A. Cover Page

Broad Agency Announcement 93-26
CURRICULUM DEVELOPMENT IN SOFTWARE ENGINEERING AND ADA

Category 1 and 3 Proposal

Proposal Title: Curriculum and Textbook Development Using Ada 9X for the Teaching of Object-Oriented Concepts

Technical Point of Contact:

Herbert L. Dershem
Department of Computer Science
Hope College
Holland, MI 49422-8000
(616) 394-7510
dershem@cs.hope.edu

(On leave 1993-4 at the following address)
Department of Computer Science
United States Air Force Academy
USAF Academy, CO 80840
(719) 472-3590
hdershem@cs.usafa.af.mil

Administrative Point of Contact:

Elliot A. Tanis
Dean for the Natural Sciences
Hope College
Holland, MI 49422-8000
(616)394-7714
tanis@physics.hope.edu

HOPE COLLEGE

DEAN FOR NATURAL SCIENCES

November 9, 1993

BAA #93-26
ARPA/SISTO
3701 North Fairfax Drive
Arlington, VA 22203-1714

TO WHOM IT MAY CONCERN:

It is with significant enthusiasm and excitement that I endorse the ideas and concepts presented by the Department of Computer Science at Hope College in this proposal entitled "Curriculum and Textbook development using Ada 9X for the Teaching of Object-Oriented concepts."

The program described in this proposal is timely and innovative. It will allow students to take advantage of object oriented and advanced programming concepts in our curriculum through Ada 9X. The proposed curricular endeavor is welcoming of students and it is understanding of the variances in backgrounds that students bring with them, especially in contrast to other available object oriented programming environments. It is designed to meet students "where they are" and help students to achieve their fullest potential of understanding.

Hope College is committed to support this program in every way. My office will work with Professor Dershem in a coordinated fashion to insure programmatic success.

We are thankful for the opportunity to submit this proposal and we look forward to developing a vital program that will provide students with the tools necessary for future success in computer science.

Sincerely,

Elliot A. Tanis

Elliot A. Tanis
Dean for the Natural Sciences

PEALE SCIENCE CENTER / 35 EAST 12TH STREET / P.O. BOX 9000
HOLLAND, MICHIGAN 49422-9000 / 616-394-7714 / FAX 616-394-7923

B. Description of the Project

Statement of Work

Scope

This project will result in the redesign of two courses, programming languages and object-oriented programming, so that they are based on the Ada 9X language. In addition, the project director will revise his programming languages textbook so that it includes a chapter on Ada 9X under the object-oriented paradigm, and will revise the use of Ada as an illustration of the imperative and concurrent paradigms to reflect changes made in Ada 9X.

Technical Approach

Hope College is a four-year liberal arts institution with enrollment of approximately 2,700. The college has had a computer science department since 1974. The department presently consists of four full-time faculty members, three of whom hold Ph.D. degrees in Computer Science. The department graduates between 10 and 15 majors each year.

Object-oriented programming is currently taught in two places in the Hope College Computer Science curriculum. First, in the programming languages course, approximately two weeks is devoted to the object-oriented paradigm. The languages Smalltalk and C++ are introduced in accordance with the presentation of the topic in the textbook which is co-authored by the Project Director [1].

The second place that object-oriented programming in found is in a topics course which is offered following the Programming Languages course and which has the Programming Languages course as a prerequisite. The language used in this course in the past has been C++.

The first edition of the textbook mentioned above uses Ada as a language to illustrate the imperative paradigm. A second edition of this textbook has just been completed and will be published in 1994. In that edition the concurrent paradigm has been added and Ada is also used as a primary illustration of that paradigm. In the textbook's discussion of the object-oriented paradigm, both editions use the languages Smalltalk and C++ as illustrations. In the second edition, a chapter is devoted to each of these languages.

Object-Oriented programming is one of the most important and popular topics in Computer Science today. Currently, the language of choice in most of industry and academe is C++. Ada 9X shows the promise that it might radically change this situation by providing object-oriented capabilities within a well-designed structured language.In order for this to happen, effective educatioal materials must be produced which present the object-oriented model in the context of the Ada 9X language.

The purpose of this project is to incorporate Ada 9X as a prototype language for the object-oriented paradigm into the programming language and object-oriented programming courses. The biggest hindrance to doing this in the near future will be the lack of textbooks which use Ada 9X

to teach the object-oriented paradigm. This project is intended to address this problem by providing a timely revision of a textbook already on the market.

Specific Tasks

Three major tasks are proposed in this project:

1. Writing a new chapter of the Dershem/Jipping textbook describing Ada 9X as it applies to the object-oriented paradigm. This chapter will be for inclusion in the next edition of this textbook.

2. Revising the discussion of Ada in the Dershem/Jipping text in relation to the imperative and concurrent paradigms to reflect the changes in Ada 9X.

3. Redesign the Programming Languages and Object-Oriented Programming courses at Hope College to base them on Ada 9X.

Time Frame of Effort

July, 1994	Draft Ada 9X Object-Oriented Chapter for Dershem/Jipping textbook
August-December, 1994	Teach Programming Languages at Hope College using new chapter Revise chapter according to the experience using it
June, 1995	Make final revision to Object-Oriented Ada 9X chapter for submission to publisher
July, 1995	Make revisions to imperative and concurrent sections of Dershem/ Jipping textbook to reflect changes in Ada 9X and send revisions to publisher
August-December, 1995	Teach Object-Oriented Programming course at Hope College using Ada 9X
May, 1996	Prepare syllabus and other materials from Object-Oriented Programming course for dissemination and final project report

Summary of Anticipated Results

This project will have two major results. The first will be a revision to the textbook Programming Languages: Structures and Models that will result in its third edition. This revision will contain a new chapter under the object-oriented paradigm that discusses Ada 9X as it represents that paradigm. It will also have revisions to the treatment of Ada under the imperative and concurrent paradigms to reflect the changes in Ada 9X..

The Table of Contents of the Second Edition of the Dershem/Jipping textbook is given below with annotations indicating the changes that will be made as a result of this project. Sections which

appear in bold will be modified to reflect changes in Ada found in Ada 9X. The new object-oriented chapter will appear between chapters 14 and 15 as indicated in the table.

I. Overview of Programming Languages

1. Introduction and Overview
    - 1.1 What is a programming language?
    - 1.2 Why study programming languages?
    - 1.3A brief history of programming languages
2. Preliminary Concepts
    - 2.1 Syntax specification
    - 2.2 Semantics specification
    - 2.3 Language translation
    - 2.4 Language design characteristics
    - 2.5 Choice of language

II. Imperative Model

3. Overview of Imperative Model
- **3.1 Data types and bindings**
- **3.2 Execution units and scope of binding**
- **3.3 Control structures**

4. Data aggregates
- **4.1 Data aggregate models**
- **4.2 Arrays**
- **4.3 Strings**
- **4.4 Records**
- **4.5 Files**
- **4.6 Sets**

5. Procedural Abstraction
- **5.1 Procedures as abstractions**
- **5.2 Procedure definition and invocation**
- **5.3 Procedure environment**
- **5.4 Parameters**
- **5.5 Value returning procedures**
- **5.6 Overloading**
- **5.7 Coroutines**
- **5.8 Procedures in Ada**
- **5.9 Exceptions**
- **5.10 Exceptions in Ada**

6. Data Abstraction
- **6.1 Abstract data types**
- **6.3 Encapsulation**
- **6.4 Parameterization**
- **6.5 Monitors**
- **6.6 Data abstraction in Ada**

7. Example Language - C
    - 7.1 Philosophy and approach
    - 7.2 Information binding
    - 7.3 Control structures

15. Smalltalk - an Object-Oriented Language
    - 15.1 Overview
    - 15.2 Smalltalk syntax
    - 15.3 Class hierarchy
    - 15.4 Abstract classes
    - 15.5 An example in Smalltalk
16. C++ - a Hybrid Object-Oriented Language
    - 16.1 Overview
    - 16.2 Components of C++
    - 16.3 An example in C++

VI. Distributed/Parallel Model

17. Overview of the Distributed/Parallel Model
    - 17.1 Process definition
    - 17.2 Invocation of processes
    - 17.3 Data sharing
    - 17.4 Interprocess communication
    - 17.5 Synchronization
18. **Concurrent Units in Ada**
    - **18.1 Process definition**
    - **18.2 Process invocation**
    - **18.3 Data sharing**
    - **18.4 Interprocess communication**
    - **18.5 Synchronization**
    - **18.6 Examples in Ada**
19. Occam - a Parallel Language
    - 19.1 Process definition
    - 19.2 Process invocation
    - 19.3 Data sharing
    - 19.4 Interprocess communication
    - 19.5 Synchronization
    - 19.6 Examples in Occam

The second result of this project will be the design of an Object-Oriented Programming course based on Ada 9X. It is anticipated that in addition to its submision as a part of the report on this project, a paper describing the course will be submitted to the SIGCSE Bulletin.

Proprietary Claims

The Project Director will retain proprietary rights to all modifications of the textbook, Programming Languages: Structures and Models. He will include an acknowledgement to DARPA in the book. He will have no other proprietary rights to any other material created as a part of this project.

C. Summary of Deliverables

1. A new chapter of the Dershem/Jipping textbook on Ada 9X as an example of the Object-Oriented paradigm.

2. A list of revisions to be included in the Third Edition of Dershem/Jipping textbook to reflect changes to Ada 9X within the imperative and concurrent paradigms.

3. A syllabus and sample projects for an object-oriented programming course based on Ada 9X. These will be submitted in the form of a paper suitable for submission to the SIGCSE Bulletin.

D. summary of Schedule and Milestones

Date	Activity
July 1994	A rough draft of an Ada 9X object-oriented chapter for Dershem/Jipping will be completed.
Aug-Dec, 1994	Programming language courses using the draft of the Ada 9X chapter will be taught at Hope College (by Dershem).
June-July, 1995	Dershem will make final revisions tothe Ada 9X object-oriented chapter, revise the imperative and concurrent model discussions of the textbook to reflect changes in Ada 9X, and design an object-oriented course based on Ada 9X. He will be assisted by a half-time undergraduate student who will code and test programs and work exercises.
Aug-Dec, 1995	Dershem will teach the Object-Oriented course using Ada 9X at Hope College
May, 1996	Dershem will prepare final report and deliverables for this project.

Key Personnel

Herbert L. Dershem (Curriculum Vitae in Appendix)
Effort expended: four months of full-time effort, directing all activities above

Undergraduate Assistant (to be selected during Spring Semester, 1995)
Effort expended: two months of half-time effort

E. Previous Related Work

Professor Dershem has been active in computer science curriculum development for more than twenty years. His first activity was in the design of a course that combined the teaching of statistics and computer science [1]. His work on that project was supported by a grant from the National Science Foundation and resulted in the publication of a laboratory manual for use in such a course [2].

Professor Dershem was also funded by NSF for the development of a modular approach to the teaching of introductory computer science [3]. As a part of this project, two modules on problem solving were produced [4] [5].

Professor Dershem has taught, worked with, and written about Ada extensively. Evidence of this is the textbook referenced previously [7] and its second edition [8] which is due for publication in 1994. In addition, the Project Director, while on leave during 1993-94 at the United States Air Force Academy, will teach a course on object-oriented programming using Ada 9X during the Spring 1994 semester. This will provide him with background information to assist in the initiation of this project.

In addition, Professor Dershem is presently completing work on a DARPA funded project to redesign the data structures course to use the Ada language. This project will be completed in June, 1994.

Professor Dershem is also the Principal Investigator for a Research Experiences for Undergraduates program funded by the National Science Foundation. This provides funds for six undergraduates to do research in each of three summers, 1992, 1993, and 1994.

Professor Dershem has also been active in curriculum development and in the activities of the Special Interest Group on Computer Science Education (SIGCSE) of the ACM. He served s program chair of the 1988 SIGCSE Symposium and edited the proceedings of that symposium [6].

This project will utilize the SUN network of the Hope College Computer Science Department. This network is describe in detail in the Appendix. Presently we run Meridian Ada on that network. The version of Ada 9X that we use for the proposed projects will depend upon the availability of compilers at the time.

Bibliography

[1] Dershem, H.L., A course on computing and statistics for social science, *Proceedings of 1972 Conference on Computers in the Undergraduate Curricula*, Atlanta, GA, 1972.

[2] Dershem, H.L., *Computer Exercises for Elementary Statistics*, Compress, Inc., 1979.

[3] Dershem, H.L., A modular introductory computer science course, *SIGCSE Bulletin*, 13(1):177-181, Feb, 1981.

[4] Dershem, H.L., *UMAP Module 477: Computer Problem Solving*, Birkhauser Boston, Inc., 1981.

[5] Dershem, H.L., *UMAP Module 478: Iteration and Computer Problem Solving*, Birkhauser Boston, Inc., 1981.

[6] Dershem, H.L. (ed.), *Proceedings of the Nineteenth SIGCSE Technical Symposium*, Association for Computing Machinery, 1988.

[7] Dershem, H.L. and Jipping, M.J., *Programming Languages: Models and Structures*, Wadsworth Publishing Company, 1990.

[8] Dershem, H.L. and Jipping, M.J., *Programming Languages: Models and Structures, Second Edition*, PWS Kent Publishing Co., 1994. [to appear]

F. Cost Breakdown

Task 1: Write a new chapter for the Third Edition of the Dershem/Jipping textbook describing Ada 9X as it applies to the object-oriented paradigm.

Two months full-time effort by project director (July, 1994 and June 1995)
Offering of course using materials (August-December, 1995) - no project cost
One month half-time assistance by undergraduate student (June 1995)

Total Cost: $17,034

Task 2: Revise text in Dershem/Jipping text in relation to imperative and concurrent paradigm to reflect changes in Ada 9X

Drafted by project director during offering of course (August-December, 1995) - no project cost
One month full-time effort by project director (July, 1995)
One month half-time assistance by undergraduate student (July, 1995)

Total Cost: $9,090

Task 3: Redesign of Programming Languages and Object-Oriented Programming courses at Hope College to base them on Ada 9X.

Course design and offering by project director (August-December, 1995) - no project cost
One month full-time effort by project director to write up course and project results (May, 1996)

Total Cost: $8,340

Budget Summary

July, 1994	Project Director's Salary (1/9 academic year salary)	$ 6,620
	Project Director's Benefits (20% of salary)	$ 1,324
June-July, 1995	Project Director's Salary (2/9 academic year salary)	$13,900
	Project Director's Benefits (20% of salary)	$ 2,780
	Half-time student assistant stipend	$ 1,500
May, 1996	Project Director's Salary (1/9 academic year salary)	$ 6,950
	Project Director's Benefits (20% of salary)	$ 1,390
Total Budget		$34,464

CURRICULM VITAE
Herbert L. Dershem

Education:

B.S. University of Dayton, 1965
M.S. (Computer Science) Purdue University, 1967
Ph.D. (Computer Science) Purdue University, 1969

Experience:

Hope College, Assistant Professor, 1969-1974
Associate Professor, 1974-1981
Professor, 1981-present
Chair of Computer Science Department, 1976-present
Oak Ridge National Laboratories, Visiting Research Scientist, 1977-1978
Boston University Overseas Program, Visiting Professor, 1982-1983
United States Air Force Academy, Distinguished Visiting Professor, 1993-1994

Honors and Awards:

NDEA Fellow, Purdue University, 1965-1968
Honeywell Corporation Fellow, Purdue University, 1968-1969
Project COMPUTe Awardee, Dartmouth College, 1972
NASA/ASEE Summer Fellow, Goddard Space Flight Center, 1976
Oak Ridge Associated Universities Summer Fellow, 1977

Grants:

Co-director, "Introduction of the Computer in the Statistics Curriculum", NSF Office of Computing Activities, 1971-1973

Director, "A Modular Approach to the Introductory Course in Computer Science", NSF Local Course Improvement Program, 1978-1980

Co-Director, "A Microcomputer Laboratory for use in Teaching Statistics", NSF Instructional Scientific Equipment Program, 1979-1980

Director, "CSNET Membership in Support of Computer Science Research", NSF RUI Program, 1987-1990

Director, "Computer Science Undergraduate Research Program", NSF REU Program, 1992-1994

Director, "Use of Ada, Laboratories, and Visualization in the Teaching of Data Structures and Discrete Mathematics", DARPA Curriculum Development Grant, 1993-1994

Publications: (23 total, those pertinent to this project listed in Bibliography)

Other major sources of support: None

Related proposals pending: None

Appendix - Description of Hope College Computer Networks

Computer Science Department Sun Network

Machine/Part	Peripherals
Sun 4/360	32 MB memory, 2.0 GB disk, 14,400 baud modem
Sun 4/470	32 MB memory, 669 MB disk
(2) Sun 4/40s	12/16 MB memory, 500MB disk, 3.5" floppy
(6) Sun 4/60s	16 MB memory 500MB disk, 3.5" floppy, GX graphics coprocessor
(3) Sun 4/65s	16 MB memory, 500MB disk, 3.5" floppy
(2) Sun SPARCstation 10s	32 MB memory, 500MB disk, 3.5" floppy
(1) Sun SPARCstation 10	32 MB memory, 1.0 GB disk, 3.5" floppy
(32) INMOS Transputers	Parallel processing units housed in Sun 4/470

Lab software includes standard distributed SunOS/Unix software. This includes a distribution of Sun's OpenWindows, which is a version of the X windowing system. In addition, several packages have been purchased from various vendors including FrameMaker, SunGKS, SunPHIGS, SunLink DNI DECnet support software, Centerline CodeCenter and ObjectCenter, SunPC, and Adobe Transcript. INMOS languages and development software are available for the Transputers. The lab uses several public domain software packages including TEX, EMACS, and DECnet utilities.

The lab's software and hardware provide access to the Internet through a college-owned Merit routing equipment.

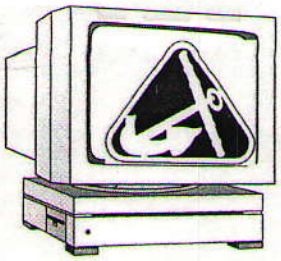
VAX Network

The college owns two VAX 4000 systems which serve the entire campus community for academic, administrative, and library applications. This system is accessible from eleven locations on campus which have a total of 144 stations that are publicly available for student access. In addition, there are many other terminals available in offices and laboratories across the campus.

A wide selection of software is available on the VAX systems including the Verdix VADS system of Ada software development.

PC Network

There is a Novell local area network in the building complex where this project will be conducted that connects 49 486 systems through a common file server. Twenty-eight of these systems are located in a computer classroom that includes one HP Laserjet III printer, and a projection system.



Hope College
Department of Computer Science
Holland, Michigan 49422-9000
(616) 395-7510

May 13, 1997

COPY

Mike Shumate
NCTAMS-LANT
9456 Fourth Avenue, Suite 200
Bldg V53, N9/N7
Naval Air Station
Norfolk, VA 23511-2199

Dear Mike,

Enclosed you will find the Final Report for Air Force Contract# F29601-94-K-0033, Curriculum and Textbook Development Using Ada 9X for the Teaching of Object-Oriented Concepts. A complete set of deliverables is also enclosed.

It has been a pleasure working with you and I greatly appreciate the funding provided under this contract. Please let me know if you need any further information.

Sincerely,

Herbert L. Dershem, Chair

Final Report
Air Force Contract # F29601-94-K-0033
Curriculum and Textbook Development Using Ada 9X for the Teaching of Object-Oriented Concepts

Herbert L. Dershem
Department of Computer Science
Hope College
Holland, MI 49422-9000

May 13, 1997

Summary of Activities

The table below indicates the activities that were carried out with support from this contract:

July, 1995	Reviewed the second edition of the textbook, *Programming Languages: Structures and Models* by Dershem and Jipping to update all Ada references to reflect the changes made in Ada 95. Conducted thorough review of features of Ada 95 with assistance from undergraduate students Andrew Van Pernis and Manuel Calderon. These students were supported by a National Science Foundation grant under the Research Experiences for Undergraduates program (CDA 9423943) This study included a thorough analysis of the object-oriented features of Ada 95. The final report of this study is included as a deliverable. Designed a course in Object-Oriented programming with a significant portion devoted to Ada 95.
August-December, 1995	Taught the course CSCI 495, Object-Oriented Programming at Hope College using Ada 95. The syllabus for this course is included as a deliverable.
June-July, 1996	Drafted a new Chapter for the proposed Third Edition of *Programming Languages: Structures and Models*. This chapter is included as a deliverable.
August-December, 1996	Taught the course CSCI 383, Programming Languages at Hope College using the newly drafted Object-Oriented Ada 95 chapter.
May, 1997	Prepare materials for final report

List of deliverables

1. *A Critical Evaluation of the Enhancements of Ada 95* by Andrew Van Pernis and Manuel Calderon

2. Syllabus for CSCI 495, Object-Oriented Programming

3. Proposed Chapter of *Programming Languages: Structures and Models, Third Edition*, The Object-Oriented Model in Ada 95.

4. Syllabus for CSCI 383, Programming Languages

5. Chapter outline for proposed Third Edition of *Programming Languages: Structures and Models*

A Critical Evaluation of the Enhancements of Ada 95

by
Andrew Pieter Van Pernis
Manuel Calderón

Hope College Computer Science
Summer Research 1995
Prof. Herb Dershem

Table of Contents

Many new features have been added to Ada 95 to enhance and expand the Ada programming language. These features affect four main categories of programming: tasking, generics, string manipulation, and object-oriented. All of the new features are improvements of Ada, but some are more useful then others. Furthermore, not all of the features are exactly what they appear to be upon a cursory examination. We will examine these new features of Ada 95, comment on their usefulness, and list any difficulties in using these new features. We will also examine how these new features apply to tasking, generics, string manipulation and object-oriented programming.

TASKING

Tasking is one of the more useful features of Ada 83. It permitted different processes to run concurrently. Ada 95 includes several new features that make tasking more useful. These new features are explained below followed by an example that uses most of them.

The features:

1. Protected Types
2. Protected Types - Entries
3. Requeue Statement
4. Task Scheduling
5. Asynchronous Transfer of Control
6. Predefined Library - I/O

Protected Types

In order to simplify tasking, Ada 95 has included the protected type feature. A protected type is a type that contains a private part, which is used to pass information between tasks or allow several tasks to share information, depending on the protected type's subprograms. Attached to the protected type are subprograms that access and manipulate the private data. Since these subprograms are executed in a mutually exclusive manner, the integrity of the data stored in the protected type is insured. Thus, protected types help to eliminate the need for additional tasks to control the passing of information between various tasks.

A protected type can be defined in two ways. First a protected variable can simply be created.

```
protected Flag is
        procedure Get (Value: Item);
        procedure Put (Value: Item);
private
        Data: Item;
end Flag;

protected  body Flag is

        -- Procedure and function bodies for the variable Flag

end Flag;
```

Second, an actual type can be defined as protected, and then variables of that type can be declared.

```
protected type Flag_Type is

...
-- As in the above example
...

Flag: Flag_Type;
```

The subprograms of the protected type are then accessed through the dot notation found throughout Ada.

```
Flag.Get (X);
Flag.Put (Y);
```

It is important to note that since functions in Ada have read-only access to the protected type, any number of tasks may execute functions on a protected type at the same time. Since procedures have the ability to change the values stored in the protected type, only one task may execute a procedure on a protected type at any given time, and no functions may access the protected type while that procedure executes.

Protected types are an incredibly useful feature when using tasks. Not only can protected types be used to create semaphores and flags, but they can also be used to prevent race conditions on data shared between tasks. Furthermore, since the actual value of the protected type remains private and can only be accessed through the defined subprograms, the abstraction of a semaphore or flag is maintained.

Protected Types - Entries

Entries are procedures or functions belonging to protected types that have a certain condition, called the barrier. When an entry is called, the condition is evaluated. If the condition is true, then the entry body executes. If it is false, the entry call is queued until its condition becomes true. When an entry is queued, the task that called it halts. Therefore entry calls to the same protected element must be made from different tasks to have any impact.

In the following example, the entry is called "win_or_lose". When this entry is called, the variable "points" must be equal to "limit" or "-limit" for the body to begin execution. At the beginning of the program, this will not be true. Task one calls the entry "win_or_lose" at the beginning. Therefore it will be queued. During the execution of other tasks, the "barrier" will become true and "win_or_lose" will eventually execute. When the person playing this game gets "limit" or "-limit" points, a win or a loss is reported.

The example:

```
-----------------------------------------
package for_game is
   ...
   protected type game is
      entry win_or_lose;
   ...
   end game;
end for_game;
-----------------------------------------
package body for_game is

   protected body game is
```

```
        ...
        entry win_or_lose when ((points = limit) or (points = -limit)) is
        -- will get executed if the barrier is broken
        -- it will display a message depending on points.
        begin
            if points = limit then
                put_line(" YOU WON !!!! ");
            else
                put_line(" YOU LOST ??? ");
            end if;
        end win_or_lose;
        ...
    end game;
end for_game;
```

Example of a call:

```
task one;
task body one is
begin
    my_game.win_or_lose;    -- this call will be queued, until the barrier
end one;                    -- becomes true.
```

Requeue Statement

When a requeue statement is executed, for example "requeue reset", entry "reset" is placed in the entry queue.

```
entry signal when true is      -- barrier is always true
begin
        ...
        requeue reset;         -- WHERE THE REQUEUE STATEMENT IS USED.
        ...
end signal;
```

In the preceding code "reset" will be placed in the entry queue. When entry "signal" finishes executing, then all executable entries waiting in the queue will be executed. Following these, "reset" will execute if its barrier condition is true. A more complete example of requeue is given later.

Task Scheduling

Ada 95 expands the capability of tasks by allowing task scheduling, giving certain tasks are given priority over other tasks. Thus, when tasks enter a queue for a processor or other resource, tasks with a higher priority are given preference over those with lower priorities.

Task scheduling is added to any tasking program with a single statement in the specification of each task.

```
task Example is
        --Entries for task example
        ...
        pragma Priority (Value);
end Example;
```

The pragma priority statement assigns the task a priority between 0 and 30, with 30 as the highest priority, and 0 the lowest. When tasks are competing for a resource, the task with the highest priority will gain control of the resource first. If several tasks with the same priority are waiting for the resource they will be handled using a FIFO queue.

Task scheduling, once understood, is easy to implement and allows for greater control over the order of execution of tasks. Furthermore, including task scheduling in Ada 95 does not invalidate any Ada 83 code, since tasks default to the original FIFO queue method of scheduling if they are not assigned priorities. A further extension of task scheduling is interrupt priorities, which can be used when tasks need to immediately gain control of resources, as in an abort situation.

Asynchronous Transfer of Control

The statement

```
select                    -- asynchronous transfer of control.
        delay 2.0;        ------------------------------------
        ...
then abort
        ...
end select;
```

seems to "not" be working. GNAT documentation says it is not yet implemented. It supposed to work so that the statements between the "then abort" and the "end select" are executed first. If they take longer to execute than the time specified after the "delay", execution is interrupted, and instead, the statements between the "delay" and the "then abort" are executed. When the statements take less time than specified in the delay, GNAT works fine. But, when execution is interrupted because time elapsed, a "segmentation fault" is reported.

Predefined Library - I/O

The Text_IO library of Ada 95 remains virtually unchanged from that of Ada 83. Several useful new features have been added to it however, such as Flush, Look_Ahead, and Get_Immediate. Furthermore, in order to support programs written under Ada 83, no previously existing functions were removed or changed in any significant fashion.

The procedure Flush immediately flushes a buffer to a file, or to the current output depending on whether a file was specified. The Flush procedure is also important when working with tasks, since most implementations will have the tasks output to a buffer that will then be sent to the current output file when the task is completed, and not before, making debugging difficult. Flush simplifies the debugging process by forcing the output of a task to be immediately sent to the output file instead of remaining in the buffer until the task finishes executing. Look_Ahead is a procedure that allows the user to determine the next character of a file and input without consuming it. The procedure Get_Immediate reads the next character from a file or the current input file if one is available, and does not skip any line or page terminators like the standard get procedure. Other additions include Modular_IO and Decimal_IO as subpackages, and the ability to use the functions Set_Error, Standard_Error, and Current_Error for error files. In general, the predefined library Text_IO remains unchanged between Ada 83 and Ada 95, only a few, helpful features have been added to it.

An Example of Tasking

The following example was taken from the Ada 95 rationale and was modified a little so that incorporates most of the Ada 95 features of tasking.

```
-----------------------------------
package for_event is
    protected type event is
        entry wait;
        entry signal;
    private
        entry reset;
        occurred: boolean:= false;
    end event;
end for_event;

-----------------------------------
package body for_event is
protected body event is

    -------------------------
    entry wait when occurred is
    begin
        put_line("wait is executed");
        flush;
    end wait;
    -------------------------
    entry signal when true is      -- barrier is always true
    begin
        put_line(" Signal is executed. ");
        flush;
        if wait'count > 0 then
            occurred := true;
            delay(1.0);            -- to make sure the waits are before requeue.
            requeue reset;         -- WERE THE REQUEUE STATEMENT IS USED.
        end if;
    end signal;
    -------------------------------
    entry reset when wait'count = 0 is
    begin
        put_line(" Reset is executed. ");
        flush;
        occurred:= false;
    end reset;

end event;
end for_event;

-----------------------------------
procedure main1 is
    my_event: event;
    ---------------------------------------------------------
    task type one;
    task body one is
        pragma Priority (30);
    begin
```

```
        put_line (" First wait");
        flush;
        my_event.wait;
        put_line (" Second wait ");
        flush;
        my_event.wait;
    end one;
    ------------------------------------------------------------
    task two;
    task body two is
        pragma Priority (29);
    begin
        delay(1.0);
        put_line(" First signal ");
        flush;
        my_event.signal;
        delay(1.0);
        put_line(" Second Signal");
        flush;
        my_event.signal;
    end two;
    ------------------------------------------------------------
    x: array(1..3) of one;
    ------------------------------------------------------------
begin
    null;
end main1;
-------------------------------------------------------------------------
```

Here is what happens when the preceding program executes:

First, the 3 "first waits" are queued (3, because there are 3 tasks of type "one"). Then the "first signal" gets called, this requeues "reset". Thus, "reset" is queued behind the 3 "waits". The 3 "waits" are now executed, and the "reset" follows. Then the same thing happens again with the "second waits" and "second signals".

The tasking features used in this example are:

Protected types:
Type "Event" is a protected type.

Protected types - Entries:
"Wait", "Signal", and "reset" are entries.

Requeue Statement:
"requeue reset" is in entry "signal".

Task Scheduling:
Task type "one" has a higher priority than task "two" therefore the tasks of the array will start execution to ensure that the 3 "first waits" are queued before the "signal" in task two is called. This enables the use of the requeue statement.

Predefined Library - I/O

"flush" is from the library I/O. Without "flush" we may be getting the output at an incorrect time and we won't really know what is happening. What happens without the "flush" is that the output goes into a buffer. This buffer is outputted to the screen only when the task finishes execution or when some other things happen. To be sure we get the output in the correct order, we use "flush". This ensures that the output is displayed on the screen when the "put" statement is reached.

GENERICS

With generics, we can save writing a lot of code. Generics allow us to write packages and procedures for an undetermined type (the generic parameter type) and instantiate the package/procedure for any type we need. For example, suppose we need a swap function for integers, one for reals, and one for characters. Instead of writing the same function 3 times, each time only changing the types in the parameters, we write a generic function, and instantiate it for integers, reals, and characters. We could also instantiate for strings and arrays, as you will see later. Ada 95 gives us new features that allow us to enhance Ada's generic capabilities, including generic packages with unconstrained types (strings, arrays, etc.). We will first explain each new feature, then we will give a complete example that implements a generic stack, using these generic features.

The features:

1. Generics - Unconstrained parameters.
2. Formal Generic Package Parameters.
3. Other Improvements:
4. Abstract Types and Subprograms.
5. Aliased Types.

Generics- Unconstrained

In Ada 83, generic packages did not accept unconstrained types as parameters. That is, we couldn't instantiate a package with the type "string" or some sort of array type. This limited us somewhat. Now, with this new feature, Ada 95 permits the use of unconstrained types as parameters to a generic.

This feature does, however, have a necessary limitation. We are not allowed to declare an uninitialized object of type T (the unconstrained parameter type) in the body of the generic. The following code will illustrate this limitation. Suppose we want to create a generic swap function for unconstrained type and let type "T" be the generic type:

```
procedure Swap (A, B: T) is
        Temp: T;                -- <---- This is not valid.
begin
        Temp := A;
        A := B;
        B := Temp;
end Swap;
```

But when we write "Temp: T" we are declaring an (uninitialized) object of a generic unconstrained type, which we are not allowed to do. There are ways around this. We could still create the swap function using pointers, but this is a complex process.

The way to make a generic-unconstrained package is:

```
generic
        type T(<>) is private;
package gen_stack is
        ...
end gen_stack;
```

Functions or procedures that use the unconstrained type are written as follows:

```
procedure push(a: in T) is
        ...
begin
        ...
end push;
```

To instantiate the package for the type "string" we write:

```
package string_gen_stack is new gen_stack(string);
```

At the end of this section I will give a complete example in which generics - unconstrained will be used and explained further.

Formal Generic Package Parameters

The concept of generic packages is expanded in Ada 95 with the addition of formal generic package parameters. Formal generic package parameters allow the user to define a generic package based on a previously compiled generic package, which is accepted as a formal parameter of the new generic package. Thus a generic package could be created for complex numbers that accepts one of the various float types for instantiation. Then another generic package that handles polar operations on complex numbers could be created and would accept any instantiation of the original complex numbers packages as a parameter upon its instantiation.

Incorporating formal generic package parameters into a program is fairly simple. If, as above, we were creating two generic packages, where one handled complex numbers and the other handled polar operations on complex numbers, these packages could be defined as

```
generic
        Float_Type is digits;
package Generic_Complex_Numbers is
        type Complex is private;

        function "+" (Left, Right: Complex) return Complex;
        --(Similarly for -,/,*, etc.)

end Generic_Complex_Numbers;
```

and

```
with Generic_Complex_Numbers;
generic
        with package Complex_Numbers is new Generic_Complex_Numbers (<>);
package Generic_Complex_Polar is
```

```
-- Types, functions, and procedures necessary for polar operations on complex number

end Generic_Complex_Polar;
```

Instantiation of these two packages is then accomplished by the following statements.

```
package Complex_Floats is new Generic_Complex_Numbers(Float);
package Polar_Complex_Floats is new Generic_Complex_Polar(Complex_Floats);
```

There are several problems with this feature of Ada 95. First, if an original generic package uses an unconstrained parameter such as

```
generic
        Float_Type is digits <>;
```

then a subtype or derived type of the unconstrained parameter must be declared if that type is needed within a second package that uses the original package as a formal generic package parameter. Second, given an instantiation of a generic package, no more than one new generic package may be instantiated within a single program using that original instantiated package as a formal generic package parameter. For example if there were three generic packages for complex numbers, GCNumbers, GCPolar, and GCVectors, where both GCPolar and GCVectors had GCNumbers as a formal generic package parameter, then the following instantiation of the three packages would be invalid.

```
package Complex_Floats is new GCNumbers(Float);
package Polar_Complex_Floats is new GCPolar(Complex_Floats);
package Vector_Complex_Floats is new GCVectors(Complex_Floats);
```

This severely restricts the usefulness of formal generic package parameters. As far as could be determined from the existing documentation for Ada 95, this is a restriction imposed by Ada 95 and not specific to any implementation.

Despite these limitations, formal generic package parameters are still quite useful for creating generic systems within Ada 95. This feature is just unable to reach its full potential with the present system, but it still does greatly expand the usability and usefulness of generic packages.

Abstract Data Types and Subprograms

An Abstract Data Type is a type that serves as parent to other types. It can have data elements, that all its children inherit, and abstract procedures and functions that, although they don't do anything, require all children to have these subprograms as well.

For example:

Suppose you have 3 different types of tires. The 3 tires have common features, but no tire has features that are a superset of the features of another tire. Thus, no tire inherits qualities from another. There is no reason that one of these tires should be the parent of any of the others. In this case, an abstract base_tire_type could contain the common qualities of all 3 tires, and then have each tire type inherit from base_tire_type.

An advantage is that this lets the programmer imitate the real world in a better way.

An example:

Consider a phone directory where information for "Basic" and for "Complete" can be stored. By "Basic," I mean relatives and all acquaintances. By "Complete" I mean close friends. Since I am mostly interested in "Complete," I want more information on them. Thus, the entries

for "Complete" will allow me to enter more information.

The base_type is "Basic_Entry," which is in package called "Base_Directory_Entry". This abstract type has a data element called "Name". This requires all directory entries to have a name, (regardless of "Basic" or "Complete"). It also has an abstract procedure called "Display", requiring all directory entries to have a way of displaying themselves.

"Normal_Entry" inherits from "Basic_Entry" and adds a data element called "Telephone". It also includes the procedure "Display". Remember, "Basic_Entry" is the abstract type and has the abstract subprogram "Display". This means that all types inheriting from it, will have to have their own "Display", such is the case here.

"Complete_Entry" inherits from "Normal_Entry and adds the data elements Address, Comment1, and Comment2. It also overwrites "Normal_Entry"'s "Display".

The code for these types follows:

```
-------------------------------------------------------
package base_directory_entry is

    type basic_entry is abstract tagged
        record
            name: string(1..100);
        end record;
        ------------------------------------------------
    procedure display(be: in out basic_entry) is abstract;
    ------------------------------------------------
end base_directory_entry;
-------------------------------------------------------------------------------


-------------------------------------------------------
package normal_directory_entry is
    package int_io is new integer_io(integer);
    use int_io;

    type normal_entry is new base_directory_entry.basic_entry with
        record
            telephone : string(1..100);
        end record;
        ------------------------------------------------
    procedure display(ne: in out normal_entry);
    ------------------------------------------------

    type complete_entry is new normal_entry with
        record
            address  : string(1..100);
            comment1 : string(1..100);
            comment2 : string(1..100);
        end record;
        ------------------------------------------------
    procedure display(ce: in out complete_entry);
-------------------------------------------------------
 end normal_directory_entry;
-------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------
package body normal_directory_entry is
   ---------------------------------------------------
   procedure display(ne: in out normal_entry) is
   begin
      put(" Name      : "); put_line(ne.name);
      put(" Telephone : "); put_line(ne.telephone);
   end display;
   ---------------------------------------------------
   procedure display(ce: in out complete_entry) is
   begin
      display(normal_entry(ce));
      put(" Address   : "); put_line(ce.address);
      Put(" Comment1  : "); put_line(ce.comment1);
      PUT(" Comment2  : "); put_line(ce.comment2);
   end display;
   ---------------------------------------------------
end normal_directory_entry;
-------------------------------------------------------
```

Aliased Types

Pointers have been greatly enhanced in Ada 95 through the use of aliased types. Previously it was not possible to create pointers to objects declared as variables or constants. The addition of aliased types to Ada 95 allows a pointer to point to an object, instead of needing to be dynamically created as before. Aliased types are used in the following manner

```
type Float_Ptr is access all Float;
type Const_Ptr is access constant Float;

Pointer1: Float_Ptr;
Pointer2: Const_Ptr;
A: aliased Float;
G: aliased constant Float := 9.8;
...

Pointer1 := A'Access;
Pointer2 := G'Access;
```

In the definition of an access type, the all qualifier indicates pointers which will point to aliased variables and thus have read and write access, whereas the constant qualifier indicates those that will only have read access to the variable to which they point.

Aliased types now allow for easy manipulation of pointers, without the worries of dangling references. Furthermore, aliased types allow pointers to point to constant values, restricting such pointers to read only, even when they point to a variable. Furthermore, constant pointers and variable pointers cannot be interchanged, but constant pointers need not point to a constant. For example

```
Pointer2 := G'Access        --a pointer to a constant
Pointer2 := A'Access        --a pointer to a variable
```

since Pointer2 is of Const_Ptr type, Pointer2 would have read access only to G or A with the above assignment statements, even though A is a variable. Conversely, a variable pointer could

not be assigned to point to a constant.

Although they are not very useful for creating large linked-lists or other pointer structures, aliased types can be used for creating small linked structures. The problem with creating large linked structures with aliased types is that each node needs to be declared as a variable. While this eliminates the problem of dangling references, it is not very efficient, nor is it as dynamic as a true linked structure. Of course, a linked structure using aliased types would be easy to rearrange and manipulate as necessary, since the pointers can be directly assigned to point to the various nodes, without the need of a temporary pointer to hold on to a disconnected node, which would become a dangling reference in a normal linked structure. Furthermore, aliased types can be used to point into the middle of composite types. Thus, it is not necessary to create a pointer to a record, to access some field of that record, but rather the pointer can be assigned to point directly to the desired field, as long as that field is marked as aliased.

An Example of Generics

The following example implements a stack of anything. That is, the statement

```
package int_stack is new stack(integer);
```

instantiates a stack of integers. The same can be accomplished with strings or any other type, constrained or unconstrained.

The same problem occurs here as in the "swap" function. To have a linked list we need to declare a record in which one of the elements is of type T (the generic unconstrained type). As you know by now, we cannot do that. The way around this is to use pointers or as they are called in Ada "access variables". That is, in the record, instead of having an element of type T, we have a pointer to it (an access variable to type T). Therefore, the record contains one element that is a pointer to T, and another that is a pointer to the next element (since the stack is implemented as a linked list).

It turns out that the "generic unconstrained" feature can be used to create many different kinds of data structures, such as stacks, binary search trees, hash tables and so on. This is very useful, because, for example, AVL trees are not trivial to rewrite each time we need them for a different type. This way, we just instantiate our generic AVL tree for the type we need and that is it.

The code for the generic stack is as follows:

```
-------------------------------------------------------------------------------
-- This implements a stack of unconstrained type T.
-- It has procedure push and function pop.
-- Push: takes an element of type T.
-- pop: returns an element of type T.
-------------------------------------------------------------------------------

with ada.text_io, ada.strings.unbounded, ada.integer_text_io, ada.float_text_io;
use  ada.text_io, ada.strings.unbounded, ada.integer_text_io, ada.float_text_io;

-------------------------------------------------------------------------------
generic
    type T(<>) is  private;
package gen_stack is
    type T_access is private;
    type list_node;
```

```
    type linked_list is  access list_node;
    type list_node is
    record
        pdata : T_access;          -- this will be pointing to the data T
        next  : linked_list;
    end record;
    i : integer;
    ll: linked_list;                          -- were the stack will be.
    ------------------------------------------------------------------------
    procedure push(a: in T);
    function  pop return T;
    ------------------------------------------------------------------------
private
    type T_access is  access  all T;        -- pointer to the generic type T
    ------------------------------------------------------------------------
end gen_stack;

-------------------------------------------------------------
package  body gen_stack is
    stack_error: exception;
-----------------------------------------
    procedure push(a: in T) is
        temp : linked_list;
    begin
        temp := new list_node'(new T'(a), ll);
        ll := temp;
    end push;
-----------------------------------------
    function pop return T is
    -- if it is null, then it will return null
        temp: linked_list;
    begin
        if  ll = null then
            raise stack_error; -- raise an exception
        else
            temp := ll;
            ll:= ll.next;
            return temp.pdata.all;
        end  if;
    end pop;
-----------------------------------------
end gen_stack;
-------------------------------------------------------------
```

The following shows how to instantiate a stack and use it.

```
-------------------------------------------------------------------------
procedure main is
        package string_gen_stack is  new gen_stack(string);
        use string_gen_stack;
        push(" Hello ");
        push(" Manuel ");
```

```
        put(pop);
        put(pop);
end main;
```

The program would output:

```
Manuel Hello
```

Strings

Previously, Ada did not support strings to a large extent. Although the string type was defined, it only allowed strings of fixed length, and the only operations defined for the string type were concatenation and comparison. Thus, a major expansion to Ada 95 is the predefined libraries for strings. There are several new packages covering fixed length, bounded, and unbounded strings, as well as character mapping between strings. Fixed length strings are strings which have a set length and must be padded if the desired string is not long enough to fill the string. Bounded length strings have a maximum length, but do not necessarily have to fill that length. Unbounded strings are strings which can be of any length.

Many new functions are defined within each of these packages for the different types of strings. Some of the most important functions are those which allow the user to convert between the three string types. The majority of the functions are for manipulating strings, such as concatenation, returning the head or tail of the string or counting the number of occurrences of a substring within a string. As well as functions for manipulating the string types, there are comparison functions defined for bounded and unbounded strings. All of these new predefined library packages add significantly to Ada 95, by making it simpler and more efficient for the user to work with strings.

Instead of detailing all of the functions and procedures defined in the new predefined library packages, only those functions or procedures used in the example program will be examined after the actual program. For more information refer to appendix A.4 of the Ada 95 Reference Manual.

An Example Program for Strings

The program in this example excepts message strings of up to 256 characters in length and encodes or decodes. The code used by the program to translate messages is entered by the user and is stored as a character mapping function.

```
with Text_IO; use Text_IO;
with Ada.Characters.Handling; use Ada.Characters.Handling;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Strings.Bounded; use Ada.Strings.Bounded;
with Ada.Strings.Fixed; use Ada.Strings.Fixed;
with Ada.Strings.Maps; use Ada.Strings.Maps;

procedure CodeMaker is

    package CodeStrings is new Generic_Bounded_Length(26); use CodeStrings;

    protected type Messages is
        function Read return Unbounded_String;
        procedure Add (NewMessage: Unbounded_String);
        procedure Clear;
```

```
private
    TheMessage: Unbounded_String := Null_Unbounded_String;
end Messages;
protected body Messages is
    function  Read return Unbounded_String is
    begin
        return TheMessage;
    end Read;

    procedure Add (NewMessage: Unbounded_String) is
    begin
        TheMessage := TheMessage & NewMessage;
    end Add;

    procedure Clear is
    begin
        TheMessage := Null_Unbounded_String;
    end Clear;
end Messages;

WillSelfDestruct,CodedMessage,DecodedMessage: Messages;

task Encoder is
    entry Incoming(Message: Unbounded_String);
    entry CodeDefn(Original,Code: Bounded_String);
    entry Quit;
    pragma Priority(5);
end Encoder;
task body Encoder is
    type MapPtr is access all Character_Mapping;

    CodePtr: MapPtr;
    Map: aliased Character_Mapping;
    Finished: Boolean := False;
begin
    loop
        select
            accept Incoming(Message: Unbounded_String) do
                put_line("<<<Encoding>>>");
                declare
                    Copy: String:= To_String(Message);
                begin
                    Copy:= To_Lower(Copy);
                    Translate(Copy,CodePtr.all);
                    CodedMessage.Add(To_Unbounded_String(Copy));
                    WillSelfDestruct.Clear;
                    DecodedMessage.Clear;
                end;
                put_line("<<<Complete>>>");
            end Incoming;
        or
            accept CodeDefn(Original,Code: Bounded_String) do
```

```
                Map := To_Mapping(To_String(Original),To_String(Code));
                CodePtr := Map'Access;
            end CodeDefn;
        or
            accept Quit do
                Finished := True;
            end Quit;
        else
            delay(1.0);
        end select;
        exit when Finished;
    end loop;
end Encoder;

task Decoder is
    entry Incoming(Message: Unbounded_String);
    entry CodeDefn(Original,Code: Bounded_String);
    entry Quit;
    pragma Priority(3);
end Decoder;
task body Decoder is
    type MapPtr is access all Character_Mapping;

    CodePtr: MapPtr;
    Map: aliased Character_Mapping;
    Finished: Boolean := False;
begin
    loop
        select
            accept Incoming(Message: Unbounded_String) do
                put_line("<<<Decoding>>>");
                declare
                    Copy: String:= To_String(Message);
                begin
                    Copy:= To_Lower(Copy);
                    Translate(Copy,CodePtr.all);
                    DecodedMessage.Add(To_Unbounded_String(Copy));
                    WillSelfDestruct.Clear;
                    CodedMessage.Clear;
                end;
                put_line("<<<Complete>>>");
            end Incoming;
        or
            accept CodeDefn(Original,Code: Bounded_String) do
                Map := To_Mapping(To_String(Code),To_String(Original));
                CodePtr := Map'Access;
            end CodeDefn;
        or
            accept Quit do
                Finished := True;
            end Quit;
        else
```

```
            delay(1.0);
        end select;
        exit when Finished;
    end loop;
end Decoder;

task CreateCode is
    entry NewCode;
    entry Quit;
    pragma Priority(7);
end CreateCode;
task body CreateCode is
    Finished: Boolean := False;
begin
    loop
        select
            accept NewCode do
                declare
                    Original: Bounded_String := Null_Bounded_String;
                    Code: Bounded_String := Null_Bounded_String;
                    OriginChar,CodeChar: Character;
                begin
                    put_line("Enter a '.' to exit");
                    loop
                        put("Enter the character of the original text: ");
                        get(OriginChar);
                        skip_line;
                        if OriginChar = '.' then
                            exit;
                        else
                            if (CodeStrings.Count(Original,To_Set(OriginChar))>0) then
                                put_line("That character has already been entered.");
                            elsif not(Is_Letter(OriginChar)) then
                                put_line("That character is not valid.");
                            else
                                OriginChar := To_Lower(OriginChar);
                                Append(Original,OriginChar);
                                loop
                                    put("Enter the character of the coded text: ");
                                    get(CodeChar);
                                    skip_line;
                                    if (CodeStrings.Count(Code,To_Set(CodeChar))>0) then
                                        put_line("That character has already been entered.");
                                    else
                                        Append(Code,CodeChar);
                                        exit;
                                    end if;
                                end loop;
                            end if;
                        end if;
                        exit when Length(Original) = 26;
                    end loop;
```

```
                    Encoder.CodeDefn(Original,Code);
                    Decoder.CodeDefn(Original,Code);
                end;
            end NewCode;
        or
            accept Quit do
                Finished := True;
            end Quit;
        else
            delay(1.0);
        end select;
        exit when Finished;
    end loop;
end CreateCode;

task Controller is
    pragma Priority(0);
end Controller;
task body Controller is
    Choice: Character;
    Finished: Boolean := False;
    Message: String(1..256);
    Length: Natural;
begin
    delay(5.0);
    loop
        Flush;
        new_line;
        put_line("Would you like to:");
        put_line("    1). Enter a Message");
        put_line("    2). Encode a Message");
        put_line("    3). Decode a Message");
        put_line("    4). Change the Code");
        put_line("    5). View the current encoded message.");
        put_line("    6). View the current decoded message.");
        put_line("    7). Clear all messages.");
        put_line("    8). Quit");
        new_line;
        put("Enter your choice (1-8): ");
        get(Choice);
        skip_line;
        new_line;
        case Choice is
            when '1' => put_line("Enter the message below: ");
                        get_line(Message,Length);
                        WillSelfDestruct.Add
(To_Unbounded_String(Message(1..Length)));
            when '2' => Encoder.Incoming(DecodedMessage.Read & WillSelfDestruct.Read);
                        put_line("The coded message is: ");
                        put_line(To_String(CodedMessage.Read));
            when '3' => Decoder.Incoming(CodedMessage.Read & WillSelfDestruct.Read);
                        put_line("The decoded message is: ");
```

```
                                        put_line(To_String(DecodedMessage.Read));
                    when '4' => CreateCode.NewCode;
                    when '5' => put_line("The current encoded message is: ");
                                        put_line(To_String(CodedMessage.Read));
                    when '6' => put_line("The current decoded message is: ");
                                        put_line(To_String(DecodedMessage.Read));
                    when '7' => WillSelfDestruct.Clear;
                                        CodedMessage.Clear;
                                        DecodedMessage.Clear;
                    when '8' => CreateCode.Quit;
                                        Encoder.Quit;
                                        Decoder.Quit;
                                        Finished := True;
                    when others => put_line("That choice is not valid.");
                end case;
                new_line;
                exit when Finished;
            end loop;
        end Controller;

    begin
        CreateCode.NewCode;
        delay(1.0);
    end CodeMaker;
```

The following functions and procedures were used in the example program and are from the library packages Ada.Characters.Handling, Ada.Strings.Unbounded, Ada.Strings.Bounded, Ada.Strings.Fixed, or Ada.Strings.Handling. It is important to note that many of the subprograms exist as both functions and procedures.

First, bounded length strings are implemented by using a generic package, so bounded length strings were instantiated to be of length 26 in order to hold the entire alphabet.

The '&' operator takes two strings of the same type (bounded, unbounded, or fixed length) and concatenates them into one string.

A character mapping function can be defined through the use of the function To_Mapping which accepts two fixed length strings and returns a function which will map a character in the first string to a corresponding character in the second.

The function To_String will convert either an unbounded or a bounded string to a fixed length string with a length matching that of the original bounded or unbounded string. The functions To_Unbounded_String and To_Bounded_String will convert fixed length strings into unbounded and bounded length strings respectively.

To_Lower is a function that takes a string and converts all of its characters to lower case.

The procedure Translate takes a string and translates it using a character mapping function provided as one of the parameters to the procedure.

The function Count counts the number of instances of a specific substring or character within a string.

The Boolean function Is_Letter determines whether or not a character is one of the letters of the alphabet.

Append is a procedure that adds a character or substring to the end of a string.

The function Length returns the length of a string.

It is important to note that get and put procedures are only defined for fixed length strings. Thus, in the example program, the size of the message is limited to 256 characters because it must be read as a fixed length string, but it is converted to an unbounded string for use within the

program. The problem of only being able to read and write fixed length strings can be solved in a number of ways. The simplest method to output either a bounded or unbounded string is to use the To_String function like so

```
put_line(To_String(The_Bounded_or_Unbounded_String);
```

Reading strings of indeterminate length as input is a little more difficult. The string could be read character by character and each character could be added onto the end of the string as it was read, which would work well for unbounded strings, but would require a large amount of error checking for unbounded strings. Or, string input could be broken up into smaller fixed length strings which are then concatenated together to form the entire string. The easiest solution, of course, is to limit the length of string being entered to a set length, which would by definition make it a fixed length string.

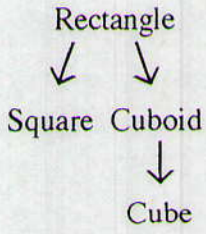
Object-Oriented Programming

The are three main components to object-oriented programming, encapsulation, inheritance, and polymorphism. Encapsulation was provided in Ada 83 through the use of packages. Ada 83 also provided inheritance with derived types. The addition of polymorphism has made Ada 95 a true object-oriented language. The new features in Ada 95 which are related to polymorphism are

1. Tagged Types
2. Class Wide Programming
3. Dynamic Type Selection
4. Public Children
5. Private Children
6. Generic Children

Tagged Types

Polymorphism is added to Ada 95 through the addition of tagged types. The inheritance of tagged types allows for the creation of derived types simply and efficiently. This inheritance is extended to all types derived from the original type. This can be seen in a simple example which has the following hierarchy



In the above example Cube inherits from Cuboid which inherits from Rectangle even though Cuboid is not defined as tagged.

Tagged types allow for object oriented classes to be defined easily. Any functions or procedures which immediately follow a tagged type definition will be the operations associated with the class defined by the tagged type and will be the operations inherited by children of the tagged type. Thus, the hierarchy above could be created as follows

```
type Rectangle is tagged with
        record
                Length: Float;
                Width: Float;
        end record;
function Size(R: in Rectangle) return Float is
begin
        return R.Length * R.Width;
end Size;

type Square is new Rectangle with null record;

type Cuboid is new Rectangle with
        record
                Height: Float;
        end record;
function Size(C: in Cuboid) return Float is
begin
        return C.Length * C.Width * C.Height;
end Size;

type Cube is new Cuboid with null record;
```

The first class, Rectangle, is the base class from which all of the other classes inherit. It consists of a record with two fields, Length and Width. There is also one function defined for the class, Size, which computes the area of the rectangle. The next class, Square, is an exact duplicate of Rectangle, since the record is not expanded, no new functions or procedures are defined, and the inherited function Size is not redefined. In a more complex structure, where a perimeter function was defined for Rectangle as well, the difference between Square and Rectangle could be more readily seen, since Square's perimeter function would be different from Rectangle's. Cuboid is also derived from Rectangle, but it expands the record by adding the field Height. Furthermore, with the addition of Height, the function Size must be redefined for Cuboid. Finally, similar to Square, Cube is an exact duplicate of its parent Cuboid.

Another important feature of tagged types is that new types derived from an original tagged type can be placed in separate packages and compiled separately. This allows for extension without disturbing the existing code further promoting encapsulation, since each class is contained within a separately compiled program unit.

Class Wide Programming

A useful extension of tagged types is class wide programming. For each tagged type T there is a type T'Class, to which any type derived from T can be converted. This is very useful because it allows T and all types derived from T to be passed as parameters to procedures or functions. Furthermore, pointers can be created to point to a class wide type declaring them to be

```
type Class_Ptr is access T'Class;
```

Then if a primitive operation is called using either the pointer or the parameter the program will implicitly choose the operation appropriate to the type of the class wide object. This choice of appropriate operation to match the class wide type is called dispatching.

Thus, as in the example hierarchy from tagged types, a pointer to a class wide type could be assigned to point to Rectangle first, then to Square, then Cuboid, and finally to Cube. After the pointer is assigned, it could be passed as a parameter to a procedure which calls the primitive

operation Size on the pointer, and since the pointer is a pointer to a class wide type the program automatically selects the appropriate Size function for each of the different types.

```
type Class_Ptr is access Rectangle'Class;
...
Shape_Ptr: Class_Ptr;
A: Rectangle;
B: Square;
C: Cuboid;
D: Cube;
...
procedure Shape_Size (Shape: in out Class_Ptr) is
        ...
begin
        ...
        Size(Shape.all);
        ...
end Shape_Size;
...
Shape_Ptr.all := A;
Shape_Size(A);
Shape_Ptr.all := B;
Shape_Size(B);
-- etc.
```

This can also be accomplished without the use of pointers. A procedure can be defined so that its parameter is a class wide type. Then the procedure can be called on each shape and again the primitive operations are called as appropriate.

```
procedure Shape_Size (Shape: in out Rectangle'Class) is
        ...
begin
        ...
        Size(Shape);
        ...
end Shape_Size;
...
Shape_Size(A);
Shape_Size(B);
-- etc.
```

It is important to note that when using pointers to class wide types, the pointer must be initialized to point to the value of the class wide type when storage is allocated to it. Thus, in the example program, in order for a pointer to point to Rectangle, an object of type Rectangle must first be defined, and then the pointer must be initialized to that object.

Dynamic Type Selection

Ada 95 is further enhanced through the addition of dynamic type selection. Basically, dynamic type selection allows the definition of pointers to subprograms. Furthermore, these pointers can be used to pass the subprograms as parameters to other subprograms, which greatly expands the capabilities of Ada 95.

Pointers to subprogram units can be defined in two ways. A pointer to a function would

be defined as follows

```
type Function_Ptr is access function (A: Type1; B: Type2; ...) return Value_Type;
```

Thus, Function_Ptr is an access type to any function which has a first parameter of Type1, second parameter of Type2, etc. and returns a value of Value_Type. A pointer to a procedure is much simpler.

```
type Procedure_Ptr is access procedure (A: Type1; B: Type2; ...);
```

Procedure_Ptr is an access type to any procedure which has parameters of Type1, Type2, etc.

Dynamic type selection has a large variety of possible uses. It can be used to construct a queue of procedures to be executed on a certain variable or variables. Or it can allow a procedure to call other functions or procedures based on selection routines within the first procedure. Also, it could be used to create a function which calls another function, passed as a parameter, repeatedly. Yet, it is the simple elegance of its use that is the most outstanding aspect of this feature.

Public Children

Ada 95 allows packages to be structured into hierarchical libraries, in which child packages can be created from existing packages. This allows for packages to be expanded without the need for recompiling or disturbing working code. A public child is a child package which is visible outside of the hierarchy in which it exists. Although its visible declarations do not have access to the private part of its parent, a public child's private part and body can both access the private part of its parent.

Child packages are easily defined in Ada 95. If for example a package for floating point complex numbers existed, and was called Complex_Numbers, then a child package which would handle vector operations on floating point complex numbers could be created as follows

```
package Complex_Numbers.Vectors is

   --Specifications for vector operations on complex numbers

end Complex_Numbers.Vectors;
package body Complex_Numbers.Vectors is

   --Function and procedure bodies for operations in specification

end Complex_Numbers.Vectors;
```

Then, to use Complex_Numbers.Vectors in a program or package a simple 'with' statement is required.

```
with Complex_Numbers.Vectors; use Complex_Numbers.Vectors;
```

It is important to note, that if elements from the package Complex_Numbers were also needed, then a 'use' statement would be required for Complex_Numbers as well, but a 'with' statement would not be necessary, since a 'with' statement is implicitly declared for a parent, when a 'with' statement is given for one of its children.

One of the main problems of the hierarchical libraries is that in order to make use of all the packages within the structure, a use statement for each package must be included, which could become cumbersome when using a library with a large number of children, grandchildren, etc. or calls to child package procedures, functions, etc. must be of the form:

<child package name>.<name of type|function|procedure|etc.>

Which in a large library would become unreadable.

Of course there are also many advantages to the hierarchical library structure. The most important advantage is the fact that existing code is not disturbed when packages are expanded through the use of a child package, because the parent package does not need to be recompiled, which makes it very useful for creating classes for object-oriented programs. Additionally, parent packages can depend on their children, through the use of with clauses, and children can depend on their siblings, again through the use of with clauses, as well as depend on completely separate packages as is normal.

Private Children

An additional expansion of the hierarchical library system of Ada 95 is the concept of private children. A private child is a child unit, in the library structure, that is completely private to its parent unit. Furthermore, a private child is only visible within the subtree rooted at its parent unit. And, since it is not visible outside of its parent's subtree, a private child can access the private part of its parent.

A private child is defined exactly like a public child, except the word private is placed at the beginning of the specification. Thus if the child package Complex_Numbers.Vectors was to be changed to be a private child, the code would be changed to look like this

```
private package Complex_Numbers.Vectors is
   ...
   -- Exactly as before
   ...
```

Private children are a useful feature conceptually, but without a large hierarchical library structure to support their use, the usefulness of private children is greatly decreased, complicating what could otherwise be a simple program.

Using private children presents some problems similar to those found in using public children. First, using private children further complicates the hierarchical library structure which must be maintained. Second, just as in the use of public children, private children require a use statement, and maybe a with statement, for each package from the structure which is to be used, or a cumbersome prefix system, if the use statement is not included.

The benefit of using private children is that packages can be expanded without the need for recompilation, and at the same time remain private outside of the tree rooted at the packages parent. But this alone is not sufficient justification for the use of private children. There must also be a complex enough library structure to warrant their use.

Generic Children

A generic child is just the same as the other kind of children, but generic, which means that it can work for different types of variables, thus making the children a lot more flexible. For example, if you want a generic child of "integer," you instantiate the package for "integer" and you are set.

Things I noticed:

The compiler didn't let me instantiate a child generic package inside the main procedure in the main program. Thus they have to be instantiated in a separate file. Since the child package needs to refer to the generic parent, the parent must also be instantiated in a separate file. In the example package "real_complex_numbers" (instantiation of "complex_numbers") and package

"real_complex_numbers_polar" (instantiation. of "complex_numbers.polar") are in separate files.

Notice that the rationale says to instantiate package "complex_numbers.polar" the following way

package float_complex_numbers_polar **is new** complex_numbers.polar

This doesn't work. Instead, the package must be instantiated by

package float_complex_numbers_polar **is new** float_complex_numbers.polar.

Generic functions, procedures, and packages are very useful. Therefore, generic children are also very useful. It is also good that the generic children have access to the private elements of the parent.

An example of a generic child:

```
-------------------------------------------------------------------------
-- This is a generic package
-------------------------------------------------------------------------
generic
    type float_type is digits<>;
package Complex_Numbers is
    type Complex is private;

    function "+" (Left, right: Complex) return Complex;
    -- similarly "-", "*", and "/"
    ...

end Complex_Numbers;

-------------------------------------------------------------------------
package body complex_numbers is

    ...
    -- Function and procedure bodies for the package complex_numbers
    ...

end complex_numbers;
-------------------------------------------------------------------------

-------------------------------------------------------------------------
-- This is a generic child package:
-------------------------------------------------------------------------
generic
package Complex_numbers.polar is

    function Polar_to_complex(R, Theta: float_type) return Complex;
    function  "abs" (right: Complex) return float_type;
    function Arg(X: Complex) return float_type;

end complex_numbers.polar;
-------------------------------------------------------------------------
```

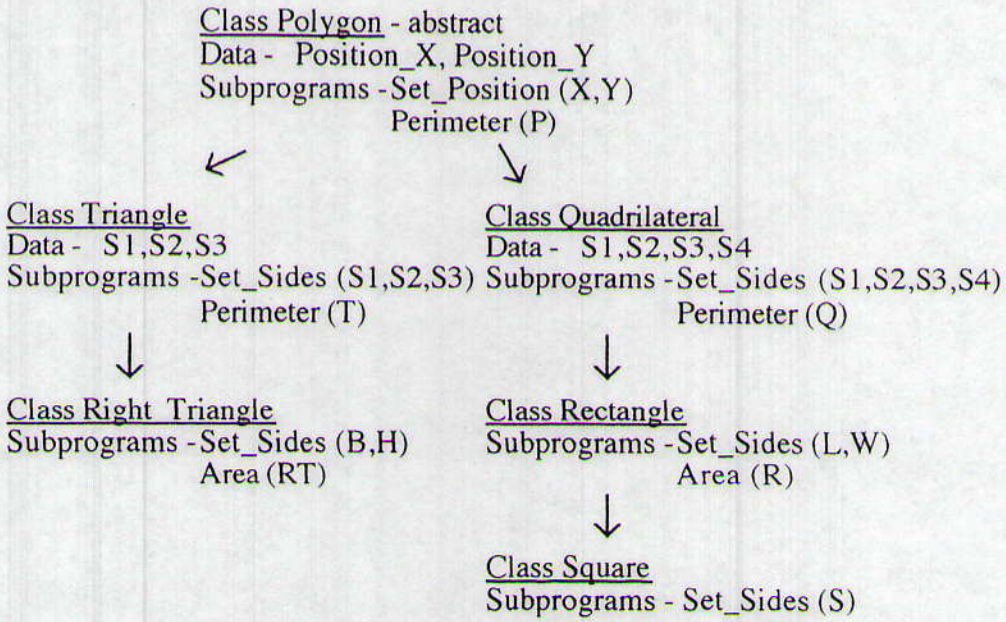
```
package body complex_numbers.polar is

   ...
   -- Function and procedure bodies for the package complex_numbers.polar
   ...

end complex_numbers.polar;
-------------------------------------------------------------------------
-- This is the package that corresponds to the instantiation of
-- "complex_numbers" with floats.
-------------------------------------------------------------------------
with complex_numbers;
   package float_complex_numbers is  new complex_numbers(float);
-------------------------------------------------------------------------
-- This is the package that correspond to the instantiation of
-- complex_numbers.polar, or actually float_complex_numbers.polar (because
-- that is the package that corresponds to complex_numbers instantiated
-- for floats).
-------------------------------------------------------------------------
with complex_numbers.polar, float_complex_numbers;
   package float_complex_numbers_polar is  new float_complex_numbers.polar;
-------------------------------------------------------------------------
```

An Example of Object-Oriented Programming in Ada 95

The following example reviews the main features of object-oriented programming in Ada 95. The example is an expansion of the example given in the section on tagged types. In this example an abstract base class called Polygon is created. The classes derived from polygon have the following hierarchy



Furthermore, in order to show true encapsulation, each of the classes will be placed in a separate package and these packages will form a hierarchical library system of parent and child packages, with the package for class Polygon as its root.

```
--<>----------<>--
-- Class Polygon
--<>----------<>--
package Poly_Pack is
    type Polygon is abstract tagged with
        record
            Position_X: Float;
            Position_Y: Float;
        end record;

    procedure Set_Position (X,Y: in Float; Shape: out Polygon);
    function Perimeter (P: Polygon) return Float is abstract;
end Poly_Pack;
package body Poly_Pack is
    procedure Set_Position (X,Y: in Float; Shape: out Polygon) is
    begin
        Shape.Position_X := X;
        Shape.Position_Y := Y;
    end Set_Position;
end Poly_Pack;
--<>----------<>--
-- Class Triangle
--<>----------<>--
package Poly_Pack.Tri_Pack is
    type Triangle is new Polygon with
        record
            S1, S2, S3: Float;
        end record;

    procedure Set_Sides (A,B,C: in Float; Shape: out Triangle);
    function Perimeter (T: Triangle) return Float;
end Poly_Pack.Tri_Pack;
package body Poly_Pack.Tri_Pack is
    procedure Set_Sides (A,B,C: in Float; Shape: out Triangle) is
    begin
        Shape.S1 := A;
        Shape.S2 := B;
        Shape.S3 := C;
    end Set_Sides;

    function Perimeter (T: Triangle) return Float is
    begin
        return T.S1 + T.S2 + T.S3;
    end Perimeter;
end Poly_Pack.Tri_Pack;
--<>----------<>--
-- Class Right Triangle
--<>----------<>--
with Ada.Numerics.Elementary_Functions; use Ada.Numerics.Elementary_Functions;
package Tri_Pack.RightTri_Pack is
    type Right_Triangle is new Triangle with null record;
```

```
    procedure Set_Sides (B,H: in Float; Shape: out Right_Triangle);
    function Area (RT: Right_Triangle) return Float;
end Tri_Pack.RightTri_Pack;
package body Tri_Pack.RightTri_Pack is
    procedure Set_Sides (B,H: in Float; Shape: out Right_Triangle) is
    begin
        Shape.S1 := B;
        Shape.S2 := H;
        Shape.S3 := Sqrt(B*B + H*H);
    end Set_Sides;

    function Area (RT: Right_Triangle) return Float is
    begin
        return 0.5 * B * H;
    end Area;
end Tri_Pack.RightTri_Pack;
--<>-------------------------<>--
-- Class Quadrilateral
--<>-------------------------<>--
package Poly_Pack.Quad_Pack is
    type Quadrilateral is new Polygon with
        record
            S1, S2, S3, S4: Float;
        end record;

    procedure Set_Sides (A,B,C,D: in Float; Shape: out Quadrilateral);
    function perimeter (Q: Quadrilateral) return Float;
end Poly_Pack.Quad_Pack;
package body Poly_Pack.Quad_Pack is
    procedure Set_Sides (A,B,C,D: in  Float; Shape: out Quadrilateral) is
    begin
        Shape.S1 := A;
        Shape.S2 := B;
        Shape.S3 := C;
        Shape.S4 := D;
    end Set_Sides;

    function Perimeter (Q: Quadrilateral) return Float is
    begin
        return Q.S1 + Q.S2 + Q.S3 + Q.S4;
    end Perimeter;
end Poly_Pack.Quad_Pack;
--<>-------------------------<>--
-- Class Rectangle
--<>-------------------------<>--
package Quad_Pack.Rect_Pack is
    type Rectangle is new Quadrilateral with  null  record;

    procedure Set_Sides (L,W: in Float; Shape: out Rectangle);
    function Area (R: Rectangle) return Float is;
end Quad_Pack.Rect_Pack;
```

```
package body Quad_Pack.Rect_Pack is
    procedure Set_Sides (L,W: in Float; Shape: out Rectangle) is
    begin
        Shape.S1 := L;
        Shape.S3 := Shape.S1;
        Shape.S2 := W;
        Shape.S4 := Shape.S2;
    end Set_Sides;

    function Area (R: Rectangle) return Float is
    begin
        return R.S1 * R.S2;
    end Area;
end Quad_Pack.Rect_Pack;
--<>------------------------<>--
-- Class Square
--<>------------------------<>--
package Rect_Pack.Sqr_Pack is
    type Square is new Rectangle with null record;

    procedure Set_Sides (S: in Float; Shape: out Square);
end Rect_Pack.Sqr_Pack;
package body Rect_Pack.Sqr_Pack is
    procedure Set_Sides (S: in Float; Shape: out Square) is
    begin
        Shape.S1 := S;
        Shape.S2 := Shape.S1;
        Shape.S3 := Shape.S1;
        Shape.S4 := Shape.S1;
    end Set_Sides;
end Rect_Pack.Sqr_Pack;
--<>------------------------<>--
-- A program using the above class hierarchy
--<>------------------------<>--
with Rect_Pack.Sqr_Pack, Tri_Pack.RightTri_Pack, Text_IO;
use Poly_Pack, Tri_Pack, RightTri_Pack, Quad_Pack, Rect_Pack, Sqr_Pack, Text_IO;
procedure Example is
    package fio is new Float_IO(Float); use fio;

    procedure PrintPerimeter (Shape: in Polygon'Class) is
    begin
        new_line;
        put("The perimeter of this shape is ");
        put(Perimeter(Shape),1,3,0);
        new_line(2);
    end PrintPerimeter;

    A: Triangle;
    B: Quadrilateral;
    C: Right_Triangle;
    D: Rectangle;
    E: Square;
```

```
begin
    Set_Sides(5.0,7.0,3.0,A);
    Set_Position(0.0,1.5,A);
    Set_Sides(6.8,E);
    Set_Position(-2.0,4.5,E);
    PrintPerimeter(A);
    PrintPerimeter(E);
    put("The area of the square is ");
    put(Area(E),1,3,0);
end Example;
```

This example program would have the following as its output

The perimeter of the shape is 15.000

The perimeter of the shape is 27.200

The area of the square is 46.240

In summary, many enhancements have been added to Ada 95. These additions have brought Ada back to the forefront of programming languages. Ada 95 has been expanded and enhanced to cover many modern programming issues, such as object-oriented programming, and yet still retains the security and readability of Ada 83. Once again Ada has proven itself to be one of the major programming languages of yesterday, today, and tomorrow.

Computer Science 495
Object-Oriented Programming
Fall 1995

Meeting Time and Place: TR 8:00 - 9:20
VZN B24

Professor: Herbert L. Dershem
Office: VWF 220
Phone: 7508
Mailbox: Username "DERSHEM"

Prerequisites: CSCI 286 and permission of the instructor

Objectives: For the student to
1. Understand the principles of the object-oriented model
2. Design classes using an object-oriented language
3. Be able to critically evaluate the object-oriented model

Approach: This course will engage the students with the fundamentals of the object-oriented paradigm. It will be an analytical and evaluative approach. Three languages, C++, Smalltalk, and Ada 95, will be used to implement this paradigm. Students will design extensive class libraries for use in the Hope College Computer Science curriculum. They will also be required to do outside readings and make presentations about aspects of object-oriented computing.

Textbook: An Introduction to Object-Oriented Programming by Timothy Budd

Exams: There will be a take-home midterm exam.

Programming Exercises: There will be a programming exercise assigned in each of the three languages, C++, Ada 95, and Smalltalk.

Project: Teams will complete class libraries for the major class project which will be due at the end of the semester. These will be implemented using C++.

Reports: Each student will prepare and deliver a report on a topic related to object-oriented computing. The report will require the reading of at least one paper on the topic. These reports are to be at least one-half hour in length and will be presented during the last three weeks of the semester.

Grading: The grading criteria will be as follows:

Midterm Exam	20%
Programming Exercises	20%
Class Project	40%
Presentation	20%

CSCI 495 - Object-Oriented Programming
Program Assignment 2

Implement a set class in Ada 95. This class represent a set of Ada strings. This class should implement the following methods:

- Make the set empty
- Test the set for empty
- Insert a string into the set
- Remove a string from the set
- Print the set, one string per line
- Test a string for inclusion in the set

You are to construct three files for this assignment. Files `set.ads` and `set.adb` will contain the specification and body of the package representing the class. In addition, you are to write a file `testset.abd` which will be a main program to test your implementation of sets. Your main program must do the following:

- Read a sequence of strings from the file `testinsert.dat` and place the strings into a set which is initially empty. If a duplicate string is read, a message should be printed alerting the user that the string is a duplicate and the program should not attempt to insert it.
- Next a sequence of strings is read from the file `testremove.dat`. Each of these strings is removed from the set as it is read. If a string is read which is not in the set, a message to that effect must be printed and the program will not attempt to remove the string.
- After both files are completely read, the final set will be printed.

The main program should declare a single object of class set and process that set by calling appropriate procedures and functions.

Attach your three files to an email message and send to "`dershem`" before 23:59 on October 5, 1995.

Chapter X
The Object-Oriented Model in Ada 95

The Ada 95 revision of the Ada programming language provides complete and extensive object-oriented capabilities. In this chapter, these capabilities are described and illustrated.

X.1 Overview

One of the major objectives of the Ada 95 revision was to include a full implementation of the object-oriented model. Ada 83 included some features of the model such as encapsulation through packages, type extension of operations through the use of derived types, and static polymorphism through overloading and generics. Those features commonly associated with the object-oriented model that were not found in Ada 83 were type extension through the addition of data components, dynamic polymorphism throughout the subclass structure, and visibility control that permits separately encapsulated classes to share non-public data and operations. All of these features have been included in Ada 95.

Another objective of the Ada 95 revision was to maintain the fundamental approach and structure of the Ada language. This objective has had a major impact on the implementation of the object-oriented paradigm in Ada 95. Class inheritance is implemented as a natural extension of derived types while encapsulation and visibility issues are handled under the package structure of Ada 83.

The remaining sections of this chapter describe the complete implementation of the object-oriented paradigm in Ada 95. In doing this, we do not distinguish the newly added features from those that were present prior to Ada 95.

X.2 Classes and Methods

In Ada, there is no distinction between a type and a class. This unifies the two concepts and makes the object-oriented model a natural extension of the Ada implementation of the Ada model.

If a class is a type, then an object is any instantiation of that type. Therefore, objects can be created in two fundamental ways: by declaring a variable and by dynamically allocating an object of the given type by means of a new statement. The object created by variable declaration will be referred to by the name of the variable. The dynamically-allocated object is referred to by means of a pointer.

The data components of a class are defined as record components. Since the class is therefore a record type, each object of that class contains every data element of the class. Consider, for example, the class `Fraction` defined by

```
type Fraction is record
    numerator : integer;
    denominator : positive;
end record;
```

This is a standard record type definition in Ada, but it also serves to define a class. Objects of type Fraction can now be instantiated in the following ways:

```
f : Fraction;
fp : access Fraction;
...
fp = new Fraction;
```

In this case, both `f` and `fp.all` are objects of class `Fraction`.

Methods of an Ada class are procedures or functions that are declared in the same compilation unit as the type declaration for the class and have parameters and/or a return value belonging to the class. Unlike other languages such as C++ and Smalltalk where methods belong to an object syntactically, in Ada they belong to a class. There is no object specified as the receiver of the message calling the method, and therefore, there is no special syntax used to differentiate a receiving object.

Commonly, the compilation unit containing the definition of a class is a package. The standard visibility control mechanisms of the package therefore apply, permitting the complete specification of a class structure to be hidden through use of the `private` specification.

For example, the specification of a package defining a few useful functions for the `Fraction` class and prohibiting direct access to the data of the class is shown in Figure X.1. The procedure `Print_Fraction` and the function `asFloat` are methods of class `Fraction` since they both have `Fraction` parameters and they are declared in the same package as `Fraction`. The function `Make_Fraction` is a method of `Fraction` because it has a `Fraction` return value. The overloaded operator `*` is also a method of `Fraction`.

One interesting possibility of the Ada specification of methods is that one function or procedure might be a method for more than one class. This will only be possible if more than one class is defined in the same package as a procedure or function which includes both classes among its parameters and return value. Standard object-oriented convention keeps the number of classes declared in each package at one, so this situation will not usually arise.

It should be noted that the encapsulation unit (in our case `Fraction_pack`) and the class (`Fraction`) are separate entities and have distinct names. A method call in Ada is no different from any other function or procedure call. For example, three of the four methods of class `Fraction` would be called in the following statement

```
Print_Fraction(f1 + Make_Fraction(6,17));
```

Figure X.1 Definition of class Fraction

```
package Fraction_pack is
  type Fraction is private;
  function Make_Fraction(num, den : in integer) return Fraction;
  procedure Print_Fraction(F : in Fraction);
  function asFloat(f : in Fraction) return float;
  function "*"(f1, f2 : in Fraction) return Fraction;
  Fraction_Error : exception;
private
  type Fraction is record
    numerator : integer;
    denominator : integer;
  end record;
end Fraction_pack;

package body Fraction_pack is
  package Int_IO is new Integer_IO(integer); use Int_IO;

  function Make_Fraction(num, den : in integer) return Fraction is
    f : Fraction;
  begin
    if den > 0 then
      f.numerator := num;
      f.denominator := den;
    elsif den < 0 then
      f.numerator := -num;
      f.denominator := -den;
    else
      raise Fraction_Error;
    end if;
    return f;
  end Make_Fraction;

  procedure Print_Fraction(f : in Fraction) is
  begin
    put(f.numerator,5);
    put("/");
    put(f.denominator,5);
  end Print_Fraction;

  function asFloat(f : in Fraction) return float is
  begin
    return float(f.numerator)/float(f.denominator);
  end asFloat;

  function "*"(f1 : in Fraction; f2 : in Fraction) return Fraction is
    result : Fraction := (f1.numerator*f2.numerator,
                          f1.denominator*f2.denominator);
  begin
    return result;
  end "*";

end Fraction_pack;
```

X.3 Inheritance

X.3.1 Derived Types and Type Extension

A limited form of inheritance is available through the use of derived types. The class of the derived type may then inherit the methods of its parent class, override them, or add new methods not present in the parent type. Figure X.2 contains an example that illustrates all of these.

In this example, the class Human is derived from class `Mammal`. The Human class actually includes four methods. The procedure blood is inherited from Mammal and does not appear in the definition of Human. Procedures `speak` and `give_name` override the procedures of the same names in `Mammal`. Function `get_legs` is a method that is not inherited at all from `Mammal` but is defined for the first time in class Human.

The overridden methods from `Mammal` are still inherited by Human, but they can only be called by type-casting a Human into a `Mammal`. Consider the following:

```
h : Human;
 ...
speak(h);   -- this calls speak from Human
speak(Mammal(h)); -- this calls speak from Mammal
```

You will notice that the `give_name` procedure for class Human actually calls the `Mammal` `give_name` by type-casting its Human parameter to class `Mammal`.

Visibility is also an important consideration here. If `Mammal` had been a `private` type, then Human would not have had access to the component names. This issue can be addressed through the use of child units as we will see in Section X.4.

Although the use of derived types for inheritance permits extension of the methods, this form of inheritance is limited by having no capability for extending the data of the class by adding to its record structure. In order to make this extension of the data of a class possible, Ada introduces the tagged type.

Figure X.2

```
package Mammal_Pack is
  type Mammal is record
    name : String(1..20);
    legs : Natural := 0;
  end record;

  procedure blood(m : in Mammal);
  procedure speak(m : in Mammal);
  procedure give_name(m : in out Mammal; mname : String);
end Mammal_Pack

package Human_Pack is

  type Human is new Mammal;

  procedure speak(h : in Human);
  function get_legs(h : in Human) return Natural;
  procedure give_name(h : in out Human; mname : string);

end Human_Pack;

package body Human_Pack is
  procedure speak(h : in Human) is
  begin
    put_line(h.name & "says 'hi'");
  end speak;

  function get_legs(h : in Human) return Natural is
  begin
    return h.legs;
  end get_legs;
  procedure give_name(h : in out Human; mname : String) is
  begin
    give_name(Mammal(h),mname);
    h.legs := 2;
  end give_name;
end Human_Pack;
```

X.3.2 Tagged Types

The major limitation of derived types with respect to the object-oriented inheritance model is the apparent inability to extend the data of a class. For example, suppose we wish to include in the Human class that was derived from Mammal, an additional data field called IQ of type Natural. This is not possible through the use of the derived type feature.

Ada includes the capability of extending the data of a derived type by adding new data elements beyond those present in the corresponding base type. This is done by declaring the base type to be tagged. For example, to accomplish our goal of adding a data component IQ to Human, we could declare the two classes as follows:

```
type Mammal is tagged record
  name : String(1..10);
  legs : Natural := 0;
end record;

type Human is new Mammal with record
  IQ : Natural;
end record;
```

These declarations indicate that type Human inherits both of the data components of Mammal (name and legs) and adds one new component, IQ.

Type conversion can be applied from the derived type to its tagged parent since it simply requires that the surplus components be dropped. Conversion from tagged parent to derived type requires that the additional components be specified, however. Consider the following example illustrating conversion in both directions:

```
m1 : Mammal("mammalname",2);
h1 : Human("Humanname",2,120);
m2 : Mammal;
h2 : Human;
...
m2 := Mammal(h1);
h2 := (m1 with IQ=>30);
```

The conversion from h1 to m2 is standard type conversion with m2 taking the Mammal subset of data elements from h1. The conversion from m1 to h2 requires the definition of all components that are in the subclass but not in the base class which is the source of the conversion.

The derivation of subclasses through the use of tagged types can be carried out to multiple levels. For example, suppose we have a new class, Student, that is derived as a subclass from Human. The class Student will add some data components such as school and year_in_school. The class Student is then declared by:

```
type Student is new Human with record
  school : String(1..20);
  year_in_school : Positive;
end record;
```

The data belonging to each of the three classes is illustrated by Figure X.3. Each new derived type

may, of course, provide additional methods as well as additional data, through the normal derived type facility.

Special consideration is given to tagged types when they are `private` types as well. As usual, `private tagged` types have their components hidden from external units. It is required, however, that all types derived from a `private tagged` type be private as well. In the above `Mammal/Human/Student` example, Mammal could have been declared as a `private tagged` type as follows:

```
  type Mammal is tagged private;
...
private
  type Mammal is record
    name : String(1..20);
    legs : Natural;
  end record;
```

Then Human must be declared as a private extension of `Mammal` by

```
  type Human is new Mammal with private;
  ...
private
  type Human is new Mammal with record
    IQ : Natural;
  end record
  ...
```

In this case, all components of `Mammal` are inherited by Human, so within the package where Human is defined, references to any of the components of `Mammal` are valid. Therefore, inside the package containing the definition of Human, we can write

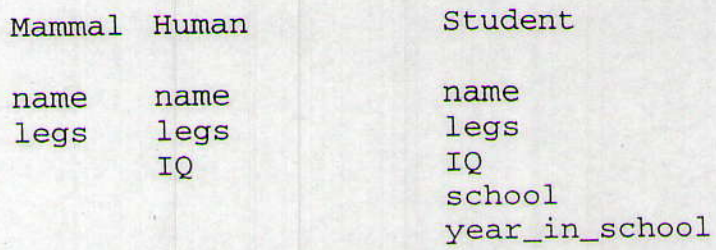
```
h : Human;
...
h.legs := 1;
```

and the reference to data component `legs`, defined in `Mammal`, is legal. A reference to h as a `Mammal`, however, would not permit direct reference to its data components, since the `Mammal` components are private to the package in which `Mammal` is defined and not accessible in the package where Human is defined. Of course, this would not be the case if both Human and `Mammal` were defined in the same package, but this is not considered good object-oriented practice. Therefore, if the two classes are defined in separate packages,

```
Mammal(h).legs := 2;
```

is illegal inside the package where Human is defined.

Figure X.3



X.3.3 Visibility and Child Units

The standard practice in Ada is to declare one class per package. One difficulty with this approach is that in order to maintain the encapsulation associated with the object-oriented model, the class should be declared as a private type. This means that the data of one class is not visible to the methods of any other class.

As an example of this restriction, consider the classes declared in the preceding section that include `Mammal`, `Human`, and `Student` and assume that each is declared in a different package. For example,

```
package Mammal_pack is
  type Mammal is tagged private;
  ...
end Mammal_pack;

with Mammal_pack; use Mammal_pack;
package Human_pack is
  type Human is new Mammal with private;
  ...
end Human_pack

with Mammal_pack, Human_pack; use Mammal_pack, Human_pack;
package Student_pack is
  type Student is new Human with private;
  ...
end Student_pack;
```

Now, since the data components of `Mammal` are defined in the `private` section of `Mammal_pack`, no method of `Human` can directly access them. What we need is the ability for the package of the subclass to access the private section of the package where its parent class is defined.

This capability is provided in Ada by means of child units. When a package is a child unit of another package, it makes the parent's private declarations visible within the private section of the child package, which includes the body of the child package.

One package is declared to be a child of another through its name. If the parent package has name `P`, for example, naming a package `P.C` would make it a child package of `P`. This naming convention can be carried out to multiple levels. For examples, a package name `P.C.G` would be a child package of `P.C`.

Child packages are very useful for providing the appropriate visibility for subclasses. In the example above, class Human could be declared in a package named Mammal_pack.Human_pack and class Student in Mammal_pack.Human_pack.Student_pack. Then all member functions of class Human could access name and legs, private components of Mammal that are inherited by Human. Likewise, Student could access all private elements of both Mammal and Human.

Although type extension and child packages are unrelated concepts, the first having to do with inheritance and the second with visibility, they are often bound together as in our Mammal/Human/Student example. By always defining a subclass in a package which is a child of the package defining its parent type, we ensure that all private elements of the parent class are visible to the child class.

X.3.4 Abstract Types

Some classes exist only to provide a base from which other classes can inherit. Such classes never have any objects instantiated directly, but rather exist to have derived classes that will have instantiations.

In Ada, this property can be enforced by making the class an abstract type. As an example, the class Mammal, which was defined earlier, could have been declared as an abstract type. This can be done by the following declaration:

```
type Mammal is abstract tagged private;
```

Class Mammal exists only so other classes can inherit data elements name and legs, and procedures blood and give_name. No objects should ever be instantiated of type Mammal. Also, note that the procedure speak is not intended to be inherited, but should be redefined for each class derived from Mammal. Therefore, it is not necessary to actually define speak in class Mammal. This can be indicated by declaring the procedure speak to itself be abstract:

```
procedure speak(m : in Mammal) is abstract;
```

The astute reader will notice that it appears that speak would not need to be defined at all in Mammal since it is never inherited. We will see later, however, how abstract procedures can be useful for dynamic binding.

Our new definition of Mammal as an abstract class is now found in Figure X.4.

Figure X.4 Abstract Version of Mammal_Pack

```
package Mammal_Pack is
  type Mammal is abstract tagged private;

  procedure blood(m : in Mammal);
  procedure speak(m : in Mammal) is abstract;
  procedure give_name(m : in out Mammal; mname : String);
private
  type Mammal is record
    name : String(1..20);
    legs : Natural := 0;
  end record;
end Mammal_Pack
```

X.4 Controlled Types

Ada provides the ability to automatically execute code when objects of a class are created or destroyed, similar in function to the constructor and destructor of C++. In Ada, this is handled through special procedures, `Initialize` and `Finalize`, that are defined in a built-in abstract class called `Controlled`. Hence, any class for which `Initialize` and/or `Finalize` are defined must be a derived class of `Controlled`., insuring that `Initialize` will be called at object creation and `Finalize` on object destruction.

A third procedure, `Adjust`, is provided in class `Controlled` to facilitate the sequence of creation and destruction that needs to occur during assignment.

For any class that is derived from `Controlled`, the procedure `Initialize` is called whenever a new object is instantiated. The `Initialize` procedure of a `Controlled` type is called in the following situations.

1. When a variable of the type is declared by a declaration which does not include an initial value.

2. When a variable is declared to be of a composite type which includes the `Controlled` class in its definition. This inclusion may be either direct or indirect. Again, if the `Controlled` object or any component containing it is initialized in a declaration, `Initialize` will not be called.

3. When an allocator is called by a new statement for an object of the `Controlled` class or a composite object containing such an object. Again, if an initial value is specified for the object in the allocation statement, `Initialize` will not be called.

If a `Controlled` class contains objects that are themselves `Controlled`, the `Initialize` procedure for the contained objects are called before the containing objects' `Initializes` are called. This calling order is illustrated by the three `Controlled` classes defined in Figure X.5. When an object of class `Parent` is declared, the `Initialize` for `Child1` is called first since it is a component of `Parent`. The `Initialize` for `Child2` will not be called until the third statement in `Initialize` for `Parent` where the allocator for `Child2` is executed.

The `Finalize` procedure of a `Controlled` class is executed at the time an object of that class ceases to exist. For declared variables, this is when the execution block containing the declaration is exited. For dynamically allocated objects, it is at the time of deallocation. A `Finalization` procedure is usually provided to free storage that has been allocated to an object that would not be deallocated by the normal deallocation of the object, such as storage referenced by the object's pointers.

A further example of `Initialize` and `Finalize` is found in Figure X.6. A `List` class is defined in this Figure. Note that the list includes both a head and tail pointer. `Initialize` sets both pointer to `null` while `Finalize` frees up all of the nodes in the list before the `List` itself is deallocated.

Figure X.5

```
with Ada.Finalization; use Ada.Finalization;

package Test_Init is
  type Child1 is new Controlled with private;
  type Child2 is new Controlled with private;
  type Parent is new Controlled with private;
private
  type Child1ptr is access all Child1;
  type Child2ptr is access Child2;
  type Child1 is new Controlled with record
    Data : Integer;
    Sibling : Child2ptr;
  end record;
  type Child2 is new Controlled with record
    Data : Integer;
    Sibling : Child1ptr;
  end record;
  type Parent is new Controlled with record
    First : aliased Child1;
    Second : Child2ptr;
  end record;

  procedure Initialize(C1 : in out Child1);
  procedure Initialize(C2 : in out Child2);
  procedure Initialize(P  : in out Parent);
end Test_Init;

package body Test_Init is
  procedure Initialize(C1 : in out Child1) is
  begin
    Put_Line("Child1 Initialize");
    C1.Data := 1;
    C1.Sibling := null;
  end Initialize;

  procedure Initialize(C2 : in out Child2) is
  begin
    Put_Line("Child2 Initialize");
    C2.Data := 2;
    C2.Sibling := null;
  end Initialize;

  procedure Initialize(P  : in out Parent) is
  begin
    Put_Line("Parent Initialize");
    P.First.Sibling := P.Second;
    P.Second := new Child2;
    P.Second.Sibling := P.First'Access;
  end Initialize;
end Test_Init;
```

Figure X.6

```
with Ada.Finalization; use Ada.Finalization;

package List_Pack is
  type List is new Controlled with private;
  procedure Add_to_Front(L : in out List; I : in Integer);
  procedure Add_to_Rear(L : in out List; I : in Integer);
  function "+"(First, Second : in List) return List;
  function "="(First, Second : in List) return Boolean;
  procedure Put(L : in List);
private
  type Node;
  type Node_ptr is access Node;
  type Node is record
    Data : Integer;
    Next : Node_ptr;
  end record;
  type List is new Controlled with record
    Head : Node_ptr;
    Tail : Node_ptr;
  end record;
  procedure Initialize(L : in out List);
  procedure Adjust(L : in out List);
  procedure Finalize(L : in out List);
end List_Pack

package body List_Pack is
  ...
  procedure Initialize(L : in out List) is
  begin
    Put_Line("Initialize called");
    L.Head := null;
    L.Tail := null;
  end Initialize;

  procedure Adjust(L : in out List) is
    Temp : Node_ptr := L.Head;
  begin
    Put_Line("Adjust Called");
    Put(L);
    L.Head := null;
    L.Tail := null;
    while Temp /= null loop
      Add_to_Rear(L,Temp.Data);
      Temp := Temp.Next;
    end loop;
  end Adjust;

  procedure Finalize(L : in out List) is
    procedure Free is new
                Ada.Unchecked_Deallocation(Node,Node_ptr);
    Temp : Node_ptr := L.Head;
    Save : Node_ptr;
  begin
    Put_Line("Finalize called");
    Put(L);
    while Temp /= null loop
      Save := Temp;
```

```
        Temp := Temp.Next;
        Free(Save);
      end loop;
      L.Head := null;
      L.Tail := null;
    end Finalize;
end List_Pack;
```

X.4.2 Overriding Assignment

An assignment operator ":=" is defined in Figure X.6. When the assignment

```
L1 := L2;
```

is executed, the components of `L2`, `Head` and `Tail`, are copied to the corresponding components of `L1`. This means that `L1` points to the same list as `L2` rather than a copy of the list. Usually, this is not the copy that we want since a later change to `L2` would also result in `L1` being modified.

Ada provides the `Adjust` procedure in the `Controlled` abstract class for just this purpose. `Adjust` is automatically executed during each assignment for a `Controlled` class. The assignment operation also results in a `Finalize` being called. The sequence of operations for the statement

```
L1 := L2;
```

is as follows:

1. Finalize(`L1`) is called.
2. Components of `L2` are copied to `L1` as in the default assignment.
3. `Adjust(L1)` is called

As a result of this sequence, the `Adjust` procedure is written under the assumption that the components themselves have already been copied. Any further adjustments must be made by the `Adjust` procedure.

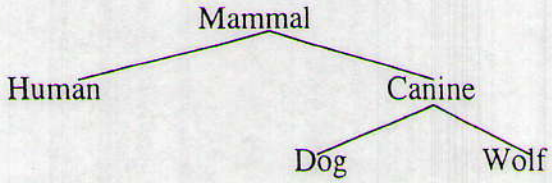
The `Adjust` procedure shown in Figure X.6 implements assignment for the `List` class of the preceding section. Since the two components, `Head` and `Tail`, have already been copied prior to the call on `Adjust`, the procedure assumes that these point to the `List` that is on the right-hand side of the assignment. Procedure `Adjust` creates a copy of this list, and sets `Head` and `Tail` to point to the appropriate places in this copy.

X.5 Polymorphism and Dynamic Binding

Polymorphism is achieved in Ada through the classwide types. These types represent all types in a subclass hierarchy and permit reference to all types in the hierarchy. Classwide types also permit the dynamic dispatching of a subprogram call.

X.5.1 Classwide Types

Every tagged type has a corresponding classwide type. The classwide type of type `T` is written `T'class`. The type `T'class` permits the representation of any type in the subclass hierarchy of `T`. For example, suppose tagged type T has the following subclass hierarchy:



Then all five of the types have a corresponding classwide type. The following table indicates the range of representation of each classwide type.

Classwide Type	Type Represented
Mammal'class	Mammal, Human, Canine, Dog, Wolf
Human'class	Human
Canine'class	Canine, Dog, Wolf
Dog'class	Dog
Wolf'class	Wolf

Classwide types can only be used in certain restricted contexts. Variables may be declared to be of a classwide type only if an initial value is provided in the declaration. For example,

```
x : Mammal'class
```

would be illegal whereas

```
d : Dog;
x : Mammal'class := d;
```

is legal. This is the case because an object of classwide type must always belong to one of the types it can represent. Once an object is bound to that type at its creation, it may not change. Therefore, though x above is of type `Mammal'class`, its tag indicates it is a `Dog` and that cannot be modified once the binding takes place.

Variables which are of access type to a classwide type can point to different objects and therefore point to objects of different types at different times. Consider the following code fragment:

```
type mamptr is access Mammal'class;
mp : mamptr;
...
mp := new Human;
...
mp := new Dog;
```

will result in `mp` pointing to a `Human` object first and then a `Dog` object.

X.5.2 Classwide Subprograms

A classwide subprogram is any procedure or function that has formal parameters that are of a classwide type. The actual parameters for such a subprogram may be of any of the types represented by the classwide type or belong to the classwide type itself.

Consider the function `same_species` defined by

```
function same_species(m1,m2: Mammal'class) return Boolean is
begin
   return m1'tag = m2'tag;
```

```
end same_species;
```

This function is a classwide subprogram because its parameters are of classwide type. It also introduces the `tag` attribute of an object of a tagged type. Two objects whose tags are the same belong to the same type within the classwide type since their tags identify their type.

The function `same_species` can be called with actual parameters that belong to any of the types represented by `Mammal'class`. For example, if variables are declared by

```
type mamptr is access Mammal'class;
d : dog;
x := Mammal'class := d;
h : Human;
m : mamptr;
```

Then all of the following calls are legal:

```
... same_species(d,h) ...
... same_species(x,h) ...
... same_species(m.all,x) ...
```

The first two calls will return false while the result of the third call depends upon the type of the object to which `m` points at the time of the call.

X.5.3. Dynamic Dispatching

While classwide subprograms provide some degree of polymorphism, another approach is dynamic dispatching. This occurs when a single call to a subprogram may be dispatched to multiple subprograms, depending on the tags of the actual parameters. Classwide subprograms have formal parameters of classwide types whereas dynamic dispatching utilizes actual parameters of classwide types.

An example of dynamic dispatching appears in the following fragment:

```
m : mamptr;
...
speak(m.all);
...
```

Since there are `speak` procedures for both `Dogs` and `Humans`, the call to `speak` could actually result in either of the two being called, depending on the tag of `m.all` at the time `speak` is called.

If there are multiple parameters of classwide type, the dispatching can be based on the combination of tags of the parameters. The return value of a function can also drive dispatching. For example, suppose the following two functions are defined:

```
function create_mammal(name:string) return dog is
   m : mamptr := new Dog;
begin
   give_name(m.all, name);
   return Dog(m.all);
end create_mammal;

function create_mammal(name:string) return Human is
```

```
   m : mamptr := new Human;
begin
   give_name(m.all, name);
   return Human(m.all);
end create_mammal;
```

Then the call

```
m.all := create_mammal("Joe");
```

would dispatch its call to the appropriate `create_mammal` function, depending on the tag of `m.all`.

CSCI 383 **Program Assignment 8** **Due November 27, 1996**

An Objective Look at Ada

You are to implement the five classes described below:

Class Date
Instance Variables: Month, Day, and Year
Instance Methods:

Set_Date (3 natural parameters plus the receiver)
Get_Month
Get_Day
Get_Year
Put

Class Transaction (abstract)
Instance Variables: Date of transaction (Date)
Account Number (String(1..10))
Amount of Transaction (Money type = delta 0.01 digist 8)
Instance Methods:

Set_Transaction (3 parameters plus the receiver)
Get_Date
Get_Acct_Num
Get_Amt
Put

Class Check
Superclass: Transaction
Instance Variables:

Check number (Natural)
Payee (Unbounded_String)

Instance Methods:

Set_Check (5 parameters plus the receiver)
Get_Check_Num
Get_Payee
Put

Class Deposit
Superclass: Transaction
Instance Variables:

Branch_Code (String(1..5))

Instance Methods:

Set_Deposit (4 parameters plus the receiver)
Get_Branch_Code
Put

Class Account

Instance Variables:	Balance (Money) Name (Unbounded_String) Transaction_List (linked list of all transactions)
Instance Methods	Initialize (start balance at 0 and list as empty) Finalize Add_Transaction Put (print all transactions in list ordered by date) Get_Balance Get_Name Set_Name Clear_Transactions (empty transaction list)

The files containing these classes along with a menu-based test program file should be submitted as an attachment to an email message by 12:00 p.m. on November 27, 1996. No late programs will be accepted, so turn in whatever you have by the deadline.

CSCI 383 **Program Assignment 9** **Due December 6, 1996**

Merry Christmas to All

Santa Claus sleeps in his shop up at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf's problem is never serious enough to wake up Santa, so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return after Santa has solved their problems. If Santa is awakened by both three elves and the last reindeer returning from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible.

You are to solve this problem using Ada tasks. You are to create three task types, Reindeer, Elf, and Santa. Define these task types and any other items you need in a package called Task_Pack.

In your main program, you will declare a Santa task and an array of nine Reindeer tasks. In addition, you will have in your main program a pointer to an Elf task and create a new Elf task every time an elf has a problem. Your program will be menu driven and the menu presented to the user will appear as follows:

Christmas Menu
1. Reindeer arrival
2. Elf has a problem
3. Check reindeer status
4. Check Santa status
5. Elves problems solved
6. Santa returns from Christmas delivery
7. Terminate program and all tasks

When selection 1 is made, the program should prompt the user for the reindeer number (1..9) and keep reprompting until a valid number is entered. If the specified reindeer has already arrived, a message should be printed and the menu reappear. When selection 2 is made, a new elf task is created and that elf goes to Santa with a problem. When selection 3 is made, the locations of all nine reindeer are reported. When selection 4 is made, Santa's status (sleeping, solving, or delivering) is reported and the number of elves currently waiting for problem-solving is printed. When selection 5 is made, Santa leaves the solving mode and returns to sleeping. When selection 6 is made, Santa returns from delivering mode to sleeping. In the case of selections 5 and 6, if Santa is not in the pertinent mode, a message is printed and the Menu reappears. When selection 7 is made, the program and all tasks are terminated.

This program is to be submitted by email by 12:00 p.m. on December 6, 1996. No late programs will be accepted, so turn in whatever you have by the deadline.

CSCI 383: Programming Languages

Fall, 1996

Course Syllabus

CSCI 383: Programming Languages - MWF 12:00-12:50, VZN B24

Survey of programming languages. Programming language syntax. Theory of computation. Control Structures. Recursion. Language extensibility. Application languages. Applicative languages. Object-oriented languages. Experience programming in Ada. Prerequisite: Computer Science 225. Alternate years, 1996-97.

Objectives: Objectives are for the student to

1. Know and understand the fundamental properties of programming languages.
2. Be able to effectively learn and utilize new programming languages.
3. Know the five major paradigms of programming languages and be able to use them.
4. Be able to choose an appropriate language for a given application.

Text: Programming Languages: Structures and Models, 2nd Edition by Dershem and Jipping

Professor: Herbert L. Dershem / VW 220 / 7508 / dershem@cs.hope.edu

Approach: Imperative and Concurrent/Parallel language features will be illustrated by the Ada programming language. Students will program in Ada to gain experience with the various features. Experience will also be gained in languages of other models including Scheme, Prolog, Smalltalk, and Java.

Prerequisite: Computer Science 225

Exams: There will be three exams in this class, tentatively scheduled as follows:

1. Exam 1 - September 27 in class
2. Exam 2 - October 30 in class
3. Final Exam - December 11 at 2:00 p.m.

Assignments: Frequent assignments will be given, including writing programs in the various languages.

Grading Criteria:

- Exam 1: 10%
- Exam 2: 10%
- Programming Exercises: 45%
- Homework: 15%
- Final Exam: 20%

CSCI 383: Programming Languages

Fall, 1996
Course Content:

Date	Textbook Reading	Assignment
Aug 28	1 Overview/History	
Aug 30	2.1-2.6 Preliminaries	HW 1 due
Sep 2	3.1 Data Types	
Sep 4	3.2 Execution Units	Program 1 (Ada) Exercises 1,2 p. 48
Sep 6	3.2-3.3 Exec. Units & Control Structures	
Sep 9	4.1-4.2 Data Aggregates - Arrays	
Sep 11	4.3-4.6 Other aggregates	Program 1 due Program 2 (Ada)
Sep 13	5.1-5.3 Procedural Abstraction	HW3 due Exercises 1,2,3, p. 131
Sep 16	5.4 Parameters	
Sep 18	5.6-5.7 Functions and Overloading	Program 2 due Program 3 (Ada)
Sep 20	Data Abstraction/Parameterization	Lab Ex. 1-5, p. 172
Sep 23	Ada Packages and Generics	
Sep 25	Critical Issues - No class	
Sep 27	Exam 1	
Sep 30	9.1-9.5 Functional Model	Program 3 due Program 4 (Ada)
Oct 2	9.6-9.7 FP	
Oct 4	10.1-10.2 Scheme	
Oct 9	10.3 Scheme	
Oct 11	10.4 Scheme	HW6 due Ada Journal due
Oct 14	12.1-12.2 Logic-Oriented Model	Program 5 (Scheme) Program 4 due

Oct 16	12.2 A Pure Logic Language	
Oct 18	13.1 Prolog	
Oct 21	13.2 Prolog	
Oct 23	13.3 Prolog	Program 6 (Prolog)
Oct 25	X.1 Introduction to Object-Oriented Model	
Oct 28	X.2 Smalltalk Syntax	
Nov 1	Exam 2	
Nov 4	X.3 More Smalltalk	Program 7 (Smalltalk)
Nov 6	Y.1 Ada 95 and the OO Model	Program 6 Due
Nov 8	Y.2 More Ada 95	
Nov 11	Y.3 Still more Ada 95	
Nov 13	Y.4 Ada 95 ad nauseum	Program 8 (Ada 95) Program 7 due
Nov 15	17.1-17.3 Distributed/Parallel Model	
Nov 18	17.4-17.6 More D/P Model	
Nov 20	18.1-18.2 Concurrent Units in Ada	
Nov 22	18.3 More Concurrent Ada	
Nov 25	18.4-18.5 Still more Concurrent Ada	Program 9 (Ada 95)
Nov 27	18.6 Examples in Ada	Program Assignment 8 due
Dec 2	Review and catchup	
Dec 4	Review and catchup	
Dec 6	Review	Program Assignment 9 due

Hope College
Department of Computer Science
Holland, Michigan 49422-9000
(616) 395-7510

March 13, 1997

COPY

David R. Dietz
PWS Publishing Company
20 Park Plaza
Boston, MA 02116-4324

Dear David:

It was good to meet you in San Jose last month. Mike Jipping and I have completed the chapter overview of the Third Edition of our textbook. As we discussed in San Jose, I am sending that to you as an enclosure.

If you have any questions or need any further information at this time, please let me know

Sincerely,

Herbert L. Dershem, Chair

Programming Languages: Structures and Models, 3rd Edition

Herbert L. Dershem and Michael J. Jipping

Chapter 1 - Introduction and Overview

This chapter is unchanged

Chapter 2 - Preliminary Concepts

This chapter is unchanged

Chapter 3 - An Overview of the Imperative Model

This chapter is unchanged except for the addition of features of Ada 95.

Chapter 4 - Data Aggregates

This chapter is unchanged except for the addition of features of Ada 95.

Chapter 5 - Procedural Abstraction

This chapter is unchanged except for the addition of features of Ada 95.

Chapter 6 - Data Abstraction

This chapter is unchanged except for the addition of features of Ada 95.

Chapter 7 - An Overview of a Functional Model

This chapter Chapter 10 of the Second Edition with no significant changes

Chapter 8 - Scheme: A Functional-Oriented Language

This chapter Chapter 11 of the Second Edition with no significant changes

Chapter 9 - ML: A Typed Functional Language

This chapter Chapter 11 of the Second Edition with no significant changes

Chapter 10 - Prolog and the Logic-Oriented Model

This eliminates the use of the LP language and introduces the Logic-Oriented model through Prolog. The coverage of Prolog is greatly expanded and improved over that in Chapter 13 of the Second Edition.

Chapter 11 - Smalltalk and the Object-Oriented Model

This eliminates the use of HOOL and introduces the Object-Oriented model directly through Smalltalk.

Chapter 12 - Java: An Object-Oriented Language

This is an entirely new chapter, replacing the Second Edition's Chapter 16 on C++. It's approach is similar to that of Chapter 16.

Chapter 13 - The Object-Oriented Approach of Ada 95

This is an entirely new chapter.

Chapter 14 - An Overview of the Distributed Parallel Model

This chapter is unchanged from Chapter 17 of the Second Edition.

Chapter 15 - Concurrent Units in Ada 95

This chapter is an update of Chapter 18 of the Second Edition with modifications to reflect Ada 95.

Chapter 16 - Concurrent Threads in Java

This is an entirely new chapter.