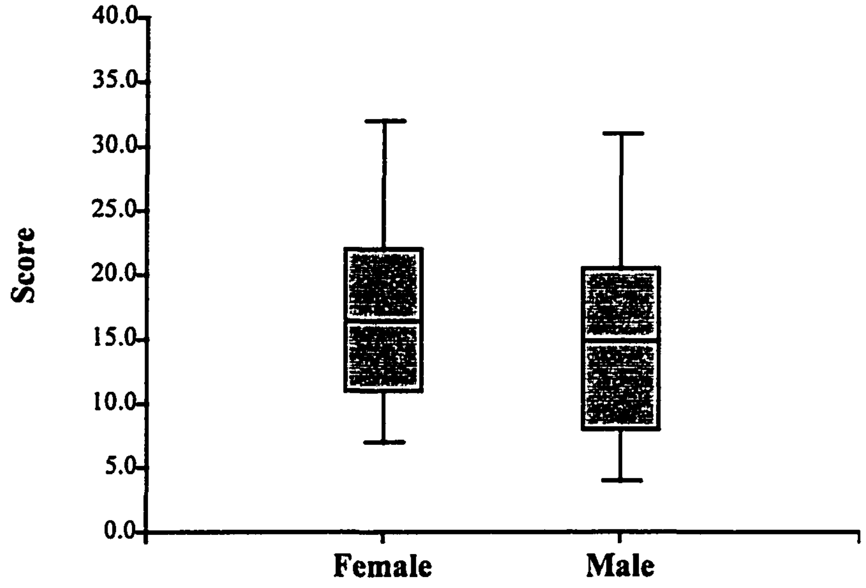


The final exam concurrency material scores for females and males are given in the table below. The female average is about 1.5 points higher than the male average. The female median is 1.5 points higher than the male median. The standard deviations for the two groups are also close (within 0.4 out of 7.3). Again, there are twice as many males as females in the comparison.

Concurrency Material	Female (0)	Male (1)
Average	16.88	15.29
Median	16.5	15
Standard Deviation	6.943	7.329
Variance	48.201	53.708
Count	24	49

The box plot in Figure 5.36 shows the final exam concurrency material scores for females and males. Females have higher percentile rank scores (100th, 75th, and so forth) than the males. The maximum differences are in the lower percentile rank scores.

Figure 5.36. Final Exam Questions -- Concurrency Material, Male / Female Comparison

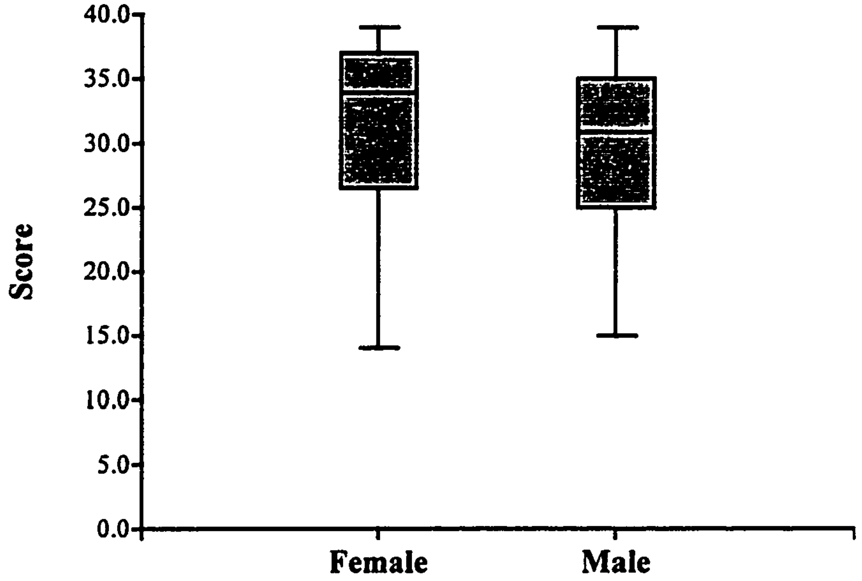


The non-zero project nine scores for the two treatment groups (concurrency material) are given in the table below. The female average is about 1.6 higher than the male average. The female median is 3.0 higher than the male median. Again, there are twice as many males as females in the comparison.

Concurrency Project Nine	Female (0)	Male (1)
Average	31.04	29.38
Median	34	31
Standard Deviation	7.544	6.896
Variance	56.911	47.559
Count	24	48

The box plot in Figure 5.37 is the project nine scores for the two treatment groups. The female scores are higher for the 75th, 50th, and 25th percentiles than the male scores. The female median is almost the 75th percentile score for the males.

Figure 5.37. Concurrency Material Project Nine Scores, Male / Female Comparison

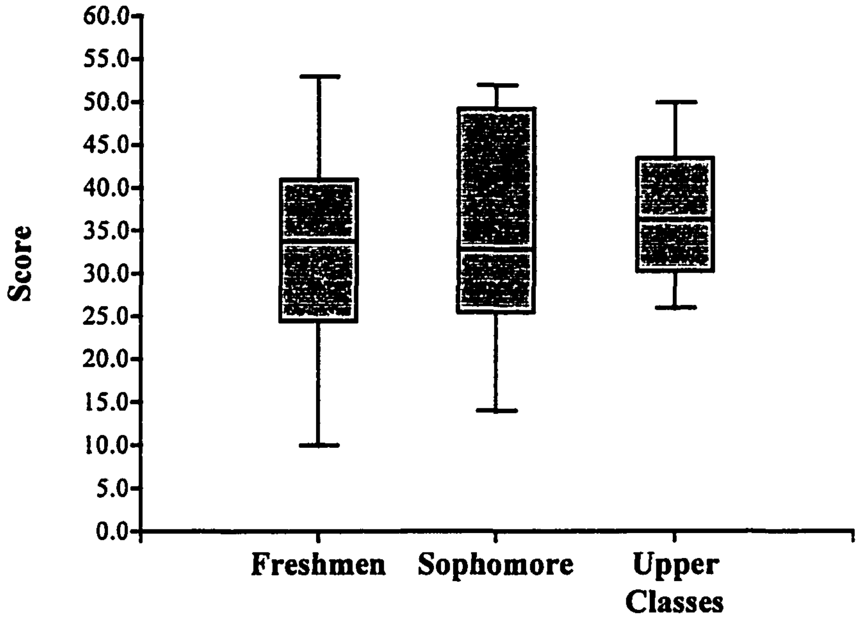


College Year. There are low positive correlations between college year and performance in sequential material, concurrency material, and project nine scores. The final exam sequential material scores are given in the table below. The freshmen had the lowest average and the upper classes (juniors, senior, and graduate students) had the highest average (with a range of about 4.3 points). The sophomore students had the lowest median and the upper classes had the highest median (with a range of 3.5 points). The number of freshmen in the comparison is almost triple the combined sophomore and upper classes population.

Sequential Material	Freshmen (1)	Sophomore (2)	Upper Classes (3, 4, 5)
Average	32.89	34.93	37.20
Median	34	33	36.5
Standard Deviation	10.322	12.487	7.997
Variance	106.543	155.918	63.956
Count	73	14	10

The box plot of the final exam sequential material scores by college year is given in Figure 5.38.

Figure 5.38. Final Exam Questions -- Sequential Material, College Year Comparison



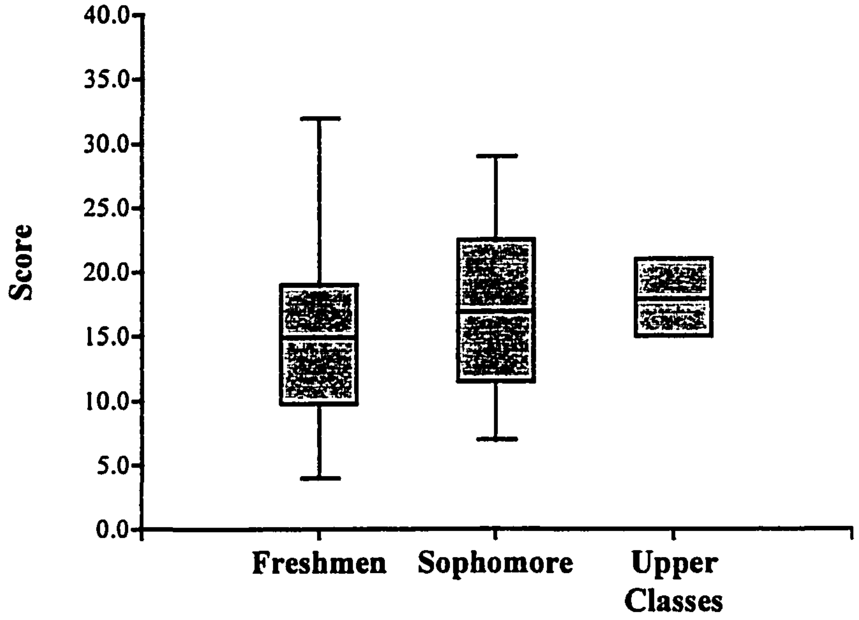
The upper classes category has higher median, 25th and 0th percentile scores than the other two college year classifications. The larger variance in the sophomore scores is likely due to the smaller number of sophomore students as compared to the freshmen students. The highest and lowest scores are in the freshmen students.

The final exam concurrency material scores for different college years are given in the table below. The freshmen average and median are lower than the averages and medians for the sophomore and other upper classes students. The sophomore average and median are within one point of the upper classes category average and median. There are almost four times more freshmen than sophomore and upper class students.

Concurrency Material	Freshmen (1)	Sophomore (2)	Upper Classes (3, 4, 5)
Average	15.38	17.20	18
Median	15	17	18
Standard Deviation	7.424	7.421	3
Variance	55.117	55.067	9
Count	58	10	5

The box plot for the final exam concurrency questions by college year comparison is given in Figure 5.39.

Figure 5.39. Final Exam Questions -- Concurrency Material, College Year Comparison



The freshmen 75th, 50th, 25th and 0th percentile scores are the lowest. Range of score is highest for the freshmen. The highest scores are freshmen scores. The box plot shows the range of score is decreasing with college year. There are only five upper class category students in the comparison.

The concurrency version of project nine scores for the two treatment groups are given in the table below. The averages for all three classifications are very close (within about 0.3 out of about 30). The freshmen have the highest median and the sophomores the lowest median. This comparison does not include the control group project nine scores. The correlation table above does include the control group project nine scores as part of the correlation computation.

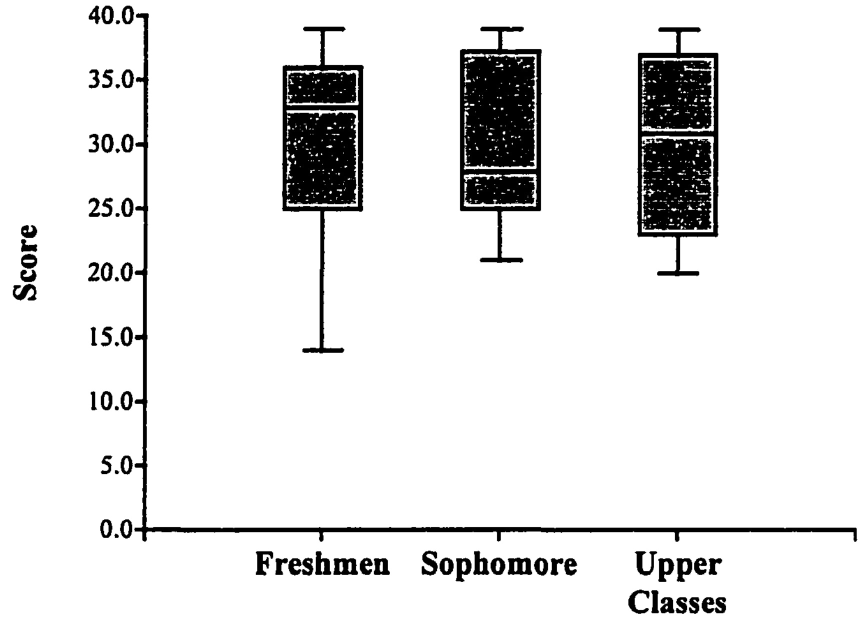
Concurrency Project Nine	Freshmen (1)	Sophomore (2)	Upper Classes (3, 4, 5)
Average	29.91	29.90	30.20
Median	33	28	31
Standard Deviation	7.281	6.641	7.463
Variance	53.010	44.100	55.700
Count	57	10	5

The box plot for the concurrency material project nine scores by college year comparison is given in Figure 5.40. It appears that the freshmen students are not disadvantaged in project nine scores on concurrency material:

- Freshmen have the same average (even with the lowest 0th percentile scores)
- Freshmen have the highest median score.
- The interquartile range (IQR) for the freshmen class is within the range of the IQR of the sophomore and other category.

Overall freshmen are performing as well on the concurrency version of project nine as the sophomore and upper classes students.

Figure 5.40. Concurrency Material Project Nine Scores, College Year Comparison



Summary. A correlation between -0.20 and +0.20 indicates a low correlation. The following low correlations were found:

- Female students doing somewhat better on concurrency questions than males
- Female students doing somewhat better on the last project, Project 9, than males
- Freshmen students doing somewhat worse on final exam question scores (for both sequential and concurrency material) than other students

The correlation between sequential material performance on the final exam and concurrency performance on the final exam is moderate at 0.68. This correlation implies sequential material performance is a positive indicator of concurrency material performance. However, the correlation is not high (above 0.80). The correlations between project nine and either sequential material or concurrency material are similar. The correlation between the last project, project nine, and the final exam sequential material question scores is moderate at 0.36. The correlation between the last project and final exam concurrency material question scores is also moderate at 0.39.

5.8 Validity and Sensitivity Check.

In general the criterion for validity is as follows -- the data used should be reliable and representative. The purpose of this subsection is to examine threats to the data to show that it is reliable. Threats to validity have been minimized. Second, the judgement of "is the data representative" is left to the reader. Data is presented in this subsection and in the appendices to support the reader making the judgement.

5.8.1 Internal Validity

Huck [Huck, 1974] identified in his text seven different sources of internal validity threats in an experiment. Five of the sources of concern were addressed in this experiment. These five threats and a quick summary of their defense were as follows:

1. Self-selection -- no defense, comparability of the groups check done
2. Mortality -- no defense, sufficient students left in all three groups
3. Maturation -- no defense, comparability of the groups check done
4. History -- some defense, two students were removed from the experiment
5. Instrumentation -- some defense, graders were not investigators in the experiment

Huck identified two additional sources of internal validity threats that are applicable only to other pseudo-experimental designs.

Self-Selection. The self-selection threat meant that the subjects of the experiment, the students, selected themselves for the individual class. Subjects were not randomly selected from a pool of students. Once in the class, each student had to pass a series of criteria to be included in the experiment. Subsection 5.1, titled Student Population, summarized the criteria for including a student in the experiment, and addressed the characteristics of the students that passed the criteria.

The self-selection of students had the potential to be the greatest threat. This was because each subject group could be different, possibly very different. It was a given in this experiment that each group was going to be different. However, techniques (controls) were used that minimized the impact of these differences. One of the simplest and most effective techniques was to calibrate each group against the others by providing identical treatments and observing the result. This was exactly what was done during the first eight weeks of each semester. All groups received the same material during the first eight weeks of the course. The relative performance of each group was calibrated against the same measure. Thus, each group's performance during the period when different treatments are applied, the second eight weeks in the course, could be adjusted for group differences (if it had been necessary).

Subsection 5.2, titled Comparability of the Groups, addressed the comparability of the groups. Significant differences in the mid-term scores and the project scores (projects two through five) were not found among any of the groups. The control group and two treatment groups were found to be comparable for the purpose of this experiment.

In summary, no control was used in the experiment for self-selection of students. Fortunately, it was not necessary.

Mortality. Mortality deals with the loss of subjects from any of the groups. The experiment could not be defended against mortality -- students could withdraw from the experiment at any time (either by course withdrawal or change of status, like to audit). Fortunately, enough students remained in each class for a statistically valid sample size to exist at the end of each semester. Students were required to take and successfully complete CSci 51 to advance to the other computer science courses. This alone "appeared" to provide sufficient motivation for students to complete the course (and provide a statistically valid sample size). The table below gives the number of student withdrawals by semester.

Category	**Control Group**	**Treatment Groups**		**Totals**
Semester	*Fall 1997*	*Spring 1998*	*Fall 1998*	
Withdrawal	5	14	11	30
Percentage	14.3	18.4	28.9	21.6

There were several different reasons given by students for withdrawal from the course. The following reasons were given by 24 of the students.

- Illness
- Change of major
- Overall poor grades
- Low mid-term grade only
- Unwilling to spend the time necessary
- Lack of time due to family and/or work

There were also three students with special cases that included the following:

- Student's fiancée had a serious illness and was near death
- Student athlete's practice requirements interfered with needed homework time
- Student completed just enough of the class to take other classes in the business school

Three students gave no reason at all.

Maturation. Maturation deals with the subjects growing maturity (both psychological and biological). The impact of this threat was that any one group may be more mature than another group. This could have impacted the results of the experiment. One of the simplest and most effective techniques was to calibrate each group against the others by providing identical instruction and observing the result. This was done as part of the experiment.

As previously stated, Subsection 5.2, titled Comparability of the Groups, addressed the comparability of the groups. Significant differences in the mid-term scores and the project scores (projects two through five) were not found among any of the groups. The control group and two treatment groups were found to be comparable for the purpose of this experiment.

In summary, no control was used in the experiment for maturation of students. It was not necessary. The period between when the mid-term exam was given and the end of the semester was always just eight weeks.

History. History deals with external non-experimental events affecting a student's performance over the period of the experiment. A significant emotional event will impact a student's performance. Controls were used to defend the experiment from history threats. Unfortunately, the controls used in the experiment defend only against known external non-experimental events. The important aspect of history threats to an experiment is that if a threat goes unobserved, a subject's performance is affected by an influence outside the experiment and may be attributed to the experimental treatment. History threats were addressed in two ways.

The first defense mechanism was to remove the affected student from the experiment. Student initiated withdrawal from the class was addressed under mortality. When the student was unselected by the instructor from the experiment due to an event, that is the history defense mechanism in action. During the experiment, this happened only twice during the Spring 1998 semester, treatment group one. These two students were included in the removed student category.

The second defense mechanism was to excuse the student from class work and grade recording for a specified period. Selected projects were to be made-up, and were graded as on-time. For example, during the control group phase, the grandmother of a student died. This student was excused from the course for three class periods. The student's makeup work was not graded. The overall work was not affected. Another example, during the Fall 1998 semester, second treatment group, a student's mother died. The student was excused from the course for three class periods. The student's mid-term score was affected. However, the projects and final exam work were not impacted.

Instrumentation. Instrumentation deals with the problems evaluating dependent variables. The instruments in this experiment included the instructor, the teaching assistant, and a paid grader. For example, differences between the control group and the experimental groups could be attributed to the instruments unless mechanisms were carefully built-in to the conduct of the experiment. The university selected the teaching assistant and the paid grader (not by the instructor). The potential for a grader learning to grade on the job did not exist. All graders had at least one semester's experience grading both exams and projects prior to their service in the experiment.

In the experiment, the same instructor was used for all groups. The same lab instructor is used for all groups, the teaching assistant.

For the purpose of the experiment, only the teaching assistant graded the mid-term exams and final exams for all groups. The dissertation committee dictated this action. The instructor prepared the grading criteria. The teaching assistant graded the exams.

Only the teaching assistant graded the projects for the Fall 1997 and Fall 1998 semesters. The teaching assistant and a paid grader graded the projects for the Spring 1998 semester. The paid grader was a college instructor at the University of Maryland, Baltimore campus. Selection of grader was entry sequence. About half the pile of project papers went to the teaching assistant and the other half to the grader. The first students to hand in a project had their projects at the bottom of the pile. The last students to hand in the project during class had their projects at the top of the pile (entry sequence). The prior class instructor, Professor Feldman, created the project grading criteria over six months before the experiment began. The prior class instructor also taught the teaching assistant how to grade the projects. The teaching assistant taught the paid grader how to grade the projects. The instructor taught the teaching assistant sufficient concurrent material such that the teaching assistant could understand and grade the concurrent projects.

In summary, sufficient controls were in place for instrumentation threats.

5.8.2 Generalization of the Experimental Results

The purpose of the subsection of the document is to explain how the results of this experiment can be generalized to a broader student population. Some of the characteristics of the source population for the experiment were as follows:

- Students enrolled at George Washington University (GWU)
- Mostly students enrolled in the School of Engineering and Applied Science (SEAS)
- Students without prior college computer science and computer programming experience

The source population was the pool of students available to enroll in the CS1 class at GWU. Some of the population characteristics in the experiment were as follows:

- Mostly students were freshmen (73 of 97 students) or sophomores (14 of 97 students)
- Many (at least 23 of 97) were in their first semester of college (hence a college grade point average was not available)
- Many (at least 48 of 97) were in their second semester of college

The population of the experiment was the students enrolled in the Introduction to Computing class that met the eligibility criteria -- students without prior college computer science and computer programming experience. Further population characteristics in the experiment are given in the table below.

Novice Category	**Control Group**	**Treatment Groups**		**Totals**
Foreign Student	3	2	3	8
SSN Student	21	50	18	89
Male	13	35	14	62
Female	11	17	7	35
Novice Total	24	52	21	97
Semester	*Fall 1997*	*Spring 1998*	*Fall 1998*	

SSN students have a social security number (SSN). Foreign students do not have a social security number. You do not have to be a citizen of the United States to have a social security number. Thus, the SSN student designation does not differentiate between United States citizens and resident aliens (with SSNs).

Additional population characteristics in the experiment are given below. These characteristics are repeated from other subsections of the document (dissertation).

Statistic	Control Group	Treatment Groups	
Semester	*Fall 1997*	*Spring 1998*	*Fall 1998*
Age			
Average	19.5	18.75	21.1
Median	19	18.5	19
Year in College			
Average	1.75	1.08	1.90
Median	1	1	2
Required Class			
Average	0.833	0.923	0.857
Median	1	1	1

There are two methods of generalizing the results of this experiment to a target population:

1. Compare the experimental population to a target population
2. Repeat the experiment with a different student population

The term target population refers to a larger, more general, population, with the implication that the results of this experiment apply to the target population. For example, the target population could be college freshman without prior college computer science and computer programming experience. Another target population could be college undergraduates without prior college computer science and computer programming experience.

Many characteristics of the student population in this experiment have been collected (such as sex, age, college year, class required, entry skill level upon entering the class, and social security number designation). These characteristics can be compared against the characteristics of the intended target population. Given that significant differences in experimental and target populations do not exist, the results of the experiment can be generalized to a specific target population. This is outside the scope of this experiment. The GWU undergraduate student population should not be considered typical of student populations for Continental U.S. colleges and universities until it is demonstrated.

The experiment can be repeated in different settings. The experiment could be repeated at different colleges and universities that have the potential to draw distinctly different student populations with similar student characteristics. For example, the experiment could be repeated at a mid-western university, a northeastern university, a southwestern university and community colleges in the same locations. Given that significant differences do not exist, the results of the combined experiments could be generalized to a very broad target population. This is outside the scope of this experiment.

Huck [Huck, 1974] identified eleven different sources of external validity threats in an experiment. Ten of the sources of concern were addressed in this experiment. These ten threats were as follows:

1. Experimentally accessible population versus target population
2. Interaction of treatment effects and subject characteristics
3. Describing the independent variable
4. Describing and measuring the dependent variable
5. Multiple-treatment interference
6. Interaction of history and treatment effects
7. Interaction of time of measurement and treatment effects
8. Rosenthal effect
9. Novelty and disruption effect
10. Hawthorne effect

Huck identified one additional source of external validity threat that was applicable only to other experimental designs. External validity is divided into two subgroups. The ability to generalize experimental results to other populations is called population validity (items 1 and 2 above). The ability to generalize experimental results to similar environments (or settings) is called ecological validity (items 3 through 10 above).

Experimentally Accessible Population versus Target Population. The experimentally accessible population was GWU students enrolling in the Introduction to Computing course. The potential target population of the experiment was college underclassmen without prior college computer science and computer programming experience. As previously stated, generalization of the experiment requires work outside the scope of the experiment. However, sufficient information was recorded for a comparison of the experimental population to a target population to be made.

Interaction of Treatment Effects and Subject Characteristics. Students in this experiment were predominately freshmen and sophomores. The ability to generalize the results of the experiment to another similar population is the threat here. For example, generalizing the results of this experiment may not be appropriate to college graduate students. Graduate students may learn concurrency much more efficiently than freshmen and sophomore students, given the same instructional methods are applied. Again, sufficient information was recorded for a comparison of the experiment's population to a target population to be made.

Describing the independent variable. Descriptions of procedure, activities, and time frames for the treatment must be of sufficient detail that the experiment can be replicated. This experiment would be very difficult to replicate without first obtaining all the instructional materials used in the classes. These materials have been given to the class director and current class instructor, Professor Feldman.

Describing and measuring the dependent variable. There are several aspects to this threat area:

- Descriptions of the dependent variables must be of sufficient detail that the experiment can be replicated
- Reliability of the measuring or test instrument
- Validity of the measuring or test instrument selected
- Reliability of the graders (in this experiment)
- Inaccurate analysis of the measured data

The above items are addressed in the succeeding paragraphs. The compilation of programs portion of this experiment is unique because the test instrument and data preparation tools are computer processes. Reliability is not the issue. Correctness is the issue. The test instrument programs were subjected to repeated testing (both for debugging and results verification).

Since the distributions collected were mostly non-normal, only non-parametric statistical tests that compare median scores between the groups could be used. The dependent variables in the experiment are the *similarities* in the distributions of the groups.

The reliability of the measuring or test instrument in this experiment concerns the degree to which the exams and projects used in the experiment test the knowledge learned by the students. Projects are designed to be both learning tools and a way of demonstrating the knowledge used. There exists a vast range of test question types available to the instructor -- from multiple choice questions to problem solving questions. Test question selection was patterned after the style of questions used by the prior class instructor. The concurrency question topic selection is very basic. The concurrency questions selected for the final exam come from very common concepts in concurrency:

- Semaphores
- Reading concurrent source code
- Message passing
- Shared memory

Another aspect of test performance is that students (not the instructor) select the questions that they are going to answer. The perception of what is easy or hard to the instructor may not be the perception of the test-taking student. Many students tend to pick the easier questions first. Other students do the test questions in the order presented. Given that a student believes that the sequential questions are fundamentally easier than concurrency questions, the sequential questions will be answered first and the concurrency questions will be left for last. The reverse is also true. Since the experiment is part of a college course, the student must be allowed to demonstrate what the student knows. For example, the final exam should not be divided into two halves -- one for concurrency and one for sequential topics. Students should be allowed to maximize their time on the topics they know best.

The validity of the measuring or test instruments selected for hypothesis testing were standard educational practice -- tests and projects that were graded.

The reliability of the graders in this experiment was addressed in Subsection 5.8.1 under the paragraph title *Instrumentation.*

The accuracy of the analysis of the measured data is not addressed explicitly. It is assumed to be correct. The reasons for this assumption to be valid are as follows:

- The distributions collected were mostly non-normal; hence, only non-parametric statistical tests that compare median scores between the groups could be used. Averages were reported (however, averages in non-normal distributions were more susceptible to the effects of outliers). Comparison of groups' medians is standard statistical practice. [Hintze, 1999]
- Equality of variance tests were done to determine which non-parametric statistical tests could be used (the results of these tests are in the Results Section of the document). Testing the equality of variance of distributions is standard statistical practice. [Hintze, 1999]
- Statistical procedures as defined by Hintze [Hintze, 1999] in using the statistical package named "Number Cruncher Statistical Systems (NCSS)" were followed.

Multiple-treatment interference. This threat occurs when subjects of an experiment are exposed to multiple treatments. The source of the multiple treatments can be one or more experiments. In this experiment the concurrency instruction is considered one treatment given over an eight-week period. An example of multiple-treatment interference would be to introduce concurrency and object-orientation simultaneously. It would be impossible to differentiate the effect of concurrency on sequential question test scores when both treatments are applied together. There is a single treatment applied in this experiment -- instruction in concurrency. The investigator is unaware of any student being involved in another experiment during the semester the student was enrolled in the class.

Interaction of History and Treatment Effects. This effect is due to historical events interacting with the subjects such that the effect of the treatment is changed. The investigator is aware of only one historical event that had the potential to impact a student's attention -- the political turmoil surrounding the President of the United States during 1998. The investigator's observation was that most students were oblivious to the event except a few students repeating late night talk show jokes. There were no other historical events that manifested themselves in classroom behavior.

Interaction of Time and Treatment Effects. This effect is due to the time between the treatment being applied and the observation (or measurement) being made. In a computer science classroom environment with one mid-term exam and one final exam, there is a delay between treatment and observation. The maximum delay results from instruction during class week nine and the final exam occurring in class week 16. There exist two observed differences between the fall and spring semesters:

- The fall final exams occurred near the end of the exam period while the spring final exam occurred in the middle of the exam period.
- Spring break occurred after the ninth week of class during the spring semester

However, since a significant difference was not found in the final exam sequential question scores for all groups, the effect should not be claimed to be significant.

Rosenthal Effect. This effect was due to the experimenter modifying the subjects' behavior unintentionally. The effect can be passed to the subjects by cues, gender, dress, appearance and so forth. In this experiment the instructor, investigator, and the experimenter were the same person. The students were in class with the instructor for one semester; thus, the instructor must have effected the students. However, each semester, or group, was exposed to the instructor for the same amount of time (and that the students all received an equal "dose" of the instructor). Given that each group got the same "dose" of the instructor, differences in the groups due to Rosenthal effect should not exist.

Novelty or Disruption Effect. This effect is due to the innovative nature of the treatment and the possible disruption of the subjects' performance due to being uncomfortable dealing with something new. The experiment was conducted as series of classes over three semesters. The newness of a class wears off over a period of several weeks. The course becomes just another class. The author is unaware of any new or novel aspects of the course presentation, except for the recording of student compilations (see Hawthorne effect discussion below).

Hawthorne Effect. Hawthorne effect deals with subjects' behavior changing due to the knowledge of being in an experiment. Although reminded numerous times over several weeks that the student compiles were being recorded, two students were caught cheating on the laboratory work that was not part of the experiment. The students forgot that they were part of an experiment. Thus, there is evidence that by week eight of the class, before the treatment had even started, the students forgot that they were part of an experiment. The course was just another class.

5.8.3 Sensitivity of the Assessment of Concurrency Instruction On Learning Sequential Material

The assessment of concurrency instruction on learning sequential material is presented in Subsection 5.3 of this document. The probability that the control group and the two treatment groups are not significantly different when tested about their medians is repeated in the following table (the table is presented in Subsection 5.3).

Method	Chi-Square (H)	Probability	Decision (0.05)
Corrected for Ties	5.420731	0.066512	Accept Ho

This probability that the "Accept Ho" decision is based upon is low. Therefore, an examination of other aspects of the comparison is warranted.

The statistical methods used in Subsection 5.3 compare the three groups together. An alternative method of comparison is to compare two groups at a time. The overall statistics for the six common sequential final exam questions are given in the table below. Both treatment groups are combined into a single group.

	Control Group	**Both Treatment Groups**
	Fall 1997	*Spring and Fall 1998*
Average	35.54	33.32
Median	38	34
Standard Deviation	11.662	9.702
Variance	135.998	94.136
Count	24	72

The table below shows the results of the normality tests (for the two groups together) and the equal-variance test on the two groups.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-0.7045	0.481097	Accept
Kurtosis Normality of Residuals	-2.1405	0.032313	Reject
Omnibus Normality of Residuals	5.0782	0.078938	Accept
Modified-Levene Equal-Variance Test	2.0399	0.156535	Accept

The omnibus test result indicates that the two groups just pass the normality test. The Modified-Levene test indicates that there is sufficient equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Mann-Whitney U test. The results are given below.

Alternative Hypothesis	**Test Value**	**Probability**	**Decision (0.05)**
D(G1) <> D(G23)	0.9991	0.317738	Accept Ho
D(G1) > D(G23)	1.0076	0.843173	Accept Ho

In a two way comparison of control group to combined treatment group, **the two groups (both control and treatment) are not significantly different when tested around their medians.** The probability that the groups are not significantly different when tested around their medians is now 0.317738 (rather than 0.066512 as in the three way comparison).

A second two way comparison is to compare the Fall 1997 control group and the Spring 1998 treatment group. However, the Spring 1998 treatment group received the question that included an "integer divide" (in the six common sequential final exam questions set) and the other groups' question set did not. This comparison is skipped.

A third two way comparison is to compare the two Fall groups on the six common sequential final exam questions. The table below shows the results of the normality tests and the equal-variance test on the two Fall groups.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-1.3287	0.183949	Accept
Kurtosis Normality of Residuals	-1.5916	0.111482	Accept
Omnibus Normality of Residuals	4.2985	0.116571	Accept
Modified-Levene Equal-Variance Test	0.1870	0.667583	Accept

The omnibus test result indicates that the two groups pass the normality test. The Modified-Levene test indicates that there is more than sufficient equality of variance for a non-parametric test with an equality of variance assumption to be used. The non-parametric statistical test selected is the Mann-Whitney U test. The results are given below.

Alternative Hypothesis	**Test Value**	**Probability**	**Decision (0.05)**
D(G1) <> D(G3)	0.3984	0.690306	Accept Ho
D(G1) < D(G3)	0.4212	0.663198	Accept Ho

In a two way comparison of control group to the Fall 1998 treatment group, **the two groups (both control and treatment) are not significantly different when tested around their medians.** The probability that the groups are not significantly different when tested around their medians is now 0.690306 (rather than 0.317738 as in the prior two way comparison).

5.8.4 Detailed Analysis of Student Withdrawals

The prior instructor in the course, Feldman, stated that the withdrawal percentage for the CS1 classes he had taught was approximately 20 percent. Thus, further analysis of the withdrawal rate for the experimental classes was appropriate, in particular for the Fall 1998 semester (withdrawal rate was 28.9 percent).

In Section 5.8.1, Internal Validity, student withdrawal statistics are presented as with regard to mortality as a threat to the experiment. The table below provides more information on student withdrawals. Illness is a major factor in the Fall 1998 withdrawal rate. If the four Fall 1998 students who withdrew from class due to illness had remained in the class, the withdrawal rate would be approximately 18.4 percent. The overall percentage of withdrawing students that are female is 30 percent; this percentage is in line with the overall percentage of female enrollment (refer to Section 5.1, Student Populations, for more details).

Category	**Control Group**	**Treatment Groups**		**Totals**
Semester	*Fall 1997*	*Spring 1998*	*Fall 1998*	
Illness	1	0	4	5
Female	4	2	3	9
Male	1	12	8	21
Total	5	14	11	30
Percentage	14.3	18.4	28.9	21.6

The following were the student provided reasons for withdrawal from the Fall 1998 class:

- Illness (can not continue class) -- 4
- Change of major -- 1
- Employment interferes -- 1
- Collegiate sports interferes -- 1

- Unwilling to spend the time necessary -- 1
- Low mid-term grade only -- 1

One student did not provide a reason, and another of the 11 withdrawing students was a repeat withdraw. In further analysis, this repeat withdraw is not counted in the Fall 1998 classes withdrawal statistics (withdrawal count used below is 10).

In order to examine the withdrawal rate, weekly withdrawal data was generated. The following process was used to obtain the withdrawal week number is as follows:

- Use the actual week of student withdrawal (used when withdraw date was recorded), or
- Use actual week of the first project not done

The withdrawal week data for the three classes is given below in week number (and count) format:

- Fall 1997 semester -- 3(2),12(2),15
- Spring 1998 semester -- 4, 6, 8(6), 10(3), 11(2), 12
- Fall 1998 semester -- 3, 4, 5, 7, 9, 10(2), 11(2), 15

For example, six students withdrew from the Spring 1998 semester class in week eight. The average withdrawal week for all three groups is week 8.5 to 9.0. This means that one-half the withdrawals occur before concurrency instruction has started or at the time the mid-term score is returned.

The table below shows the results of the normality tests (for all three groups together) and the equality of variance test on the combined groups.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Skewness Normality of Residuals	-0.5448	0.585914	Accept
Kurtosis Normality of Residuals	-0.3551	0.722493	Accept
Omnibus Normality of Residuals	0.4229	0.809415	Accept
Modified-Levene Equal-Variance Test	2.2199	0.128793	Accept

The omnibus test result indicates that three groups combined together could form a normal distribution. The Modified-Levene test indicates that there is minimal equality of variance for a non-parametric test with an equality of variance assumption to be used. The number of withdrawals in the Fall 1997 semester (five) is ordinarily too low for statistical tests to be used; however, there is no alternative available. The remaining 29 withdraws (the repeat withdrawal is not included) were compared using the non-parametric Kruskal-Wallis One-Way ANOVA on Ranks test to find the probability that the three classes were derived from the same population. The results of the test are given in the table below.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Corrected for Ties	0.301092	0.856797	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.**

In conclusion, the profile of student withdrawals during the three semesters of this experiment does not yield anything unusual. Finally, the overall withdrawal rate for the three semesters in the experiment is in line with the prior withdrawal rate stated by Feldman, when concurrency was not taught.

5.9 Hypothesis and Findings

Hypothesis One.

Hypothesis: Students who have had concurrency training will have significantly lower sequential test question scores than those who have not had concurrency training.

Null Hypothesis: There is no difference in the sequential test question scores between a student who has had concurrency training and a student who has not had concurrency training.

An assessment of the impact of concurrency instruction on learning sequential material was done as part of the final exam. Six sequential questions were given to the control group and to the two treatment groups. The test scores of the three groups were compared for significant differences. The test selected was the Kruskal-Wallis One-Way ANOVA on Ranks; the test is a non-parametric statistical test. The result of comparing the three distinct groups is given in the table below.

Method	Chi-Square (H)	Probability	Decision (0.05)
Corrected for Ties	5.420731	0.066512	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.** The comparison was repeated using five of the six sequential questions, due to an "integer divide" difference in one of the questions for the Spring 1998 treatment group. Again, the test selected was the Kruskal-Wallis One-Way ANOVA on Ranks; the test is a non-parametric statistical test. The result of comparing the three distinct groups is given in the table below.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Corrected for Ties	5.329297	0.069624	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.**

Finally, the two treatment groups combined into a single group. This results in a control group to combined treatment group comparison. The test selected was the Mann-Whitney U test, a non-parametric statistical test. The result is given in the table below.

Alternative Hypothesis	**Test Value**	**Probability**	**Decision (0.05)**
D(G1) <> D(G23)	0.9991	0.317738	Accept Ho

In a two way comparison of control group to combined treatment group, **the two groups (both control and treatment) are not significantly different when tested around their medians.** The probability that the groups are not significantly different when tested around their medians is now 0.317738 (rather than 0.066512 as in the three way comparison).

The hypothesis must be rejected. The three groups (both control and treatment) are not significantly different when tested around their medians. The null hypothesis is accepted. There is no significant difference in the sequential test question scores between a student who has had concurrency training and a student who has not had concurrency training.

Hypothesis Two.

Hypothesis: Students, with both sequential and concurrency instruction, will score significantly different on concurrency test questions than on sequential test questions.

Null Hypothesis: There is no difference in the concurrency test question scores and sequential test question scores.

A comparison of the students' ability to solve sequential questions and concurrency questions was done as part of the final exam. Four final exam questions given to the treatment groups were questions on concurrency material. The students selected the order in which the questions were answered.

The sequential test scores and concurrency test scores within each treatment group were compared for significant differences. The test selected was the Mann-Whitney U test; the test is a non-parametric statistical test. The result of running the test for the Spring 1998 treatment group is given in the table below.

Alternative Hypothesis	Test Value	Probability	Decision (0.05)
D(S) <> D(C)	5.4078	0.000000	Reject Ho
D(S) > D(C)	5.4144	1.000000	Accept Ho

The result of running the test for the Fall 1998 treatment group is given in the table below.

Alternative Hypothesis	Test Value	Probability	Decision (0.05)
D(S) <> D(C)	3.5994	0.000319	Reject Ho
D(S) > D(C)	-3.6245	0.999855	Accept Ho

Two alternative hypotheses were presented. The "D(S) <> D(C)" hypothesis is the null hypothesis (the probability that there is no significant difference between the two groups). The second hypothesis shown is the difference hypothesis with the highest probability (in this case the sequential distribution, "D(S)", is greater than concurrency distribution," D(C)" with a probability at or near 1.0 in both treatment groups).

The null hypothesis must be rejected. The hypothesis accepted is as follows -- students, with both sequential and concurrency instruction, will score significantly lower on concurrency test questions than on sequential test questions. There are significant differences between the sequential test question scores and the concurrency test scores with each group.

Hypothesis Three.

Hypothesis: Students who use concurrent methods on a "large" project will have significantly lower project scores than those who use sequential methods.
Null Hypothesis: There is no difference in "large" project scores between a student using concurrent methods and sequential methods.

The large project in the Introduction to Computing course was the last project, project nine. In this project the students were asked to create a small database and to provide rudimentary set of capabilities. All groups were given a crude character based menu interface to modify. The control group class did the project using all sequential methodology. The treatment group classes did the project using concurrent methodology. The project scores of the three groups were compared for significant differences. Only non-zero project scores were included in the comparison (i.e., the student had to do the project). The test selected was the Kruskal-Wallis One-Way ANOVA on Ranks; the test is a non-parametric statistical test. The result of running the test is given in the table below.

Method	Chi-Square (H)	Probability	Decision (0.05)
Corrected for Ties	0.4230784	0.809338	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.**

The hypothesis must be rejected. The three groups (both control and treatment) are not significantly different when tested around their medians. The null hypothesis is accepted. There is no difference in "large" project scores between a student using concurrent methods and sequential methods.

Findings.

A measure of relative work (by compilations) was studied in this experiment. The measure of relative work used was a ratio:

$$\text{Ratio} = \frac{\text{Compilations for project nine (P9)}}{\text{Compilations for projects two through five (P2-5)}}, \quad \text{per student}$$

Projects two through five were performed over a five week period. The period was longer if project five was completed late. Project nine was performed over a three week period. Thus, the "ratio" studied was derived from eight weeks of student compilation activity. The ratios for the control group and treatment groups were compared. The test selected was the Kruskal-Wallis One-Way ANOVA on Ranks; the test is a non-parametric statistical test. The result of running the test is given in the table below.

Method	Chi-Square (H)	Probability	Decision (0.05)
Not Corrected for Ties	1.593634	0.450761	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.** Duplicate values do not exist within any single group; thus, the statistic is not corrected for ties. Significant differences in the compilation "ratio" are not found, even though the treatment group students used concurrency methodology in project nine.

The ratio of total error messages to distinct error messages was studied for the control group and the two treatment groups. Compiler error messages were compressed into distinct error messages. The method used to create distinct error and warning messages was as follows:

- Replace variable names with a representative symbol (such as "token")
- Replace numbers with a representative symbol (such as the value "99")

This ratio was measure of the number of times the "average" error message occurs. The test selected was the Kruskal-Wallis One-Way ANOVA on Ranks; the test is a non-parametric statistical test. The result of running the test is given in the table below.

Method	**Chi-Square (H)**	**Probability**	**Decision (0.05)**
Not Corrected for Ties	0.3133233	0.854993	Accept Ho

The "Accept Ho" decision means the following: **the three groups (both control and treatment) are not significantly different when tested around their medians.** The very high probability is significant, it implies that compilation error resolution is not affected by concurrency.

6 CONCLUSIONS AND FUTURE RESEARCH

This section of the document presents the conclusions and future research topics. The conclusions are presented first.

6.1 Conclusions

Several conclusions can be drawn from the results of this experiment. They are presented in this subsection of the document.

First, concurrency can be taught to CS1 students. The students do learn the concurrency material. Both treatment groups were able to do the large project using concurrency methods (refer to Section 5.5). The control and treatment groups were not significantly different in performance on the large project. The treatment groups students were able to demonstrate learning concurrent material on the final exam (refer to Section 5.4); however, the treatment groups' concurrency test question scores were significantly lower than the treatment groups' sequential test question scores.

Second, teaching concurrency to CS1 students does no harm -- the treatment group students learned the sequential material successfully. In a two way comparison of the control group to the combined treatment group, the two groups (both control and treatment) are not significantly different (see Section 5.8.3). Further, there is a probability of 0.663 that the Fall 1998 treatment group did better on the final exam sequential test questions than did the Fall 1997 control group.

Third, compilation error resolution is not affected by concurrency -- differences in distinct error ratio (DER) between using sequential methods and concurrent methods to do the large project were not observed (refer to Section 5.6). Restated, for the programming language Ada 95 (a programming language with imbedded concurrency), differences in the novice students' ability to solve syntax errors in either concurrent programs or sequential programs were not observed. Compilation listing and syntax error recording was an essential aspect of this experiment. This recording capability allowed the experimenter to derive the following items:

- Distinct error ratios
- Distinct error messages
- Relative level of effort for the large project as compared to other projects

Much mcre information could be "mined" from the compilation listing and syntax error data recorded during this experiment.

Fourth, the methodology used in the experiment performed as expected -- validity problems did not get into the experiment. Threats to the experiment due to internal validity problems were handled with a minimum of defensive action (refer to Section 5.8.1). Students were placed in one of four categories: novice, experienced, removed, and withdrawal. Only novice students were included in the experiment. There were sufficient novice students in each class, after filtering for experienced, removed, and withdrawn students, for the experiment to be run (refer to Section 5.1). Internal validity threats such as self-selection, mortality, and maturation required no defense at all. Threats to the experiment due to external validity problems were handled without defense action being needed (refer to Section 5.8.2). Situations arose during the experiment that provide credence to the assertion that the students were unaware they were part of an experiment after several weeks had passed.

Finally, based on the first three conclusions, **introducing concurrency into an introductory computer science course can be done.**

6.2 Future Research

Several areas of future research topics can be drawn from this experiment. They are presented in this subsection of the document.

6.2.1 Extending the Experiment

This experiment can be repeated in different collegiate settings. The experiment could be repeated at different colleges and universities that have the potential to draw distinctly different student populations with similar student characteristics. Given that similar results are derived in these different settings, this would extend the finding of this experiment to a larger, more general, population. It would also validate the findings of this experiment.

This experiment can be repeated in senior high school settings. The experiment could be repeated in high schools that have advance placement in computer science. Extending the experiment to high schools would have several interesting aspects (as compared to a college student population):

- The student population would be younger.
- The student population may be less mature.
- The student population would probably have much less opportunity to skip or withdraw from the class.

Performing this experiment in a high school setting would require that the definition of an experienced student be rewritten (the current definition includes college course work).

6.2.2 Sequencing of Course Materials

The finding that significant differences in final exam sequential test question scores were not found whether the students did or did not received concurrency instruction is important. The investigator knows of three possible implications of this finding:

- Instruction in concurrency and programming using concurrent methods does not logically conflict with sequential instruction (restated -- sequential and concurrent instruction "dovetail" together in such a way that the sequential instruction loss is minimal)
- Beginning college level programming instruction provides more information than the students can absorb and replicate in a given topic area -- sequential programming
- The additional sequential material presented and the additional time spent reviewing already presented sequential material to the control group students may not reinforce already learned sequential material

The second item is fascinating because it resembles Miller's theorem that states that the average person can remember (memorize) only a fixed count of numbers (five through nine) at a time. Restated, the difference between sequential methods and concurrency methods is enough of a change in subject, that the concurrent method learning continues to be recorded by the students. This implies that there may be a point of diminishing returns in teaching sequential methods and that the sequence of material presented to entry level college programmers should be examined. From a different perspective, this would imply a "teach the basics philosophy" for multiple computer science areas (or topics) in the introductory course. It may provide a key to understanding some withdrawal and low grade behavior in introductory computer science courses.

The third item implies that the control group students had already learned certain sequential material and it was time to "move on" to new material.

Taking items two and three together has further implications -- the sequencing of introductory computer science material should be reexamined. The examples given below serve to illustrate the point.

The order that introductory materials are presented could be modified to give students more time with more difficult concepts. LOOP structures are somewhat more difficult for students to initially grasp. Restated, the time on task to comprehend LOOPs may be longer. First, moving the introduction of LOOP structures to earlier in the course would provide more time on the material before examination and the students would be more familiar with the material. Second, moving the introduction of LOOP structures to earlier in the course increases the variation in projects available.

Ada task syntax appears to be more difficult for students to grasp. Moving the introduction of tasks to earlier in the course is probably not a good idea. However, if the semester were 18 weeks long (two weeks longer), the final exam concurrency question scores should improve. The Ada task syntax could have been well rehearsed by the students.

In summary, the sequence of material presented and time of presentation of introductory computer science materials should receive further study.

6.2.3 Language and Compiler Characteristics

The finding that students did better on final exam sequential test question scores compared to final exam concurrency test question scores could be influenced due to Ada task syntax appearing to be more difficult for students to grasp. Since the results indicate that novice students can and do learn concurrent material, the next question is "are these results generalizable to another computer language". Two possible languages are Java and C++. Java has the advantage of having threads built into the language. C++ has the advantages of being accepted as a language used for teaching object-oriented programming and a language heavily used in business. C++ has the disadvantage of concurrency methods being an add-on to the language or done through system calls. However, these advantages and disadvantages can be shown to be secondary to the clarity of error and warning messages from the compiler. The target population is college entry level computer science students. Beginning computer science students need excellent error and warning messages. Easy to understand error and warning messages make it much easier for novice programmers to learn. Older languages have the potential to have more revisions to an existing compiler. For example, GNU C++ compiler has been revised and updated numerous times. Most of these revisions can and do have improvements to the compile time diagnostics -- the error and warning message system. Finally, the instructor's familiarity with the computer language is important.

Significant differences in the Distinct Error Ratio (Total Errors / Distinct Errors), DER, for the sequential and concurrent large projects were not found; this finding is based on the programming language Ada. Again, the next question is "are these results generalizable to another computer language". The computer language selected should have concurrency built into the language. Java has the advantage of having threads built into the language. C++ does not have concurrency as an intrinsic part of the language. Another computer language with intrinsic concurrent capabilities is Occam. The language Occam uses indentation as the syntax to designate loop structure. Novice programming students may have difficulty with the syntax used in Occam..

Setting concurrency aside, the tools developed in this experiment can be adapted to another computer language. An experiment should be created to compare learning introductory computer science with one class using Ada and the other class using another language (preferably C++ or Java). Given that both classes are taught the same material and in the same order, a direct comparison of the languages for novice students could be done. Student learning curves for syntax and semantics could be developed and compared. Ideally the compilers should both come from the same "software house". For example, one class could use GNU's GNAT Ada compiler and the other class could use GNU's C++ compiler. The compilers sharing the same origins (i.e., the compilers are developed by GNU) provides a greater possibility for the compilers to be developed with the same philosophy for the error and warning message system and that the compilers share the object code generator.

There is a second facet of the DER that should be examined. Many years ago, the investigator was a programming group manager for a large consulting firm in the Washington area. The programming staff at the firm was required to be multi-lingual (computer languages). As the staff learned new computer languages, the staff's ability to recognize compiler errors and solve them quickly appeared to improve with experience. *However, the order in which languages were learned was a factor.* College students may exhibit similar behavior. It would be very interesting (at least to the investigator) to compile statistics on DER behavior based on student performance. The long term effect of the order that computer languages are introduced to students could be studied from an DER perspective. The table below is an DER profile for given college. The table shows the order in which the programming languages are introduced.

Programming Language	Semesters Programming					
	1	2	3	4	5	6
Ada	n.nn	n.nn				
C++			n.nn	n.nn	n.nn	n.nn
Java						

In the table above, "n.nn" is the median DER for a given group at College A. This information could then be compared with a similar Error Ratio table compiled at another college. The impact of the order in which computer languages are taught in college would have a new known metric.

BIBLIOGRAPHY

[Adams, 1996] Adams, Joel C., "Object-Centered Design, A Five Phase Introduction To Object-Oriented Programming In CS1-2", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 78-82.

[Allen, 1997] Allen, Michael, Wilkerson, Barry, and Ailey, James, "Parallel Programming for the Millenium: Integration Throughout the Undergraduate Curriculum", Conference Proceedings, Second Forum on Parallel Computing Curricula, Newport Rhode Island, June 1997, http://www.cs.dartmouth.edu/FPCC/papers/Wilkerson/index.html.

[Arnow, 1995] Arnow, David M., XDP: A Simple Library For Teaching A Distributed Programming Module", The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, SIGCSE Bulletin, Volume 27, Number 1 March 1995, pages 82-86.

[Ashton, 1997] Ashton, Paul, "Using Interaction Networks for Visualization of Message Passing", The Papers of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education, San Jose, California, SIGCSE Bulletin, Volume 29, Number 1, March 1997, pages 272-276.

[Aki, 1989] Aki, Selim, G., The Design and Analysis of Parallel Algorithms, Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[Andrews, 1991] Andrews, Gregory R., Concurrency Programming, Benjamin / Cummings, Redwood City, California, 1991.

[Ausubel, 1970] Ausubel, David P., The Use of Ideational Organizers in Science Teaching, Occasional Paper 3, ERIC Document Reproduction Service, Number ED050930, March 1970.

[Bachus, 1996] Bachus, Bruce D., Determining the Feasibility of Introducing Concurrent Programming into the Lower-Level Curriculum via a Controlled Experiment, Doctor of Science Dissertation, George Washington University, Washington, D.C., 1996

[Baldwin, 1996] Baldwin, Doug, "Discovery Learning In Computer Science", ", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 222-226.

[Barsalou, 1992] Barsalou, Lawrence W., Cognitive Psychology, An Overview for Cognitive Scientists, Lawrence Erlbaum Associates, Hillsdate, New Jersey, 1992.

[Ben-Ari, 1990] Ben-Ari, M., Principles of Concurrent and Distributed Programming, Prentice Hall, New York, New York, 1990.

[Ben-Ari, 1996] Ben-Ari, Mordechai, "Using Inheritance To Implement Concurrency", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 180-184.

[Berk, 1996] Berk, Toby S., "A Simple Student Environment For Lightweight Process concurrent Programming Under SunOS", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 165-169.

[Brilliant, 1996] Brilliant, Susan S., and Wiseman, Timothy R., "The First Programming Paradigm And Language Dilemma", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 338-342.

[Brown, 1998] Computer Science Department, Brown University, http://www.cs.brown.edu/courses.

[Burkhart, 1997] Burkhart, Helmar, "Parallel Programming Using Public Domain Software", The Papers of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education, San Jose, California, SIGCSE Bulletin, Volume 29, Number 1, March 1997, pages 224-228.

[Burns, 1995] Burns, Alan, and Wellings, Andy, Concurrency in Ada, Cambridge University Press, 1995.

[Bustard, 1990] Bustard, David W., Concepts of Concurrent Programming, Curriculum Module CM-24, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1990.

[Bynum, 1996] Bynum, Bill, and Camp, Tracy, "After You, Alfonse: A Mutual Exclusion Toolkit", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 170-174.

[Carnegie, 1998] Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, http://www.cs.cmu.edu/csd/undergrad/ugcourses.html.

[Chandra, 1994] Chandra, Rohit, Gupta, Anoop, and Hennessy, John L., "COOL: An Object-Based Language for Parallel Programming", Computer Volume 27, Number 8, August 1994, pages 13-26.

[Cohen, 1986] Cohen, Norman H., Ada As A Second Language, McGraw-Hill, New York, New York, 1986.

[Cormen, 1990] Cormen, Thomas H., Leiserson, Charles E., and Rivest, Ronald L., Introduction to Algorithms, MIT Press, Cambridge, Massachusetts, 1990.

[Dartmouth, 1998] Department of Computer Science, Dartmouth College, Hanover, New Hampshire, http://www.cs.dartmouth.edu/.

[DeClue, 1996] DeClue, Tim, "Object-Orientation And The Principles of Learning Theory: A New Look At Problems and Benefits", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 232-235.

[Decker, 1994] Decker, Rick, and Hirschfield, Stuart, "The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught In CS1", The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education, Phoenix, Arizona, SIGCSE Bulletin, Volume 26, Number 1, March 1994, pages 51-55.

[Dillon, 1997] Dillon, Eric, Dos Santos, Carlos Gamboa, and Guyard, Jacques, "Teaching an Engineering Approach for Networking Computing", The Papers of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education, San Jose, California, SIGCSE Bulletin, Volume 29, Number 1, March 1997, pages 229-232.

[Ductworth, 1994] Duckworth, James R., "Introducing Parallel Processing Concepts Using The Maspar MP-1 Computer", The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education, Phoenix, Arizona, SIGCSE Bulletin, Volume 26, Number 1, March 1994, pages 353-356.

[Elenbogen, 1996] Elenbogen, Bruce S., "Parallel and Distributed Algorithms Laboratory Assignments In Joyce / Linda", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 14-17.

[Ercal, 1996] Ercal, Fikret, Class Notes, CSc 387 - Parallel Processing: Algorithms, Architectures, and Languages, November 1996, http://www.cs.umr.edu/~ercal/387/387.html.

[Feldman, 1990] Feldman, Michael, B., Language and System Support for Concurrent Programming, Curriculum Module CM-25, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, April 1990.

[Feldman, 1992] Feldman, Michael, B., "The Portable Dining Philosophers: a Movable Feast of Concurrency and Software Engineering", The Papers of the Twenty-third SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, SIGCSE Bulletin, Volume 24, Number 1, March 1992, pages 276-280.

[Feldman, 1996] Feldman, Michael, B, and Koffman, Elliot, B., Ada 95 Problem Solving and Program Design, Addison-Wesley, Reading, Massachusetts, 1996.

[Feldman, 1997] Handouts and Program Files for Spring 1997 Semester, as taught by Feldman, Michael B., http://www.seas.gwu.edu/classes/csci51/spring97/index.html.

[Fisher, 1991] Fisher, Allan L., and Gross, Thomas, "Teaching the Programming of Parallel Computers", The Papers of the Twenty-second SIGCSE Technical Symposium on Computer Science Education, San Antonio, Texas, SIGCSE Bulletin, Volume 23, Number 1, March 1991, pages 102-107.

[Fisher, 1992] Fisher, Allan L., and Gross, Thomas, "Teaching Empirical Performance Analysis of Parallel Programs", The Papers of the Twenty-third SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, SIGCSE Bulletin, Volume 24, Number 1 March 1992, pages 309-313.

[Forsythe, 1996] Forsythe, Ronald G., and Mavrovouniotis, Michael L., "On The Need For Object-Oriented Programming In Engineering Curricula", Computers In Education, Volume 6, Number 2, April-June 1996, pages 50-55.

[Foster, 1997] Foster, Ian, Building and Designing Parallel Programs (Online), Concepts and Tools Parallel Software Engineering, Addison-Wesley, Reading Massachusetts, 1997, http://www.cs.rdg.ac.uk/dbpp/text/node1.html.

[Fox, 1997] Fox, Geoffrey C. and Furmanski, Wojtek, "Java for parallel computing and as a general language for scientific and engineering simulation and modeling", Concurrency: Practice and Experience, Volume 9, Number 6, June 1997, pages 415-426.

[Gehani, 1991] Gehani, Narain, Ada: Concurrent Programming, Silicon Press, Summit New Jersey, 1991.

[Ginat, 1996] Ginat, David, "Efficiency Of Algorithms For Programming Beginners", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 256-260.

[Gomaa, 1993] Gomaa, Hassan, Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley, Reading, Massachusetts, 1993.

[Goudreau, 1997] Goudreau, Mark W. "Unifying Software and Hardware in a Parallel Computing Curriculum", Conference Proceedings, Second Forum on Parallel Computing Curricula, Newport Rhode Island, June 1997, http://www.cs.dartmouth.edu/FPCC/papers/goudreau.html.

[Guzdial, 1995] Guzdial, Mark, "Centralized Mindset: A Student Problem with Object-Oriented Programming", The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, SIGCSE Bulletin, Volume 27, Number 1 March 1995, pages 182-185.

[Harlan, 1995] Harlan, Robert M., and Akulis, Joseph G., "Parallel Threads: Parallel Computation Labs For CS3 And CS 4", ", The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, SIGCSE Bulletin, Volume 27, Number 1 March 1995, pages 141-145.

[Hartley, 1992] Hartley, Stephen, J. "Experience with the Language SR in an Undergraduate Operating Systems Course", The Papers of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, SIGCSE Bulletin, Volume 24, Number 1, March 1992, pages 176-180.

[Hartman, 1991] Hartman, Janet, and Sanders, Dean, "Teaching a Course in Parallel Processing with Limited Resources", The Papers of the Twenty-second SIGCSE Technical Symposium on Computer Science Education, San Antonio, Texas, SIGCSE Bulletin, Volume 23, Number 1, March 1991, pages 97-101.

[Hartman, 1993] Hartman, Janet, and Sanders, Dean, "Data Parallel Programming: A Transition from Sequential to Parallel Computing", The Papers of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education, Indianapolis, Indiana, SIGCSE Bulletin, Volume 25, Number 1 March 1993, pages 96-100.

[Hill, 1991] Hill, Jane C., and Wayne, Andrew, "A CYK Approach To Parsing In Parallel: A Case Study", The Papers of the Twenty-second SIGCSE Technical Symposium on Computer Science Education, San Antonio, Texas, SIGCSE Bulletin, Volume 23, Number 1, March 1991, pages 240-245.

[Hintze, 1999] Hintze, Jerry L., User's Guide, NCSS 2000, Statistical System for Windows, Number Cruncher Statistical Systems, January 1999.

[Howard, 1996] Howard, Richard A., Carver, Curtis A., and Lane, William D., "Felder's Learning Styles, Bloom's Taxonomy, And Kolb Learning Cycle: Tying It All Together In The CS2 Course", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 227-231.

[Huck, 1974] Huck, Schuyler W., Cormier, William H., Bounds, William G. Jr., Reading Statistics and Research, Harper Collins, 1974.

[Huck, 1996] Huck, Schuyler W., Cormier, William H., Reading Statistics and Research, Harper Collins, 1996.

[Hummel, 1997] Hummel, Susan F., Ngo, Ton, and Srinivasan, Harini, "SPMD programming in Java", Concurrency: Practice and Experience, Volume 9, Number 6, June 1997, pages 621-631.

[Hurley, 1994] Hurley, Stephen, Department of Computer Science, University of Wales at Cardiff, England, http://www.cs.cf.ac.uk.

[Indiana, 1998] Computer Science Department, Indiana University, http://www.cs.indiana.edu/dept/acad.

[Jackson, 1991] Jackson, David, "A Mini-Course on Currency", The Papers of the Twenty-second SIGCSE Technical Symposium on Computer Science Education, San Antonio, Texas, SIGCSE Bulletin, Volume 23, Number 1, March 1991, pages 92-96.

[JaJa, 1992] JaJa, Joseph, An Introduction to Parallel Algorithms, Addison-Wesley, Reading Massachusetts, 1992.

[Jin, 1995] Jin, Lan, and Yang, Lan, "A Laboratory For Teaching Parallel Computing On Parallel Structures", The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, SIGCSE Bulletin, Volume 27, Number 1, March 1995, pages 71-75.

[John, 1992] John, David, "Integration of Parallel Computation into Introductory Computer Science", The Papers of the Twenty-third SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, SIGCSE Bulletin, Volume 24, Number 1 March 1992, pages 281-285.

[John, 1994] John, David J., "NSF Supported Projects: Parallel Computation as an Integrated Component in the Undergraduate Curriculum in Computer Science", The Papers of the Twenty-fifth SIGCSE Technical Symposium on Computer Science Education, Phoenix, Arizona, SIGCSE Bulletin, Volume 26, Number 1, March 1994, pages 357-361.

[Katsinis, 1994] Katsinis, Constantine, "The Development of a Multi-Processor Personal Computer in a Senior Computer Design Laboratory", The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education, Phoenix, Arizona, SIGCSE Bulletin, Volume 26, Number 1, March 1994, pages 349-352.

[King, 1992] King, K. N., "The Evolution of the Programming Language Course", The Papers of the Twenty-third SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, SIGCSE Bulletin, Volume 24, Number 1, March 1992, pages 213-219.

[Kitchen, 1992] Kitchen, Andrew T, Schaller, Nan C., and Tymann, Paul T., "Game Playing As A Technique For Teaching Parallel Computing Concepts, SIGCSE Bulletin, Volume 24, Number 3, September 1992, pages 35-38.

[Koffman, 1984] Koffman, Elliot B., Miller, P. L., Wardle, Caroline, E., "Recommended Curriculum for CS1, 1984", Communications of the ACM, Volume 27, Number 10, October 1984, pages 998-1001.

[Koffman, 1985] Koffman, Elliot B., Stemple, David, Wardle, Caroline, E., "Recommended Curriculum for CS2, 1985", Communications of the ACM, Volume 28, Number 8, August 1985, pages 815-818.

[Kolling, 1996] Kolling, Michael, and Rosenberg, John, "An Object-Oriented Program Development Environment For The First Programming Course", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 83-87.

[Kotz, 1995] Kotz, David, "A Data-Parallel Programming Library For Education (DAPPLE)", The Papers of the Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, SIGCSE Bulletin, Volume 27, Number 1 March 1995, pages 76-81.

[Langan, 1993] Langan, David D., "A Multi-Purpose Dataflow Simulator", The Papers of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education, Indianapolis, Indiana, SIGCSE Bulletin, Volume 25, Number 1 March 1993, pages 87-90.

[Leska, 1996] Leska, Chuck, Barr, John, and Smith King L. A., "Multiple Paradigms In CS 1", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 343-347.

[Lewis, 1997] Lewis, John, and Loftus, William, Java Software Solutions, Addison-Wesley, Reading, Massachusetts, 1997, table of contents and outline available at http://cseng.aw.com/bookdetail.qry?ISBN=0-201-57164-1&ptype=0.

[Liu, 1996] Liu, Mei-Ling, and Blanc, Lori, "On The Retention of Female Computer Science Students", ", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 32-36.

[Manchester, 1998] Computer Science Department, University of Manchester, Manchester, England, http://www.cs.man.edu.uk.

[Mason, 1998] Course Description Spring 1998, Department of Computer Science, George Mason University, Fairfax, Virginia, http://www.cs.gmu.edu/syllabus/syllabi-spring98/cs112.html.

[Mayer, 1992] Mayer, Richard E., "Cognition and Instruction: Their Historic Meeting Within Education Psychology", Journal of Educational Psychology, Volume 84, Number 4, pages 405-412.

[McDonald, 1992] McDonald, Chris, "Teaching Concurrency with Joyce and Linda", The Papers of the Twenty-third SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, SIGCSE Bulletin, Volume 24, Number 1, March 1992, pages 46-52.

[McDonald, 1997] McDonald, Chris, and Kazemi, Kamram, "Improving the PVM Teaching Environment", The Papers of the Twenty-eighth SIGCSE Technical Symposium on Computer Science Education, San Jose, California, SIGCSE Bulletin, Volume 29, Number 1, March 1997, pages 219-223.

[McMaster, 1999] Graduate Courses, McMaster University Computer Science, Hamilton, Ontario, Canada, http://www.dcss.mcmaster.ca/graduate/courses/.

[Meredith, 1992] Meredith, Marsha J., "Introducing Parallel Computing into the Undergraduate Computer Science Curriculum: a Progress Report", The Papers of the Twenty-Third SIGCSE Technical Symposium on Computer Science Education, Kansas City, Missouri, SIGCSE Bulletin, Volume 24, Number 1, March 1992, pages 187-191.

[Miller, 1994] Miller, Russ, "The Status of Parallel Processing Education", Special Report, Computer Volume 27, Number 8, August 1994, pages 40-43.

[Millersville, 1999] Department of Computer Science, Millersville University, Millersville, Pennsylvania, http://iml.millersv.edu/cs360.html.

[Monash, 1998] Computer Science Department, Monash University, Melbourne, Australia, http://www.cs.monash.edu.au.

[NPAC, 1994] High Performance Fortran Forum, Northeast Parallel Architectures Center, Syracuse University, Syracuse, New York, http://www.npac.syr.edu/hpfa/hpff2/html/.

[Olsson, 1995] Olsson, Ronald A., and McNamee, Carole M., "Tools for Teaching CCRs, Monitors, and CSP Concurrent Programming Concepts", SIGCSE Bulletin, Volume 27, Number 2 June 1995, pages 31-40.

[Olszewski, 1993] Olszewski, Jacek, "CSP Laboratory", The Papers of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education, Indianapolis, Indiana, SIGCSE Bulletin, Volume 25, Number 1 March 1993, pages 91-95.

[Oracle, 1992] Oracle Corporation, Oracle7 Server Concepts Manual, Part Number 6693-70-1292, December 1992, pages 25-1 - 25-5.

[Osborne, 1993] Osborne, Martin, and Johnson, James L., "An Only Undergraduate Course in Object-Oriented Technology", The Papers of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education, Indianapolis, Indiana, SIGCSE Bulletin, Volume 25, Number 1 March 1993, pages 101-106.

[Ostle, 1963] Ostle, Bernard, Statistics In Research, Iowa State University Press, Ames, Iowa, 1963.

[Pacheco, 1997] Pacheco, Peter, "Using MPI to Teach Parallel Computing", Conference Proceedings, Second Forum on Parallel Computing Curricula, Newport Rhode Island, June 1997, http://www.cs.dartmouth.edu/FPCC/papers/pacheco.html.

[Papoulis, 1991 Papoulis, Athanasios, Probability, Random Variables, and Stochastics Processes, McGraw-Hill, New York, New York, 1991.

[Paralogic, 1996] Paralogic Inc., Bethlehem, Pennsylvania, 1996, http://www.plogic.com/para-per.html.

[Penn, 1998] School of Engineering and Applied Science, University of Pennsylvania, http://www.seas.upenn.edu/class.html.

[Perrott, 1987] Perrott, R. H., Parallel Programming, Addison-Wesley, Wokingham, England, 1987.

[Queens, 1998] Department of Computer Science, The Queen's University of Belfast, Belfast, Northern Ireland, http://www.cs.qub.ac.uk/CS.

[Reading, 1997] Department of Computer Science, The University of Reading, http://www.cs.rdg.ac.uk/cs/teaching/units.html.

[Reid, 1994] Reid, Richard J., "Introductory Object-Oriented Programming Projects Using Simulation And Animation", Computers In Education, Volume 4, Number 1, January-March 1994, pages 11-15.

[Rensselaer, 1998] Computer Science Department, Rensselaer University, http://www.cs.rpi.edu/undergrad/BS.html.

[Reynolds, 1996] Reynolds, Charles, and Fox, Christopher, "Requirements For A Computer Science Curriculum Emphasizing Information Technology Subject Area: Curriculum Issues", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 247-251.

[Schaller, 1995] Schaller, Nan C., Kitchen, Andrew T., "Experiences in Teaching Parallel Computing - Five Years Later", SIGCSE Bulletin, Volume 27, Number 3 September 1995, pages 15-20.

[Schwartz, 1988] Schwartz, Mischa, Telecommunications Networks: Protocols, Modeling and Analysis, Addison-Wesley, Reading, Massachusetts, 1988.

[Sebesta, 1989] Sebesta, Robert W., Concepts of Programming Languages, Benjamin / Cummings, Redwood City, California, 1989.

[Sisal, 1996] SISAL, Computer Research Group at Lawrence Livermore National Laboratory, November 1996, ftp://sisal.llnl.gov/pub/sisal.

[Smith, 1996] Smith, Harry F., and Plusnick, Patrick, "Image Processing As An Example Of Parallelism Applied To Graphics", The Papers of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, SIGCSE Bulletin, Volume 28, Number 1, March 1996, pages 363-367.

[Toll, 1995] Toll, William E., "Decision Points In The Introduction Of Parallel Processing Into The Undergraduate Curriculum", The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, SIGCSE Bulletin, Volume 27, Number 1 March 1995, pages 136-140.

[Toll, 1997] Toll, William E., "Parallel Processing Integration in the Computer Science Curriculum: A Question of Balance", Conference Proceedings, Second Forum on Parallel Computing Curricula, Newport Rhode Island, June 1997, http://www.cs.dartmouth.edu/FPCC/papers/Toll/toll.html.

[Trono, 1994] Trono, John A. "A New Exercise in Concurrency", SIGCSE Bulletin, Volume 26, Number 3, September 1994, pages 8-10.

[Turner, et al, 1991] ACM/IEEE-CS Joint Curriculum Task Force, "Computing Curricula 1991", Communications of the ACM, Volume 34, Number 6, June 1991, pages 68-84.

[VanScoy, 1994] VanScoy, Frances L., "Power Point Documents in Support of an Ada-Based CS 1 Course", Asset Source for Software Engineering Technology (ASSET), 1994, http://www.asset.com/WSRD/abstracts/ABSTRACT_813.html.

[Villanova, 1998] Computer Sciences Department, Villanova University, Philadelphia, Pennsylvania, http://www.csc.vill.edu/courses.shtml.

[Wake, 1998] Department of Mathematics and Computer Science, Wake Forest University, http://www.mthcsc.wfu.edu/info.html.

[Washington, 1998] Department of Computer Science, School of Engineering and Applied Science, Washington University in St. Louis, http://www.wustl.edu/~klg/cs101/home.html.

[Washington, 1999] Department of Computer Science, School of Engineering and Applied Science, Washington University in St. Louis, http://www.cs.wustl.edu/~sandholm/cs520.html.

[Wein, 1997] Wein, Joel, "An Active Learning Approach to Teaching Parallel Algorithms", Conference Proceedings, Second Forum on Parallel Computing Curricula, Newport Rhode Island, June 1997, http://www.cs.dartmouth.edu/FPCC/papers/wein.html.

[Wheeler, 1996] Wheeler, David A., Ada 95 The Lovelace Tutorial, Springer-Verlag, New York, New York, 1996.

[Yang, 1995] Yang, Lan, and Jin, Lan, "Integrating Parallel Algorithm Design With Parallel Machine Models", The Papers of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, SIGCSE Bulletin, Volume 27, Number 1 March 1995, pages 131-135.

[Yue, 1991] Yue, Kwok-bun, "Dining Philosophers Revisited, Again", SIGCSE Bulletin, Volume 23, Number 2, June 1991, pages 60-64.

[Yue, 1994] Yue, Kwok-bun, "An Undergraduate Course in Concurrent Programming Using Ada", SIGCSE Bulletin, Volume 26, Number 4, December 1994, pages 59-62.

APPENDIX A. DEFINITIONS

Concurrent -- "two or more things can be done independently, but not necessarily on different processors [one or more processors]" [Paralogic, 1996]

Concurrent Program -- "a set of ordinary sequential programs which are executed in abstract parallelism" [Ben-Ari, 1990]

Dependent Variable -- "the data that the researcher analyzes" [Huck, 1974]

External Validity -- "refers to representativeness or generalizability" of a study, "the extent [] the results of a study can be generalized to other populations, settings, treatments, or measurement variables" [Huck, 1974]

Independent Variable -- that which "is manipulated by the researcher" [Huck, 1996] in a study

Internal Validity -- "refers to casual relationships" between the independent and dependent variables such that a "researcher [can] infer a cause-and-effect relationship" [Huck, 1974]

Novice (as in student programmer) -- student entering their first computer science programming course who has received no or inconsequential exposure to collegiate programming, professional programming, and tool creation programming.

Parallel -- "two or more things can be done independently at the same time on different processors" [Paralogic, 1996]

Parallel Computer -- "a collection of processors, typically of the same type, interconnected in a certain fashion to allow the coordination of their activities and the exchange of data" [Jaja, 1992]

Parallel Program -- a concurrent program designed for execution on parallel hardware (implies more than one processor)

Semaphore -- "an integer-valued variable which can take only non-negative values" [Ben-Ari, 1990]. In this experiment semaphores are modeled as a collection of distinct states.

APPENDIX B. SYLLABUS AND INSTRUCTIONS

The syllabus and instructions for both the control and treatment groups are the same with the following exceptions:

- Schedule of instruction for the last eight weeks of each group
- Dates instruction given

The syllabus and instructions given in this section are extracts of the Web pages for the Fall 1998 class, a treatment group. Therefore, the schedule of instruction for the control group, Fall 1997, is given on the next page (preceding the Web pages). As 20 March 1999, the complete set of Web pages for the experiment were Web accessible at the URLs below:

- Control group -- http://www.seas.gwu.edu/classes/csci51/fall97/index.html
- Treatment group one -- http://www.seas.gwu.edu/classes/csci51/spring98/index.html
- Treatment group two -- http://www.seas.gwu.edu/classes/csci51/fall98/index.html

The syllabus and instructions on the following pages are in Web page format. The page format for this dissertation is different than the original Web page format of the text. Therefore, the appearance of the text is different when viewed using a browser.

Michael Feldman wrote the vast majority of this material. Dr. Feldman taught the class prior to the experiment. The investigator made modifications to the material.

Excluded from this section are the Unix and compiler instruction materials that were presented to the students at the beginning of the course.

Class Schedule, Control Group, Fall 1997

8/26-8/28	Week 1	Chapter 1	Introduction
9/2-9/4	Week 2	Chapter 2	Introduction to Programming with Ada 95
9/9-9/11	Week 3	Chapter 3	Introduction to Design; Enumeration Types; the Spider
9/16-9/18	Week 4	Chapter 3	Using Packages
9/23-9/25	Week 5	Chapter 4	Decision Statements
9/30-10/2	Week 6	Chapter 4	Writing Functions and Packages
10/7-10/9	Week 7	Chapter 5	Counting Loops; Introduction to External Files
10/14	Week 8	———	Review for Midterm Exam
10/16	——	———	MIDTERM EXAM - covers Chapters 1-5
10/21-10/23	Week 9	Chapter 6	General Loops; Exception Handling
10/28-10/30	Week 10	Chapter 6	Writing Procedures; Parameter Modes; Robust Input
11/4-11/6	Week 11	Chapter 7	Case Statements; Math Library; Random Numbers
11/11-11/13	Week 12	Chapter 8	Composite Types: Records
11/18-11/20	Week 13	Chapter 8	Composite Types: Arrays
11/25	Week 14	Chapter 9	A Systematic View of Strings and Files
12/2-12/4	Week 15	Chapter 9	Strings and Files, continued
12/9	Week 16	———	Review for Final Exam (in Reading Period)
12/16	——	———	FINAL EXAM - covers Chapters 1-9

READ THIS AND KEEP IT HANDY! IT IS VERY IMPORTANT!

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science

CSci 51 -- Introduction to Computing -- Fall 1998

Lecture Times: Tuesday/Thursday, 5:45 - 7:00 PM
Lab/Recitation Times: 50 minutes, various times
Chester B. Lund
Phone: 703-610-2954 (office)
Electronic Mail: clund@seas.gwu.edu
Office hours: After class on Thursday
Optional Extra Help Classes: Friday 5:45 - 7:00 PM (Starting October 23)

Required Textbook:

Feldman and Koffman, *Ada 95: Problem Solving and Program Design (2nd edition).* Addison Wesley, 1996.
ISBN 0-201-87009-6 (textbook alone);
ISBN 0-201-30485-6 (textbook bundled with Aonix ObjectAda Special Edition CD-ROM).

Read each chapter during the week it is assigned. The book discusses much more than I can cover in class, and I will cover things not in the book. You will get much more out of the class if you are well-prepared.

Course Outline:

8/25-8-27	Week 1	Chapter 1	Introduction
9/1-9/3	Week 2	Chapter 2	Introduction to Programming with Ada 95
9/8-9/10	Week 3	Chapter 3	Introduction to Design; Enumeration Types; the Spider
9/15	Week 4	Chapter 3	Using Packages
9/17	——	Chapter 4	Decision Statements
9/22-9/24	Week 5	Chapter 4	Writing Functions and Packages
9/29-10/1	Week 6	Chapter 5	Counting Loops; Introduction to External Files
10/6-10/8	Week 7	———	Computer Architecture; Models of Computation; Mid-term Review
10/13	Week 8	Chapter 6	Exception Handling; General Loops
10/15	——	———	MIDTERM EXAM - covers Chapters 1-5
10/20-10/22	Week 9	Chapter 6	Exception Handling; Writing Procedures; Parameter Modes; Tasks *
10/27-10/29	Week 10	Chapter 8	Composite Types: Arrays
11/3-11/5	Week 11	Chapter 8	Composite Types: Records; Concurrent Modeling and Constructs *
11/10-11/12	Week 12	Chapter 7	Case Statements; Math Library; Message Passing *
11/17-11/19	Week 13	———	More on Data Types; Last Project, Number 9; Concepts of Concurrent Programming *
11/24	Week 14	Chapter 9	Systematic View of Strings and Files (started); Shared Memory *
11/26	Week 14	———	Thanksgiving Day Holiday
12/1-12/3	Week 15	Chapter 9	Systematic View of Strings and Files (concluded); Efficiency and Amdahl's Law *
12/8	Week 16	———	Final Exam Review (Last Class)
12/15	——	———	FINAL EXAM (Tentative Date)

Supplimental materials:

Topics related to concurrency are marked in the course outline with an "*". Supplimental materials about concurrency are provided to the student beginning with course week 9.

Attendance:

Attendance is *required* in both lecture and lab, and important work will be done in both. If you have an unavoidable need to be absent, you do not need special permission, but *you are responsible for the work covered even if you are not in class.*

Office hours:

Office hours are after class on Thursday. Office hours, both the lecturer's and the lab instructor's, are an important way for you to get help or to discuss anything you have on your mind. We are there to help you; that is an important part of our job. Please make good use of these hours; you are cheating yourself if you do not.

Electronic mail (e-mail):

Part of your first-week assignment is to learn to write and send e-mail. Both the lecturer and the lab instructor read e-mail at least once a day; you are sure to get a quick response if you make good use of this system. If you have never used e-mail before, you are in for a treat--it is fun!

Programming Projects:

I will assign a project every week, which will be due the following week. Each project will build on the work done in previous projects, so it is in your interest to keep up with the project work. There will be about 10 projects.

Each project will be graded on a 0-20 point basis. An incomplete submission is better than none; you will get credit where credit is due. I will accept late projects, subject to a "late fee" of 4 points per week of lateness. Each project is due *at the start of the class* on the due date; projects submitted after the lecture has begun will be counted as one week late.

Examinations:

There will be a two-hour midterm and a two-hour final exam, both strictly timed. These will be open-book, open-notes exams. If you are coming to class regularly, and keeping up with the reading and the projects, the exams should not be difficult for you. Exams will require a mixture of reading and interpreting short program segments, writing short program segments, and short "essay" questions.

Quizzes:

There may be one or more unnanounced quizzes during the lecture period. The best way to avoid unannounced quizzes is to come to class regularly, participate actively, and keep up with the reading.

Grading:

Your semester grade will be calculated as follows:

- Midterm Exam 25%
- Final Exam 39%
- Projects 36% (about 4% per project / last project counts double)

The last project, Project 9, and the Final Exam together are 47% of the total grade; they both occur at the semester's end.

I will eliminate the lowest project grade for each student from Project #1 thru Project #8 only (Project #9 will not be eliminated). However, I will not give a semester grade that is more than one grade higher than the project average. That is, if your project average is a "C", you will not be able to get a semester grade higher than "B". Quizzes, if any, will be counted in according to how many there are.

I keep grade records strictly "by the numbers"; any conversion to letter grades, and any "curving" of the final grade results, is done only at the very end of the semester, when I have all the semester data.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Introduction to Computing

Preparation and Grading of Programming Projects: the Importance of Professionalism

Introduction

This course is taught under the assumption that the students have no previous experience with programming or program design. Projects are therefore of a difficulty and complexity that a beginner can handle.

It is, however, important for you to realize that one of the purposes of this course is to help you begin your preparation for a profession. If you continue in computing courses, you will find that your experience here will serve you well. Even if you never take another course in computing, you will have learned much about the kinds of techniques professionals use and especially about the courtesy that they are expected to show to others needing to read and understand their work.

The conventional scheme used in universities to assess a student's accomplishments is the grade. We therefore use a grading system to assess your success not only in writing a program that "works," but also in completing a project that meets the standards of professionalism we have set. Meeting these standards is neither especially difficult, nor especially burdensome in terms of your time.

We hope that you will realize that we are not just being bureaucratic, and that you will understand the benefits of what we are asking you to do. In any event, we are giving you, through the grading system, a tangible incentive to do it.

Project Preparation

In developing a project, you will use the Case Study form as illustrated many times in the Feldman/Koffman book. Specifically, you need to prepare and submit all the sections of the Case Study: Problem Statement, Analysis, Data Requirements, Algorithm with refinements, Test Plan, Coding, and Testing. It is acceptable to attach a copy of the project handout and mark it as Problem Statement.

Programming projects are to be submitted in paper form, because the graders will need to make notes on them. Projects will be turned in at the start of lecture and returned during lab. Be sure your name and lab section are on each piece, in case they get accidentally separated. Put the pieces together with a paper clip, not a staple.

Your submission must contain:

- Your Case Study document, printed from a computer or *neatly* hand-written. The document must show, on the first page, your name, e-mail address, lab section, project number, date submitted).
- One or more listing (.lss/.lsb) files, according to the project assignment sheet.
- One or more compile/link/execute scripts, according to the assignment sheet. Run the program with enough test data that you can show that it works as advertised. Choose test data carefully!
- Each program source file must have, at the beginning of the program, a "banner comment" in the following form:

```
---------------------------------------------------------------
-- Name: Elvis Presley
-- E-mail Address: elvis@seas.gwu.edu
-- Lab Section: CSci 51-30
--
-- Project #1
-- Date: Feb. 1, 1996
--
-- Brief Project Description:
--
-- This project requires the computation of the American
-- price for a quantity of food purchased in Canada,
-- where the food is purchased in kilos and the price
-- is charged in Canadian dollars.
--
-------------------------------------------------------------
```

Indentation and comments in the program should follow the style of those in the book, and reflect the major steps of the algorithm as given in the Design Document. Use comments to document each identifier, e.g.

```
Quantity: Float; -- input - weight of food in kilos
USPrice: Float; -- output - US price of the food
```

Project Grading

Your grade will be assigned on the basis of 20 points; partial credit is always given where appropriate.

- up to 6 points, or 30%, are given for the Analysis, Data Requirements, and Algorithm/Refinements part of your Case Study document.
- up to 4 points, or 20%, are given for the Testing part of your document, in which you will describe and justify the test cases you will use to test your program. The grader will read your test plan and compare it with the actual test shown in your script.

- up to 4 points, or 20%, are given for the structure and style of your program, including indentation, consistent capitalization, blank lines for readability, appropriate level of comments, etc.
- up to 6 points, or 30%, are given for the correctness of your program as shown by the source listing and test run(s).

Note that a complete and correct design and test plan will earn 50% of the grade, even if you do not complete the program, and that a correctly developed and formatted program will earn an additional 20%, even if it does not work. This weighting of grades is quite intentional, and is typical of modern computing courses. It is also typical in industry for the coding part of a project to absorb only 30% of the resources.

Schedule

Your project is due on the assigned date *at the beginning of class . A project brought in during the class will normally be counted as late.*

Many teachers refuse to accept late projects at all, but in this course you are permitted to turn in projects at any time. Your grade on the project will, however, be reduced by 4 points, or 20%, for each week it is late.

Note that if you are extremely busy, you can buy an additional week of time for a 4-point price. You do not need permission to do so; just turn it in late. The "lateness fee" will be waived *only* for documented medical situations or other unusual circumstances. "The computer was down" is *not* an unusual circumstance; our response will always be "the computer often goes down; you should have allowed yourself more time."

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science

CSci 51 -- Introduction to Computing

PLAGIARISM AND COLLABORATION ON PROGRAMMING PROJECTS

The project work you turn in must represent your own work and not the work of someone else. On the other hand, it is unreasonable to expect that you will work in a complete vacuum, without ever speaking to a classmate. The purpose of this note is to give you some guidance about the areas in which it is appropriate to discuss project phases with your classmates. Violating these guidelines may result in a charge of academic dishonesty.

Plagiarism

The term *plagiarism* describes an attempt to claim work as your own, which you have copied from another person, whether that other person knows about it or not. In a class like this, plagiarism includes copying program code, data, documentation, etc. Plagiarism is simply not allowed. If you submit another student's work as your own, you will be charged with a violation of the GW Academic Integrity Code.

Collaboration

Collaboration is defined as two or more students working together on a phase of a project. Working together does not mean that one student does the work and the other student just copies it! Collaboration is allowed under certain conditions, as long as you are honest about it.

You are taking this class to learn important fundamental things about computing, and I must give you a grade that fairly represents what I think you've learned. Therefore, I need to know that your work is *your* work, so I need to limit the collaboration somewhat. For purposes of projects in this class, here are some guidelines as to which phases of a project are appropriate for collaboration, and which are inappropriate.

OK	Preliminary Analysis of Problem
OK	Developing the Algorithm
NO	Developing a Test Plan
NO	Coding in the Programming Language
NO	Proof-reading the program before compiling
OK	Interpreting errors at compilation time
OK	Interpreting errors at execution time
NO	Writing Up the Case Study

"Truth in Advertising"

If you collaborate with another student, for each permitted phase of the project, you must give your "partner's" name in your documentation for that phase.

Save Your Projects!

You are required to save all your projects until the end of the semester, after grades have been reported. Be prepared to re-submit these to the instructor if he or she asks you to do so.

Protect Yourself

If you suspect that another student is misusing your work (for example, one of your printouts disappeared), report this immediately to the instructor, to protect yourself against a charge of plagiarism if your work is copied by another student.
Read the University Academic Integrity Code carefully.

APPENDIX C. STUDENT ACKNOWLEDGEMENT FORM

Acknowledgement Form
===================

This form has been reformatted to fit a single page in this document.

Introduction

This offering of CSci 51 includes some experimental material that is part of a doctoral research study. The information resulting from your participation in this class will be used to help the researchers understand (1) how well students learn certain material and (2) how quickly students learn certain material. By your participation, you will be contributing to computer science education research.

Procedure

Your involvement in this study includes attending class, doing homework, and taking examinations. There are no additional requirements imposed on you. However, as a part of the research, your compilation and linking of Ada programs is being recorded.

Risk and Benefits

The only risk to you is that some of your computer usage is being recorded. The benefit to you is that the instructor can assist you with your homework by accessing your recorded computer usage. The recorded information (compiling and linking of Ada programs) is NOT used in computing your grade.

Confidentially

Your individual data can and will be discussed only among the researchers. If the results are made available to any other persons, your identity will in no way be made known, or associated with your results.

Rights

The recorded material (compiling and linking of Ada programs) is NOT used in computing your grade. The recording process is automatic for all students, so the only effective way to withdraw from the recording process is to withdraw from the course.

Contacts and Questions

The researchers involved in this study are: Chester Lund and Dr. Michael Feldman of The George Washington University. If you have any questions regarding this consent form or this study, please contact Chester Lund at clund@seas.gwu.edu.

We appreciate your participation in this study.

My signature below indicates I have read the above information and agree to participate in this study.

Your Name (printed) ________________________________

Your Name (signed) ________________________________

Student ID ____________________ Date _______________

APPENDIX D. PROJECT ASSIGNMENTS, CONTROL GROUP

The project assignments for the control group and the treatment groups were prepared as Web pages. The page format for this dissertation is different than the original Web page format of the text. Therefore, the appearance of the text is different when viewed using a browser. Further, in order to make selected information fit on a single page (as it did in the project assignment handouts), changes to font size and paragraph spacing have been made.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #0
Due Date: 5 PM, Friday, August 29, 1997

This initial "mini-project" will not be graded, but you must do it on time. The purpose of is to get you onto the computer and using the editor and e-mail system to compose a message. *You will get assistance for this during the first lab meetings on Friday. Of course, it is acceptable to do it earlier.*

1. Get your Unix account set up at the School of Engineering and Applied Science Computing Facility (SEASCF), 4th floor, Tompkins Hall. Do this immediately after class.
2. Log in to felix. Type the following to get your account set up:

   ```
   . ~csada/setup-51
   ```

   This must be typed in *exactly* as you see it, including the initial dot and the "tilde." If you have files in your file system from other courses, the setup script will leave those untouched. The setup process will, however, modify your `.kshrc` and `.profile` files to give you read-only access to the shared directories like `programs51` and `info`. You will also start receiving a "news flash" from the professor and lab instructor every time you log in.
3. Once the setup is done, copy the file `info/survey.txt` into your file system:

   ```
   cp info/survey.txt ~
   ```

4. Bring the survey form into the editor:

   ```
   vi survey.txt
   ```

5. Use the editor to fill in the requested information on the survey form, then save the survey back in your file system.

6. Send the survey as an e-mail message to the prof and lab instructor:

   ```
   elm clund <survey.txt
   elm koeun <survey.txt
   ```

7. Log off.

We will both acknowledge receipt by sending a "thank you" message back to you. You will then know that we can communicate with each other.

Please do this project before the deadline.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #1
Due Date: start of class, Thursday, September 11, 1997

The purpose of this project is to help you become familiar with the GNAT compilation system and the editor on `felix` (the Sun/Solaris server in the SEAS Computing Facility).

The first part will help you to become familiar with the compilation system.

Part 1:

Compile, link, and execute the programs from Chapter 2. Note that all the programs in the book are available to you in the programs51 subdirectory. Each program's file name is the same as its program name, except that the file name is in lower case. (Example: the program `Distance`, Program 2.5, is in the file `distance.adb`.) Choose *one* of the last few programs to compile, link, and execute with turnin running.

You will find that all the programs compile without errors *except* Program 2.10. Print out and turn in the listing file from Program 2.10, showing the error messages given by the compiler.
The second part will help you learn to use the software development method as discussed in Chapter 2.

Part 2:

Problem: You are taking a vacation in the beautiful country of LaLa Land. You rent a car there, and you're driving on the highway. Then you notice that the distances are measured in *furlongs.* Each furlong is 1/8 mile (really!).

Not only that, but speeds are measured in *furlongs per fortnight (fpf).* Each fortnight is two weeks or 14 days (really). The highway speed limits are, of course, given in these units.

Worse still, the speedometers on the cars show miles per hour (mph) as is used here in the United States. So how do you know if you are exceeding the speed limit?

What you need is a quick calculator program, so that if your speedometer reads, for example, 65 mph, you can input this number and the calculator will tell you immediately what your speed is in fpf, so you can compare it with the speed limit signs.

Your job is to design and code such a program, testing it with some typical highway speeds. For details on what to turn in, please read the handout *Preparation and Grading of Programming Projects: the Importance of Professionalism.*

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #2
Due Date: start of class, 18 September 1997

The purpose of this project is to help you begin to use standard and class-specific packages. Everything you need is in Chapters 1-3; you need not, and should not, use any "extra" statements or anything from later chapters. Part 1 is just a "spider program" for which you need only to turn in a listing file. No Case Study is needed. Part 2 is a word problem requiring a Case Study.

Part 1:

First compile the `Screen` and `Spider` packages:

```
gcompile screen.ads
gcompile screen.adb
gcompile spider.ads
gcompile spider.adb
```

Now write and test a program that instructs the spider to draw a pattern in the shape of a triangle, that is,

```
X
X X
X   X
XXXXXX
```

Hints: Start the spider facing West, draw the top line, etc. Also note that you can get the spider to draw a "blank" by changing its color to black.

Part 2:

Here is the specification for a `Min_Max` package (we'll look at the body in Chapter 4):

```
PACKAGE Min_Max IS
------------------------------------------------------------
--| specifications of functions provided by Min_Max package
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: July 1995
------------------------------------------------------------

  FUNCTION Minimum (Value1, Value2: Integer) RETURN Integer;
  -- Pre: Value1 and Value2 have been assigned values
  -- Post: Returns the smaller of the two input values
  FUNCTION Maximum (Value1, Value2: Integer) RETURN Integer;
  -- Pre: Value1 and Value2 have been assigned values
  -- Post: Returns the larger of the two input values

END Min_Max;
```

First compile the `Min_Max` package:

```
gcompile min_max.ads

gcompile min_max.adb
```

Now develop and test a program that finds the largest, smallest, and average of *six* integers read from the terminal.

Hint: Declare variables for the sum, current smallest, and current largest values. You do not need to store all six values; read them in one at a time, using the `Minimum` and `Maximum` functions from `Min_Max` to compare each new value to the current smallest and largest.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #3
Due Date: start of class, 25 September 1997

The purpose of this project is to help you become more familiar with the basics of Ada and especially the IF statement. You should be reading Chapter 4 by now; everything you need to do this project will be in chapters 1-4.

Many states have recently raised the speed limits on some of their highways. The state police would like to collect statistical data on the actual speeds of cars under the new laws, and have hired us to develop a computer program to help them. In the next few projects, we will design and build such a program. The first step, Project 3, is to write and test a function to classify a speed into one of the following classifications:

- Class 1: `0 < speed<= 45` miles per hour (m.p.h.)
- Class 2: `45 < speed <= 55`
- Class 3: `55 < speed <= 65`
- Class 4: `65 < speed <= 75`
- Class 5: `75 > speed`

For this project, declare the function inside a main program, by analogy with Program 4.6. The main program's declaration part should contain these declarations:

- an enumeration type to define the classes, as follows:
  `TYPE SpeedClasses IS (Class1, Class2, Class3, Class4, Class5);`
- a subtype to specify the realistic range of speeds on the highway:
  `SUBTYPE Speeds IS Natural RANGE 0..130;`
- the function specification:
  `FUNCTION Classify (Speed: Speeds) RETURN SpeedClasses;`
- the function body

and the main program should test the function according to a test plan you design. For each test, input a speed from the user, call the function to classify it, and display the classification using an instantiation of `Ada.Text_IO.Enumeration_IO`.

As usual, submit the Case Study document, a printout of the listing file, and a test run executed with turnin.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997

Project #4

Due Date: start of class, 7 October 1997

The goal of this project is to give you some more experience in working with packages and enumeration types. The project depends upon material in Chapters 3 and 4.

You are to write a program that prompts the user for a date, then displays the day of the week on which that date occurs (e.g. `September 24, 1992 is on a Thursday`).

The month will be entered as a 3-letter enumeration literal (Jan, Feb, etc). The month and the day of the week should be displayed as above, with the *only* the first letter in uppercase.

The day will be entered as an integer in the predefined subtype `Ada.Calendar.Day_Number`; the year will be entered as an integer in the predefined subtype `Ada.Calendar.Year_Number`.

To find the day of the week, use the package `DayWeek` in the programs directory. The specification is in `dayweek.ads`; the body is in `dayweek.adb`. You will need to compile the specification, then the body, of this package in order to use it. This package is not in the book, but it is online in the programs directory. A simple program, `DayTest`, that demonstrates the package is in `daytest.adb`.

Of course `Ada.Calendar` does not need to be compiled; it is part of the Ada system libraries.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #5
Due Date: start of class, 16 October 1997

This project depends upon Chapters 1-5, and provides experience with writing loops, and with file redirection.

Part 1:

Write a `Spider` program `Checker_Board`, which causes the spider to draw a checkboard pattern on the screen, like

```
X X X X
 X X X X
X X X X
 X X X X
```

Use loops wherever appropriate.

Part 2:

Revise `Project 3` so that a series of 30 speeds is processed as follows.

Instead of reading the speeds from the keyboard, create a file of 30 speeds -- call it, say, `speeds.dat` -- with an editor, one speed per line in the file, and use input redirection to read and process the 20 speeds. In addition to classifying each speed and displaying its classification, find the minimum, maximum, and average speeds, and the number of speeds in each class.

If your program is called `speeds.exe`, using input redirection you can process the speeds by

```
gexecute speeds.exe <speeds.dat
```

This program will be much easier to do correctly if you design the algorithm carefully before starting to code!

As usual, submit the Case Study document, a printout of the listing file, and a test run executed with turnin.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #6
Due Date: start of class, 28 October 1997

The purpose of this project is to help you get more familiar with loops and packages, and with creating and writing output text files.

Your project is to use the DayWeek package to create a disk file called `calendar.dat`. Your program will prompt the user for a starting month/year pair and an ending month/year pair, then write, into the external file, a line for each day in the given range. Enumeration types are to be used for the month and day abbreviations. For example, if the user enters

```
APR 1993
MAY 1993
```

as the starting and ending months, the file will contain, after the program is done, 61 lines. The first line will say

```
THU APR 1 1993
```

and the last line will say

```
FRI MAY 31 1993
```

Of course, if the user enters different years for the starting and ending values, the file will be much larger! Design your algorithm carefully before even *thinking* about code.

Notes about Files:

To create a file into which your program will write, your program will contain a variable declaration

```
MyFile: Ada.Text_IO.File_Type;
```

To associate the file name with a file in the file system, include this statement after the `BEGIN` of your program:

```
Ada.Text_IO.Create
  (File=>MyFile,Mode=>Text_IO.Out_File,Name=>"calendar.dat");
```

To write an integer value into this file, use the file-oriented `Ada.Text_IO` operations, for example, if Today is an integer variable, use

```
Ada.Integer_Text_IO.Put(File=>MyFile,Item=>Today,Width=>2);
```

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #7
Due Date: start of class, 6 November 1997

This project will give you some practice with exceptions and exception loops, and with modifying packages. Everything you need for this project is in Chapter 6 and earlier chapters. In this project, you will modify the `DayWeek` package, and the program `DayTest`, so that the date handling is robust ("bulletproof").

Part 1:

Modify the body of `DayWeek`. Use exception handling to check each value as it is entered, to be sure that the month is in the range `Jan..Dec`, the day is in the range of `Ada.Calendar.Day_Number`, and the year is in the range `Ada.Calendar.Year_Number`.

Part 2:

Currently, the function `DayWeek.DayOfWeek` returns a meaningless value if it receives a bad combination of inputs (e.g., April 31, or February 29 of a non-leap year). As it happens, `Ada.Calendar` provides a built-in way to check the validity of a date. The function `Ada.Calendar.Time_Of` has the specification

```
  FUNCTION Time_Of (Year : Year_Number;
                    Month : Month_Number;
                    Day : Day_Number;
                    Seconds : Day_Duration:=0.0) RETURN Time;
```

and returns a value of type `Ada.Calendar.Time`, if and only if the month/day/year combination forms a valid date. Otherwise, `Time_Of` raises the exception `Ada.Calendar.Time_Error`. Let's use this to check the validity of a date. First we define our own exception: change the specification of `DayWeek`, to include the line

```
Date_Error: EXCEPTION;
```

Next, modify the function `DayOfWeek`. Declare a variable of type `Ada.Calendar.Time`, for example

```
TestDate: Ada.Calendar.Time;
```

then call `Ada.Calendar.Time_Of` with the 3 parameters received by `DayOfWeek`.

```
Test.Date :=
  Ada.Calendar.Time_Of(Month => Month, Day => Day, Year => Year);
```

put an exception handler in the body of DayOfWeek that looks like

```
EXCEPTION
  WHEN Ada.Calendar.Time_Error =>
    RAISE Date_Error;
```

Finally, add to your main program (your modified DayTest) a handler for DayWeek.Date_Error. Also change the program so that the user is requested to input *five* valid dates.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #8
Due Date: start of class, 13 November 1997

This project involves an extension of the speed-monitoring program you did for Project 5. It depends on material from Chapter 7, especially random-number generation as shown in Random Numbers (Program 7.6) and Drunken Spider (Program 7.7), and the calendar operations in Time of Day (Program 7.1).

This program will be a simulation of a speed-monitoring session on the Beltway. Instead of creating a file full of data as in Project 5, consider that on the Beltway, cars pass a given point at random time intervals, going at more-or-less random speeds. With the Beltway speed limit set at 55, we can assume that almost all speeds lie between, say, 40 and 80. And at most times of the day, cars pass the monitoring equipment every few seconds.

Part 1:

You're going to develop a package that contains the speed types from Project 5, the speed-classification function, and a new function that will deliver a random speed. If this were a real speed monitor, this function would be connected to the radar gun instead of the random number generator. The specification of this package will look like this:

```
PACKAGE Speeds IS

  TYPE SpeedClasses IS (Class1, Class2, Class3, Class4, Class5);
  SUBTYPE Speeds IS Natural RANGE 0..130;

  FUNCTION Classify (Speed: Speeds) RETURN SpeedClasses;
  -- Pre: Speed is defined
  -- Post: Returns the class of the given speed

  FUNCTION DeliverSpeed RETURN Speed;
  -- Pre: None
  -- Post: Delivers a random speed in the range 40..80

END Speeds;
```

In the body of this package, put the function bodies as usual. Also, you will need to instantiate Ada.Numerics.Discrete Random for the range 40..80. This instance, and the generator variable G, should appear at the top of the package body.

Part 2.

Now develop a main program, based on the one you wrote for Project 5, that simulates the speed monitor's operation for one hour beginning with the current time. Here you'll need a *second* instance of the random number generator, for the subtype

```
SUBTYPE ArrivalTimes IS Positive RANGE 1..60;
```

Call this instance RandomArrival, and the generator variable A. If T represents the time of day, then a new time is calculated by

```
T := T + Duration(RandomArrival.Random(Gen=>A));
```

For each new time, get a speed from DeliverSpeed, classify it, and display a line like:

```
THU NOV 7 1996 12:40:17: Speed 57, Class 3
```

At the end of about one hour in simulated time, display the speed statistics as you did in Project 5.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1997
Project #9
Due Date: start of class, December 4, 1997
Late projects accepted (subject to 20% fee) until 5 PM, December 11, 1997
in Chester Lund's faculty mailbox.

This project counts double, that is, as two projects

This project involves setting up a database for car records, similar to the one used by the State Department of Motor Vehicles. Attached is the source code for an interactive user interface program, Cars_UI, and the spec and body of a package Cars. Cars_UI, in operation, looks something like this:

```
Select one of the operations below.

C Clear the database
R Read database from disk
W Write database to disk
A Add car to database
D Delete car from database
F Find a car in the database

P Display all records in the database
Q Exit the program

Please type a command, then press Enter > a

Thank you for correct input.
```

You'll find it easiest to do the project step-by-step, as follows:

Step 1: Compile these three programs and also the package Simple_Dates. Then link Cars_UI and run it, just to see how it behaves.

Step 2: Modify Simple_Dates according to the attached package spec. Use ideas and code from Project_7.

Step 3: Implement the following operations in Cars and tie them into Cars_UI:

```
ReadDatabase
DisplayDatabase
```

We will provide a test data file `cars.dat`. You will be able to test these operations by running Cars_UI, entering an R command, then a P command.

Step 4: Now implement the operations

```
Put
WriteDatabase
AddCar
```

so you can read in the data, add a few `AddCar` transactions, then display and write the database to disk. You can then examine the disk file with `vi` or `cat`.

Step 5: Complete the other operations in the database package and tie them into Cars_UI. Here is the `Simple_Dates` interface with the desired modifications:

```
WITH Ada.Calendar;
PACKAGE Simple_Dates IS
------------------------------------------------------------------
--| Specification for package to represent calendar dates
--| in a form convenient for reading and displaying.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: April 1996
------------------------------------------------------------------

  TYPE Months IS
    (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

  TYPE Date IS PRIVATE;

  PROCEDURE Get(Item: OUT Date);
  -- Pre: None
  -- Post: Reads a date ROBUSTLY in mmm dd yyyy form, returning it in
Item

  PROCEDURE Put(Item: IN Date);
  -- Pre: Item is defined
  -- Post: Displays a date in mmm dd yyyy form

  PROCEDURE Get(File: IN Ada.Text_IO.File_Type; Item: OUT Date);
  -- Pre: None
  -- Post: Reads a date in mmm dd yyyy form from the given file,
  -- returning it in Item

  PROCEDURE Put(File: IN Ada.Text_IO.File_Type; Item: IN Date);
  -- Pre: Item is defined
  -- Post: Writes a date in mmm dd yyyy form to the given file

  FUNCTION Today RETURN Date;
  -- Pre: None
  -- Post: Returns today's date
```

```
PRIVATE

  TYPE Date IS RECORD
    Month: Months;
    Day: Ada.Calendar.Day_Number;
    Year: Ada.Calendar.Year_Number;
  END RECORD;

END Simple_Dates;
```

APPENDIX E. PROJECT ASSIGNMENTS, TREATMENT GROUPS

The project assignments for the control group and the treatment groups were prepared as Web pages. The page format for this dissertation is different than the original Web page format of the text. Therefore, the appearance of the text is different when viewed using a browser. Further, in order to make selected information fit on a single page (as it did in the project assignment handouts), changes to font size and paragraph spacing have been made.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #0
Due Date: 5 PM, Friday, 28 August 1998

This initial "mini-project" will not be graded, but you must do it on time. The purpose of is to get you onto the computer and using the editor and e-mail system to compose a message. *You will get assistance for this during the first lab meetings on Friday. Of course, it is acceptable to do it earlier.*

1. Get your Unix account set up at the School of Engineering and Applied Science Computing Facility (SEASCF), 4th floor, Tompkins Hall. Do this immediately after class.
2. Log in to felix. Type the following to get your account set up:

   ```
   . ~csada/setup-51
   ```

   This must be typed in *exactly* as you see it, including the initial dot and the "tilde." If you have files in your file system from other courses, the setup script will leave those untouched. The setup process will, however, modify your `.kshrc` and `.profile` files to give you read-only access to the shared directories like programs51 and info. You will also start receiving a "news flash" from the professor and lab instructor every time you log in.
3. Once the setup is done, copy the file `info/survey.txt` into your file system:

   ```
   cp info/survey.txt ~
   ```

4. Bring the survey form into the editor:

   ```
   vi survey.txt
   ```

5. Use the editor to fill in the requested information on the survey form, then save the survey back in your file system.

6. Send the survey as an e-mail message to the prof and lab instructor:

   ```
   elm clund <survey.txt
   elm koeun <survey.txt
   ```

7. Log off.

We will both acknowledge receipt by sending a "thank you" message back to you. You will then know that we can communicate with each other.

Please do this project before the deadline. Remember, this project must be successfully completed before any other homework assignment can be done.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998

Project #1

Due Date: start of class, Thursday, 10 September 1998

The purpose of this project is to help you become familiar with the GNAT compilation system and the editor on `felix` (the Sun/Solaris server in the SEAS Computing Facility).
The first part will help you to become familiar with the compilation system.

Part 1:

Compile, link, and execute the programs from Chapter 2. Note that all the programs in the book are available to you in the programs51 subdirectory. Each program's file name is the same as its program name, except that the file name is in lower case. (Example: the program `Distance`, Program 2.5, is in the file `distance.adb`.) Choose *one* of the last few programs to compile, link, and execute with "turnin" running.

You will find that all the programs compile without errors *except* Program 2.10. Print out and turn in the listing file from Program 2.10, showing the error messages given by the compiler. Second, *correct* the compilation errors in Program 2.10; then, print out and turn in the *corrected* listing file from Program 2.10.

The second part will help you learn to use the software development method as discussed in Chapter 2.

Part 2:

Problem: You are taking a vacation in the beautiful country of LaLa Land. You rent a car there, and you're driving on the highway. Then you notice that the distances are measured in *furlongs.* Each furlong is 1/8 mile (really!).

Not only that, but speeds are measured in *furlongs per fortnight (fpf).* Each fortnight is two weeks or 14 days (really). The highway speed limits are, of course, given in these units.
Worse still, the speedometers on the cars show miles per hour (mph) as is used here in the United States. So how do you know if you are exceeding the speed limit?

What you need is a quick calculator program, so that if your speedometer reads, for example, 65 mph, you can input this number and the calculator will tell you immediately what your speed is in fpf, so you can compare it with the speed limit signs.

Your job is to design and code such a program, testing it with some typical highway speeds. For details on what to turn in, please read the handout *Preparation and Grading of Programming Projects: the Importance of Professionalism.*

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #2
Due Date: start of class, Thursday, 17 September 1998

The purpose of this project is to help you begin to use standard and class-specific packages. Everything you need is in Chapters 1-3; you need not, and should not, use any "extra" statements or anything from later chapters. Part 1 is just a "spider program" for which you need only to turn in a listing file. No Case Study is needed. Part 2 is a word problem requiring a Case Study.

Part 1:

First compile the Screen and Spider packages:

```
gcompile screen.ads
gcompile screen.adb
gcompile spider.ads
gcompile spider.adb
```

Now write and test a program that instructs the spider to draw a pattern in the shape of a rhombus, that is,

```
    XXXXXXXXX
   X       X
  X       X
 X       X
XXXXXXXXX
```

Hints: Start the spider facing West, draw the top line, etc. Also note that you can get the spider to draw a "blank" by changing its color to black. No case study is needed for Part 1.

Part 2:

Here is the specification for a Min_Max package (we'll look at the body in Chapter 4):

```
PACKAGE Min_Max IS
------------------------------------------------------------------
--| specifications of functions provided by Min_Max package
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: July 1995
------------------------------------------------------------------

  FUNCTION Minimum (Value1, Value2: Integer) RETURN Integer;
  -- Pre: Value1 and Value2 have been assigned values
  -- Post: Returns the smaller of the two input values
  FUNCTION Maximum (Value1, Value2: Integer) RETURN Integer;
  -- Pre: Value1 and Value2 have been assigned values
  -- Post: Returns the larger of the two input values

END Min_Max;
```

First compile the Min_Max package:

```
gcompile min_max.ads

gcompile min_max.adb
```

Now develop and test a program that finds the largest, smallest, and average of ***seven*** integers read from the terminal. Remember, a case study is necessary for Part 2.

Hint: Declare variables for the sum, current smallest, and current largest values. You do not need to store all six values; read them in one at a time, using the `Minimum` and `Maximum` functions from `Min_Max` to compare each new value to the current smallest and largest.

The George Washington University
WASHINGTON DC

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #3
Due Date: start of class, Thursday, 24 September 1998

The purpose of this project is to help you become more familiar with the basics of Ada and especially the IF statement. You should be reading Chapter 4 by now; everything you need to do this project will be in chapters 1-4.

Many states have recently raised the speed limits on some of their highways. The state police would like to collect statistical data on the actual speeds of cars under the new laws, and have hired us to develop a computer program to help them. In the next few projects, we will design and build such a program. The first step, Project 3, is to write and test a function to classify a speed into one of the following classifications:

- Class 1: `00 < speed <= 25` miles per hour (m.p.h.)
- Class 2: `25 < speed <= 35`
- Class 3: `35 < speed <= 50`
- Class 4: `50 < speed <= 65`
- Class 5: `65 < speed <= 80`
- Class 6: `80 > speed`

For this project, declare the function inside a main program, by analogy with Program 4.6. The main program's declaration part should contain these declarations:

- an enumeration type to define the classes, as follows:
  ```
  TYPE SpeedClasses IS (Class1, Class2, Class3, Class4, Class5,
  Class6);
  ```
- a subtype to specify the realistic range of speeds on the highway:
  ```
  SUBTYPE Speeds IS Natural RANGE 0..120;
  ```
- the function specification:
  ```
  FUNCTION Classify (Speed: Speeds) RETURN SpeedClasses;
  ```
- the function body

and the main program should test the function according to a test plan you design. For each test, input a speed from the user, call the function to classify it, and display the classification using an instantiation of `Ada.Text_IO.Enumeration_IO`.

As usual, submit the Case Study document, a printout of the listing file, and a test run executed with turnin.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #4
Due Date: start of class, Thursday, 1 October 1998

The goal of this project is to give you some more experience in working with packages and enumeration types. The project depends upon material in Chapters 3 and 4.

You are to write a program that prompts the user for a date, then displays the day of the week on which that date occurs (e.g. Thursday is on 24 September 1992.).

The month will be entered as a 3-letter enumeration literal (Jan, Feb, etc). The month and the day of the week should be displayed as above, with the *only* the first letter in uppercase. This means that enumeration types are not to be used for output.

The day will be entered as an integer in the predefined subtype `Ada.Calendar.Day_Number`; the year will be entered as an integer in the predefined subtype `Ada.Calendar.Year_Number`.

To find the day of the week, use the package DayWeek in the programs directory. The specification is in `dayweek.ads`; the body is in `dayweek.adb`. You will need to compile the specification, then the body, of this package in order to use it. This package is not in the book, but it is online in the programs directory. A simple program, `DayTest`, that demonstrates the package is in `daytest.adb`.

Of course `Ada.Calendar` does not need to be compiled; it is part of the Ada system libraries.

As usual, submit the Case Study document, a printout of the listing file, and a test run executed with turnin.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #5
Due Date: start of class, Tuesday, 13 October 1998

This project depends upon Chapters 1-5, and provides experience with writing loops, and with file redirection.

Part 1:

Write a Spider program Checker_Board, which causes the spider to draw a checkboard pattern on the screen, like

```
      # # # #
       # # # #
      # # # #
       # # # #
```

Use loops wherever appropriate. No case study is necessary for Part 1. Once the program is working correctly, make the output of the program a comment in the program and rerun again.

Part 2:

Revise Project 3 so that a series of 25 speeds is processed as follows.

Instead of reading the speeds from the keyboard, create a file of 25 speeds -- call it, say, speeds.dat -- with an editor, one speed per line in the file, and use input redirection to read and process the 25 speeds. In addition to classifying each speed and displaying its classification, find the minimum, maximum, and average speeds, and the number of speeds in each class.

If your program is called speeds.exe, using input redirection you can process the speeds by

```
gexecute speeds.exe <input_speeds.dat
```

This program will be much easier to do correctly if you design the algorithm carefully before starting to code! Remember, that a speed of zero is not in any of these classes. Program output should be formatted as follows:

```
SPEEDS STATITSTICS PROGRAM, NAMED speeds

Class Name   Average   Maximum   Minimum   Number of Speeds
==========   =======   =======   =======   ================
CLASS1           999       999       999                999
...
CLASS6           999       999       999                999
```

As usual, submit the Case Study document, a printout of the listing file, and a test run executed with turnin.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #6
Due Date: start of class, Tuesday, 27 October 1998

The purpose of this project is to help you get more familiar with loops and packages, and to introduce PROCEDUREs and TASKs.

Part One: Using A PROCEDURE

In Part One you use the DayWeek package to create a disk file called `calendar.dat`. Your program will prompt the user for a starting month/year pair and an ending month/year pair, then display a line for each day in the given range. Enumeration types are to be used for the month and day abbreviations. For example, if the user enters

```
MAR 1997
APR 1997
```

as the starting and ending months, the file will contain, after the program is done, 61 lines. The first line will say

```
SAT MAR 1 1997
```

and the last line will say

```
WED APR 30 1997
```

Of course, if the user enters different years for the starting and ending values, the file will be much larger! Design your algorithm carefully before even *thinking* about code.

Your program should use a PROCEDURE named proc_display_month. This PROCEDURE will display all the days of a single month. See the specification below:

```
PROCEDURE proc_display_month (the_year : IN Natural; the_month : IN
Natural; the_days : IN Natural);
```

To obtain a file output of your program, use the redirection character ">" and do ***not*** use "gexecute" command. Instead, use the example given below.

```
proj_06p.exe >calendar.dat
```

Part Two: Using Two TASKs

Replace the PROCEDURE in Part One with two TASKs running concurrently. Create the TASK TYPE task_display_month. Each TASK will display all the days of a single month. See the specification below:

```
TASK TYPE task_display_month (the_year : Natural; the_month : Natural;
the_days : Natural);
```

There is only one TASK BODY definition given in the program. Use a DECLARE block, as shown below, to declare the two tasks. Also, note that the number of days per month varies from month to month.

```
DECLARE         -- Create one task per month
     task_one : task_display_month (the_year  = year_current,
                                    the_month = month_current,
                                    the_days  = total_days_in_month);
BEGIN
     task_one.start;
END;
```

In Part Two only, start the date range on an odd numbered month (e.g., Jan, Mar, etc), and end the date range with an even numbered month (e.g., Oct, Dec, etc.). Use redirection to obtain a file output of your program.

Do not do a case study for Part Two. Do a case study for Part One only.

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #7
Due Date: start of class, Thursday, 5 November 1998

The purpose of this project is to help you get more familiar with FUNCTIONs, PROCEDUREs, and TASKs and to introduce arrays.

You are going to create a PACKAGE named "Stats". This package provides the following:

```
TYPE sort_array IS ARRAY (1..100) of Float;
PROCEDURE sort    (in_array  : IN sort_array;
                   out_array : OUT sort_array;
                   in_count  : IN Natural);
FUNCTION  average (in_array : IN sort_array;
                   in_count : IN Natural) RETURN Float;
FUNCTION  median  (in_array : IN sort_array;
                   in_count : IN Natural) RETURN Float;
FUNCTION  max     (in_array : IN sort_array;
                   in_count : IN Natural) RETURN Float;
FUNCTION  min     (in_array : IN sort_array;
                   in_count : IN Natural) RETURN Float;
```

The package specification `stats.ads` is already written. In addition, part of the package body `stats.adb` is already written. Also, stored with the package is ALL the test data for this project. The files are stored in the directory named `~csada/clund/proj_07`. To copy the files to your current directory use the following Unix command: `cp ~csada/clund/proj_07/*.* .`

You will use this package to analyze student test scores. These test scores have values in the range from 0.0 to 100.0. In addition, there are some comments and junk in the input data. Your program will use exception handling to bypass the non-numeric data. Finally, your program will read its input from the file scores.dat. There are three sets of test data in the file scores.dat; your program must process all three sets of data.

Part One: Simple Program

In Part One you will complete writing the package named `stats`, and write a program that reads and processes the student grade data. Remember to follow the program organization given in class; this program minimizes the differences between Part One and Part Two. The format of the program report should be the following:

```
Comments detected in input data, comment ignored
Score detected out of range, value ignored

Set Number 1
=======================
    Students =   999
    Average  =   999.9
    Median   =   999.9
    Minimum  =   999.9
    Maximum  =   999.9
```

Part Two: Using TASKs

Revise the program created in Part One such that reading the input file and writing the report are done concurrently (by separate tasks). A simple hint for doing this is included in the file proj_samp.adb.

Do not do a case study for Part Two. Do a case study for Part One only.

Notes about Files:

To open an input file from which your program will read, your program will contain a variable declaration:

```
scores : Ada.Text_IO.File_Type;
```

To associate the file name with an input file in the file system, include this statement after the BEGIN of your program:

```
Ada.Text_IO.open (File => scores,           --  Open input file
                  Mode => Ada.Text_IO.in_file,
                  Name => "scores.dat");
```

To read a float value from this file, use the file-oriented Ada.Text_IO operations, for example, if student_grade is a float variable, use

```
Ada.Float_Text_IO.get (File = scores, Item = student_grade);
```

Remember to include into your exception handler the following statement:

```
Ada.Text_IO.skip_line (File = scores);
```

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #8
Due Date: start of class, Thursday, 12 November 1998

This project involves five students over a weekend; they are either eating or sleeping. That is it. The five students are sitting at a large round table in the student union building. They eat at the table and they sleep at the table (these student are not in condition to leave the table after eating). This project simulates their behaviour over the weekend. The weekend is 48 hours long. Use one second on the computer to represent one hour of weekend time. The simulation rules are as follows:

- There are five students -- one task per student
- There are five chopsticks
- There is only one chopstick *between* each student
- A student must "pick_up" two chopsticks to eat (only the chopsticks next to the student)
- A student must "put_down" two chopsticks after eating
- Only one student can have a chopstick at a time
- There is an huge supply of food in the center of the table (enough for the entire weekend)
- Students eat for periods of 1, 2, or 3 hours at a time (consecutively)
- Students sleep for periods of 1, 2, 3, or 4 hours at a time (consecutively)
- At no time can all five students hold just one chopstick and be waiting for a second chopstick (deadlock)
- Students must be allowed to complete their eating and sleeping cycle even after the 48 hour period is over (they had a terrible prior week)
- All task communication is by message passing only
- Tasks are given their name and number after they are running

This project uses the DELAY statement in Ada. The DELAY statement uses variables of SUBTYPE Duration as input. The output of the program is in the following format:

```
Student Version of Dining Philosophers
  This program uses five diner tasks.
  <Place other descriptive information here>

Diner 1 - Nikki is eating.
Diner 1 - Nikki finished eating.
Diner 1 - Nikki is sleeping.
Diner 1 - Nikki is finished sleeping.
.
.
Passing quit to diner tasks
Diner 1 - Nikki is finished sleeping.
Diner 1 - Nikki TASK is terminating
.
.
All tasks have quit.  Program completed.
```

The George Washington University
School of Engineering and Applied Science
Department of Electrical Engineering and Computer Science
CSci 51 -- Fall 1998
Project #9
Due Date: start of class, Thursday, 3 December 1998
Late projects accepted (subject to 20% fee) until 4 PM, 10 December 1998
in Chester Lund's faculty mailbox.

This project counts double, that is, as two projects

This for car project involves setting up a database records, similar to the one used by the State Department of Motor Vehicles. Attached is the source code for an interactive user interface program, proj_09ui, and the spec and body of a package cars.

proj_09ui, in operation, looks something like this:

```
Select one operation below:

C - Table, Commit Table
R - Table, Rollback Table
S - Table, Select all Rows
T - Table, Truncate Table
A - Row, Add One Row
D - Row, Delete One Row
F - Row, Find a Row
U - Row, Update One Row
Q - Quit the Program
Please type a command, then press Enter > C

Thank you for correct input.
```

You'll find it easiest to do the project step-by-step, as follows:

Step 1: Compile these three programs and also the package simple_dates. Then link proj_09ui and run it, just to see how it behaves.

Step 2: Modify simple_dates according to the attached package spec. Use the concepts developed in Project 7.

Step 3: Implement the following task in proj_09ui:

```
database_writer
```

The database_writer task reads from input file into the DB_Commit array and the DB_Pending array. The database_writer writes only from the DB_Commit array (every so many seconds, you chose the time interval). We will provide a test data file `cars.dat`. You will be able to test these operations by running proj_09ui program.

Step 4: Now implement the operations

```
Commit_Database     --  Copy DB_Pending to DB_Commit
Rollback_Database   --  Copy DB_Commit to DB_Pending
Select_Database     --  Display all rows in DB_Pending
```

The database_writer task and the Commit_Database procedure require synchronization -- use the task synchronize_commit for this purpose.

Step 5: Now implement the operations

```
Put
Add_Row
Find_Row
```

so you can read in the data, add a few Add_Row transactions, then display and write the database to disk. You can then examine the disk file with vi or cat. Hint: Rewrite the Procedure Select_Database to use the Procedure Put.

Step 6: Complete the other operations in the database package and tie them into proj_09ui. The Ada source code files necessary to start this project are stored in the directory "~csada/clund/proj_09". Use the Unix copy ("cp") command to obtain the files.

Here is the `Simple_Dates` interface with the desired modifications:

```
WITH Ada.Calendar;
PACKAGE Simple_Dates IS
------------------------------------------------------------------
--| Specification for package to represent calendar dates
--| in a form convenient for reading and displaying.
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: April 1996
------------------------------------------------------------------

  TYPE Months IS
    (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);

  TYPE Date IS PRIVATE;

  PROCEDURE Get(Item: OUT Date);
  -- Pre: None
  -- Post: Reads a date ROBUSTLY in mmm dd yyyy form, returning it in
Item

  PROCEDURE Put(Item: IN Date);
  -- Pre: Item is defined
  -- Post: Displays a date in mmm dd yyyy form
  PROCEDURE Get(File: IN Ada.Text_IO.File_Type; Item: OUT Date);
  -- Pre: None
  -- Post: Reads a date in mmm dd yyyy form from the given file,
  -- returning it in Item

  PROCEDURE Put(File: IN Ada.Text_IO.File_Type; Item: IN Date);
  -- Pre: Item is defined
  -- Post: Writes a date in mmm dd yyyy form to the given file

  FUNCTION Today RETURN Date;
  -- Pre: None
  -- Post: Returns today's date

PRIVATE

  TYPE Date IS RECORD
    Month: Months;
    Day: Ada.Calendar.Day_Number;
    Year: Ada.Calendar.Year_Number;
  END RECORD;

END Simple_Dates;
```

APPENDIX F. MID-TERM EXAMINATION

The mid-term examination was given in two parts:

- Part One -- examination in classroom, 75 minutes, open book, six problems
- Part Two -- examination in lab room, 45 minutes, open book using computer

The part one exam is included in the experiment. The part one examination for all three groups is included in the next 18 pages; these pages include the questions, answers, and grading instructions. The point structure for the part one exam is given below:

- Problem 1 -- 20 points
- Problem 2 -- 10 points
- Problem 3 -- 15 points
- Problem 4 -- 10 points
- Problem 5 -- 12 points
- Problem 6 -- 12 points

A sample part two exam problem is presented in the last two pages of this appendix; these two pages include the question, grading instructions, and sample program. This sample part two exam was given to the Fall 1997 group. As previously stated in this document, the part two exam is NOT included in the experiment.

The exams for the control group and the treatment groups were prepared as plain text files. The page format for this dissertation is different than the plain text file format. Therefore, the appearance of the exam text in this document is different. Further, changes to font size and paragraph spacing have been made (when needed).

Mid-Term Examination, Control Group, Fall 1997

```
PROBLEM 1 (20 total points, 2 points each)
Given these declarations:                            Grading Notes (1-> 8):
     int : Integer;                                    1 point computation
     nat : Natural;                                    1 point format
     flo : Float;                 either side of float, counts as correct
Show what will be displayed by each of these program fragments.
If a fragment will lead to an error, briefly explain the cause
of the error.  Use the letter "b" for a blank space.
                                                      Answers:
     --  Part 1
     int := 2 * 3 + 4 * 5;                                         bbb26
     Ada.Integer_Text_IO.put (Item => int, Width => 5);

     --  Part 2
     int := 2 + 3 / 4 * 5;                                         bbbb2
     Ada.Integer_Text_IO.put (Item => int, Width => 5);

     --  Part 3
     int := 2 ** 3 * 4 + 5;                                        bbb37
     Ada.Integer_Text_IO.put (Item => int, Width => 5);

     --  Part 4
     nat := 6 + 7 * 8 - 9;                                            53
     Ada.Integer_Text_IO.put (Item => nat, Width => 1);

     --  Part 5
     nat := 6 * 7 - 8 * 9;                             constraint error
     Ada.Integer_Text_IO.put (Item => nat, Width => 1);

     --  Part 6
     nat := 6 * 7 / 8 / 9;                                             0
     Ada.Integer_Text_IO.put (Item => nat, Width => 1);

     --  Part 7
     flo := 2.5 * 6.0 / 12.0;                                   bbbb1.25
     Ada.Float_Text_IO.put (Item=>flo, Fore=>5, Aft=>2, Exp=>0);

     --  Part 8
     flo := 6.0 * 7.0 - 8.0 * 9.0;                              bb-30.00
     Ada.Float_Text_IO.put (Item=>flo,Fore=>5, Aft=>2, Exp=>0);

     --  Part 9                                                 TRUE(+2)
     IF 6 + 7 * 8 - 9 >= 2 + 3 / 4 * 5 THEN
         Ada.Text_IO.put (Item => "TRUE");
     ELSE
         Ada.Text_IO.put (Item => "FALSE");
     END IF;

     --  Part 10                                                TRUE(+2)
     IF 18 + 4 /= 2 * 3 + 4 * 5 THEN
         Ada.Text_IO.put (Item => "TRUE");
     ELSE
         Ada.Text_IO.put (Item => "FALSE");
     END IF;
```

Mid-Term Examination, Control Group, Fall 1997

PROBLEM 2 (10 points total)

Part 1 (4 points)
Explain, in your own words, what a data type is.
Do NOT copy your answer from the book.

Grading : (see pages 67 in Feldman text)

key words to use: "a set of values (2 points) and
a set of operations (2 points)"

if one or more of the above missing, then partial credit words:
predefined type (1) if others missing
can define own (1) if others missing

Part 2 (6 point total, 1 point each type)
Name six data types of variables. Given a sample declaration for each type with a value assigned using a literal.

One set possible answers:

```
c : character := 'a';           Grading : one point per type

f : float     := 1.2;

i : integer   := -12;

n : natural   :=  12;           -- Actually integer subtype
                                -- Accepted here
s : string (1..5) := "hello";

TYPE my_enum IS (aaa, bbb, ccc); -- This is an enumeration type.

e : my_enum := aaa;
```

Mid-Term Examination, Control Group, Fall 1997

PROBLEM 3 (15 points total)
The eight dwarfs are sneezy, dopey, bashful, grumpy, doc, sleepy, happy, and cheapy. Write a small program named "dwarfs":

- o The program displays the names of all eight drawfs. One dwarf per line.
- o The program displays the number of dwarfs (use an attribute).

Your program must use a FOR LOOP and demonstrate three enumeration type attributes. The following statements are assumed:

```
     WITH Ada.Text_IO;
     WITH Ada.Integer_Text_IO;
```

The instructor's program is 14 lines including the WITH statements.

One possible answer is:

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;
PROCEDURE dwarfs IS
    TYPE dwarf IS (sneezy, dopey, bashful, grumpy, doc,
                   sleepy, happy, cheapy);
    PACKAGE dwarf_IO IS NEW Ada.Text_IO.Enumeration_IO (enum => dwarf);
BEGIN
        FOR d IN dwarf'first..dwarf'last LOOP
            dwarf_IO.put (Item => d);
            Ada.Text_IO.new_line;
        END LOOP;
        Ada.Text_IO.put (Item => "The number of dwarfs is ");
        Ada.Integer_Text_IO.put (Item => dwarf'Pos(cheapy) + 1);
END dwarfs;
```

Scoring:

PROCEDURE statement	+1	
BEGIN	+1	
END statement	+1	
TYPE statement	+1	
PACKAGE dwarf_IO	+1	
FOR LOOP	+3	
Dwarf Count Display	+1	
ENUM Attributes	+6	(2 points per attribute used correctly)

Mid-Term Examination, Control Group, Fall 1997

PROBLEM 4 (10 points total)
Nancy has 2 dogs and 3 cats. The dogs and cats are identified by two variables named "animal" and "number". Using the logic table below, write the IF statements that display the text given below.

animal	number	text to be displayed
1	1	dog named Puddles
1	2	dog named Jake
2	1	cat named Midnight
2	2	cat named Tom
2	3	cat named Cleo

The instructor's IF statements are approximately 15 lines long.

One possible answer is:

```
IF animal = 1 THEN
     IF    number = 1 THEN
         Ada.Text_IO.put ("dog named Puddles");
     ELSE
         Ada.Text_IO.put ("dog named Jake");
     END IF;
ELSE
     IF    number = 1 THEN
         Ada.Text_IO.put ("cat named Midnight");
     ELSIF number = 2 THEN
         Ada.Text_IO.put ("cat named Tom");
     ELSE
         Ada.TEXT_IO.put ("cat named Cleo");
     END IF;
END IF;
```

```
Grading : First level  IF THEN ELSE +3
          Second level IF THEN ELSE +3  (or use AND)
          Use END IF                +2
          Logic / Good syntax       +2  (RESERVE words,punctuation)
```

Mid-Term Examination, Control Group, Fall 1997

PROBLEM 5 (12 total points, 4 points per FUNCTION)
Write three functions named "pi_it". The input parameter name is "x".

Input	Output type	FUNCTION output
Character	Integer	-1
Float	Float	input * 3.14159
Integer	Natural	input * 3, output is always positive

The instructor's FUNCTION's are four lines, four lines, and eight lines long.

```
One possible answer is:                       -- Grading Info:

     FUNCTION pi_it (x : IN Character) RETURN Integer IS
     BEGIN
          RETURN -1;                          -- 1 point per line item
     END pi_it;

     FUNCTION pi_it (x : IN Float) RETURN Float IS
     BEGIN
          RETURN x * 3.14159;                 -- 1 point per line item
     END pi_it;

     FUNCTION pi_it (x : IN Integer) RETURN Natural IS
     BEGIN
          IF x < 0 THEN                       -- 2 points for FUNCTION
               RETURN 0 - x * 3;              -- 2 points for IF
          ELSE
               RETURN x * 3;
          END IF;
     END pi_it;
```

Mid-Term Examination, Control Group, Fall 1997

PROBLEM 6

Part 1 (6 points)
Write a PACKAGE specification named "small_package" that includes the three FUNCTIONs in problem 5.

```
     PACKAGE small_package IS

          FUNCTION pi_it (x : IN Character) RETURN Integer;

          FUNCTION pi_it (x : IN Float) RETURN Float;

          FUNCTION pi_it (x : IN Integer) RETURN Natural;

     END small_package;
```

Grading : one point per line, one point for same name.

Part 2 (3 points)
What is polymorphic about the PACKAGE? One sentence answer, please.

FUNCTIONs have the same name (pi_it is overloaded). (from class)

Grading : FUNCTIONs have same name +3

Part 3 (3 points)
What is encapsulated in the PACKAGE? One sentence answer, please.

Three FUNCTIONs are all in the same PACKAGE. (from class)

Grading : stuff in PACKAGE +3

Mid-Term Examination, Treatment Group One, Spring 1998

```
PROBLEM 1 (20 total points, 2 points each)
Given these declarations:                           Grading Notes (1-> 8):
     int : Integer;                                   1 point computation
     nat : Natural;                                   1 point format
     flo : Float;                either side of float, counts as correct
Show what will be displayed by each of these program fragments.
If a fragment will lead to an error, briefly explain the cause
of the error.  Use the letter "b" for a blank space.
     --  Part 1                                        Answers:
     int := 6 * 7 + 8 * 9;                                      bbb114
     Ada.Integer_Text_IO.put (Item => int, Width => 6);

     --  Part 2
     int := 6 + 7 / 8 * 9;                                      bbbbb6
     Ada.Integer_Text_IO.put (Item => int, Width => 6);

     --  Part 3
     nat := 3 ** 2 * 8 + 9;                                     bbbb81
     Ada.Integer_Text_IO.put (Item => nat, Width => 6);

     --  Part 4
     int := 8 - 9 * 6 - 7;                                         -53
     Ada.Integer_Text_IO.put (Item => int, Width => 1);

     --  Part 5
     nat := 9 / 8 / 7 * 6;                                           0

     Ada.Integer_Text_IO.put (Item => nat, Width => 1);

     --  Part 6
     nat := 7 / 8 / 9 - 6;                          constraint error
     Ada.Integer_Text_IO.put (Item => nat, Width => 1);

     --  Part 7
     flo := 18.0 / 12.0 * 2.5;                               bbbbb3.75
     Ada.Float_Text_IO.put (Item=>flo, Fore=>6, Aft=>2, Exp=>0);

     --  Part 8
     flo := 9.0 / 8.0 - 13.0 / 8.0;                          bbbb-0.50
     Ada.Float_Text_IO.put (Item=>flo, Fore=>6, Aft=>2, Exp=>0);

     --  Part 9
     IF 9 / 8 / 7 * 6 >= 6 + 7 / 8 * 9 THEN
         Ada.Text_IO.put (Item => "TRUE");                 FALSE (+2)
     ELSE
         Ada.Text_IO.put (Item => "FALSE");
     END IF;

     --  Part 10
     IF 102 /= 6 * 7 + 8 * 9 THEN
         Ada.Text_IO.put (Item => "TRUE");                  TRUE (+2)
     ELSE                                              (see page 132)
         Ada.Text_IO.put (Item => "FALSE");
     END IF;
```

Mid-Term Examination, Treatment Group One, Spring 1998

PROBLEM 2 (10 points total)

Part 1 (4 points)
Explain, in your own words, what is the difference between a variable and a data type.

```
Grading :                                      (see pages 42 and 67)

key words to use:
    data type -- "a set of values (1 point) and
                  a set of operations (1 point)"


    variable  -- "is an identifier," page 42, and
                  are "used in a program for storing results",
                  page 43, which have a value

                  identifier or identification  (1 point)
                  storing a result or something (1 point)

    Partial credit:  predefined type   (1) if others missing
                     can define own    (1) if others missing
```

Part 2 (6 point total, 1 point each type)
Name six different data types by giving a sample declaration for each type, and assign an initial value using a literal (do not use positive or negative [subtypes])

One set possible answers:

```
c : character := 'a';          Grading : one point per type

f : float     := 1.2;

i : integer   := -12;

n : natural   :=  12;          -- Actually integer subtype
                               -- Accepted here
s : string (1..5) := "hello";

TYPE my_enum IS (aaa, bbb, ccc); -- This is an enumeration type.

e : my_enum := aaa;
```

Mid-Term Examination, Treatment Group One, Spring 1998

PROBLEM 3 (15 points total)
The seven dwarfs are sneezy, dopey, bashful, grumpy, doc, sleepy, and happy. Write a small program named "dwarfs", using these rules:

- The program displays the names of all drawfs.
- Two dwarfs per line, with two spaces between the names (except last line).
- The program displays the number of dwarfs (use an attribute).

Your program must use a FOR LOOP, IF statement, and demonstrate three enumeration type attributes. The following statements are assumed:

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;
```

The instructor's program is 16 lines including the WITH statements.

One possible answer is:

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;
PROCEDURE dwarfs IS
    TYPE dwarf IS (sneezy, dopey, bashful, grumpy, doc, sleepy, happy);
    PACKAGE dwarf_IO IS NEW Ada.Text_IO.Enumeration_IO (enum => dwarf);
BEGIN
    FOR d IN dwarf'first..dwarf'last LOOP
        dwarf_IO.put (Item => d);
        Ada.TEXT_IO.put (Item => "  ");
        If dwarf'pos(d) MOD 2 = 0 THEN
            Ada.Text_IO.new_line;
        END IF;
    END LOOP;
    Ada.Text_IO.put (Item => "The number of dwarfs is ");
    Ada.Integer_Text_IO.put (Item => dwarf'Pos(cheapy) + 1);
END dwarfs;
```

Scoring:	PROCEDURE statement	+1
	BEGIN	+1
	END statement	+1
	TYPE statement	+1
	PACKAGE dwarf_IO	+1
	FOR LOOP	+2
	IF logic	+2
	ENUM Attributes	+6 (2 points per attribute used correctly)

Mid-Term Examination, Treatment Group One, Spring 1998

PROBLEM 4 (10 points total)
Nora has seven children. The children are identified by two variables named "age" and "gender". Using the logic table below, write the IF statements that display the names of the children.

Name of child	age	gender
Ralph	12	M
Nancy	10	F
Eric	8	M
Jane	8	F
Art	6	M
Sue	6	F
Oops	1	M

The instructor's IF statements are approximately 19 lines long.

One possible answer is:

```
IF    age = 12 THEN
      Ada.Text_IO.put (Item => "Ralph");
ELSIF age = 10 THEN
      Ada.Text_IO.put (Item => "Nancy");
ELSIF age =  8 THEN
    IF gender = 'M' THEN
          Ada.Text_IO.put (Item => "Eric");
    ELSE
          Ada.Text_IO.put (Item => "Jane");
    END IF;
ELSIF age = 6
    IF gender = 'M' THEN
         Ada.Text_IO.put (Item => "Art");
    ELSE
         Ada.Text_IO.put (Item => "Sue");
    END IF;
ELSE
         Ada.Text_IO.put (Item => "Oops");
END IF;
```

```
Grading : First level  IF THEN ELSE +3
          Second level IF THEN ELSE +3 (or use AND)
          Use END IF                +2
          Logic / Good syntax       +2 (RESERVE words,punctuation)
```

Mid-Term Examination, Treatment Group One, Spring 1998

PROBLEM 5 (12 total points, 4 points per FUNCTION)
Write three functions named "george".
The input parameter name is "in_x".

Input	Output type	FUNCTION output
Character	Integer	
in_x < 'M'		-1
or else		0
Float	Float	input * input
Integer	Natural	
in_x < 3		0
or else		the three time input minus 3

The instructor's FUNCTION's are six lines, four lines, and eight lines long.

```
One possible answer is:                 -- Grading Info:

     FUNCTION george (in_x : IN Character) RETURNS Integer IS
     BEGIN
          IF in_x < 'M' THEN RETURN -1;  --  2 points for FUNCTION
          ELSE               RETURN  0;  --  2 point for IF
          END IF;
     END george;

     FUNCTION george (in_x : IN Float) RETURNS Float IS
     BEGIN
          RETURN in_x * in_x;            --  1 point per line item
     END george;

     FUNCTION george (in_x : IN Integer) RETURNS Natural IS
     BEGIN
          IF in_x < 3
               RETURN 0;                 --  2 points for FUNCTION
          ELSE                           --  2 points for IF
               RETURN 3 * in_x - 3;
          END IF;
     END george;
```

Mid-Term Examination, Treatment Group One, Spring 1998

PROBLEM 6

Part 1 (6 points)
Write a PACKAGE specification named "small_package" that includes the three FUNCTIONs in problem 5.

```
PACKAGE small_package IS                      --  (see page 121)

    FUNCTION george (x : IN Character) RETURNS Integer;

    FUNCTION george (x : IN Float) RETURNS Float;

    FUNCTION george (x : IN Integer) RETURNS Natural;

END small_package;
```

Grading : one point per line, one point for same name.

Part 2 (3 points)
What is polymorphic about the PACKAGE? One sentence answer, please.

FUNCTIONs have the same name (overloaded name).   (from class)

Grading : FUNCTIONs have same name +3

Part 3 (3 points)

What is encapsulated in the PACKAGE? One sentence answer, please.

FUNCTIONs are all in the same PACKAGE. (from class)

Grading : stuff in PACKAGE +3

Mid-Term Examination, Treatment Group Two, Fall 1998

```
PROBLEM 1 (20 total points, 2 points each)
Given these declarations:                         Grading Notes (1-> 8):
     int : Integer;                                  1 point computation
     nat : Natural;                                  1 point format
     flo : Float;              either side of float, counts as correct
Show what will be displayed by each of these program fragments.
If a fragment will lead to an error, briefly explain the cause
of the error.  Use the letter "b" for a blank space.
                                                   Answers:
     --  Part 1
     int := 7 * 3 + 3 * 9;                          --       bb48
     Ada.Integer_Text_IO.put (Item => int, Width => 4);

     --  Part 2
     int := 7 + 2 / 3 * 9;                          --       bbb7
     Ada.Integer_Text_IO.put (Item => int, Width => 4);

     --  Part 3
     nat := 7 ** 2 * 3 + 9;                         --       b156
     Ada.Integer_Text_IO.put (Item => nat, Width => 4);

     --  Part 4
     int := 7 - 2 * 9 + 3;                          --         -8
     Ada.Integer_Text_IO.put (Item => int, Width => 1);

     --  Part 5
     nat := 7 / 3 / 2 - 9;                      --  constraint error
     Ada.Integer_Text_IO.put (Item => nat, Width => 1);

     --  Part 6
     nat := 7 / 3 / 2 * 9;                          --          9
     Ada.Integer_Text_IO.put (Item => nat, Width => 1);

     --  Part 7
     flo := 39.0 / 6.0 * 1.5;                       --    bbb9.75
     Ada.Float_Text_IO.put (Item=>flo, Fore=>4, Aft=>2, Exp=>0);

     --  Part 8
     flo := 9.0 / 3.0 - 7.0 / 2.0;                  --    bb-0.50
     Ada.Float_Text_IO.put (Item=>flo, Fore=> 4, Aft=>2, Exp=>0);

     --  Part 9
     IF 7 * 3 + 3 * 9 >= 7 + 2 / 3 * 9 THEN
         Ada.Text_IO.put (Item => "TRUE");          --   TRUE (+2)
     ELSE
         Ada.Text_IO.put (Item => "FALSE");
     END IF;

     --  Part 10
     IF 7 - 2 * 9 + 3 < 21 THEN
         Ada.Text_IO.put (Item => "TRUE");          --   TRUE (+2)
     ELSE                                          (see page 132)
         Ada.Text_IO.put (Item => "FALSE");
     END IF;
```

Mid-Term Examination, Treatment Group Two, Fall 1998

PROBLEM 2 (10 points total)

Part 1 (4 points)
Explain, in your own words and the textbooks, what is a variable and a data type.

```
     Grading :                                   (see pages 42 and 67)


     key words to use:

         data type -- "a set of values (1 point) and
                       a set of operations (1 point)"

         variable  -- "is an identifier," page 42, and
                       are "used in a program for storing results",
                       page 43, which have a value

                       identifier or identification  (1 point)
                       storing a result or something (1 point)

         Partial credit:  predefined type   (1) if others missing
                          can define own    (1) if others missing
```

Part 2 (6 point total, 1 point each type)
Name six different data types by giving a sample declaration for each type, and assign an initial value using a literal (do not use positive or negative as data types [subtypes]).

One set possible answers:

```
      c : character := 'a';           Grading : one point per type

      f : float     := 1.2;

      i : integer   := -12;

      n : natural   :=  12;           -- Actually integer subtype
                                      -- Accepted here
      s : string (1..5) := "hello";

      TYPE my_enum IS (aaa, bbb, ccc); -- This is an enumeration type.

      e : my_enum := aaa;
```

Mid-Term Examination, Treatment Group Two, Fall 1998

PROBLEM 3 (15 points total)
The seven dwarfs are sneezy, dopey, bashful, grumpy, doc, sleepy, and happy. Write a small program named "dwarfs", using these rules:

- o Display the names of all drawfs.
- o Display one dwarf per line.
- o Display the number of dwarfs (use an attribute).
- o Display the text "is a bad dwarf" with grumpy.

Your program must use a FOR LOOP, IF statement, and demonstrate three enumeration type attributes. The following statements are assumed:

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;
```

The instructor's program is about 18 lines including the WITH statements.

One possible answer is:

```
WITH Ada.Text_IO;
WITH Ada.Integer_Text_IO;
PROCEDURE dwarfs IS
    TYPE dwarf IS (sneezy, dopey, bashful, grumpy, doc, sleepy, happy);
    PACKAGE dwarf_IO IS NEW Ada.Text_IO.Enumeration_IO (enum => dwarf);
BEGIN
    FOR d IN dwarf'first..dwarf'last LOOP
        dwarf_IO.put (Item => d);
        IF d = grumpy THEN
            Ada.Text_IO.put (Item => " is a bad dwarf");
            Ada.Text_IO.new_line;
        END IF;
    END LOOP;
    Ada.Text_IO.put (Item => "The number of dwarfs is ");
    Ada.Integer_Text_IO.put (Item => dwarf'Pos(happy) + 1);
    Ada.Text_IO.new_line;
END dwarfs;
```

Scoring:	PROCEDURE statement	+1
	BEGIN	+1
	END statement	+1
	TYPE statement	+1
	PACKAGE dwarf_IO	+1
	FOR LOOP	+2
	IF logic	+2
	ENUM Attributes	+6 (2 points per attribute used correctly)

Mid-Term Examination, Treatment Group Two, Fall 1998

PROBLEM 4 (10 points total)
Sandy has seven pets. The pets are identified by two variables named "animal" and "gender". Using the logic table below, write the IF statements that display the names of the pets.

animal	gender	name of pet	what the letters mean
d	b	Rover	b = boy
c	g	Cleo	c = cat
d	g	Suzie	d = dog
c	b	Max	f = fish
f	g	Wanda	g = girl
f	b	Sushi	p = pig
p	b	Bacon	

The instructor's IF statements are about 12 to 20 lines long (depends upon whether the "put" statement is on the same line as the "THEN")

One possible answer is:

```
IF gender = 'b' THEN
     IF    animal = 'd' THEN  Ada.Text_IO.put (Item => "Rover");
     ELSIF animal = 'c' THEN  Ada.Text_IO.put (Item => "Max");
     ELSIF animal = 'f' THEN  Ada.Text_IO.put (Item => "Sushi");
     ELSE                     Ada.Text_IO.put (Item => "Bacon");
     END IF;
ELSE    --  gender = 'g'
    IF      animal = 'd' THEN  Ada.Text_IO.put (Item => "Suzie");
    ELSIF   animal = 'c' THEN  Ada.Text_IO.put (Item => "Cleo");
    ELSE                       Ada.Text_IO.put (Item => "Wanda");
    END IF;
END IF;
```

```
Grading : First level  IF THEN ELSE +3
          Second level IF THEN ELSE +3 (or use AND)
          Use END IF                +2
          Logic / Good syntax       +2 (RESERVE words,punctuation)
```

Mid-Term Examination, Treatment Group Two, Fall 1998

PROBLEM 5 (12 total points, 4 points per FUNCTION)
Write three functions named "happy".
The input parameter name is "in_x".

Input	Output type	FUNCTION output
Integer	Integer	
in_x < 0		-1
in_x = 0		0
in_x > 0		1
Float	Float	two times input times 3.14159
Natural	Integer	
in_x < 199		input minus 200
otherwise		100

The instructor's FUNCTION's are 7 to 10 lines, four lines, and eight lines long.

```
One possible answer is:                     -- Grading Info:

     FUNCTION happy (in_x : IN Integer) RETURNS Integer IS
     BEGIN
          IF    in_x < 0 THEN RETURN -1;   --  2 points for FUNCTION
          ELSIF in_x = 0 THEN RETURN  0;   --  2 point for IF
          ELSE                RETURN  1;
          END IF;
     END happy;

     FUNCTION happy (in_x : IN Float)   RETURNS Float IS
     BEGIN
          RETURN 2.0 * in_x * 3.14159;     --  1 point per line item
     END happy;

     FUNCTION happy (in_x : IN Natural) RETURNS Integer IS
     BEGIN
          IF inx_x < 199 THEN
               RETURN in_x - 200;          --  2 points for FUNCTION
          ELSE                             --  2 points for IF
               RETURN 100;
          END IF;
     END happy;
```

Mid-Term Examination, Treatment Group Two, Fall 1998

PROBLEM 6

Part 1 (6 points)
Write a PACKAGE specification named "small_package" that includes the three FUNCTIONs in problem 5.

```
     PACKAGE small_package IS                    --  (see page 121)

         FUNCTION happy (x : IN Integer) RETURNS Integer;

         FUNCTION happy (x : IN Float) RETURNS Float;

         FUNCTION happy (x : IN Natural) RETURNS Integer;

     END small_package;
```

Grading : one point per line, one point for same name.

Part 2 (3 points)
What is polymorphic (or overloaded) about the PACKAGE?
One sentence answer, please.

FUNCTIONs have the same name.   (from class)

Grading : FUNCTIONs have same name +3

Part 3 (3 points)
What is encapsulated in the PACKAGE? One sentence answer, please.

FUNCTIONs are all in the same PACKAGE. (from class)

Grading : stuff in PACKAGE +3

Sample Lab Exam Question -- Material Not Included In The Experiment

PROBLEM 7

```
Grading of this problem: Case study   6 pcints
                         Test plan    4 points
                         Correctness 10 points
                         Style        4 points
```

Write a program named "iso_tri". This program produces a triangle of "x" like the triangle shown below (with each line number identified):

```
x      Line number is ONE
xx     Line number is TWO
xxx    Line number is THREE
xxxx   Line number is FOUR
xxxxx  Line number is FIVE
xxxxxx Line number is SIX
```

Your input is to be text that represents a number. The input prompt in the instructor's version of the program is as follows:

```
Enter a number from one to twelve;
using the name of the number (two). >
```

Your output should line up in columns as shown below:

```
x    Line number is ONE
xx   Line number is TWO
xxx  Line number is THREE
xxxx Line number is FOUR
```

The rules below must be followed to get maximum credit for this problem.

- o No IF statements
- o No FUNCTIONs and PROCEDUREs
- o NO PACKAGEs
- o FOR LOOPs and Enumeration Types are to be used
- o Run only two good input "gexecute" of the program
- o Document only bad input test data

The instructor's program is 22 lines long without comments.

Place your documentation in a separate file. The following documentation should be included in the program:

- o Analysis -- keep it short (Maximum of ten lines)
- o Data requirements and formulas -- keep it short
- o Design -- initial algorithm and any refinements
- o Test plan

Remember to include your documentation in your "turnin" script. Given that the name of documentation file is "mid_term.doc", use the Unix "cat" command to include the documentation in the turnin script:

```
 cat mid_term.doc
```

The preferred way to hand in PROBLEM 7 is to E-mail your "turnin" script to the instructor. Given that the name of your script is "mid_term.turnin"; E-mail your script as follows:

```
  cat mid_term.turnin | elm -s mid_term clund
```

Grading detail:

Case study 6 points	
Analysis	1 point
Program Inputs	1 point
Program Outputs	1 point
Types/ subtypes / intermediate variables	1 point
Algorithm & Refinement	2 point
Test plan 4 points	
Good data	2 point
Boundary conditions	1 point
Data Error or Constraint Error	1 point
Correctness 10 points	
Input logic correct	2 point
Triangle logic detected	2 point
Other output logic correct	2 point
Compile and link	2 point
Go!	2 point
Style 4 points	
Indentation / Other	2 point
Gift (for trying to write code)!!	2 point

One possible solution is:

```
WITH Ada.Text_IO;
PROCEDURE iso_tri IS
      TYPE number IS (one, two, three, four, five, six,
                      seven, eight, nine, ten, eleven, twelve);
      PACKAGE number_IO IS NEW Ada.Text_IO.Enumeration_IO
          (Enum => number);
     num_name : number;
BEGIN
     Ada.Text_IO.put (Item => "Enter a number from one to twelve; ");
     Ada.Text_IO.put (Item => "using the name of the number (two). >");
     number_IO.get (Item => num_name);
     FOR i IN 1..number'pos(num_name)+1 LOOP
          FOR j IN 1..i LOOP
               Ada.Text_IO.put (ITEM => 'x');
          END LOOP;
          FOR j IN i..number'pos(num_name)+1 LOOP
               Ada.Text_IO.put (Item => ' ');
          END LOOP;
          Ada.Text_IO.put (Item => "Line number is ");
          number_IO.put (Item => number'val(i-1));
          Ada.Text_IO.new_line;
     END LOOP;
END iso_tri;
```

APPENDIX G. FINAL EXAMINATION

The final examination is an open book, 120 minute test given in the classroom. The point structure for the common six sequential test questions is given below:

- Problem 1 -- 7 points
- Problem 3 -- 10 points
- Problem 5 -- 8 points
- Problem 9 -- 10 points
- Problem 10 -- 10 points
- Problem 11 -- 12 points

The total number of points for the above problems is 57. Two points in Problem 3 are not included in the comparison. The next 24 pages of this appendix are the six sequential final exam test questions; these pages include questions, answers, and grading instructions.

The four concurrency final exam test questions follow the sequential questions. The same concurrency questions were asked of both treatment groups. The point structure for the four concurrency test questions is given below.

- Problem 2 -- 10 points
- Problem 4 -- 12 points
- Problem 7 -- 10 points
- Problem 8 -- 9 points

The concurrency test question pages include the questions, answers, and grading instructions.

The final two pages of this appendix present a sample Problem 6 that is not part of the experiment.

The exams for the control group and the treatment groups were prepared as plain text files. The page format for this dissertation is different than the plain text file format. Therefore, the appearance of the exam text in this document is different. Further, changes to font size and paragraph spacing have been made (when needed).

Final Examination Sequential Problem, Control Group, Fall 1997

Problem 1. (7 points total)
What is the output of the program given below:

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
WITH Ada.Integer_Text_IO;
PROCEDURE prob_01 IS
     a : Natural := 1;
     b : Natural := 0;
     t : Natural;
BEGIN
     put ("Program prob_01 -- Print Number Sequence");
     new_line (Spacing => 2);
     put ("    Integer    New Number");  new_line;
     put ("    -------    ----------");  new_line;

     FOR c IN 2..11 LOOP
          t := a;  a := a + b;  b := t;
          Ada.Integer_Text_IO.put (Item => c, Width =>  9);
          Ada.Integer_Text_IO.put (Item => a, Width => 14);
          new_line;
     END LOOP;
END prob_01;
```

Answer:

```
Program prob_01 -- Print Number Sequence

    Integer    New Number
    -------    ----------
        2             1
        3             2
        4             3
        5             5
        6             8
        7            13
        8            21
        9            34
       10            55
       11            89
```

Grading:

"Program" line	1 point
Column headers	1 point
Integer column indent	1 point
Integer column values	1 point
New Number column indent	1 point
New Number column values	2 points

Final Examination Sequential Problem, Control Group, Fall 1997

Problem 3. (10 points total)
The PACKAGE BODY timer is defined below. There are five items to be answered.

```
PACKAGE BODY timer IS

     FUNCTION total_seconds (days, hours, minutes,
                             seconds: IN Natural)
          RETURNS Natural IS
     BEGIN
          RETURN days * 24 * 60 * 60 +
                 hours     * 60 * 60 +
                 minutes        * 60 +
                 seconds;
     END total_seconds;

     FUNCTION total_minutes (days, hours, minutes: IN Natural)
          RETURNS Natural IS
     BEGIN
          RETURN total_seconds (days, hours, minutes, 0) / 60;
     END total_seconds;

END timer;
```

Part a. Write the PACKAGE specification for the PACKAGE BODY timer (2 points)

Part b. Give an example using the function total_minutes (include a "WITH" statement only) (1 point)

Part c. Give an example using the function total_minutes (include a "WITH" and "USE" statements both) (1 point)

Part d. What is encapsulated in the PACKAGE timer (2 points)

Part e. Why does Ada support both PACKAGE specifications and PACKAGE BODYs (4 points)

Part a answer:

```
PACKAGE timer IS
     FUNCTION total_seconds (days, hours, minutes,
                             seconds: IN Natural)
          RETURNS Natural;

     FUNCTION total_minutes (days, hours, minutes: IN Natural)
          RETURNS Natural;
END timer;
```

Grading: 1 point per FUNCTION and
-1 point for PACKAGE / END error

Problem 3. (continued)

Part b answer:

```
WITH timer;
timer.total_minutes (days => 1, hours => 12, minutes => 32);
```

Grading: 1 point

Part c answer:

```
WITH timer;  USE timer;
total_minutes (days => 1, hours => 12, minutes => 32);
```

Grading: 1 point

Part d answer:

Two functions are encapsulated in the package distance.
The functions are named total_seconds and total_minutes.

Grading: 1 point per FUNCTION named.

Part e answer:

PACKAGE specifications are interfaces to people to define PACKAGE funcationality. PACKAGE specifications allow the distribution and use of PACKAGEs without the internals of the BODY being known and seen by people. PACKAGE BODYs are interfaces to the compiler to define the internal operations of the PACKAGE to the compiler.

Grading: 2 points for specification
2 points for body
-1 mostly correct, but not above language

Final Examination Sequential Problem, Control Group, Fall 1997

```
Problem  5.  (8 points total)
What is the output of the program given below:

     WITH Ada.Text_IO;  USE Ada.Text_IO;
     PROCEDURE prob_05 IS
          max  : CONSTANT Integer := 10;
          TYPE new_array IS ARRAY (1..max) of Character;
          text : new_array :=('a','b','c','d','e','f','g','h','i','j');
     BEGIN
          FOR i IN 1..max LOOP
               FOR j IN 1..i LOOP
                    put (Item => text(max-j));
               END LOOP;
               new_line;
          END LOOP;
          EXCEPTION
               WHEN Constraint_Error =>
                    put (Item => "Ham and Cheese Sub, Please");
                    new_line;
               WHEN Data_Error       =>
                    put (Item => "Call For Pizza and Beer");
                    new_line;
     END prob_05;

     Answer for Prob_05 is:
          i
          ih
          ihg
          ihgf
          ihgfe
          ihgfed
          ihgfedc
          ihgfedcb
          ihgfedcba
          ihgfedcbaHam and Cheese Sub, Please
```

Grading:	Positive logic:	Triangular shape found	+3
		Computation correct	+3
		Constraint error found	+2
	Negative logic:	Index off	-1
		Wrong direction	-1
		Loop order	-1
		No new_line	-1
		One loop missing	-1

Final Examination Sequential Problem, Control Group, Fall 1997

```
Problem  9.  (10 points total)

  Part a.  Rewrite the FUNCTION as a PROCEDURE.
           Also rewrite the calling statement.  (4 points)

        FUNCTION next_value (current_value : Integer) RETURN Integer IS
        BEGIN
           RETURN 120000 + ((current_value - 120000) * 6791) MOD 10000;
        END next_value;

        --  calling statement
        my_num := next_value (current_value => my_num);

  Part b.  Write a PROCEDURE that solves the following expression:
           (3 points)
                  2
              a (b ) + 2 c
           r = ------------, where a, b, c, d, d, e, and r are integers
                d e - a

  Part c.  Write a PROCEDURE that solved the following expression:
           (3 points)
                       2            2
              (vl - v2) + (hl - h2)
           r = ---------------------, where vl, v2, hl, h2, and r are
                         10                  naturals

     Answer for Part a:
          PROCEDURE next_value (current_value : Integer;
                                result : OUT Integer) IS
          BEGIN
               result :=120000+((current_value-120000)*6791) MOD 10000;
          END next_value;

          --  calling statement
          next_value (current_value => my_num), result => my_num;

     Answer for Part b:
          PROCEDURE x (a, b, c, d, e : IN Integer; r : OUT Integer) IS
          BEGIN
               r := (a * b * b + 2 * c) / (d * e - a);
          END x;

     Answer for Part c:
          PROCEDURE y (vl, v2, hl, h2 : IN Natural; r : OUT Natural) IS
          BEGIN
               r : = ((vl - v2) ** 2 + (hl - h2) ** 2) / 10;
          END y;
```

Grading:

```
          Routine type / BEGIN / END                        1 point
                  type is either FUNCTION or PROCEDURE
          Arithmetic assignment statement                   1 point
          Parameters                                        1 point
          Calling statement (Part a only)                   1 point
```

Final Examination Sequential Problem, Control Group, Fall 1997

Problem 10. (10 points total)
Ada supports FUNCTIONs and PROCEDUREs and provides different parameter modes.

Part a. Name the parameter modes and given a one sentence description. Which parameter mode is the default? (4 points)

Part b. Which parameter modes can be used by FUNCTIONs? (3 points)

Part c. Which parameter modes can be used by PROCEDUREs? (3 points)

Answer Part a:

IN mode -- parameter's value is passed to the called routine, and parameter's value can not be changed in called routine.
IN mode is the default.

OUT mode -- parameter's value is passed out to the calling routine parameter's value must be assigned by called routine

IN OUT mode -- parameter's value is passed to the called routine, and parameter's value may be changed by the called routine, and parameter's value is passed out to the calling routine

Grading: 1 point per mode / 1 point for correct default

Answer for Part b:
FUNCTIONs use "IN" mode parameters only. *Grading:* 3 points

Answer for Part c:
PROCEDUREs use all three modes of parameters.

Grading: 3 points for all three / 2 for two / 1 for one

Final Examination Sequential Problem, Control Group, Fall 1997

Problem 11. (12 points total)
The map below provides information about each family on the block.

```
                              Cedar Street
.---------------------------------------------------------------------.
| 1703 Cedar St.  | 1705 Cedar St.  | 1707 Cedar St.  | 1709 Cedar St.  |
|                 |                 |                 |                 |
| Jones family    | Harris family   | Davis family    | Mr. Nelson      |
|                 |                 |                 |                 |
|  married        |  married        |  married        |  divorced       |
|  3 children     |  3 children     |  1 child        |  no children    |
|  1 dog          |  2 dogs         |  2 dogs         |  1 truck        |
|  1 car          |  2 cats         |  1 cat          |                 |
|                 |  2 cars         |  3 cars         |                 |
|-----------------+-----------------+-----------------+-----------------|
| 1702 Oak St.    | 1704 Oak St.    | 1706 Oak St.    | 1708 Oak St.    |
|                 |                 |                 |                 |
| Peters family   | Murray family   | Mrs. Menna      | Adams family    |
|                 |                 |                 |                 |
|  married        |  married        |  widowed        |    unmarried    |
|  2 children     |  1 child        |  2 children     |    7 children   |
|  2 cars         |  1 dog          |  1 dog          |    4 cats       |
|  1 truck        |  1 cat          |  1 car          |    2 cars       |
|                 |  1 car          |  1 truck        |    2 trucks     |
'---------------------------------------------------------------------'
                              Oak Street
```

Part a. Write the Ada statements to declare a record that hold information about a family. (4 points)

Part b. Write the Ada statements to declare an array of records (one record per family). (4 points)

Part c. Write the Ada statements place the "Murray" family Information into a record number 6. (4 points)

One possible solution for part a:

```
TYPE marital IS (married, unmarried, divorced, widowed);
TYPE family  IS RECORD
     name     : String (1..10);
     address  : String (1..20);
     status   : marital;
     children : Natural;
     dogs     : Natural;
     cats     : Natural;
     cars     : Natural;
     trucks   : Natural;
END RECORD;
```

One possible solution for part b:

```
max    : CONSTANT Natural := 8;
TYPE family_array IS ARRAY (1..max) of family;
family_data : family_array;
```

One possible solution for part c:

```
        family_data(6).name     := "Murray    ";
        family_data(6).address  := "1704 Oak Street     ";
        family_data(6).status   := married;
        family_data(6).children := 1;
        family_data(6).dogs     := 1;
        family_data(6).cats     := 1;
        family_data(6).cars     := 1;
        family_data(6).trucks   := 0;
```

```
Grading for part a:
     Enumeration declaration      1 point
     TYPE … Record / END RECORD   1 point
     Identifier Definitions       2 points, as qualified below:
          Missing identifiers    -1 point (more than one missing)
          Incorrect definition   -1 point (such as, "x : string;")
Grading for part b:
     TYPE declaration             2 points
     Array Declaration            2 points
Grading for part c:
     Valid syntax                 2 points
     Most values assigned         1 point
     Lack of careless errors      1 point
```

Final Examination Sequential Problem, Treatment Group One, Spring 1998

```
Problem  1.  (7 points total)
What is the output of the program given below:

     WITH Ada.Text_IO;  USE Ada.Text_IO;
     WITH Ada.Integer_Text_IO;
     PROCEDURE Final_01 IS
          x : Natural := 1;
          y : Natural := 0;
          t : Natural;
     BEGIN
          put ("Program Final_01 -- Print Number Sequence");
          new_line (Spacing => 2);
          put ("    Increment    Next Number");  new_line;
          put ("    ---------    -----------");  new_line;

          FOR c IN 3..11 LOOP
               t := x;  x := 2*y + x;  y := t;
               Ada.Integer_Text_IO.put (Item => c, Width => 10);
               Ada.Integer_Text_IO.put (Item => x, Width => 15);
               new_line;
          END LOOP;
     END Final_01;

     Answer:

          Program Final_01 -- Print Number Sequence

              Increment    Next Number
              ---------    -----------
                      3              1
                      4              3
                      5              5
                      6             11
                      7             21
                      8             43
                      9             85
                     10            171
                     11            341


     Grading:  "Program" line             1 point
               Column headers             1 point
               Increment column indent    1 point
               Increment column values    1 point
               Next Number column indent  1 point
               Next Number column values  2 points
```

Final Examination Sequential Problem, Treatment Group One, Spring 1998

Problem 3. (10 points total)
The PACKAGE BODY distance is defined below. There are four items to be answered.

```
     PACKAGE BODY distance IS

          FUNCTION total_feet
               (miles, furlongs, yards, feet, inches: IN Integer)
               RETURN Float IS
          BEGIN
               RETURN Float (miles * 5280 +
                             furlongs * 8 +
                             yards * 3    +
                             feet) +
                      Float (inches) / 12.0;
          END total_feet;

          FUNCTION total_yards
               (miles, furlongs, yards, feet, inches: IN Integer)
               RETURN Float IS
          BEGIN
               RETURN total_feet
                      (miles, furlongs, yards, feet, inches) / 3.0 ;
          END total_yards;

     END distance;
```

Part a. Write the PACKAGE *specification for* the PACKAGE BODY Distance (2 points)

Part b. Write a new FUNCTION total_furlongs that is outside the PACKAGE, given that the first line of the program is as follows: "WITH distance; USE distance;" (2 points)

Part c. What is encapsulated in the PACKAGE distance (2 points)

Part d. Why does Ada support both PACKAGE specifications and PACKAGE BODYs (4 points)

Part a answer:

```
   PACKAGE distance IS
      FUNCTION total_feet
           (miles, furlongs, yards, feet, inches: IN Integer)
           RETURN Float ;

      FUNCTION total_yards
           (miles, furlongs, yards, feet, inches: IN Integer)
           RETURN Float ;
   END distance;
```

Grading: 1 point per FUNCTION and
-1 point for PACKAGE / END error

Part b answer:

```
   FUNCTION total_furlongs
        (miles, furlongs, yards, feet, inches: IN Integer)
        RETURN Float IS
   BEGIN
        RETURN total_feet (miles,furlongs,yards,feet,inches) / 660.0;
   END total_furlongs;
```

Grading: 1 point for FUNCTION / BEGIN / END syntax
1 point for RETURN and computation

Part c answer:

Two functions are encapsulated in the package distance.
The functions are named total_feet and total_yards.

Grading: 1 point per FUNCTION named.

Part d answer:

PACKAGE specifications are interfaces to people to define PACKAGE funcationality. PACKAGE specifications allow the distribution and use of PACKAGEs without the internals of the BODY being known and seen by people. PACKAGE BODYs are interfaces to the compiler to define the internal operations of the PACKAGE to the compiler.

Grading: 2 points for specification
2 points for body
-1 mostly correct, but not above language

Final Examination Sequential Problem, Treatment Group One, Spring 1998

Problem 5. (8 points total)
What is the output of the program given below:

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
PROCEDURE final_05 IS
     limit : CONSTANT Integer := 10;
     TYPE new_array IS ARRAY (1..limit) of Character;
     text : new_array :=('F','I','N','A','L','*','E','X','A','M');
BEGIN
     FOR i IN 2..limit LOOP
          FOR j IN i/2..i+1 LOOP
               put (Item => text(j));
          END LOOP;
          new_line;
     END LOOP;
     EXCEPTION
          WHEN Constraint_Error =>
               put_line (Item => "Go to Pizza Hut and Relax");
          WHEN Data_Error       =>
               put_line (Item => "Go to Hamburger Hamlet");
END final_05;
```

Answer for final_05 is:

```
FIN
FINA
INAL
INAL*
NAL*E
NAL*EX
AL*EXA
AL*EXAM
L*EXAMGo to Pizza Hut and Relax
```

Grading:	Positive logic:	Triangular shape found	+3
		Computation correct	+3
		Constraint error found	+2
	Negative logic:	Index off	-1
		Wrong direction	-1
		Loop order	-1
		No new_line	-1
		One loop missing	-1

Final Examination Sequential Problem, Treatment Group One, Spring 1998

Problem 9. (10 points total)

Part a. Rewrite the FUNCTION as a PROCEDURE.
Also rewrite the calling statement. (4 points)

```
FUNCTION lex (par : Float) RETURN Float IS
BEGIN
     RETURN 3.14159 + ((par - 21.0) * 91.67) / 1.719;
END lex;

--  calling statement
my_num := lex (par => my_num);
```

Part b. Write a FUNCTION that solves the following expression:
(3 points)

```
    2        1
a (x ) - b (x ) + c
-------------------, where a, b, c, d, d, and e are integers
        e
       d
```

Part c. Write a PROCEDURE that solved the following expression:
(3 points)

```
     e          2     3
r = (g ) + (j - g)  * (c ),
    where a, c, e, g, j, and r are naturals
```

Answer for Part a:

```
PROCEDURE next_value (par : IN Float; result : OUT Float) IS
BEGIN
     result := 3.14159 + ((par - 21.0) * 91.67) / 1.719;
END next_value;

--  calling statement
lex (par => my_num, result => my_num);
```

Answer for Part b:

```
FUNCTION x (a, b, c, d, e : IN Integer) RETURN Integer IS
BEGIN
     RETURN (a * x * x - b * x + c) / (d ** e);
END x;
```

Answer for Part c:

```
PROCEDURE y (a, c, e, g, j : IN Natural; r : OUT Natural) IS
BEGIN
     r : = g ** e + ((j - g) ** 2) * c ** 3
END y;
```

Grading:	Routine type / BEGIN / END type is either FUNCTION or PROCEDURE	1 point
	Arithmetic assignment statement	1 point
	Parameters	1 point
	Calling statement (Part a only)	1 point

Final Examination Sequential Problem, Treatment Group One, Spring 1998

Problem 10. (10 points total)
Ada supports FUNCTIONs and PROCEDUREs and provides different parameter modes.

Part a. Name the parameter modes and given a one sentence description. Which parameter mode is the default?(4 points)

Part b. Which parameter modes can be used by FUNCTIONs? (2 points)

Part c. Which parameter modes can be used by PROCEDUREs? (2 points)

Part d. Which parameter modes were used in class with TASKs? (2 pts)

Answer Part a:

IN mode -- parameter's value is passed to the called routine, and parameter's value can not be changed in called routine.
IN mode is the default.

OUT mode -- parameter's value is passed out to the calling routine parameter's value must be assigned by called routine

IN OUT mode -- parameter's value is passed to the called routine, and parameter's value may be changed by the called routine, and parameter's value is passed out to the calling routine

Grading: 1 point per mode / 1 point for correct default

Answer for Part b:
FUNCTIONs use "IN" mode parameters only. *Grading:* 2 points

Answer for Part c:
PROCEDUREs use all three modes of parameters.

Grading: 2 points for all three / 1 for two

Answer for Part d:
TASKs used in class use "IN" mode parameters only.

Grading: 2 points for "IN"

Final Examination Sequential Problem, Treatment Group One, Spring 1998

Problem 11. (12 points total)
The map below provides information about part of a elementary school:

```
.-------------------------------------------------------------------.
| Room 121       | Room 123       | Room 125       | Room 127       |
|                |                |                |                |
| Ms. Harris     | Ms. Johnson    | Ms. Menna      | Ms. Meanny     |
|  (married)     |  (married)     |  (unmarried)   |  (divorced)    |
|                |                |                |                |
|  1st grade     |  2nd grade     |  3rd grade     |  4th grade     |
|  17 boys       |  15 boys       |  15 boys       |  16 boys       |
|  14 girls      |  21 girls      |  18 girls      |  16 girls      |
|  1  hamster    |  2  hamsters   |  1  snake      |  no pets       |
|                |  1  frog       |  2  hamsters   |                |
|----------------+----------------+----------------+----------------|
|                                                                   |
|                              Hallway                              |
|                                                                   |
|----------------+----------------+----------------+----------------|
| Room 122       | Room 124       | Room 126       | Room 128       |
|                |                |                |                |
| Ms. Peters     | Ms. Solo       | Ms. Myers      | Ms. Ford       |
|  (married)     |  (widowed)     |  (divorced)    |  (unmarried)   |
|                |                |                |                |
|  5th grade     |  6th grade     |  Kindergarden  |  Nursery       |
|  20 boys       |  18 boys       |  15 boys       |  12 boys       |
|  16 girls      |  17 girls      |  15 girls      |  14 girls      |
|  no pets       |  30 frogs      |  4  hamsters   |  no pets       |
'-------------------------------------------------------------------'
```

Part a. Write the Ada statements to declare a record that hold
information about a class, (4 points)

Part b. Write the Ada statements to declare an array of records
(one record per class). (4 points)

Part c. Write the Ada statements place Ms. Johnson's class
information
into record number 2. (4 points)

One possible solution for part a:

```
        TYPE marital IS (married, unmarried, divorced, widowed);
        TYPE class   IS RECORD
             teacher  : String (1..10);
             status   : marital;
             room_num : Natural;
             grade    : Character;
             boys     : Natural;
             girls    : Natural;
             hamsters : Natural;
             snakes   : Natural;
             frogs    : Natural;
        END RECORD;
```

One possible solution for part b:

```
        max    : CONSTANT Natural := 8;
        TYPE class_array IS ARRAY (1..max) of family;
        class_data : class_array;
```

One possible solution for part c:

```
        this : family; -- is assumed

        this.teacher   := "Johnson    ";
        this.status    := married;
        this.room_num  := 123;
        this.grade     := '2';
        this.boys      := 15;
        this.girls     := 21;
        this.hamsters  := 2;
        this.snakes    := 1;
        this.frogs     := 1;

        family_data(2) := this;
```

Grading for part a:

Enumeration declaration	1 point
TYPE … Record / END RECORD	1 point
Identifier Definitions	2 points, as qualified below:
Missing identifiers	-1 point (more than one missing)
Incorrect definition	-1 point (such as, "x : string;")

Grading for part b:

TYPE declaration	2 points
Array Declaration	2 points

Grading for part c:

Valid syntax	2 points
Most values assigned	1 point
Lack of careless errors	1 point

Final Examination Sequential Problem, Treatment Group Two, Fall 1998

Problem 1. (7 points total)
What is the output of the program given below:

```
WITH Ada.Text_IO;  USE Ada.Text_IO;
WITH Ada.Integer_Text_IO;
PROCEDURE Final_01 IS
     x : Natural := 1;
     y : Natural := 0;
     t : Natural;
BEGIN
     put ("Program Final_01 -- Print Number Sequence");
     new_line (Spacing => 2);
     put ("    Increment    Next Number");  new_line;
     put ("    ---------    -----------");  new_line;

     FOR c IN 2..10 LOOP
          t := x;  x := 2*y + c;  y := t;
          Ada.Integer_Text_IO.put (Item => c, Width => 10);
          Ada.Integer_Text_IO.put (Item => x, Width => 15);
          new_line;
     END LOOP;
END Final_01;
```

Answer:

```
Program Final_01 -- Print Number Sequence

    Increment    Next Number
    ---------    -----------
         2              2
         3              5
         4              8
         5             15
         6             22
         7             37
         8             52
         9             83
        10            114
```

Grading:

"Program" line	1 point
Column headers	1 point
Increment column indent	1 point
Increment column values	1 point
Next Number column indent	1 point
Next Number column values	2 points

Final Examination Sequential Problem, Treatment Group Two, Fall 1998

Problem 3. (10 points total)
The PACKAGE BODY distance is defined below. There are four items to be answered.

```
PACKAGE BODY distance IS

     FUNCTION total_feet
          (miles, furlongs, yards, feet, inches: IN Integer)
          RETURN Float IS
     BEGIN
          RETURN Float (miles * 5280 +
                        furlongs * 8 +
                        yards * 3    +
                        feet) +
                 Float (inches) / 12.0;
     END total_feet;

     FUNCTION total_yards
          (miles, furlongs, yards, feet, inches: IN Integer)
          RETURN Float IS
     BEGIN
          RETURN total_feet
                 (miles, furlongs, yards, feet, inches) / 3.0 ;
     END total_yards;

END distance;
```

Part a. Write the PACKAGE specification for the PACKAGE BODY Distance (2 points)

Part b. Write a new FUNCTION total_furlongs that is outside the PACKAGE, given that the first line of the program is as follows: "WITH distance; USE distance;" (2 points)

Part c. What is encapsulated in the PACKAGE distance (2 points)

Part d. Why does Ada support both PACKAGE specifications and PACKAGE BODYs (4 points)

Part a answer:

```
PACKAGE distance IS
   FUNCTION total_feet
          (miles, furlongs, yards, feet, inches: IN Integer)
          RETURN Float ;

   FUNCTION total_yards
         (miles, furlongs, yards, feet, inches: IN Integer)
         RETURN Float ;
END distance;
```

Grading: 1 point per FUNCTION and
-1 point for PACKAGE / END error

Part b answer:

```
FUNCTION total_furlongs
     (miles, furlongs, yards, feet, inches: IN Integer)
     RETURN Float IS
BEGIN
     RETURN total_feet (miles,furlongs,yards,feet,inches) / 660.0;
END total_furlongs;
```

Grading: 1 point for FUNCTION / BEGIN / END syntax
1 point for RETURN and computation

Part c answer:

Two functions are encapsulated in the package distance.
The functions are named total_feet and total_yards.

Grading: 1 point per FUNCTION named.

Part d answer:

PACKAGE specifications are interfaces to people to define PACKAGE funcationality. PACKAGE specifications allow the distribution and use of PACKAGEs without the internals of the BODY being known and seen by people. PACKAGE BODYs are interfaces to the compiler to define the internal operations of the PACKAGE to the compiler.

Grading: 2 points for specification
2 points for body
-1 mostly correct, but not above language

Final Examination Sequential Problem, Treatment Group Two, Fall 1998

```
Problem  5.  (8 points total)
What is the output of the program given below:

     WITH Ada.Text_IO;  USE Ada.Text_IO;
     PROCEDURE final_05 IS
          limit : CONSTANT Integer := 10;
          TYPE new_array IS ARRAY (1..limit) of Character;
          text : new_array :=('F','I','N','A','L','*','E','X','A','M');
     BEGIN
          FOR i IN 2..limit LOOP
               FOR j IN 1..i LOOP
                    put (Item => text(limit-j));
               END LOOP;
               new_line;
          END LOOP;
          EXCEPTION
               WHEN Constraint_Error =>
                    put_line (Item => "Go to Pizza Hut and Relax");
               WHEN Data_Error       =>
                    put_line (Item => "Go to Hamburger Hamlet");
     END final_05;

     Answer for final_05 is:

          AX
          AXE
          AXE*
          AXE*L
          AXE*LA
          AXE*LAN
          AXE*LANI
          AXE*LANIF
          AXE*LANIFGo to Pizza Hut and Relax

      Grading:  Positive logic:  Triangular shape found  +3
                                 Computation correct     +3
                                 Constraint error found  +2
                Negative logic:  Index off               -1
                                 Wrong direction         -1
                                 Loop order              -1
                                 No new_line             -1
                                 One loop missing        -1
```

Final Examination Sequential Problem, Treatment Group Two, Fall 1998

```
Problem  9.  (10 points total)

  Part a.  Rewrite the FUNCTION as a PROCEDURE.
           Also rewrite the calling statement.  (4 points)

           FUNCTION lex (par : Float) RETURN Float IS
           BEGIN
                RETURN 3.14159 + ((par - 21.0) * 91.67) / 1.719;
           END lex;

           --  calling statement
           my_num := lex (par => my_num);

  Part b.  Write a FUNCTION that solves the following expression:
           (3 points)
               2        1
           a (x ) - b (x ) + c
           -------------------, where a, b, c, d, and e are integers
                    e
                   d

  Part c.  Write a PROCEDURE that solved the following expression:
           (3 points)
                 e           2     3
           r = (g ) + (j - g)   * (c ),
               where a, c, e, g, j, and r are naturals

     Answer for Part a:
          PROCEDURE next_value (par : IN Float; result : OUT Float) IS
          BEGIN
               result := 3.14159 + ((par - 21.0) * 91.67) / 1.719;
          END next_value;

          --  calling statement
          lex (par => my_num, result => my_num);

     Answer for Part b:
          FUNCTION x (a, b, c, d, e : IN Integer) RETURN Integer IS
          BEGIN
               RETURN (a * x * x - b * x + c) / (d ** e);
          END x;

     Answer for Part c:
          PROCEDURE y (a, c, e, g, j : IN Natural; r : OUT Natural) IS
          BEGIN
               r : = g ** e + ((j - g) ** 2) * c ** 3
          END y;

     Grading:  Routine type / BEGIN / END                     1 point
                       type is either FUNCTION or PROCEDURE
               Arithmetic assignment statement                1 point
               Parameters                                     1 point
               Calling statement (Part a only)                1 point
```

Final Examination Sequential Problem, Treatment Group Two, Fall 1998

Problem 10. (10 points total)
Ada supports FUNCTIONs and PROCEDUREs and provides different parameter modes.

Part a. Name the parameter modes and given a one sentence description. Which parameter mode is the default?(4 points)

Part b. Which parameter modes can be used by FUNCTIONs? (2 points)

Part c. Which parameter modes can be used by PROCEDUREs? (2 points)

Part d. Which parameter modes were used in class with TASKs? (2 pts)

Answer Part a:

IN mode -- parameter's value is passed to the called routine, and parameter's value can not be changed in called routine.
IN mode is the default.

OUT mode -- parameter's value is passed out to the calling routine parameter's value must be assigned by called routine

IN OUT mode -- parameter's value is passed to the called routine, and parameter's value may be changed by the called routine, and parameter's value is passed out to the calling routine

Grading: 1 point per mode / 1 point for correct default

Answer for Part b:
FUNCTIONs use "IN" mode parameters only. *Grading:* 2 points

Answer for Part c:
PROCEDUREs use all three modes of parameters.

Grading: 2 points for all three / 1 for two

Answer for Part d:
TASKs used in class use "IN" mode parameters only.

Grading: 2 points for "IN"

Final Examination Sequential Problem, Treatment Group Two, Fall 1998

Problem 11. (12 points total)
The map below provides information about part of a elementary school:

```
.-----------------------------------------------------------------.
| Room 121        | Room 123        | Room 125        | Room 127        |
|                 |                 |                 |                 |
| Ms. Harris      | Ms. Johnson     | Ms. Menna       | Ms. Meanny      |
|  (married)      |  (married)      |  (unmarried)    |  (divorced)     |
|                 |                 |                 |                 |
|  1st grade      |  2nd grade      |  3rd grade      |  4th grade      |
|  17 boys        |  15 boys        |  15 boys        |  16 boys        |
|  14 girls       |  21 girls       |  18 girls       |  16 girls       |
|  1  hamster     |  2  hamsters    |  1  snake       |  no pets        |
|                 |  1  frog        |  2  hamsters    |                 |
|-----------------+-----------------+-----------------+-----------------|
|                                                                       |
|                              Hallway                                  |
|                                                                       |
|-----------------+-----------------+-----------------+-----------------|
| Room 122        | Room 124        | Room 126        | Room 128        |
|                 |                 |                 |                 |
| Ms. Peters      | Ms. Solo        | Ms. Myers       | Ms. Ford        |
|  (married)      |  (widowed)      |  (divorced)     |  (unmarried)    |
|                 |                 |                 |                 |
|  5th grade      |  6th grade      |  Kindergarden   |  Nursery        |
|  20 boys        |  18 boys        |  15 boys        |  12 boys        |
|  16 girls       |  17 girls       |  15 girls       |  14 girls       |
|  no pets        |  30 frogs       |  4  hamsters    |  no pets        |
'-----------------------------------------------------------------'
```

Part a. Write the Ada statements to declare a record that hold information about a class, (4 points)

Part b. Write the Ada statements to declare an array of records (one record per class). (4 points)

Part c. Write the Ada statements place Ms. Johnson's class Information into record number 2. (4 points)

One possible solution for part a:

```
TYPE marital IS (married, unmarried, divorced, widowed);
TYPE class   IS RECORD
     teacher  : String (1..10);
     status   : marital;
     room_num : Natural;
     grade    : Character;
     boys     : Natural;
     girls    : Natural;
     hamsters : Natural;
     snakes   : Natural;
     frogs    : Natural;
END RECORD;
```

One possible solution for part b:

```
max    : CONSTANT Natural := 8;
TYPE class_array IS ARRAY (1..max) of family;
class_data : class_array;
```

One possible solution for part c:

```
this : family; -- is assumed

this.teacher   := "Johnson   ";
this.status    := married;
this.room_num  := 123;
this.grade     := '2';
this.boys      := 15;
this.girls     := 21;
this.hamsters  := 2;
this.snakes    := 1;
this.frogs     := 1;

family_data(2) := this;
```

Grading for part a:

Enumeration declaration	1 point
TYPE … Record / END RECORD	1 point
Identifier Definitions	2 points, as qualified below:
Missing identifiers	-1 point (more than one missing)
Incorrect definition	-1 point (such as, "x : string;")

Grading for part b:

TYPE declaration	2 points
Array Declaration	2 points

Grading for part c:

Valid syntax	2 points
Most values assigned	1 point
Lack of careless errors	1 point

Final Examination Concurrency Problem For Both Treatment Groups

Problem 2. (10 points total)
In class, we discussed semaphores and busy waits. Write two small tasks as specified below:

Part a: Write a semaphore task that has the following four states in the order specified: pink, mauve, gray, beige. The task must also be able to terminate correctly (pick any state for termination). Write only the TASK BODY. (5 points)

Part b: Write a busy wait task that repeats the word "Hello" until the task receives the message goodbye. Write both the TASK specification and the TASK BODY. (5 points)

One possible solution for Part a:

```
TASK BODY semaphore IS
BEGIN
     LOOP
          ACCEPT pink;
          ACCEPT mauve;
          ACCEPT gray;
          SELECT
               ACCEPT beige;
          OR
               ACCEPT quit;  EXIT;
          END SELECT;
     END LOOP;
END semaphore;
```

One Possible solution for Part b:

```
TASK busy_wait IS  --  or TASK TYPE busy_wait IS
     ENTRY goodbye;
END busy_wait;

TASK BODY busy_wait IS
BEGIN
     LOOP
          put_line (Item => "Hello");
          SELECT
               ACCEPT goodbye;  EXIT;
          ELSE
               NULL;
          END SELECT;
     END LOOP;
END busy_wait;
```

Grading Part a:	Syntax	2 points
	Logic	2 points
	Exit Loop	1 point
	Nothing off for extra code	
Part b:	Syntax	2 points
	Logic	2 points
	Entry / Accept / Exit	1 Point
	Multiple goodbyes	-1 point
	Nothing off for extra code	

Final Examination Concurrency Problem For Both Treatment Groups

Problem 4. (12 points total)
The program below contains two tasks. Read the program and then answer the questions after the program.

```
WITH Ada.Text_IO;         USE Ada.Text_IO;
WITH Ada.Float_Text_IO;   USE Ada.Float_Text_IO;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
PROCEDURE game IS

     time_left : Duration := 23.0;

     TASK TYPE player (my_id, offset, pause : Natural);

     TASK BODY player IS
     BEGIN
          DELAY Duration(offset);
          LOOP
               put (Item => "Player");    put (Item => my_id, Width => 2);
               IF    time_left = 0.0 THEN put (Item => " won! ");  EXIT;
               ELSIF time_left = 1.0 THEN time_left := 0.0;
                                          put (Item => " lost!");  EXIT;
               ELSIF time_left = 2.0 THEN time_left := 1.0;
                                          put (Item => " time left is  1.0");
               ELSE  put (Item => " time left is");
                     put (Item => Float(time_left), Fore=>3, Aft=>1, Exp=>0);
                          time_left := time_left - Duration (my_id);
               END IF;
               new_line;
               DELAY Duration(pause);
          END LOOP;
     END player;

          player_one : player (my_id => 1, offset => 1, pause => 2);
          player_two : player (my_id => 2, offset => 0, pause => 2);
     BEGIN
          NULL;
     END game;
```

Part a. What is the output of the program given above. (6 points)

Part b. The player_one and player_two lines of the program are changed to read as shown below. Which player wins? (4 points)

```
player_one : player (my_id => 1, offset => 1, pause => 1);
player_two : player (my_id => 2, offset => 1, pause => 1);
```

Part c. The variable "time_left" is outside the tasks. What is this variable called? (2 points)

Problem 4. (continued)

```
                                                   Grading:

Answer a:  Player 2 time left is 23.0    Correct start    1 point
           Player 1 time left is 21.0
           Player 2 time left is 20.0    ----------------
           Player 1 time left is 18.0
           Player 2 time left is 17.0
           Player 1 time left is 15.0
           Player 2 time left is 14.0
           Player 1 time left is 12.0    Correct middle  3 points
           Player 2 time left is 11.0       Some math errors -1
           Player 1 time left is  9.0       Format problems  -1
           Player 2 time left is  8.0
           Player 1 time left is  6.0
           Player 2 time left is  5.0
           Player 1 time left is  3.0
           Player 2 time left is  1.0    -----------------
           Player 1 lost!Player 2 won!   Correct Ending  2 points

Answer b:  Do not know!  Race condition between two tasks.
           Non-deterministic behavior.

           Grading: any of three statements explained    4 points
                      or
                    any similar concept                  2 points

Answer c:  Shared memory variable.                       2 points
           Partial credit for "global" used by "both"    1 point
```

Final Examination Concurrency Problem For Both Treatment Groups

Problem 7. (10 total points)
Write two TASKs (one named "send" and one named "receive"). The two tasks communicate by message passing. The task "send" sends the lower case letters of the alphabet, one character at a time, to the task named "receive". The task "send" communicates to the task "receive" that it is time to stop execution by sending the tilde character ('~').

One possible solution is:

```
                                                  Grading:
TASK send;                                            1 point

TASK receive IS                                       1 point
     ENTRY input (message : IN Character);
END receive;

TASK BODY send IS
     TYPE new_array IS ARRAY (1..26) of Character;
     letters : new_array :=  ('a','b',........,'y','z');
BEGIN
     FOR i IN 1..26 LOOP                          --+
          receive.input (message => letters(i) );   | 2 points
     END LOOP;                                    --+
     receive.input (message => '~');    --  the rest 2 points
END send;

TASK BODY receive IS
     x : character;
BEGIN
     LOOP
                                                  --+
          ACCEPT input (message : IN Character) DO  | 3 points
               x := message;                        |
          END input;                              --+
          IF x = '~' THEN  EXIT;  END IF;
          put (Item => x);                  -- the rest 1 point
     END LOOP;
END receive;
```

Final Examination Concurrency Problem For Both Treatment Groups

Problem 8. (9 points total)
The prior problem, problem 7, uses message passing. In this problem, use shared memory concepts to answer the questions below. The shared memory variable is as follows:

```
    sml : Character := ' ';  -- shared memory letter
```

Part a. Describe how access to shared memory variables is controlled. (2 points)

Part b. Describe how access to shared memory variables was controlled in project 9. (2 points)

Part c. Rewrite the TASK BODY send (of problem 7) for shared memory access. DO NOT WRITE any other task; just assume what is needed exits. (5 points)

Answer a:

Access to shared memory is controlled as a "critical region". Only one task at a time can enter a critical region. Critical regions can be established using semaphores. (2 points)

Answer b: (same answer as Part a, just related to project 9)

Access to shared memory is controlled as a "critical region". Only one task at a time can enter a critical region. Critical regions can be established using semaphores. (2 points)

Answer c:

```
TASK BODY send IS
     TYPE new_array IS ARRAY (1..26) of Character;
     letters : new_array := ('a','b', to ,'y','z');
BEGIN
     FOR i IN 1..26 LOOP
          synch.lock;  sml := letters(i);  synch.unlock;
          delay 0.0;
     END LOOP;
     synch.lock;  sml := '~';  synch.unlock;
END send;
```

Grading of Part c:

TASK / BEGIN / END syntax	2 points
Critical region entry	1 point
sml := letters(i);	1 point
Critical region leave	1 point
See any message passing	-1 point

Sample Final Exam Question 6 -- Material Not Included In The Experiment

```
Problem  6.  (12 total points)
This problem involves the seven dwarfs and moving dwarfs around.
Complete the program named "final_06" by doing the following:

  a.  Assign the seven dwarfs to the array 'x'
  b.  Display the names of the dwarfs in order
  c.  Input the name of one dwarf
  c.  Swap the dwarf name with another dwarf name that is as "far away"
      from its original position as possible
  d.  Display the names of the dwarfs in the new order
  e.  Sample program execution is given below:

      The original dwarf order is:
      sleepy  bashful  dopey  grumpy  happy  doc  sneezy

      Enter dwarf name -> doc

      The new dwarf order is:
      doc  bashful  dopey  grumpy  happy  sleepy  sneezy
      ^                                   ^
```

One possible solution is:

```
     WITH Ada.Text_IO;  USE Ada.Text_IO;
     PROCEDURE final_06 IS
          TYPE dwarf IS (sleepy, bashful, dopey, grumpy,
                         happy, doc, sneezy);
          PACKAGE dwarf_io IS NEW
               Ada.TEXT_IO.Enumeration_IO (enum => dwarf);
          TYPE dwarf_array IS ARRAY (1..7) of dwarf;
          x :  dwarf_array;
```

The answer is as follows:

```
          dwarf_name : dwarf;
          num        : Natural;
     BEGIN
          FOR i IN 1..7 LOOP                         --  Code Group 1
               x(i) := dwarf'val(i-1);
          END LOOP;

          put_line (Item => "The original dwarf order is: ");
          FOR i IN 1..7 LOOP                         --  Code Group 2
               put (Item => "  ");
               dwarf_io.put (Item => x(i));
          END LOOP;
          new_line;

          put (Item => "Enter dwarf name -> ");      --  Code Group 3
          dwarf_io.get (Item => dwarf_name);
          new_line;
     num := dwarf'pos(dwarf_name) + 1;          --  Code Group 4
     IF num > 4 THEN
         x(1)   := dwarf_name;
         x(num) := dwarf'val(0);
     ELSE
         x(7)   := dwarf_name;
         x(num) := dwarf'val(6);
     END IF;

     put_line (Item => "The new dwarf order is:");
     FOR i IN 1..7 LOOP                         --  Code Group 5
          put (Item => "  ");
          dwarf_io.put (Item => x(i));
     END LOOP;
     new_line (spacing => 2);
END final_06;
```

Grading: Problem is designed as extra credit problem.
Students told in class to do this one last.

Code Group	Spring 1998
1	3 points
2	2 points
3	2 points
4	3 points
5	2 points

APPENDIX H. CLASS SURVEY

CSci 49/50/51/131 Student Survey Form - Fall 1997 Semester

This form is set up to make it fairly easy for you to enter the
desired information. Just start typing at the beginning of each line.
______________________________ Course and Section
______________________________ Your Last (family) Name
______________________________ Your First(given) Name
______________________________ Your phone number
______________________________ Your e-mail address
______________________________ Your Major or Desired Major
______________________________ Year in Your Major (1,2,3,4,Grad,None)
______________________________ Are you required to take this course?

Describe any programming experience you have, including languages you have
studied (high school, other college-level course, work-related, etc.)

____
____
____

Describe the most important things you desire to gain from taking this course.

____
____
____

If you have a computer in your home, please describe it briefly.
(Amiga, Atari, Commodore, Apple II, Macintosh, IBM or compatible,
which model, memory, disk, modem, printer, etc.)

____
____
____

APPENDIX I. STUDENT DEMOGRAPHIC DATA

Student Demographic Data -- Students In The Experiment, Control Group, Fall 1997

Class	Subject	M / F	Age	College Year	Major	Class Required	Prior Skills
1	01	F	21	1	Med E.	Yes	None
1	03	F	20	3	CS	Yes	Lost
1	05	M	19	2	CS	Yes	Lost
1	06	F	19	1	CE	Yes	None
1	07	M	23	5	CS	Yes	HS-2
1	08	F	18	1	CS	Yes	None
1	10	F	20	1	Sys Anl	Yes	None
1	13	M	19	1	CS	Yes	None
1	14	F	19	1	CS	Yes	None
1	15	F	18	1	CS	Yes	None
1	16	F	20	3	Intl Aff.	No	None
1	17	F	18	1	CS	Yes	None
1	18	M	22	2	CS	Yes	Some
1	21	M	18	1	CE	Yes	HS-1
1	22	M	18	1	CS	Yes	HS-1
1	23	M	20	3	Math	No	None
1	24	M	18	1	CS	Yes	HS-1
1	27	F	19	1	Sys Anl	Yes	None
1	28	M	18	1	CE	Yes	None
1	29	F	19	2	Econ	No	None
1	30	M	19	None	CE	Yes	Lost
1	32	M	24	5	CS	No	HTML
1	34	M	18	1	Econ	Yes	None
1	35	M	20	2	CE	Yes	HS-2

Additional legend information is presented below:

- A-1 -- Introduction to Computing, CSci 51, repeat student (Ada-1)
- C++ -- college, business, military course(s) and experience in C++
- C-1 -- one college level programming course
- C-2 -- two college level programming courses
- C-X -- college level programming course experience
- xx -- student refused or neglected to provide request information

Section 5.1 of the document, titled Student Population presents the coding of prior skills for students in the experiment.

Student Demographic Data -- Students *Not* In Experiment, Control Group, Fall 1997

				College		Class	Prior
Class	**Subject**	**M / F**	**Age**	**Year**	**Major**	**Required**	**Skills**
Excluded Due to Experience							
1	04	M	24	1	CS	Yes	C++
1	11	M	19	2	CE	Yes	C-1
1	31	M	20	3	CS	Yes	C-1
1	33	M	21	3	Math	No	C-1
Excluded Due to Withdrawal or Lack of Participation (Withdrawal = 5)							
1	02	F	18	1	CE	Yes	None
1	09	F	xx	5	CS	Yes	Lost
1	12	M	19	xx	xx	xx	xx
1	19	F	24	5	None	No	None
1	20	F	24	2	CS	Yes	None
1	25	F	19	2	History	No	None
1	26	M	18	1	CS	No	None

Student Demographic Data -- Students In The Experiment, Treatment Group One, Spring 1998

Class	Subject	M / F	Age	College Year	Major	Class Required	Prior Skills
2	01	F	18	1	CS	Yes	None
2	02	F	19	1	CS	Yes	None
2	05	M	18	1	CS	Yes	None
2	06	M	19	1	CS	Yes	HS-1
2	07	F	20	2	None	No	None
2	12	M	18	1	CS	Yes	HS-1
2	14	M	19	1	CS	Yes	HS-1
2	18	M	27	1	CE	Yes	None
2	19	F	18	1	CS	Yes	HS-1
2	20	M	18	1	CE	Yes	None
2	21	F	19	1	Crim	No	None
2	22	M	19	1	CE	Yes	HTML
2	23	M	19	1	CS	Yes	HS-1
2	24	M	19	1	CS	Yes	Some
2	25	F	18	1	CS	Yes	HS-1
2	26	M	19	1	CE	Yes	HS-1
2	29	M	19	1	CS	Yes	None
2	30	M	19	1	CE	Yes	None
2	31	M	18	1	CS	Yes	None
2	32	F	18	1	CS	Yes	HTML
2	33	M	19	1	CS	Yes	None
2	34	F	19	1	CS	Yes	None
2	35	F	18	1	CS	Yes	HS-1
2	37	M	18	1	None	No	HS-1
2	38	M	18	1	CS	Yes	None
2	42	M	19	1	CE	Yes	None
2	43	M	18	1	CS	Yes	None
2	45	F	18	1	CS	Yes	None
2	46	M	18	1	CE	Yes	None
2	47	F	20	1	Sys Anl	Yes	None
2	48	M	19	1	CS	Yes	None
2	49	F	19	1	CS	Yes	None
2	50	M	19	1	CE	Yes	HS-1
2	51	M	18	1	CS	Yes	None
2	52	F	18	1	CE	Yes	None
2	54	M	20	1	Social	No	Lost
2	57	M	20	2	Poli. Sci	Yes	None
2	58	M	18	1	CS	Yes	HS-1

Class	Subject	M / F	Age	College Year	Major	Class Required	Prior Skills
2	59	M	18	1	CS	Yes	HS-2
2	60	M	18	1	CE	Yes	None
2	61	F	18	1	Sys Anl	Yes	None
2	62	F	18	1	Sys Anl	Yes	None
2	63	F	18	1	CS	Yes	None
2	64	M	19	1	CS	Yes	Self
2	66	M	18	1	CS	Yes	None
2	68	M	21	3	Psych	Yes	None
2	69	M	19	1	CS	Yes	HS-2
2	70	M	19	1	CS	Yes	HS-2
2	72	M	20	1	CS	Yes	HTML
2	73	M	18	1	CS	Yes	Self
2	75	M	17	1	CE	Yes	None
2	76	F	17	1	CS	Yes	None

Student Demographic Data -- Students *Not* In Experiment, Treatment Group One, Spring 1998

Class	Subject	M / F	Age	College Year	Major	Class Required	Prior Skills
Excluded Due to Experience							
2	28	M	20	2	Psych	No	C-1
2	40	M	19	1	CS	Yes	C++
2	41	M	27	2	CE	Yes	C++
2	56	M	19	2	CE	Yes	C-2
2	67	M	22	2	CE	Yes	C-1
Excluded Due to Lack of Participation or Suicidal Final Exam Score							
2	08	M	19	1	CS	Yes	HTML
2	11	M	20	2	CS	Yes	Lost
2	13	M	19	1	CS	Yes	xx
2	16	M	18	1	CS	Yes	xx
2	27	M	19	1	Finance	No	None
Excluded Due to Withdrawal (Withdrawal = 14)							
2	03	M	18	1	Sys Anl	Yes	None
2	04	M	xx	xx	xx	xx	xx
2	09	M	xx	2	Law	No	None
2	10	M	18	1	CS	Yes	HS-1
2	15	F	20	2	CS	Yes	A-1
2	17	M	xx	xx	xx	xx	xx
2	36	M	29	1	CE	Yes	None
2	39	M	25	1	CS	Yes	C-X
2	44	M	19	1	CE	Yes	HS-2
2	53	M	20	1	CS	Yes	None
2	55	M	18	1	CS	Yes	None
2	65	M	19	1	CS	Yes	None
2	74	F	19	2	Business	No	xx
2	71	M	18	1	CE	Yes	HS-1

Student Demographic Data -- Students In The Experiment, Treatment Group Two, Fall 1998

Class	Subject	M / F	Age	College Year	Major	Class Required	Prior Skills
3	01	M	18	1	CE	Yes	None
3	02	M	20	1	CE	Yes	None
3	04	M	43	1	CS	Yes	Lost
3	05	M	19	2	Sys Anl	Yes	None
3	06	M	20	1	CE	Yes	None
3	07	M	21	1	CS	Yes	HTML
3	09	M	19	2	CS	Yes	Some
3	10	M	18	1	CE	Yes	None
3	11	F	21	4	Comm	No	None
3	12	M	20	2	Econ	Yes	None
3	15	F	19	1	None	Yes	Java
3	16	F	18	2	Sys Anl	Yes	None
3	17	M	21	3	CE	Yes	HS-2 *
3	19	F	18	1	None	No	None
3	22	M	19	3	Econ	Yes	None
3	23	F	22	2	CE	Yes	None
3	25	M	19	2	CS	Yes	None
3	27	F	19	2	Info Sys	No	HS-1
3	32	F	18	1	CS	Yes	HS-2
3	34	M	32	5	CS	Yes	Lost
3	35	M	19	2	CS	Yes	HTML

* Equivalent experience to HS-2

Student Demographic Data -- Students *Not* In Experiment, Treatment Group Two, Fall 1998

Class	Subject	M / F	Age	College Year	Major	Class Required	Prior Skills
Excluded Due to Experience							
3	14	F	25	5	CS	Yes	C-2
Excluded Due To Handicap Interferes With Performance							
3	3	M	25	1	CS	Yes	Lost
Excluded Due To Prior Csci 51 Class Attendance							
3	21	F	18	2	CS	Yes	A-1
3	24	M	19	2	CS	Yes	A-1
3	36	M	18	2	CS	Yes	A-1
Excluded Due to Withdrawal or Lack of Participation (Withdrawals = 10)							
3	8	M	20	2	Info Sys	No	None
3	13	F	17	2	CS	Yes	None
3	18	F	21	2	Econ	No	None
3	20	M	21	1	CS	Yes	HTML
3	26	F	25	2	CS	Yes	Lost
3	28	M	18	1	CS	Yes	HTML
3	29	M	22	N/A	None	No	None
3	30	M	19	2	ER Med.	No	None
3	31	M	19	2	Russ	Yes	None
3	33	M	22	5	None	No	None
3	37	M	19	1	Sys Anl	Yes	None
3	38	M	29	1	CE	Yes	A-1

APPENDIX J. EXAMINATION AND PROJECT SCORES

This section of the document presents the examination and project scores used in the experiment. The organization of this appendix is as follows:

1. Mid-Term Examination Scores
2. Project 2 Through 5 Scores
3. Project 9 Scores -- The Last and Large Project
4. Final Examination Sequential Question Scores
5. Final Examination Concurrency Question Scores

The student codes for all experimental subjects are included in the tables.

Mid-Term Scores.

Mid-Term Scores, Control Group, Fall 1997

Student Code	Question 1	Question 2	Question 3	Question 4	Question 5	Question 6	Total Score
1_01	13	6	11	3	3	5	41
1_03	13	9	9	3	5	8	47
1_05	17	8	3	9	0	8	45
1_06	8	5	3	5	9	8	38
1_07	19	7	13	10	11	11	71
1_08	17	10	9	10	11	10	67
1_10	14	8	6	10	9	12	59
1_13	16	10	15	10	10	9	70
1_14	11	8	5	9	8	2	43
1_15	17	9	14	9	12	9	70
1_16	19	9	15	10	12	11	76
1_17	12	8	7	10	10	7	54
1_18	19	8	13	7	7	11	65
1_21	16	10	15	10	12	12	75
1_22	16	9	15	10	12	8	70
1_23	13	7	3	7	6	11	47
1_24	20	10	12	10	12	12	76
1_27	14	7	4	8	4	10	47
1_28	17	9	15	10	10	12	73
1_29	16	9	9	5	10	4	53
1_30	13	8	13	10	12	9	65
1_32	19	9	15	10	8	10	71
1_34	13	7	8	9	0	8	45
1_35	16	10	11	10	12	10	69

Mid-Term Scores, Treatment Group One, Spring 1998

Student Code	Question 1	Question 2	Question 3	Question 4	Question 5	Question 6	Total Score
2_01	14	8	6	10	11	11	60
2_02	9	7	8	0	2	2	28
2_05	18	7	8	10	8	11	62
2_06	11	7	11	9	6	9	53
2_07	12	7	8	10	7	2	46
2_12	13	5	13	10	9	11	61
2_14	14	8	5	8	12	5	52
2_18	17	8	6	9	8	6	54
2_19	11	8	8	9	10	12	58
2_20	16	8	5	10	9	10	58
2_21	13	7	8	10	9	11	58
2_22	17	7	7	10	4	4	49
2_23	13	8	13	9	9	12	64
2_24	20	6	10	9	12	12	69
2_25	13	6	9	7	4	7	47
2_26	13	9	6	9	10	10	57
2_29	13	6	10	9	10	7	55
2_30	16	7	9	10	12	7	61
2_31	18	10	11	10	9	12	70
2_32	12	5	5	4	8	8	42
2_33	16	9	9	8	12	10	64
2_34	7	3	5	2	3	0	20
2_35	10	8	5	8	2	2	35
2_37	14	7	15	8	11	12	67
2_38	19	9	13	10	12	6	69
2_42	15	8	5	7	8	11	54
2_43	6	7	1	6	7	2	29
2_45	14	8	5	10	9	12	58
2_46	10	5	2	10	12	4	43
2_47	11	5	13	8	8	9	54
2_48	20	8	11	10	9	9	67
2_49	15	9	2	10	11	12	59
2_50	10	8	11	8	10	6	53
2_51	19	9	15	10	10	12	75
2_52	16	6	9	10	12	11	64
2_54	6	6	0	8	6	6	32
2_57	11	9	12	8	8	10	58
2_58	14	9	13	10	10	12	68
2_59	19	9	14	10	11	12	75

Student Code	Question 1	Question 2	Question 3	Question 4	Question 5	Question 6	Total Score
2_60	15	6	5	10	11	10	57
2_61	15	7	6	8	10	4	50
2_62	19	8	10	10	12	10	69
2_63	17	8	12	9	7	12	65
2_64	20	8	11	10	12	12	73
2_66	16	8	8	10	5	9	56
2_68	9	6	7	8	7	2	39
2_69	11	6	11	10	7	7	52
2_70	9	7	10	10	6	6	48
2_72	17	10	14	9	7	10	67
2_73	18	8	13	10	9	12	70
2_75	18	9	11	10	10	11	69
2_76	9	6	6	10	4	3	38

Mid-Term Scores, Treatment Group Two, Fall 1998

Student Code	Question 1	Question 2	Question 3	Question 4	Question 5	Question 6	Total Score
3_01	20	10	14	10	12	12	78
3_02	9	6	3	1	6	3	28
3_04	4	7	13	8	11	6	49
3_05	15	9	8	3	0	3	38
3_06	10	2	0	5	9	2	28
3_07	17	8	8	9	7	9	58
3_09	13	8	9	10	12	12	64
3_10	19	9	11	10	12	12	73
3_11	18	8	8	9	12	12	67
3_12	18	9	10	10	11	12	70
3_15	16	6	6	8	11	12	59
3_16	19	8	13	10	9	12	71
3_17	15	7	10	8	11	6	57
3_19	14	8	3	10	11	11	57
3_22	17	6	4	0	9	10	46
3_23	10	3	3	4	3	2	25
3_25	18	9	10	9	8	12	66
3_27	17	8	11	10	5	12	63
3_32	20	8	11	10	12	9	70
3_34	16	5	5	0	0	1	27
3_35	15	5	7	5	12	9	53

Project Two Through Five Scores.

Projects Two Through Five, Control Group, Fall 1997

Student Code	Project 2	Project 3	Project 4	Project 5	Project 2->5 Average	Project 2->5 Non-Zero
1_01	18	20	20	15	18.25	18.25
1_03	20	19	18	18	18.75	18.75
1_05	20	20	14	18	18	18
1_06	20	20	20	0	15	
1_07	20	17	17	17	17.75	17.75
1_08	20	20	20	20	20	20
1_10	17	19	20	19	18.75	18.75
1_13	20	17	19	15	17.75	17.75
1_14	20	20	15	11	16.5	16.5
1_15	20	16	16	0	13	
1_16	20	19	19	19	19.25	19.25
1_17	20	20	20	12	18	18
1_18	15	18	18	0	12.75	
1_21	20	20	20	19	19.75	19.75
1_22	20	20	15	16	17.75	17.75
1_23	14	16	14	0	11	
1_24	20	20	20	9	17.25	17.25
1_27	20	16	17	13	16.5	16.5
1_28	20	19	19	16	18.5	18.5
1_29	20	20	18	17	18.75	18.75
1_30	19	20	16	18	18.25	18.25
1_32	19	20	19	19	19.25	19.25
1_34	14	16	20	0	12.5	
1_35	17	18	19	20	18.5	18.5

Projects Two Through Five, Treatment Group One, Spring 1998

Student Code	Project 2	Project 3	Project 4	Project 5	Project 2->5 Average	Project 2->5 Non-Zero
2_01	20	20	20	20	20	20
2_02	20	17	20	11	17	17
2_05	20	19	19	18	19	19
2_06	16	19	20	19	18.5	18.5
2_07	17	17	9	18	15.25	15.25
2_12	20	20	20	18	19.5	19.5
2_14	17	18	18	18	17.75	17.75
2_18	20	12	20	14	16.5	16.5
2_19	20	19	20	20	19.75	19.75
2_20	14	19	20	20	18.25	18.25
2_21	19	18	20	18	18.75	18.75
2_22	20	19	19	19	19.25	19.25
2_23	20	19	19	0	14.5	
2_24	20	20	20	19	19.75	19.75
2_25	19	19	20	16	18.5	18.5
2_26	20	19	20	20	19.75	19.75
2_29	20	17	20	15	18	18
2_30	20	19	20	19	19.5	19.5
2_31	19	18	20	19	19	19
2_32	19	20	0	19	14.5	
2_33	20	18	20	19	19.25	19.25
2_34	20	19	20	15	18.5	18.5
2_35	20	15	20	18	18.25	18.25
2_37	20	20	20	18	19.5	19.5
2_38	20	20	20	19	19.75	19.75
2_42	20	20	20	20	20	20
2_43	12	18	19	11	15	15
2_45	0	17	19	17	13.25	
2_46	20	19	20	16	18.75	18.75
2_47	20	19	20	19	19.5	19.5
2_48	20	16	20	19	18.75	18.75
2_49	14	18	20	20	18	18
2_50	20	18	20	20	19.5	19.5
2_51	19	17	20	20	19	19
2_52	20	19	20	20	19.75	19.75
2_54	16	16	14	9	13.75	13.75
2_57	20	17	19	20	19	19
2_58	15	19	20	19	18.25	18.25
2_59	20	17	20	20	19.25	19.25

Student Code	Project 2	Project 3	Project 4	Project 5	Project 2->5 Average	Project 2->5 Non-Zero
2_60	20	20	20	19	19.75	19.75
2_61	20	20	19	19	19.5	19.5
2_62	20	19	20	16	18.75	18.75
2_63	20	20	20	19	19.75	19.75
2_64	20	20	18	19	19.25	19.25
2_66	20	0	19	15	13.5	
2_68	19	19	19	18	18.75	18.75
2_69	15	17	19	17	17	17
2_70	19	17	20	18	18.5	18.5
2_72	20	18	20	18	19	19
2_73	20	20	20	19	19.75	19.75
2_75	20	19	19	0	14.5	
2_76	6	16	15	18	13.75	13.75

Projects Two Through Five, Treatment Group Two, Fall 1998

Student Code	Project 2	Project 3	Project 4	Project 5	Project 2->5 Average	Project 2->5 Non-Zero
3_01	20	20	19	19	19.5	19.5
3_02	16	19	14	13	15.5	15.5
3_04	20	19	20	14	18.25	18.25
3_05	19	18	14	13	16	16
3_06	16	18	14	14	15.5	15.5
3_07	18	18	18	15	17.25	17.25
3_09	17	0	18	16	12.75	
3_10	17	19	20	20	19	19
3_11	19	18	20	19	19	19
3_12	17	20	18	20	18.75	18.75
3_15	20	19	20	19	19.5	19.5
3_16	20	20	20	20	20	20
3_17	20	19	20	18	19.25	19.25
3_19	13	19	16	14	15.5	15.5
3_22	17	20	20	19	19	19
3_23	19	18	20	14	17.75	17.75
3_25	20	19	19	0	14.5	
3_27	20	19	19	20	19.5	19.5
3_32	20	18	19	10	16.75	16.75
3_34	16	19	18	20	18.25	18.25
3_35	16	19	20	20	18.75	18.75

Project Nine Scores -- The Last and Large Project

Project Nine Scores

Control Group Fall 1997	
Student Code	Score
1_01	37
1_03	34
1_05	24
1_06	13
1_07	35
1_08	37
1_10	35
1_13	13
1_14	30
1_15	0
1_16	35
1_17	19
1_18	0
1_21	38
1_22	36
1_23	19
1_24	38
1_27	12
1_28	28
1_29	40
1_30	0
1_32	39
1_34	19
1_35	39

Treatment Group One, Spring 1998			
Student Code	Score	Student Code	Score
2_01	31	2_43	17
2_02	21	2_45	15
2_05	35	2_46	19
2_06	19	2_47	38
2_07	21	2_48	38
2_12	34	2_49	36
2_14	20	2_50	27
2_18	27	2_51	32
2_19	34	2_52	38
2_20	23	2_54	34
2_21	36	2_57	39
2_22	37	2_58	0
2_23	36	2_59	37
2_24	27	2_60	17
2_25	34	2_61	35
2_26	34	2_62	37
2_29	38	2_63	39
2_30	35	2_64	19
2_31	28	2_66	27
2_32	14	2_68	20
2_33	33	2_69	33
2_34	28	2_70	27
2_35	30	2_72	37
2_37	25	2_73	31
2_38	33	2_75	15
2_42	36	2_76	33

Treatment Group Two, Fall 1998	
Student Code	Score
3_01	33
3_02	37
3_04	35
3_05	26
3_06	25
3_07	22
3_09	26
3_10	38
3_11	39
3_12	38
3_15	38
3_16	37
3_17	35
3_19	26
3_22	26
3_23	29
3_25	22
3_27	34
3_32	22
3_34	31
3_35	27

Final Examination Sequential Question Scores

Final Examination Sequential Question Scores, Control Group, Fall 1997

Student Code	Question 1	Question 3 *	Question 5	Question 9	Question 10	Question 11	Total Score
1_01	5	3	3	7	8	9	35
1_03	5	6	0	9	6	9	35
1_05	3	0	3	4	4	5	19
1_06	2	5	2	0	3	3	15
1_07	5	7	4	10	10	12	48
1_08	5	5	6	9	8	5	38
1_10	5	4	0	5	0	5	19
1_13	5	7	8	8	10	10	48
1_14	7	4	7	8	10	8	44
1_15	5	7	4	7	8	7	38
1_16	5	7	7	10	10	11	50
1_17	5	8	0	8	0	5	26
1_18	5	6	7	10	10	2	40
1_21	7	6	8	10	10	12	53
1_22	7	6	3	9	8	10	43
1_23	3	4	2	4	8	5	26
1_24	6	7	2	8	9	11	43
1_27	4	5	1	3	5	0	18
1_28	5	6	8	8	10	9	46
1_29	5	2	4	2	8	3	24
1_30	7	4	5	8	0	0	24
1_32	7	6	8	9	10	2	42
1_34	5	6	0	5	9	5	30
1_35	5	6	7	9	10	12	49

* The question three score includes question three part 3a (2 points), part 3d (2 points), part 3e (4 points) for a total of 8 points.

Final Examination Sequential Question Scores, Treatment Group One, Spring 1998

Student Code	Question 1	Question 3 *	Question 5	Question 9	Question 10	Question 11	Total Score
2_01	4	5	0	4	9	8	30
2_02	2	4	0	7	9	8	30
2_05	4	6	2	7	8	10	37
2_06	5	7	4	9	8	4	37
2_07	2	4	2	5	8	5	26
2_12	6	2	4	3	10	11	36
2_14	4	3	1	7	8	8	31
2_18	4	5	3	6	9	6	33
2_19	3	5	2	0	7	3	20
2_20	4	1	0	1	9	2	17
2_21	5	7	0	5	9	9	35
2_22	4	5	8	9	8	7	41
2_23	6	3	0	2	3	1	15
2_24	7	7	6	10	10	9	49
2_25	7	7	4	8	6	5	37
2_26	5	4	3	7	8	5	32
2_29	5	3	2	4	5	5	24
2_30	6	6	2	6	6	6	32
2_31	5	3	4	4	4	8	28
2_32	0	4	0	3	1	2	10
2_33	6	6	1	9	9	10	41
2_34	4	2	7	1	7	3	24
2_35	2	4	2	4	8	5	25
2_37	5	5	2	10	8	9	39
2_38	4	4	8	7	6	9	38
2_42	4	7	3	4	5	5	28
2_43	3	6	0	4	5	3	21
2_45	4	4	2	0	8	8	26
2_46	4	0	2	9	9	0	24
2_47	4	3	5	10	9	8	39
2_48	6	6	6	9	6	8	41
2_49	6	2	4	9	8	5	34
2_50	4	4	8	9	7	9	41
2_51	5	6	6	8	10	5	40
2_52	6	5	8	9	7	5	40
2_54	6	2	2	5	7	0	22
2_57	4	6	2	6	7	10	35
2_58	6	2	4	4	0	4	20

Student Code	Question 1	Question 3 *	Question 5	Question 9	Question 10	Question 11	Total Score
2_59	6	5	7	3	8	8	37
2_60	3	1	0	0	8	2	14
2_61	7	7	3	10	9	12	48
2_62	5	5	6	10	6	9	41
2_63	3	6	2	6	9	6	32
2_64	7	7	7	10	10	10	51
2_66	2	7	0	2	7	2	20
2_68	5	3	0	7	8	5	28
2_69	6	3	6	8	0	11	34
2_70	5	2	1	1	4	12	25
2_72	3	5	6	8	2	3	27
2_73	4	7	7	7	0	9	34
2_75	5	6	4	7	8	0	30
2_76	5	5	2	5	7	10	34

* The question three score includes question three part 3a (2 points), part 3c (2 points), part 3d (4 points) for a total of 8 points.

Final Examination Sequential Question Scores, Treatment Group Two, Fall 1998

Student Code	Question 1	Question 3 *	Question 5	Question 9	Question 10	Question 11	Total Score
3_01	7	5	8	8	10	12	50
3_02	4	3	0	2	8	4	21
3_04	5	6	5	6	8	6	36
3_05	3	2	3	6	0	0	14
3_06	2	0	0	4	5	2	13
3_07	5	7	8	4	8	11	43
3_09	5	7	2	6	9	12	41
3_10	5	6	7	9	10	9	46
3_11	4	5	0	6	8	11	34
3_12	7	7	8	10	10	10	52
3_15	7	6	4	7	9	8	41
3_16	6	6	7	9	10	12	50
3_17	5	5	5	6	10	7	38
3_19	5	8	6	4	9	7	39
3_22	7	5	5	7	8	8	40
3_23	5	7	0	4	9	4	29
3_25	5	2	6	0	9	6	28
3_27	6	8	6	9	10	12	51
3_32	7	8	7	9	9	8	48
3_34	5	4	4	7	8	3	31
3_35	3	5	2	8	9	4	31

* The question three score includes question three part 3a (2 points), part 3c (2 points), part 3d (4 points) for a total of 8 points.

Final Examination Concurrency Question Scores

Final Examination Concurrency Question Scores, Treatment Group One, Spring 1998

Student Code	Qu. 2a	Qu. 2b	Tot. 2	Qu. 4a	Qu. 4b	Qu. 4c	Tot. 4	Qu. 7	Qu. 8a	Qu. 8b	Qu. 8c	Tot. 8	Total Score
2_01	4	4	8	2	0	0	2	2	0	0	0	0	12
2_02	5	1	6	0	2	0	2	3	0	0	0	0	11
2_05	4	2	6	4	4	0	8	6	1	2	0	3	23
2_06	2	2	4	2	0	0	2	4	2	2	0	4	14
2_07	0	1	1	3	2	0	5	3	2	2	1	5	14
2_12	2	5	7	2	2	2	6	4	2	2	0	4	21
2_14	3	2	5	2	0	0	2	0	0	0	0	0	7
2_18	2	5	7	2	0	0	2	0	2	2	0	4	13
2_19	2	4	6	0	0	2	2	3	2	2	1	5	16
2_20	3	1	4	0	0	0	0	0	0	0	0	0	4
2_21	4	2	6	5	4	0	9	4	1	2	1	4	23
2_22	4	5	9	0	0	2	2	4	2	2	2	6	21
2_23	2	3	5	0	0	0	0	2	1	0	0	1	8
2_24	3	5	8	6	0	2	8	5	2	2	5	9	30
2_25	5	3	8	5	0	0	5	2	2	2	0	4	19
2_26	4	2	6	2	0	2	4	5	0	2	0	2	17
2_29	5	3	8	0	0	0	0	0	0	0	0	0	8
2_30	3	1	4	2	2	0	4	4	0	2	0	2	14
2_31	4	3	7	0	1	0	1	4	0	0	0	0	12
2_32	0	1	1	5	0	0	5	2	1	0	1	2	10
2_33	4	5	9	4	0	0	4	9	2	2	3	7	29
2_34	4	3	7	0	0	0	0	1	0	2	1	3	11
2_35	3	2	5	2	4	0	6	2	0	1	1	2	15
2_37	3	5	8	0	0	2	2	2	2	2	0	4	16
2_38	5	5	10	0	2	2	4	9	2	2	2	6	29
2_42	2	5	7	5	0	0	5	4	2	1	0	3	19
2_43	3	1	4	2	0	0	2	1	0	0	1	1	8
2_45	5	2	7	0	0	0	0	0	0	0	0	0	7
2_46	0	1	1	3	0	2	5	0	0	0	0	0	6
2_47	5	5	10	3	2	2	7	9	2	2	2	6	32
2_48	2	2	4	2	0	2	4	3	2	2	0	4	15
2_49	3	2	5	0	0	0	0	3	1	0	0	1	9
2_50	3	2	5	0	0	0	0	0	0	0	0	0	5
2_51	2	5	7	1	0	0	1	1	2	0	1	3	12
2_52	2	2	4	0	0	0	0	3	1	2	0	3	10
2_54	5	5	10	3	0	2	5	2	0	2	0	2	19
2_57	2	3	5	4	0	2	6	5	2	2	1	5	21
2_58	0	0	0	2	0	0	2	3	0	0	0	0	5

Student Code	Qu. 2a	Qu. 2b	Tot. 2	Qu. 4a	Qu. 4b	Qu. 4c	Tot. 4	Qu. 7	Qu. 8a	Qu. 8b	Qu. 8c	Tot. 8	Total Score
2_59	5	3	8	0	0	0	0	4	1	2	0	3	15
2_60	2	2	4	0	2	0	2	1	0	0	0	0	7
2_61	5	5	10	6	2	2	10	7	0	2	0	2	29
2_62	3	4	7	3	0	2	5	1	2	2	0	4	17
2_63	3	4	7	1	0	2	3	3	2	2	0	4	17
2_64	5	5	10	6	2	2	10	9	1	0	1	2	31
2_66	2	0	2	1	0	0	1	1	1	0	0	1	5
2_68	3	2	5	3	0	0	3	2	2	2	1	5	15
2_69	3	1	4	1	0	0	1	3	2	0	0	2	10
2_70	2	3	5	3	0	0	3	0	0	0	0	0	8
2_72	3	4	7	0	0	2	2	4	1	0	1	2	15
2_73	3	5	8	0	0	0	0	4	1	2	0	3	15
2_75	4	5	9	2	0	2	4	0	0	0	0	0	13
2_76	5	2	7	2	0	2	4	2	1	2	3	6	19

Final Examination Concurrency Question Scores, Treatment Group Two, Fall 1998

Student Code	Qu. 2a	Qu. 2b	Tot. 2	Qu. 4a	Qu. 4b	Qu. 4c	Tot. 4	Qu. 7	Qu. 8a	Qu. 8b	Qu. 8c	Tot. 8	Total Score
3_01	3	5	8	4	0	1	5	6	2	2	4	8	27
3_02	3	0	3	0	0	2	2	2	0	1	2	3	10
3_04	3	5	8	1	4	0	5	3	0	0	0	0	16
3_05	3	5	8	1	0	1	2	1	2	0	0	2	13
3_06	2	2	4	0	0	0	0	3	0	0	0	0	7
3_07	4	4	8	1	0	0	1	5	2	2	1	5	19
3_09	3	4	7	2	0	2	4	2	2	2	3	7	20
3_10	5	5	10	1	0	2	3	1	2	2	0	4	18
3_11	3	2	5	0	4	1	5	2	2	2	2	6	18
3_12	5	5	10	2	0	2	4	7	2	2	4	8	29
3_15	3	5	8	2	4	2	8	3	2	0	2	4	23
3_16	4	4	8	4	4	2	10	2	2	2	3	7	27
3_17	5	5	10	2	0	2	4	3	2	2	0	4	21
3_19	5	5	10	0	0	2	2	3	0	0	0	0	15
3_22	3	5	8	0	0	2	2	1	2	2	0	4	15
3_23	0	0	0	5	0	1	6	0	1	0	0	1	7
3_25	3	5	8	0	0	2	2	1	2	2	1	5	16
3_27	4	5	9	0	0	0	0	1	2	2	4	8	18
3_32	4	4	8	6	0	2	8	4	2	2	2	6	26
3_34	3	5	8	2	2	0	4	3	2	2	2	6	21
3_35	3	2	5	0	0	2	2	0	0	0	0	0	7

APPENDIX K. COMPILATION DATA

This section of the document presents the compilation data used in the experiment. The organization of this appendix is by group. The following information is provided for each group (both control and treatment).

1. Ratio, Project 9 Compilations / Project 2 Through 5 Compilations
2. Distinct Error Messages, Project 9
3. Total Errors / Distinct Errors, Project 9

The student codes for all experiment subjects are included in the tables

Compilation and Error Data, Control Group, Fall 1997

Student Code	Projects 2 -> 5 Compiles (PC25)	Project 9 Compiles (PC9)	Ratio (PC9 / PC25)	Project 9 Distinct Errors & Warnings	Project 9 Total Errors & Warnings	Ratio of Total / Distinct
1_01	548	127	0.231752	55	375	6.8182
1_03	246	110	0.447154	33	220	6.6667
1_05	60	47	0.783333	25	96	3.8400
1_06	**					
1_07	261	110	0.421456	29	175	6.0345
1_08	265	150	0.566038	37	441	11.9189
1_10	212	134	0.632075	29	221	7.6207
1_13	180	32	0.177778	11 *	60 *	5.4545
1_14	144	226	1.569444	49	502	10.2449
1_15	**					
1_16	85	199	2.341176	51	318	6.2353
1_17	73	83	1.136986	56	351	6.2679
1_18	**					
1_21	62	88	1.419355	23	160	6.9565
1_22	283	205	0.724382	52	548	10.5385
1_23	**					
1_24	113	173	1.530973	35	253	7.2286
1_27	163	45	0.276074	18	69	3.8333
1_28	101	112	1.108911	65	731	11.2462
1_29	161	29	0.180124	17	49	2.8824
1_30	**					
1_32	150	187	1.246667	54	217	4.0185
1_34	**					
1_35	82	30	0.365854	3 *	11 *	3.6667

* Data presented yet not used

** Insufficient project homework recorded

Compilation and Error Data, Treatment Group One, Spring 1998

Student Code	Projects 2 -> 5 Compiles (PC25)	Project 9 Compiles (PC9)	Ratio (PC9 / PC25)	Project 9 Distinct Errors & Warnings	Project 9 Total Errors & Warnings	Ratio of Total / Distinct
2_01	264	146	1.808219	60	398	6.6333
2_02	95	217	0.437788	39	424	10.8718
2_05	136	276	0.492754	48	324	6.7500
2_06	86	242	0.355372	38	306	8.0526
2_07	148	208	0.711538	57	266	4.6667
2_12	**					
2_14	175	230	0.76087	38	350	9.2105
2_18	157	107	1.46729	44	386	8.7727
2_19	160	210	0.761905	47	340	7.2340
2_20	273	169	1.615385	61	408	6.6885
2_21	192	197	0.974619	50	406	8.1200
2_22	435	594	0.732323	76	1137	14.9605
2_23	10					
2_24	206	110	1.872727	66	333	5.0455
2_25	115	226	0.50885	47	208	4.4255
2_26	241	115	2.095652	57	339	5.9474
2_29	175	238	0.735294	58	530	9.1379
2_30	193	142	1.359155	65	438	6.7385
2_31	85	152	0.559211	46	178	3.8696
2_32	**					
2_33	**			55	334	6.0727
2_34	227	190	1.194737	44	414	9.4091
2_35	322	409	0.787286	83	925	11.1446
2_37	31	77	0.402597	18	64	3.5556
2_38	237	208	1.139423	61	717	11.7541
2_42	465	225	2.066667	58	627	10.8103
2_43	**					
2_45	78	182	0.428571	32	193	6.0313
2_46	115	235	0.489362	28	247	8.8214
2_47	365	218	1.674312	73	534	7.3151
2_48	131	104	1.259615	21	89	4.2381
2_49	300	177	1.694915	54	674	12.4815
2_50	131	190	0.689474	24	71	2.9583
2_51	82	109	0.752294	34	166	4.8824
2_52	422	294	1.435374	64	1001	15.6406
2_54	73	154	0.474026	44	248	5.6364
2_57	127	168	0.755952	35	162	4.6286

Student Code	Projects 2 -> 5 Compiles (PC25)	Project 9 Compiles (PC9)	Ratio (PC9 / PC25)	Project 9 Distinct Errors & Warnings	Project 9 Total Errors & Warnings	Ratio of Total / Distinct
2_58	**					
2_59	364	302	1.205298	75	1221	16.2800
2_60	58	150	0.386667	32	407	12.7188
2_61	200	136	1.470588	64	388	6.0625
2_62	156	76	2.052632	31	172	5.5484
2_63	191	181	1.055249	47	251	5.3404
2_64	124	180	0.688889	40	185	4.6250
2_66	**					
2_68	**					
2_69	78	186	0.419355	10 *	27 *	2.7000*
2_70	16	100	0.16			
2_72	248	205	1.209756	56	397	7.0893
2_73	95	202	0.470297	38	198	5.2105
2_75	44			32	90	2.8125
2_76	190	249	0.763052	56	317	5.6607

* Data presented yet not used

** Insufficient project homework recorded

Compilation and Error Data, Treatment Group Two, Fall 1998

Student Code	Projects 2 -> 5 Compiles (PC25)	Project 9 Compiles (PC9)	Ratio (PC9 / PC25)	Project 9 Distinct Errors & Warnings	Project 9 Total Errors & Warnings	Ratio of Total / Distinct
3001	227	132	1.719697	45	447	9.9333
3002	350	132	2.651515	89	2142	24.0674
3004	414	180	2.3	70	806	11.5143
3005	115	254	0.452756	29	179	6.1724
3006	67	89	0.752809	18	504	28.0000
3007	172	160	1.075	51	260	5.0980
3009	138	185	0.745946	62	511	8.2419
3010	88	136	0.647059	37	150	4.0541
3011	142	176	0.806818	55	524	9.5273
3012	324	112	2.892857	82	695	8.4756
3015	110	204	0.539216	42	192	4.5714
3016	112	137	0.817518	40	245	6.1250
3017	**					
3019	79	123	0.642276	48	311	6.4792
3022	52	94	0.553191	39	118	3.0256
3023	83	173	0.479769	34	202	5.9412
3025	**					
3027	282	185	1.524324	62	593	9.5645
3032	**					
3034	120	252	0.47619	24	266	11.0833
3035	60	317	0.189274	17	57	3.3529

** Insufficient project homework recorded

APPENDIX L. CONCURRENCY PROGRAMMING (NON-HYPOTHESIS) PROJECT SCORES

The tables below show the unadjusted concurrent programming scores for three projects not included in hypothesis statistics. These scores are for program syntax, correctness, and completeness. The two tables present only scores for students who did the concurrent parts of the projects. The term unadjusted means that late penalties are not included in the scores. The scores are on a 10 point scale. The interesting aspect of the averages, to the investigator, is that the averages are increasing over the three projects. The second table shows the individual student scores.

Treatment Group One Spring 1998				Treatment Group Two Fall 1998			
	Project 6	Project 7	Project 8		Project 6	Project 7	Project 8
Average	8.276596	8.742857	9.695652		6.578947	8.142857	8.8
Median	10	10	10		6	8.5	9.5

Treatment Group One Spring 1998				Treatment Group Two Fall 1998			
Student Code	Project 6	Project 7	Project 8	Student Code	Project 6	Project 7	Project 8
2_01	10	10	9	3_01	5		9.5
2_02	8		10	3_02	8	10	10
2_05		10	10	3_04	9		10
2_06	5	8	9.5	3_05	7	10	9.5
2_07	4		10	3_06	5	7	9
2_12	10			3_07			9.5
2_14	5	7	10	3_09	8	4	9.5
2_18	8	5	7	3_10	5	10	10
2_19	8	10	10	3_11	5	10	10
2_20	10	10	9.5	3_12	9	10	4.5
2_21	10	10	10	3_15	6	10	10
2_22	10	9	10	3_16	10	8	7
2_23	8		10	3_17	5		
2_24	10		10	3_19	2	5	10
2_25	10	10	10	3_22	7		7
2_26		10	9.5	3_23	5	5	7
2_29	10	8	10	3_25	6		10

Treatment Group One Spring 1998				Treatment Group Two Fall 1998			
Student Code	Project 6	Project 7	Projecт 8	Student Code	Project 6	Project 7	Project 8
2_30	10	10		3_27	9	9	10
2_31	10		9.5	3_32			4
2_32	8	7	8.5	3_34	8	8	9.5
2_33	10	10	10	3_35	6	8	10
2_34	10	10	9				
2_35	6	7	10				
2_37	3		10				
2_38	10	10	10				
2_42	10	10	10				
2_43							
2_45	6	6	10				
2_46	10		10				
2_47	10	10	10				
2_48	8		10				
2_49	10	10	8.5				
2_50	8	6					
2_51	8		10				
2_52	10		10				
2_54	4		8				
2_57	10	8	10				
2_58	6						
2_59	8	10	9.5				
2_60	10	10					
2_61	8	7	10				
2_62	3	7	10				
2_63	10	10	10				
2_64	10	9	10				
2_66			9.5				
2_68			10				
2_69	8	4	10				
2_70	4	8	10				
2_72	10	10	10				
2_73	8	10	10				
2_75	10	10	10				
2_76	7		9				

APPENDIX M. STATISTICAL TESTS

There are three statistical tests used in this experiment to examine the normality of a distribution; they are as follows:

- Skewness Normality of Residuals
- Kurtosis Normality of Residuals
- Omnibus Normality of Residuals

A skewness test measures the direction and degree of asymmetry of a distribution. The result of a skewness test means the following:

- Result < 0, the distribution is long-tailed to the left [Hintze, p. 93]
- Result = 0, the distribution is symmetric
- Result > 0, the distribution is long-tailed to the right

A residual is defined as the vertical deviation of each point from a regression line. The regression line for a normal distribution is straight line with a slope of 1.0. The residuals may though of as vertical lines that connect each observation to the regression line. Skewness results are reported as shown in the table below. The test value is the test statistic. The probability is a p-value for a two-tailed test for normality [Hintze, p. 218]

Assumption	Test Value	Probability	Decision (0.05)
Skewness Normality of Residuals	-2.5731	0.010078	Reject

A kurtosis test measures the heaviness of the tails of a distribution. A normal distribution has a skewness statistic equal 0.0 and a kurtosis statistic of 3.0. The result of a kurtosis test means the following:

- Result < 3, a unimodal distribution with lighter tails
- Result > 3, a unimodal distribution with heavier tails

For example, unimodal distributions with lighter tails tend to have a broader peak than the classic normal distribution. Kurtosis results are reported as shown in the table below.

Assumption	Test Value	Probability	Decision (0.05)
Kurtosis Normality of Residuals	-0.7914	0.428726	Accept

The test value is the test statistic. The probability is a p-value for a two-tailed test for normality [Hintze, p. 218]. The omnibus test combines the skewness and kurtosis tests into a single statistic of the overall normality of a distribution. The omnibus test is considered a better statistic because it combines the skewness and kurtosis test. Omnibus results are reported as shown in the table below.

Assumption	Test Value	Probability	Decision (0.05)
Omnibus Normality of Residuals	7.2473	0.026685	Reject

There are problems in the application of the normality tests:

- With a sample size < 25, the power of these normality test is questionable
- With a sample size < 50, only extreme examples of non-normality can be detected [Hintze, p.80]
- With a sample size < 100, there may be not be enough evidence in your data to reject normality

In this experiment, the data group sizes are as follows:

- 24 for the Fall 1997 class, control group
- 52 for the Spring 1998 class, treatment group one
- 21 for the Fall 1998 class, treatment group two

The data group sizes are large enough in many cases to disprove normality. In all cases the data group sizes are too small to prove normality. Therefore, although the normality tests are run, the statistical tests chosen are always non-parametric.

Some of the statistical tests used in this experiment assume that the data groups have equal variances. The equal-variance (modified Levene) test is an excellent test for the equality of variances. The test is one of the most robust and powerful tests for equality of variance [Hintze, p.219]. This test can be important when the variance of a smaller group (like the Fall 1997 class) is larger than the variance of a larger group (like the Spring 1998 class). The test results are reported as shown in the table below.

Assumption	**Test Value**	**Probability**	**Decision (0.05)**
Modified-Levene Equal-Variance Test	1.1602	0.317881	Accept

The test value is the result of a one-way analysis of variance of the absolute values of the median differences, and the corresponding p-value is shown in the probability column. Probabilities of 0.3 and above are common in this experiment, and these probabilities are more than sufficient to justify using statistical tests with equality of variance assumptions.

In this experiment there are two statistical test used that include the assumption of equality of variance:

- Mann-Whitney U Test
- Kruskal-Wallis One-Way ANOVA on Ranks Test

The Mann-Whitney U Test is the non-parametric test used instead of the equal-variance T-test; this test is used in comparing two groups. The Mann-Whitney U Test is based on the following assumptions:

- The data groups have equal variances
- The data groups are independent populations
- The data is ordinal -- the data is order strictly by magnitude
- The data groups are random samples of their respective populations
- The data is continuous -- the continuous data assumption is met with five or more unique values

The test results are reported as shown in the table below. Two alternative hypotheses are presented. The "D(1) <> D(2)" hypothesis is the null hypothesis (the probability that there is no significant difference between the two groups). The second hypothesis shown is the difference hypothesis with the highest probability (in this case distribution one is probably less than distribution two). The test value shown is statistic value.

Alternative Hypothesis	Test Value	Probability	Decision (0.05)
D(1) <> D(2)	5.1836	0.000000	Reject Ho
D(1) < D(2)	5.1902	1.000000	Accept Ho

The Kruskal-Wallis One-Way ANOVA on Ranks test is the non-parametric test used instead of the One-Way ANOVA test; this test is used in comparing three or more groups. The Kruskal-Wallis One-Way ANOVA on Ranks test is based on the following assumptions:

- The data groups have equal variances
- The data groups are independent populations
- The data is ordinal -- the data is order strictly by magnitude
- The data groups are random samples of their respective populations
- The data is continuous -- the continuous data assumption is met with five or more unique values

The number of degrees of freedom associated with the Kruskal-Wallis One-Way ANOVA on Ranks test is one less than the number of groups. In this experiment, three groups are compared; thus, the number of degrees of freedom is two. The test results are reported as shown in the table below. The Kruskal-Wallis One-Way ANOVA on Ranks test is slightly sensitive to the presence of tie observations in a single sample. An example of tie observations is two or more students with the score of 75 on a test. Therefore, the "corrected for ties" statistic value is reported. Ties typical affect the 3rd or 4th decimal position of the probability.

Method	Chi-Square (H)	Probability	Decision (0.05)
Corrected for Ties	1.961352	0.375058	Accept Ho

In the event that the data groups do not pass an equality of variance test (Modified-Levene), the non-parametric Kolmogorov-Smirnov test is used. This test is used to detect a significant difference in two samples or distributions. The Kolmogorov-Smirnov test is based on the following assumptions:

- The data groups are independent populations
- The data is ordinal -- the data is order strictly by magnitude
- The data groups are random samples of their respective populations
- The data is continuous -- the continuous data assumption is met with five or more unique values

The test results are reported as shown in the table below. The test results indicate that the two distributions are not the same.

Hypothesis	**Criterion Value**	**Probability**	**Decision (0.05)**
D(S) <> D(2)	0.303922	0.0112	Reject Ho

The correlations are computed using the Spearman's rho (row-wise deletion) technique. The Spearman's rho correlation coefficient measures the monotonic association between two variables in terms of ranks. The Spearman's rho is a non-parametric technique. The technique works well with data that has the following characteristics:

- Non-normal
- Non-linear relationships
- Unequal variance between groups compared

A Spearman's rho coefficient of 0.00 indicates zero correlation between the variables. A coefficient of 0.95 indicates a very high positive correlation between the variables. A coefficient of -0.95 indicates a very high negative correlation between the variables. Correlation coefficients between -0.20 and 0.20 indicate low correlations (either negative or positive). The Spearman's rho correlation coefficient is computed by using the Pearson correlation technique applied to ranks of data.

APPENDIX N. INTRODUCTORY ADA CONCURRENCY SUMMARY

The Introductory Ada Concurrency Summary presented on the following pages is in Web page format. The page format for this dissertation is diffcrent than the original Web page format of the text. Therefore, the appearance of the text is different when viewed using a browser.

Computer Science 1

Introductory Ada Concurrency Summary

Tasks -- Just The Basics

The purpose of this document is to provide just enough Ada Concurrency programming information for an entry level student to succeed in a first programming class (Computer Science CS1). As in any introductory programming class, some topics are omitted. This document is based on a student having several weeks (about eight) of sequential programming experience.

The reserved words are in upper case (like TASK) for easy recognition.

TASK Index

TASKS

■TASK Introduction

Tasks are the basic unit of concurrency in Ada. Tasks are declared and have TASK BODYs. Tasks begin once they are created (like when the program begins). Tasks are not called, unlike FUNCTIONs and PROCEDURES. Tasks can communicate with other tasks:

- Tasks can pass messages, called **message passing**
- Tasks can share variables, called **shared memory**

Both programs and tasks are scheduled. Unlike programs, the effects of task scheduling are often visible. Thus, this narrative provides a discussion on the states (scheduling states) of a task. Tasks can have one or more "entry" points. Explicitly named "entry" points are declared in the task declaration. See the ENTRY and ACCEPT Statements narrative below.

■TASK Basic Structure

The following program declares one task type named "task_intro". Three individual tasks named Task_1, Task_2, and Task_3 are declared to be of type "task_intro". The three tasks start running when the program begins execution.

```
WITH Ada.Text_IO;                  --  Include Text_IO Library
WITH Ada.Integer_Text_IO;          --  Include Integer Text_IO Library
PROCEDURE task_demo_01 IS
                                   --  Task Type Specification
     TASK TYPE intro_task (message : Integer);

     TASK BODY intro_task IS       --  Task Body Definition
     BEGIN
          FOR count IN 1..5 LOOP
               Ada.Text_IO.put (Item = "Display from Task ");
               Ada.Integer_Text_IO.put (Item = message, Width = 1);
               Ada.Text_IO.new_line;
          END LOOP;
     END intro_task;
     --  Unlike procedures, these tasks are not called.
     --  These three tasks are activated once the program begins.
     Task_1 : intro_task (message = 1);
     Task_2 : intro_task (message = 2);
     Task_3 : intro_task (message = 3);
BEGIN
     NULL;
END task_demo_01;
```

The output of above program is shown below. Once the program begins execution, each of the tasks is scheduled and starts executing. The order of task execution is not the same as the order of task declaration.

```
      Display from Task 3
      Display from Task 3
      Display from Task 3
      Display from Task 3
      Display from Task 3
      Display from Task 2
      Display from Task 2
      Display from Task 2
      Display from Task 2
      Display from Task 2
      Display from Task 1
      Display from Task 1
      Display from Task 1
      Display from Task 1
      Display from Task 1
```

ENTRY And ACCEPT Statements

The ENTRY statement declares an entry point into a task. The ACCEPT statement marks (or places) an entry point into a task. More complex ENTRY and ACCEPT statements are explained later in the document (see Message Passing).

```
WITH Ada.Text_IO;                  --  Include Text_IO Library
WITH Ada.Integer_Text_IO;          --  Include Integer Text_IO Library
PROCEDURE task_demo_02 IS
                                   --  Task Type Specification
     TASK TYPE intro_task (message : Integer) IS
          ENTRY start;             --  Entry Point Into The Task
     END intro_task;

     TASK BODY intro_task IS       --  Task Body Definition
     BEGIN
          ACCEPT start;            --  Entry Point Into The Task
          FOR count IN 1..5 LOOP
               Ada.Text_IO.put (Item = "Display from Task ");
               Ada.Integer_Text_IO.put (Item = message, Width = 1);
               Ada.Text_IO.new_line;
          END LOOP;
     END intro_task;
     --  Unlike procedures, these tasks are not called.
     --  These three tasks are activated once the program begins.
     Task_1 : intro_task (message = 1);
     Task_2 : intro_task (message = 2);
     Task_3 : intro_task (message = 3);
BEGIN
     Task_1.start;
     Task_2.start;
     Task_3.start;
END task_demo_02;
```

The output of above program is shown below. Once the program begins execution, each of the tasks is scheduled and starts executing. However, the first statement after the BEGIN is the ACCEPT statement. Each task waits for a message from the main program, a "start message". Once the "start" message is received ("accepted"), the task proceeds. The order of task completion follows the order of the "Task.start" statements in the program.

```
      Display from Task 1
      Display from Task 1
      Display from Task 1
      Display from Task 1
      Display from Task 1
      Display from Task 2
      Display from Task 2
      Display from Task 2
      Display from Task 2
      Display from Task 2
      Display from Task 3
      Display from Task 3
      Display from Task 3
```

```
Display from Task 3
Display from Task 3
```

■DELAY Statement

Execution of a program or task is delayed by at least the delay time specified in a DELAY statement.

```
DELAY expression;  --  or

DELAY 0.0;         --  use of a literal
```

The DELAY statement causes a task to be rescheduled. This is shown in the example below.

```
WITH Ada.Text_IO;                  --  Include Text_IO Library
WITH Ada.Integer_Text_IO;          --  Include Integer Text_IO Library
PROCEDURE task_demo_03 IS
                                   --  Task Type Specification
     TASK TYPE intro_task (message : Integer) IS
          ENTRY start;             --  Entry Point Into The Task
     END intro_task;

     TASK BODY intro_task IS       --  Task Body Definition
     BEGIN
          ACCEPT start;            --  Entry Point Into The Task
          FOR count IN 1..5 LOOP
               Ada.Text_IO.put (Item = "Display from Task ");
               Ada.Integer_Text_IO.put (Item = message, Width = 1);
               Ada.Text_IO.new_line;
               delay 0.0;          --  Task is delayed and rescheduled
          END LOOP;
     END intro_task;
     --  Unlike procedures, these tasks are not called.
     --  These three tasks are activated once the program begins.
     Task_1 : intro_task (message = 1);
     Task_2 : intro_task (message = 2);
     Task_3 : intro_task (message = 3);
BEGIN
     Task_1.start;
     Task_2.start;
     Task_3.start;
END task_demo_03;
```

The output of above program is shown below. Once the program begins execution, each of the tasks is scheduled and starts executing. However, due to the DELAY statement, the tasks are now delayed and rescheduled at the end of each pass through the FOR LOOP. Each task periodically "goes to sleep" and waits to be rescheduled. The Ada task scheduler controls the order in which the tasks take turns executing. Coding tasks using a DELAY statement such that the tasks appear to be taking turns executing is called *cooperative multitasking*.

```
Display from Task 1
Display from Task 1
Display from Task 2
Display from Task 1
Display from Task 2
Display from Task 1
Display from Task 3
Display from Task 2
Display from Task 1
Display from Task 3
Display from Task 2
Display from Task 3
Display from Task 2
Display from Task 3
Display from Task 3
```

■SELECT and ACCEPT

SELECT and ACCEPT statements together mark entry points for messages into a task that are alternatives (or optional). In the example below, the "ACCEPT stop" entry point into the task is bypassed if no other task has sent a "stop" message.

```
WITH Ada.Text_IO;                  --  Include Text_IO Library
WITH Ada.Integer_Text_IO;          --  Include Integer Text_IO Library
PROCEDURE task_demo_05 IS
                                   --  Task Type Specification
     TASK TYPE intro_task (message : Integer) IS
          ENTRY start;             --  Entry Point Into The Task
          ENTRY stop;              --  Entry Point Into The Task
     END intro_task;

     TASK BODY intro_task IS       --  Task Body Definition
     BEGIN
          ACCEPT start;            --  Entry Point Into The Task
          LOOP
              Ada.Text_IO.put (Item = "Display from Task ");
              Ada.Integer_Text_IO.put (Item = message, Width = 1);
              Ada.Text_IO.new_line;
              DELAY 0.0;           --  Task Is Delayed and Rescheduled
              SELECT
                  ACCEPT stop;     --  Entry Point Into The Task
                  EXIT;            --  Exit The LOOP
              ELSE
                  NULL;            --  Do Nothing In The Select
              END SELECT;
          END LOOP;
     END intro_task;
     --  Unlike procedures, these tasks are not called.
     --  These three tasks are activated once the program begins.
     Task_1 : intro_task (message = 1);
     Task_2 : intro_task (message = 2);
     Task_3 : intro_task (message = 3);
BEGIN
     Task_1.start;  Task_2.start;  Task_3.start;
     delay 0.05;
     Task_1.stop;   Task_2.stop;   Task_3.stop;
END task_demo_05;
```

The output of the above program is shown below. In a SELECT block (SELECT ... END SELECT;), if there are no message waiting for an ACCEPT, the ELSE condition is executed. The "SELECT ... ELSE NULL; END SELECT;" logic allows an infinite LOOP to "check-in" for messages. If a message is waiting, the statements following the ACCEPT are executed. If no message is waiting, the LOOP is repeated. Coding tasks using SELECT blocks such that the task continues to execute while waiting for a message is called *busy waiting*.

```
Display from Task 1
Display from Task 1
Display from Task 2
Display from Task 1
Display from Task 2
Display from Task 1
Display from Task 3
Display from Task 2
..
..
..
Display from Task 1
Display from Task 3
Display from Task 2
Display from Task 1
Display from Task 3
Display from Task 2
Display from Task 3
Display from Task 2
Display from Task 3
Display from Task 3
```

In addition to *busy waiting*, the Ada language also supports the concept of *selective waiting*. The SELECT block can contain many ACCEPT statements, separated by the reserve word "OR". Messages sent to the receiving task are processed in the select block in the order they are received. The following program shows both the "OR" and "ELSE" within a single SELECT block. The program models a toy that waits for a command at one-half second intervals (once started). As in the previous example, if no message is waiting, the LOOP is repeated (see ELSE statement).

```
WITH Ada.Text_IO;                       -- Include Text_IO Library
WITH Ada.Integer_Text_IO;               -- Include Integer Text_IO Library
PROCEDURE task_demo_08 IS
                                        -- Task Type Specification
     TASK TYPE intro_task IS
          ENTRY start;                  -- Entry Points Into The Task
          ENTRY turn_left;
          ENTRY turn_right;
          ENTRY stop;
     END intro_task;
     TASK BODY intro_task IS            --  Task Body Definition
     BEGIN
          ACCEPT start;                 --  Entry Point Into The Task
          Ada.Text_IO.put_line (Item = "Toy is running");
          LOOP
              SELECT
                  ACCEPT turn_left;     --  Entry Point Into The Task
                      Ada.Text_IO.put_line (Item = "Turning left");
                  OR
                  ACCEPT turn_right;    --  Entry Point Into The Task
                      Ada.Text_IO.put_line (Item = "Turning right");
                  OR
                  ACCEPT stop;          --  Entry Point Into The Task
                      Ada.Text_IO.put_line (Item = "Toy has stopped");
                      EXIT;             --  Exit The LOOP
                  ELSE
                      Ada.Text_IO.put_line (Item = "Moving straight");
              END SELECT;
              DELAY 0.5;
          END LOOP;
     END intro_task;
     --  Unlike procedures, tasks are not called.
     --  This task are activated once the program begins.
     Task_1 : intro_task;
BEGIN
     Task_1.start;        DELAY 2.0;
     Task_1.turn_left;    DELAY 2.0;
     Task_1.turn_right;   DELAY 2.0;
     Task_1.stop;
END task_demo_08;
```

The output of the above program is shown below.

```
      Toy is running
      Moving straight
      Moving straight
      Moving straight
      Moving straight
      Turning left
      Moving straight
      Moving straight
      Moving straight
      Turning right
      Moving straight
      Moving straight
      Moving straight
      Toy has stopped
```

■Declaring Tasks

There are many ways to define and refer to tasks. Three are shown in the following table. In the first example, titled "TASK Only", the task is declared without using a TASK TYPE declaration. This works when declaring only a single instance of the task.

In the second example, titled "TASK TYPE", a task type is declared first and then followed by the declaration of the individual tasks. The second example works for declaring multiple copies of the same task type.

In the third example, titled "TASK TYPE & DECLARE BLOCK", a declare block is used to allow for tasks to be declared within the procedural part of the program (after the BEGIN). The logic within the program can determine whether a task should be declared and how often a new task should be declared.

Example	Task Declaration	Task Invocation

TASK Only	`TASK sample_1 IS`
	`     ENTRY go;`
	`END  sample_1;`
	
	`TASK BODY sample_1 IS`
	`     ..`
	`     ..`
	`END  sample_1;`
		`sample_1.go;`

TASK TYPE	`TASK TYPE sample_2 IS`
	`     ENTRY go;`
	`END  sample_1;`
	
	`TASK BODY sample_2 IS`
	`     ..`
	`     ..`
	`END  sample_2;`
	
	`Task_2a : sample_2;`
	`Task_2b : sample_2;`
		`Task_2a.go;`
		`Task_2b.go;`

TASK TYPE &
DECLARE BLOCK
	`TASK TYPE sample_3 IS`
	`     ENTRY go;`
	`END  sample_3;`
	
	`TASK BODY sample_3 IS`
	`     ..`
	`     ..`
	`END  sample_3;`
		`DECLARE`
		`     Task_3a : sample_3;`
		`     Task_3b : sample_3;`
		`BEGIN`
		`     Task_3a.go;`
		`     Task_3b.go;`
		`END;`

■Task State Model

Tasks are not always executing; they have several different states. The diagram below shows a simplified model of task states. The states are in bold print. The event or condition to occur causing a change of state is shown between the states. For example, when a running task performs a delay, the task is blocked (or waits) until the specified time has passed. Once the delay specified time has passed, the task is now ready to be rescheduled to execute.

```
                 Ready <------+
                    |         ^
               Task |         |
          Scheduled |         |
                    |         |
                    |         |
                    |         |
Completed <---- Running       | Message (for ACCEPT)
     |              |         |
     |              |         | Delay time has passed
     |              | ACCEPT  |
     |              | DELAY   |
     |              |         |
    \/              |         |
Terminated <--- Blocked ----->+
```

The task states are as follows:

- Ready -- the task is competing for a resource

  The processor is a resource
  Another task is using the processor

- Running -- the task is executing on a processor

- Blocked -- the task is waiting for an event or condition to occur

  ACCEPT processed waiting for a message
  DELAY processed, waiting for specified time to pass

- Completed -- the task body has finished executing

- Terminated -- the task is about to go out of existence, the final state

A blocked task can be killed; thus the change of state from blocked to terminated.

■Message Passing, When Tasks Rendezvous

Two or more tasks may communicate by sending messages. Message passing syntax in Ada is bi-directional. Since this is an introductory class, the message passing syntax used is limited to one directional. One or more tasks can send messages to the same task entry point; the messages are received in the order sent. Examples of message passing are shown below.

Example	Sending Task	Receiving Task

Simple Message	worker.go;	TASK worker IS
		     ENTRY go;
		END worker;
		
		TASK BODY worker IS
		     ..
		     ACCEPT go;
		     ..
		END worker;

Message With One Parameter	worker.run
	    (repeat = 5);
		TASK worker IS
		     ENTRY run (repeat : IN Integer);
		END worker;
		
		TASK BODY worker IS
		     ..
		     ACCEPT run (repeat : IN Integer) DO
		         save_repeat := repeat;
		     END run;
		     ..
		END worker;

Message With Multiple Parameters	worker.run
	    (repeat = 5,
	     length = 12);
		TASK worker IS
		     ENTRY run (repeat : IN Integer;
		                length : IN Integer);
		END worker;
		
		TASK BODY worker IS
		     ..
		     ACCEPT run (repeat : IN Integer;
		                 length : IN Integer) DO
		         save_repeat := repeat;
		         save_length := length;
		     END run;
		     ..
		END worker;

Message passing characteristics include the following items:

- Message passing is tied to object-oriented methodology
- In Ada message passing is done by rendezvous (a method of synchronization where sending and receiving tasks WAIT)
- Tasks exchange messages to synchronize activities
- Tasks exchange messages to pass data (as in the following example)

```
task_one.start (id_num = 3);           --  Sends a message

ACCEPT start (id_num : IN Natural) DO  --  Receives the message
    my_num := id_num;
END Start;
```

- Each task has a unique identifier (see CSci 131 for details)
- A message is sent to a single task
- When messages are send to all task, that is called a broadcast
- Message passing works on computers with distributed memory (memory is associated with a single processor)
- Message passing works on distributed computers on a network (like a collection of PCs)
- Message must know the target task (for example: "task_one.quit;" where the target task name and entry point are specified
- Target task may not need to know source task
- The programmer determines pattern of communication

Task Parameter Modes

Tasks have the same parameter modes as PROCEDURES in Ada. The TASK parameters are defined in the table below.

Parameter Mode	Legal In Tasks	Used In Class	Description
IN	Legal	Yes	Message contents passed into the task, are constants in the TASK, and may not be changed
OUT	Legal	No	Message contents passed out of the task, the parameter value is defined (assigned) inside the subprogram
INOUT	Legal	No	Message contents passed into the task, the parameter value is defined (assigned) inside the subprogram

Shared Memory

Shared Memory -- a global memory space that is accessible by two or more tasks. Tasks can communicate with each other by writing and reading into the global shared memory.

```
.-------------------------------------------------------------------.
|                          Shared Memory                            |
'-------------------------------------------------------------------'
      |                |                 |                     |
      |                |                 |                     |
.-----------.   .-----------.   .-------------.          .---------.
| Task One  |   | Task Two  |   | Task Three  |   ....   | Task N  |
'-----------'   '-----------'   '-------------'          '---------'
```

Shared memory solves the intertask communication problem, and introduces a new problem -- several tasks accessing the same memory location simultaneously

```
.--------------------------.
|       Shared Memory      |
|          x := 0;         |
'--------------------------'

      |               |
      |               |
.-----------.    .-----------.
| Task One  |    | Task Two  |
| x := x+1; |    | x := x+2; |
'-----------'    '-----------'
```

In the diagram above (starting with a shared memory variable x := 0), the value of x can be any of the following x is 1, x is 2, or x is 3.

- Possibility one -- the tasks start at the same time, plus x is 1 if task one finishes last
- Possibility two -- the tasks start at the same time, plus x is 2 if task two finishes last
- Possibility three -- either task starts after the other task completes -- this means x is 3

The value of x is determined by factors outside programmer control. This is an example of non-determinant behavior. This behavior is caused by race conditions.

Shared Variables -- variables in common between two or more tasks:

- Only one task at a time can write to a shared variable (without causing problems)
- One or more tasks can read from a shared variable at the same time
- Shared variables do not have to be accessed by each task

```
task_one                task_two               task_three
     |                   |    |                     |
     shared variable "a"   shared variable "b"
```

- Shared variables should have access control in order to avoid race conditions -- like use of a *critical section*
- Relatively simple to program
- Multiple processors can access a central memory (for example, Felix has one memory and four working processors / all memory is available to every processor except for small sections of memory that may be exclusively used by one processor for very short time periods)

Critical Section -- Controlling Access to Shared Memory

- Critical section as part of an infinite LOOP (see Semaphores)

```
LOOP
     non-critical section
     entry protocol
     critical section  --  <<<<<<<<<<<<<<<<<<
     exit protocol
     non-critical section
END LOOP;
```

- Definitions

Critical section -- a sequence of statements that access a shared item (like the same location in shared memory)
Mutual exclusion -- at most one task at a time accesses the shared item (like the same location in shared memory)
Contention -- two or more tasks competing for the same resource (like the same location in shared memory)
Communication -- two or more tasks passing information from one task to another task

■Semaphores

A semaphore is a collection distinct states used to control access to a critical section. A semaphore is most often an integer-based variable that can take only non-negative values. In this class a semaphore is modeled as task that is a collection of distinct states.

```
--   Two state semaphore (Lock or Unlock), Infinite LOOP
--
--   BEGIN --->+---> Lock ---> Unlock --+
--             |                        |
--             +<-----------------------+

TASK semaphore IS             --  Declare a TASK
     ENTRY lock;
     ENTRY unlock;
END  semaphore;

TASK BODY semaphore IS        --  TASK BODY demonstrates semaphore
BEGIN
     LOOP                     --  LOOP is an infinite loop
         ACCEPT lock;    put_line (Item = "locked");
         ACCEPT unlock;  put_line (Item = "unlocked");
     END LOOP;
END semaphore;
```

The example above is a model of a semaphore with two states: lock and unlock. After task "A" sends a successful "lock" message to the semaphore, the semaphore task is waiting for an "unlock" message. If task "B" sends a "lock" message to the semaphore after task "A" does, then task "B" must wait for task "A" to send an "unlock" message to the semaphore. Hence a semaphore can be used to control access to shared variables. Access to the shared variables is limited to one task at a time. The example below is a three-state semaphore.

```
--    Three state semaphore (Green, Yellow, Red), Infinite LOOP
--
--    BEGIN --->+---> green ---> yellow ---> red --+
--              |                                  |
--              +<---------------------------------+

TASK TYPE stop_light IS        --  Declare a task type
     ENTRY turn_green;
     ENTRY turn_yellow;
     ENTRY turn_red;
END stop_light;

TASK BODY stop_light IS        --  TASK BODY demonstrates semaphore
BEGIN
     LOOP                      --  LOOP is an infinite loop
         ACCEPT turn_green;    put_line (Item = "Green Light");
         ACCEPT turn_yellow;   put_line (Item = "Yellow Light");
         ACCEPT turn_red;      put_line (Item = "Red Light");
     END LOOP;
END stop_light;

     traffic_light  : stop_light;
```

The three-state semaphore proceeds through the ordered set of states: green, yellow, red, and then repeats. A four state semaphore can be created by adding another ACCEPT statement. A five state semaphore can be created by adding two ACCEPT statements.

In the previous section on shared memory, it was stated that shared variables should have access control in order to avoid race conditions. The two-state semaphore is an ideal protocol to the control the access to variables shared between tasks. In a program where multiple tasks have access to shared variables, the use of the semaphore insures that only one task at a time can access the shared variables (this is called *mutual exclusion*). The use of a semaphore task is shown in the example below.

```
semaphore.lock;     --  The entry protocol to the critical section

--  Critical Section
--  Ada source code to update variables shared between tasks

semaphore.unlock;   --  The exit protocol to the critical section
```

The examples above are infinite loops without an exit. In order for the task to complete and then terminate, a SELECT block with an EXIT must be added to the TASK BODY. The semaphore examples in the table below contain the entry point quit. These examples complete and terminate correctly.

Example	Semaphore Declaration	Semaphore Usage

Simple Semaphore	TASK semaphore IS
	     ENTRY lock;
	     ENTRY unlock;
	     ENTRY quit;
	END semaphore;
	
	TASK BODY semaphore IS
	BEGIN
	     LOOP
	          ACCEPT lock;
	          SELECT
	               ACCEPT unlock;
	          OR
	               ACCEPT quit;
	               EXIT;
	          END SELECT;
	     END LOOP;
	END semaphore;
		semaphore.lock;
		-- Critical activity
		semaphore.unlock;

Array of Semaphores	TASK TYPE semaphore IS
	     ENTRY lock;
	     ENTRY unlock;
	     ENTRY quit;
	END semaphore;
	
	TASK BODY semaphore IS
	BEGIN
	     LOOP
	          ACCEPT lock;
	          SELECT
	               ACCEPT unlock;
	          OR
	               ACCEPT quit;
	               EXIT;
	          END SELECT;
	     END LOOP;
	END semaphore;
	
	TYPE semaphore_array IS
	    ARRAY (1..5) OF
	semaphore;
	
	this_semaphore :
	    semaphore_array;
		this_semaphore(i).lock;
		-- Critical activity
		this_semaphore(i).unlock;

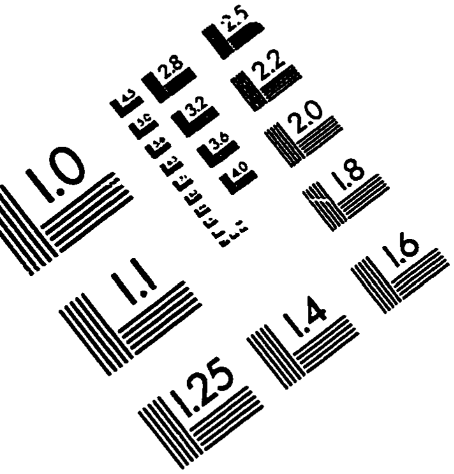
■Terms And Definitions

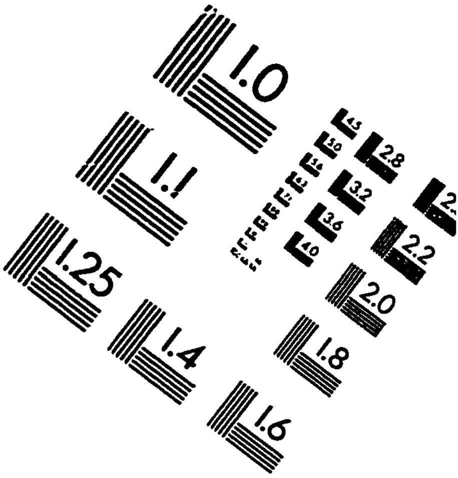
Numerous definitions related to concurrency are introduced in the class lecture. The complete collection of these definitions is placed at the end of this document (one convenience location).

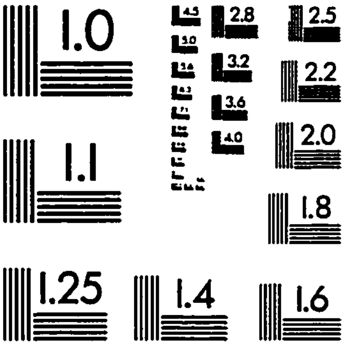
Term	Definition
Asynchronous	Refers to communication between tasks, the sending task leaves the message where the receiving tasks can get it (the implication is that the sending task does NOT wait for the receiving task(s) to acknowledge the message)
Communication	Two or more tasks passing information from one task to another task
Concurrent Program	A program specifying two or more sequential activities to be preformed as tasks
Contention	Two or more tasks competing for the same resource (like the same location in memory)
Critical Section	A section of code that performs an operation that must NOT be executed by more than one task at a time; such as, a sequence of statements that access a shared item (like the same location in memory)
Deadlock	A state where a task is waiting for an event that will NOT occur
Lockout	Indefinite postponement
Message Passing	A mechanism for enabling one task to make information available to other tasks by directing it to the tasks concerned
Multi-Tasking	The sharing of a single processor among a set of competing tasks
Mutual Exclusion	A mechanism to ensure that only one task at a time performs a specified action; such as, at most one task at a time accesses the shared item (like the same location in memory)
Non-Determinism	A program property, where results are in part determined by factors external to the program
Parallel Program	A concurrent program designed for execution on parallel hardware (implies more than one processor)
Process (Task)	A sequential program
Race Condition	Two or more tasks competing for a resource or state
Semaphore	A collection of distinct states used to control access to a critical section
Synchronous	Refers to communication between tasks, message passing between tasks such that the sending and receiving tasks rendezvous (the implication is that the sending task waits for the receiving task(s) to acknowledge the message)
Waiting	A task awaiting a change in state or a message
Wait, Busy	A task executes a loop awaiting a change in state or a message
Wait, Simple	Just waiting (no other activity occurring in the task)

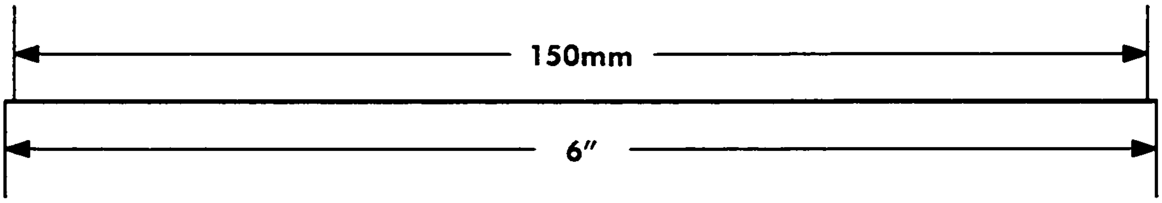
Return to Index

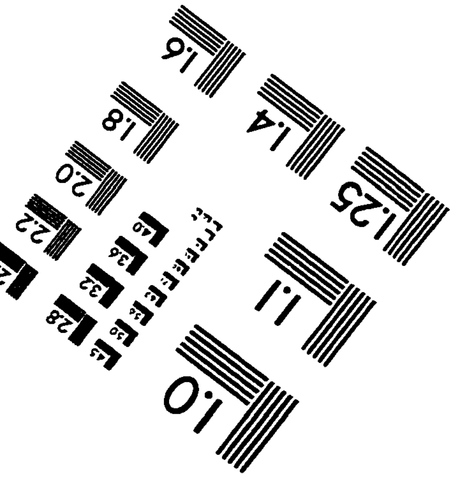
IMAGE EVALUATION TEST TARGET (QA-3)

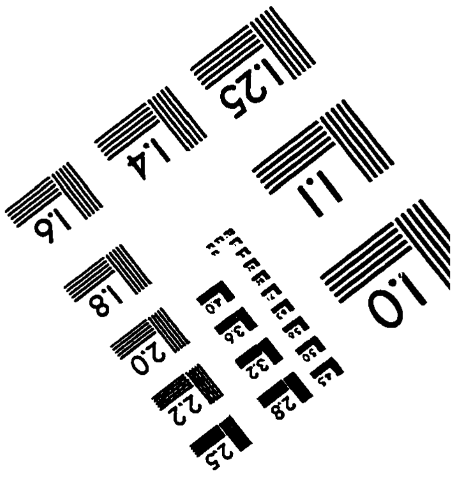












APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989