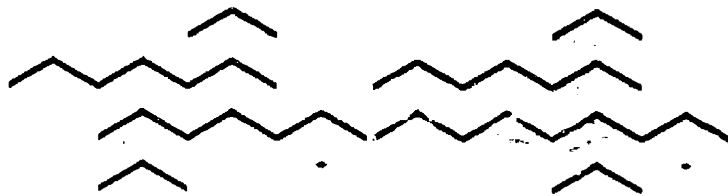
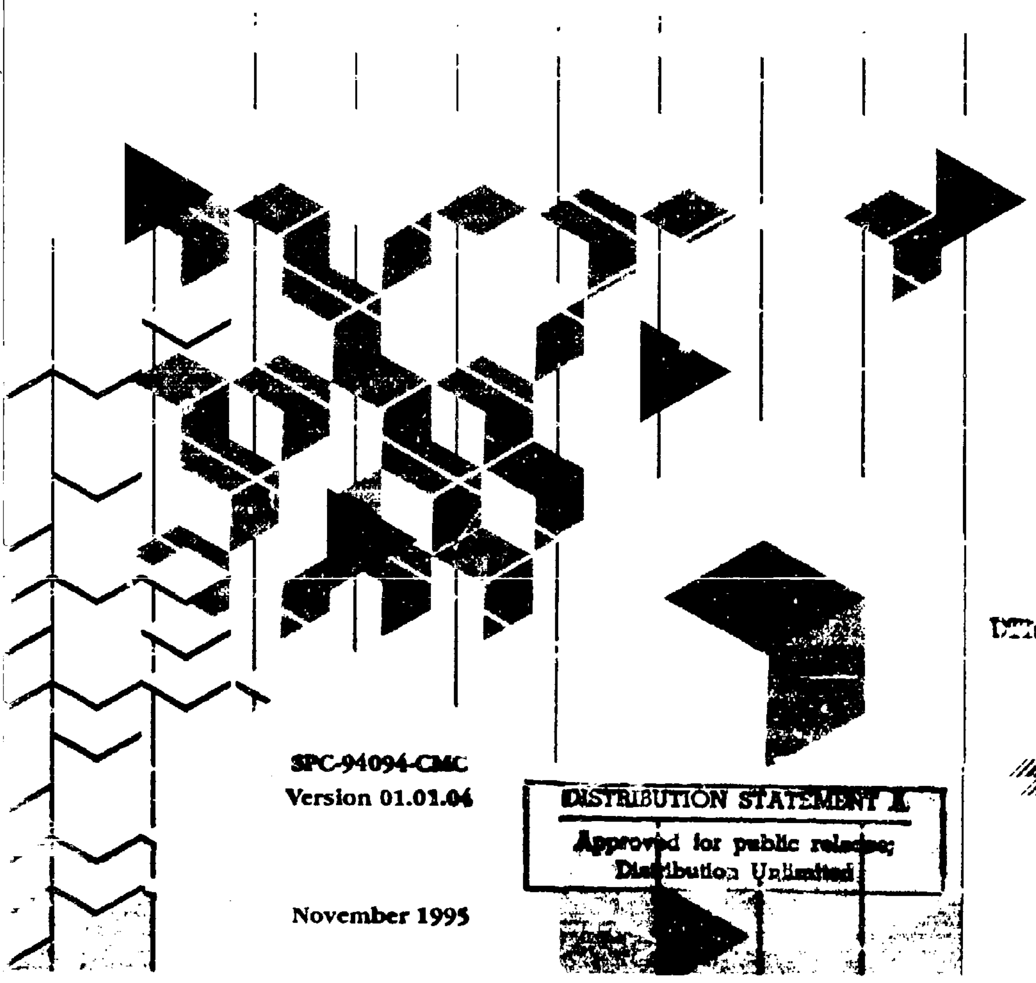
Software Productivity Consortium



Megaprogramming in Ada Course: Lectures and Exercises

Department of Defense Ada Joint Program Office



SPC-94094-CMC
Version 01.01.04

November 1995

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070
(703) 742-8877

Megaprogramming in Ada Course: Lectures and Exercises

SPC-94094-CMC

Version 01.01.04

November 1995

Prepared for the
Department of Defense Ada Joint Program Office

Produced by the
SOFTWARE PRODUCTIVITY CONSORTIUM

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

Copyright © 1995, Software Productivity Consortium, Herndon, Virginia. This document can be copied and distributed without fee in the U.S., or internationally. This is made possible under the terms of the DoD Ada Joint Program Office's royalty-free, worldwide, non-exclusive, irrevocable license for unlimited use of this material. This material is based in part upon work sponsored by the DoD Ada Joint Program Office under Advanced Research Projects Agency Grant #MDA972-92-J-1018. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred. The name Software Productivity Consortium shall not be used in advertising or publicity pertaining to this material or otherwise without the prior written permission of Software Productivity Consortium, Inc. SOFTWARE PRODUCTIVITY CONSORTIUM, INC. MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS MATERIAL FOR ANY PURPOSE OR ABOUT ANY OTHER MATTER, AND THIS MATERIAL IS PROVIDED WITHOUT EXPRESS OR IMPLIED WARRANTY OF ANY KIND.

Approved for public release;
Distribution Unlimited

IBM is a registered trademark of International Business Machines Corporation.
Macintosh is a registered trademark of Apple Computer, Inc.
Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.
Windows and Windows 95 are trademarks of Microsoft Corporation.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1995	Technical Report - Final

4. TITLE AND SUBTITLE

Megaprogramming in Ada Course: Lectures and Exercises

5. FUNDING NUMBERS

G MDA972-92-J-1018

6. AUTHOR(S) R. Christopher, L. Finneran, S. Wartik
Produced by Software Productivity Consortium under contract to Virginia Center of Excellence

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
Software Productivity Consortium
SPC Building
2214 Rock Hill Road
Herndon, VA 22070

8. PERFORMING ORGANIZATION REPORT NUMBER

SPC-94094-CMC, Version 01.01.01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
ARPA/SISTO
Suite 400
801 N. Randolph Street
Arlington, VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

N/A

12a. DISTRIBUTION / AVAILABILITY STATEMENT	12b. DISTRIBUTION CODE
No Restrictions	1

13. ABSTRACT *(Maximum 200 words)*

This is a short course that introduces novice programmers to software engineering concepts and illustrates them using the Ada programming language. The course, which takes about two weeks to teach, is aimed at advanced placement computer science high school classes. It stresses problems that arise in programming in the large, particularly those caused by change, communication, and complexity. It shows how software engineers employ abstraction, information hiding, and software reuse to deal with these problems. The solutions shown are expressed in Ada. The students see and appreciate how Ada can help them solve real problems.

The course material contains viewgraphs instructors can use as the basis of lectures. Each viewgraph has accompanying notes that show how to present the viewgraph and suggest topics for discussion. The course is divided into four units; following each unit are summaries, suggested group activities, and homework assignments. A comprehensive examination and an evaluation form are also included.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Software engineering, software reuse, course, Ada, information hiding, abstraction	162
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

CHANGE HISTORY

Version Number	Date of Change	Change Description
Version 01.00.05	April 1995	Original document.
Version 01.01.04	November 1995	Title of course, copyright notice, and slide format is changed.

This page intentionally left blank.

CONTENTS

	Tab
Unit 1: Software Engineering	1
Unit 1: Software Engineering, Workbook	2
Unit 2: Abstraction	3
Unit 2: Abstraction, Workbook	4
Unit 3: Information Hiding	5
Unit 3: Information Hiding, Workbook	6
Unit 4: Reuse	7
Unit 4: Reuse, Laboratory	8
Test and Survey	9

This page intentionally left blank.

ACKNOWLEDGMENTS

Lisa Finneran and Steve Wartik created and wrote this course. Christine Ausnit and Jeff Facemire reviewed it for the Consortium; Jerry Berry and Lydotta Taylor served as external reviewers. Bob Christopher supervised the course's development. Bobbie Troy lent her technical editing skills, and Deborah Tipeni and Debbie Morgan proofread and corrected the final version.

This page intentionally left blank.

Megaprogramming in Ada Course: Lectures and Exercises

This material is based in part upon work sponsored by the Advanced Research Projects Agency under Grant #MDA972-92-J-1018.
The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

DISCUSSION

To set the stage for this course, ask your students to write down their answers to the following questions:

- How do you write a software program now?
- How large are the software programs you write now?
- How would you write a software program that was over a million lines of code?

High school and college computer science courses typically center around programming languages, data structures, etc. Students do not have a clear understanding of problems software engineers face in their day-to-day jobs. Completing a software project in industry today requires a team effort (e.g., a team of engineers to gather requirements, develop a design, implement the design, test the product, fix bugs found in the product, and maintain the product). High school and college students rarely get an opportunity to work in large group settings because typical homework assignments are done on an individual basis. The students' focus is on writing code so they never realize that code is only a small part of developing a software system. This course will introduce additional factors (e.g., building software entails more than just writing code) that influence how software is built today.

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Explain the problems software engineers face today (e.g., complex systems, redirection, and change are the norm)
- Explain the importance of software engineering; how it helps with two major issues: communication and change
- Explain why the Ada language was developed and how it addresses software engineering needs

1. Software Engineering
2. Abstraction
3. Information Hiding
4. Reuse

Unit 1

Software Engineering

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Think of the implications of developing and maintaining 200 million lines of code. The actual coding is a small part of the software process. We will be discussing these implications, referred to as programming in-the-large, throughout this course.

Programming in-the-large involves many facets. Several companies are usually involved in the design and development of a single, large, software system. These companies are often geographically separated (i.e., in different parts of the country or world). No single person can comprehend all of the information associated with a large software system. As the number of people assigned to a project increases, the time devoted to communication increases while software development time decreases. The space shuttle and strategic defense initiative are some examples of programming in-the-large. (The strategic defense initiative is a proposed space-based monitoring system that tracks potential threats against the United States.)

Also consider the effect of change on 1 million lines of code. A major problem facing today's software engineers is the impact of changing customer requirements or modifying software designs. Consider the various sources for requirement/design changes for the space shuttle, for example. Congress changes their requirements indirectly through budget allocations, NASA releases software upgrades based on maintenance, changing mission requirements (e.g., launching satellites, repairing satellites, etc.).

Programming in-the-small, however, usually consists of one or two people working side by side to design and write the code. Programming in-the-small does not contain the complexity of large systems; small systems can be developed using only a few people.

STUDENT INTERACTIONS

- What do you think the major problems are in trying to develop 200 millions lines of code? Maintain 200 million lines of code? (Change, complexity, and communication.)
- State the differences between programming in-the-large and programming in-the-small. (Program size, program complexity, geographical separation, number of people.)

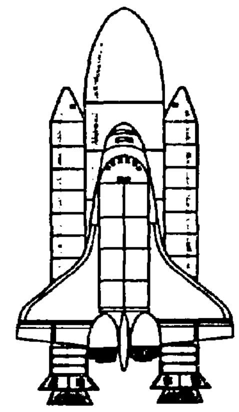
OBJECTIVES

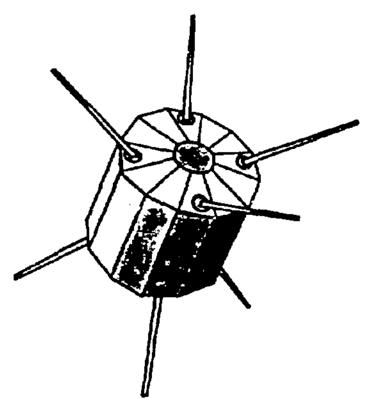
The students should be able to:

- Explain why communication is so important in developing large software systems
- Explain why coding time is a small percentage of the total development time

Programming in-the-Large

- Space Shuttle
  - Over 1 million lines of code
  - 5 onboard processors
  - Over 6 companies developing software
- Strategic Defense Initiative
  - 10 million lines of code
  - Several companies developed software
  - Program listings taller than a 5-story building!





Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Unfortunately, large software systems are often late (behind schedule), cost much more than originally budgeted, and frequently do not satisfy the end user. Large software systems are often unreliable because they fail to operate under normal conditions. Changing requirements and designs are the norm for large software development efforts. Large software projects *are extremely complex. Think of the engineering complexity that went into the design and implementation of the* space shuttle.

These *characteristics are also true of commercial products.* Consider Microsoft Corporation's announcement of *the new* Windows 95 *release.* They had anticipated that the new version of Windows would be released in the beginning of 1994. They delayed until middle of 1994 and then slipped the delivery to January 1995. Now they are saying that Windows will be released by the summer of 1995. Microsoft is trying to be responsive to market needs and it is delaying their release (e.g., changing requirements).

The reasons that large software systems are late and cost more than originally budgeted are that the code was badly organized, and different pieces were so interrelated that nobody could keep it all straight. Changes to one part *of the software rippled throughout all* the code. Often requirements and design information were never recorded.

These problems have been referred to as the **software crisis**.

Software engineering was introduced in the 1970s in an effort to solve the software crisis by bringing an engineering discipline to software development.

STUDENT INTERACTIONS

- How could you prevent these problems (e.g., changing requirements, behind schedule) from occurring? (Designs must *plan for changing* requirements, obtain customer agreement along the software development life cycle; in other words, communication!)
- Can you list additional factors that would make software late? (Changing requirements, not knowing all of the customer requirements.)

OBJECTIVE

The students should be able to:

- State the problems that led to a software engineering discipline

Software Crisis

- Few software products
  - Are delivered on time and on budget
  - Satisfy customer needs
  - Work reliably
- Large software systems
  - Must accommodate changing requirements and design
  - Are extremely complex

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Let's take a look at the major factors for change surrounding large software system development. From the previous slide, we see that large software systems must accommodate changing requirements and design. When thinking about the types of changes, realize that large software systems exist for very long periods of time. Consider the space shuttle. The software that ran on the first shuttle in 1981 is still being used today and is evolving 14 years later in 1995.

- The need for a software system to change can occur quite quickly. There are several reasons for this: enhancements, requirements changes, design changes, upgrades, etc.
- The changes mentioned in this chart reflect changes that software systems had to undergo when they were in operation.
- The largest reason for change was due to enhancements. Enhancements include adding new features/functionality and/or modifying existing features/functionality.
- The cost of these changes is quite expensive—as much as 75% of a company's information technology budget is consumed by software modification (as opposed to new development).
- These changes cause a lot of project overruns and, at worst, lead to project cancellations.

STUDENT INTERACTIONS

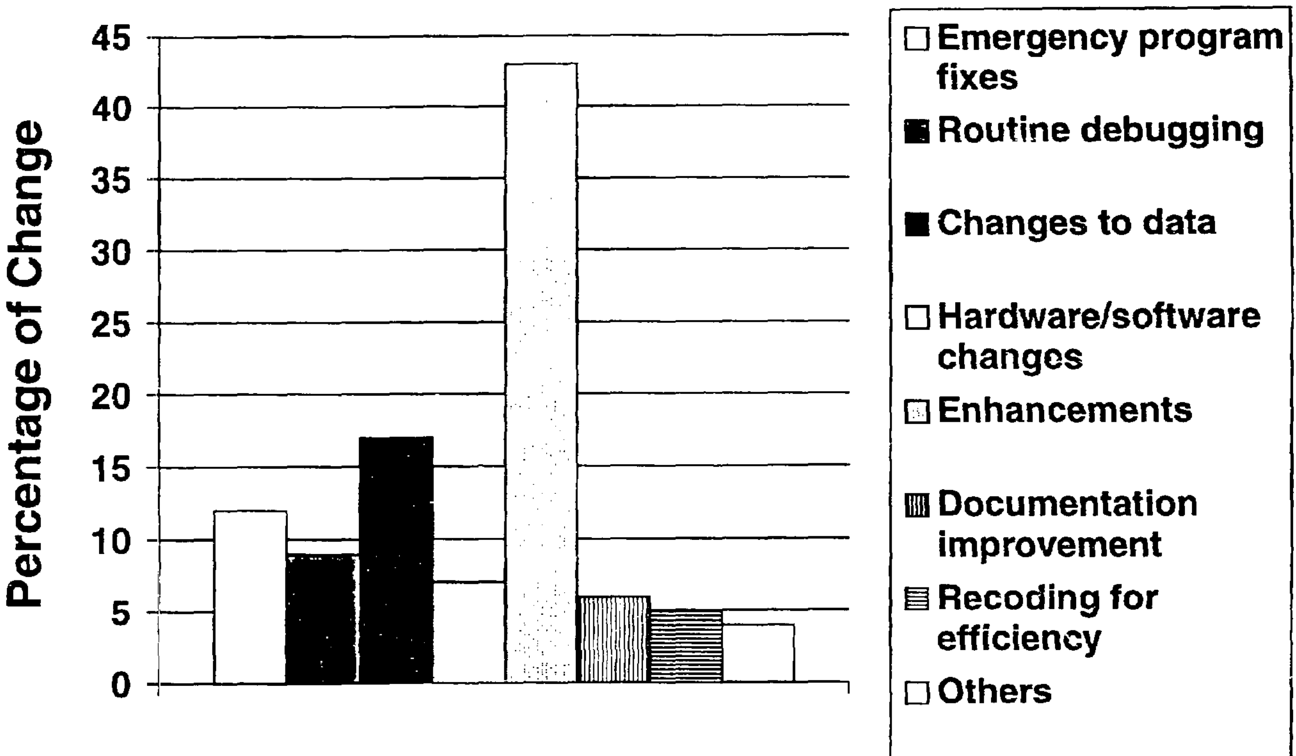
- Why do you think the largest reason for change is enhancements? (Once the software is delivered and being used, users want new features.)
- How do you think communication affects these changes? (These changes must be communicated and their effects on the design and implementation must be considered.)

OBJECTIVE

The students should be able to:

- Identify the types of changes affecting software development

Reasons for Software Changes



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Software engineers do much more than just code; data on software development shows only 15% of their time is spent in coding. Gathering requirements and producing a design involve various communication skills. For example, usually a team of software engineers elicits requirements from one or more customers. These requirements must be interpreted and communicated back to the customer to make sure that the requirements are valid.

The requirements and design also need to be communicated to other project members. These project members can work for the same organization or different organizations/companies.

The software engineers must report status to their management and sometimes to management in other companies (especially, if one or more companies have teamed to develop the software).

There are many documents that need to be written for each software project. These include requirements documents, design documents, user manuals, installation guides, etc.

STUDENT INTERACTIONS

- Can you list some problems you might encounter in working with large groups? (How do you make sure that information is communicated to all the appropriate people?)
- Can you list some problems you might encounter in working with the people organized in the example organization chart? (What happens if Company A and Company B are geographically separated?)
- How would you deal with these problems? (Set up regular meetings, use e-mail, phone calls/conference calls, etc.)

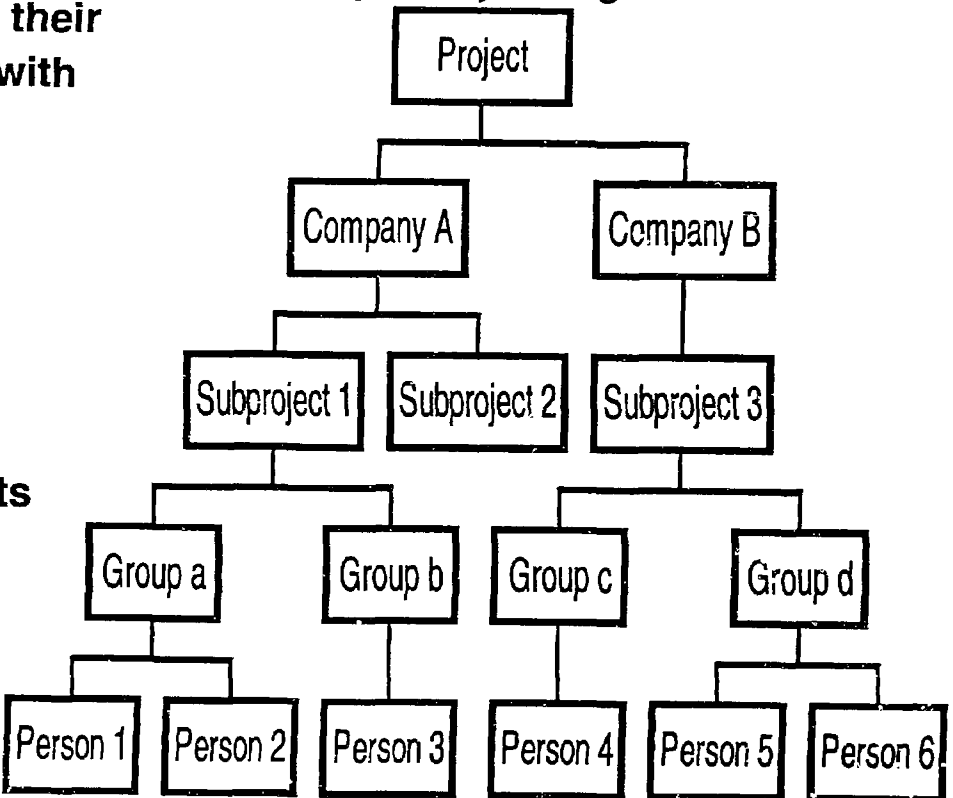
OBJECTIVE

The students should be able to:

- Understand why communication is so important and difficult in developing software

Complexity Requires Communication

- Software engineers spend the majority of their time communicating with
  - Managers
  - Customers
  - Peers
- They must also write
  - Technical documents
  - Technical reports
  - Presentations



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

As you can see from this pie chart, software engineers spend the majority of their time communicating. In fact, they spend over 35% of their day communicating! Communicating encompasses many different aspects:

- Talking or listening to other software engineers and/or customers
- Talking with a manager

Software engineers spend 16% of their time reading programs and manuals; 13% of their time writing programs. They actually spend less time coding than communicating!

Software engineers, unlike students, do not have the luxury of simply receiving a homework assignment and producing code. Software engineers need to focus on communicating what is required and coordinating activities across groups and organization.

STUDENT INTERACTIONS

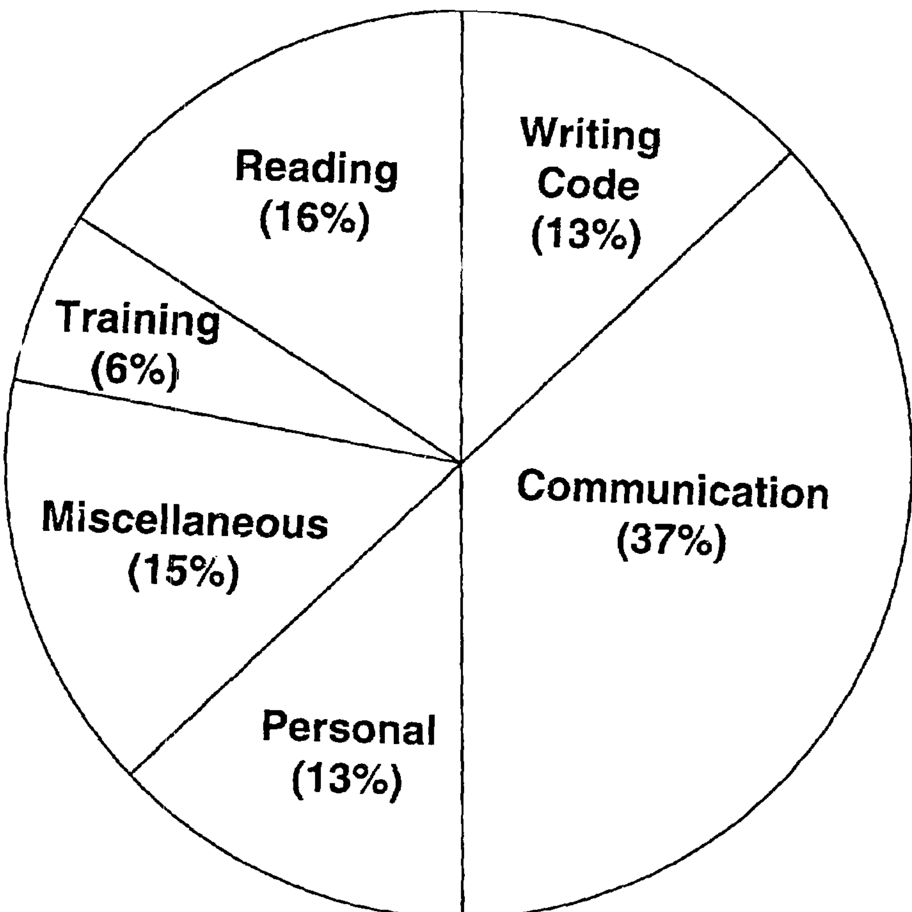
- Why do you think software engineers spend more time communicating than they do writing code? (They need to be able to communicate the complexity of their designs; they need to understand their requirements, they need to work in groups, etc.)

OBJECTIVE

The students should be able to:

- Explain why communication plays such an important role for a software engineer

How Do Software Engineers Spend Their Time?



Software engineers spend half of their lay communicating and writing code!

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

The discipline of software engineering emerged to solve the software crisis. Software engineering uses sound engineering principles in the development of software systems. The main objective of software engineering is to facilitate the production of high quality software systems within budget and on time.

This course concentrates on three aspects of software engineering:

- How software engineering helps with complexity
- How software engineering helps with communication
- How software engineering helps with modularity or the ability to minimize the effect of change

You use design methods as your building blocks to deal with complexity. Design methods help you break a problem into manageable subsets so that individual software engineers can design and implement a piece of the problem. Design methods also guide you through the many decisions you make during design to help avoid things falling through the cracks. Design methods guide you in what to write down to capture the design so that software change is easier, i.e., a form of written communication between designer and maintainer. Design methods also guide you to write reusable code. Software reuse is using previously written software on a new project. Unit 4 is dedicated solely to issues related to software reuse.

Communication is important for all phases of the software development life cycle. **Requirements** specify what you are to build. **Design** specifies how you will build the software. **Code** is the actual implementation of your design. Finally, **test** allows you to verify that you have built the software according to the requirements. To minimize the effect of change, you want to solve relatively independent pieces of the problem and bring them together as a set of cooperating modules. Think of the puzzle pieces as parts of the software problem you would divide among various companies. These parts are referred to as modules.

STUDENTS INTERACTIONS

- Relate the process on Slide 1-7 to the process you use to do your homework. (Teacher hands you your requirements in terms of a homework assignment; you then design a solution on a piece of paper; type in your program; compile, link, and execute.)

OBJECTIVES

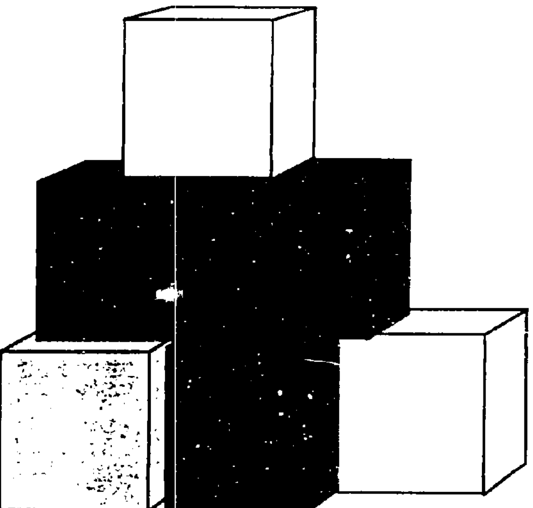
The students should be able to:

- Explain that software engineering helps in managing complex systems
- Explain that software engineering aids in communication for all phases of the life cycle
- Explain that software engineering helps in modularity

Large Projects Demand Software Engineering

Complexity

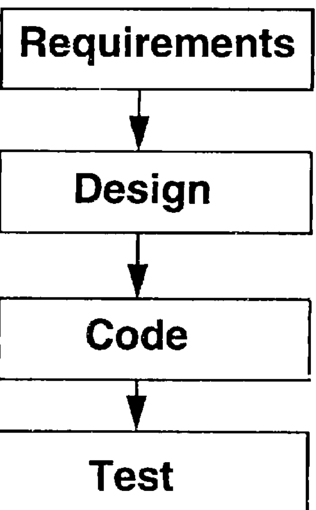
- Design methods
- Reuse



"Building Blocks"

Communication

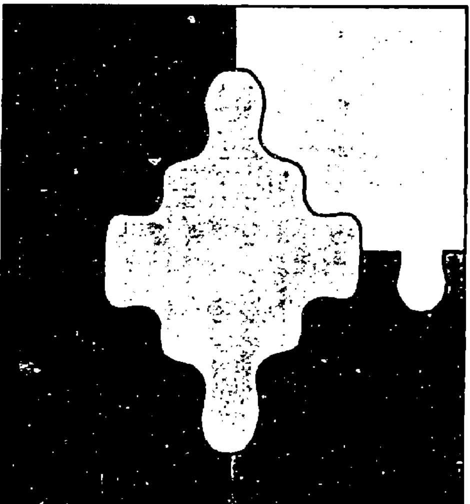
- All phases of software development



"Process"

Change

- Modularity
- Interfaces
- Information hiding



"Cooperating Modules"

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

The U.S. Department of Defense (DoD) is a major developer and user of computer software. In 1974, the DoD and its contractors were using more than 100 different programming languages. This made it difficult to reuse existing software and transfer software engineers from one project to another. A single standard programming language was needed.

The DoD was also looking to develop a programming language that facilitated sound software engineering principles to address the problems identified as the **software crisis** (e.g., systems were difficult to change, behind schedule, etc.)

The Ada programming language was developed and standardized in 1983. The language underwent a formal revision process starting in 1988 and ending in 1995. Ada is named after the world's first computer programmer, Lady Ada Augusta, Countess of Lovelace, daughter of the poet Lord Byron. She was a talented mathematician who worked with Charles Babbage (he designed the first programmable computer).

What does it mean for a programming language to be standardized? For Ada, it means that all validated Ada compilers strictly adhere to the Ada standard. This means that there are not different dialects of the language. As a result, programs written in Ada will compile on any computer containing a validated Ada compiler.

In subsequent units, we will talk about each one of these features (e.g., complexity, communication, change, and reuse) and how Ada helps you with these features.

STUDENT INTERACTIONS

- How does the programming language(s) you are familiar with support (or not support) software engineering principles (e.g., complexity, communication, change, and reuse)?

OBJECTIVE

The students should be able to:

- Explain that the Ada programming language was developed to address software engineering principles

Ada Programming Language Supports Software Engineering

- Ada introduced and standardized in 1983; revised in 1995
- Supports complexity
  - — Ada provides mechanisms for modularity
- Supports communication
  - — Ada is easy to read/understand
- Supports change
  - — Ada mechanisms localize changes to a single module
- Supports reuse
  - — Ada mechanisms promote reusable components

Copyright © 1995, Software Productivity Consortium.

UNIT 1: SOFTWARE ENGINEERING

UNIT SUMMARY

Much industrial software development produces very large software systems, consisting of millions of lines of code. Such a system takes years to develop. It's a team effort, not an individual activity. In fact, a software system is typically a joint effort by several companies.

Your Computer Science courses have probably concentrated on writing code. To write code requires many skills. These skills include mastering a programming language, using algorithms and data structures, and using a compiler—all fundamental skills you need each time you create a program. Yet despite the need for these skills, they are not the most important ones professional programmers possess. Writing code is the smallest, easiest part of developing software. There are other activities that consume far more time and require much greater skill.

Programming in-the-large (that is, developing large software systems) is very hard. Most large software systems are delivered later than planned and contain bugs. Many people even believe this country faces a **software crisis** because programming in-the-large is so difficult.

Why is developing large software systems so hard? Much of the reason stems from two factors: **change** and **complexity**. We will study change and complexity in this course.

Once software is written, it changes. This is a fact of life. When you write software, you seldom anticipate all the ways it will be used. (Consider that, as of this writing, Microsoft Corporation has produced six versions of DOS, six versions of Word, and three versions of Windows.) Also, you seldom discover all the bugs. New uses and bugs call for change. The problem with change is not that it occurs, but that implementing a seemingly simple change often requires huge amounts of work. If you've ever made a change that rippled throughout your program, you understand why. Then too, think of how much more work you would have if you also needed to change user's manuals, installation guides, and other supporting material that accompanies large software systems.

Programs are complex because of the sheer number of details inherent in them. No doubt you realize that the larger your program, the more things you need to keep track of. But really, complexity only truly manifests itself in team settings. Though you may understand your own code well enough, you'll experience troubles explaining its inner workings to someone else (you'll have a chance to try in Unit 2). Therefore, when you write software in a team, you spend much of your time communicating with other team members about the software. You also spend much of your time writing technical documents that describe your work. Management briefings, user's manuals, and design reports are examples of such documents. Therefore, complexity necessitates **communication**.

Change and communication make developing software potentially very problematic. For this reason, this course shows software development to be an exercise in **engineering**. This is why Unit 1 introduces **software engineering** as the preferred way to develop software. The dictionary defines engineering as the disciplined application of science and mathematics in making systems that are useful to humanity. Software engineering is applying science and mathematics to help you make useful software systems.

When you practice software engineering, you follow a **software development process**, shown in Figure 1. The process breaks software development into a set of coherent steps. In each step, you focus on a particular aspect of developing software:

- In the Requirements step, you focus on the problem you must solve.
- In the Design step, you focus on organizing a solution to the problem you defined in the Requirements step.
- In the Code step, you write source code, implementing the plan you created in the Design step.
- In the Test step, you test the software you created in the Code step to be certain it meets the requirements you defined.

Focusing on specific areas in each step helps you deal with change and communication problems.

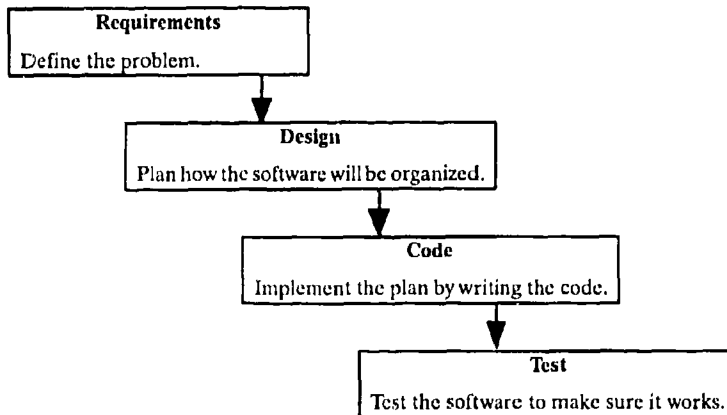


Figure 1. The Software Development Process

Programming languages can also help you deal with change and communication problems. In this course, you will learn about the programming language Ada. You will see how a software developer, through careful and correct use of Ada's features, can facilitate communication of necessary information to other software developers in a team. You will also see how developers can use these features to lessen the workload in response to change.

UNIT 1: SOFTWARE ENGINEERING

GROUP ACTIVITY

COMMUNICATION

Your Student Government Association has decided to purchase a vending machine and wants you to build it from the following parts:

1. A money acceptor
2. A change dispenser
3. A set of food dispensers
4. An item selector

Split into groups. Allocate the parts among the people in your group. Working independently, everyone must write down which other parts they think will interact with their own part. When everyone is finished, get together and compare your results.

HOMEWORK

Look up the definition of engineering in a dictionary. What does it say? Based on this definition, what do you think software engineering is?

UNIT 1: SOFTWARE ENGINEERING

TEACHER NOTES FOR EXERCISES

GROUP ACTIVITY

COMMUNICATION

Your Student Government Association has decided to purchase a vending machine and wants you to build it from the following parts:

1. A money acceptor
2. A change dispenser
3. A set of food dispensers
4. An item selector
5. A coin return button

Each of these is visible to the person operating the machine. Behind the scenes, however, they work together to provide people with vending services.

Split into groups. Allocate the parts among the people in your group. Working independently, everyone must write down which other parts they think will interact with their own part. When everyone is finished, get together and compare your results.

Figure 2 depicts one possible set of interactions.

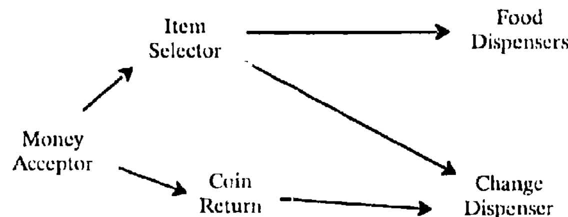


Figure 2. Interaction Among Vending Machine Parts

That is, the money acceptor lets the item selector know how much money has been fed in to date. The item selector notifies the food dispenser when the person makes a choice and arranges for change to be dispensed if the person has fed in more money than the item costs. The money acceptor also notifies the coin return how much has been fed in; if the person presses the coin return button, then the coin return has the change dispenser provide change.

This is only one possible solution. When students do this activity, they will probably come up with conflicting ideas of how the parts interact. Of course, if they had agreed on how each part behaves beforehand, they would not have had this difficulty.

This activity illustrates the need for communication in software. Too often, group members begin working without a clear idea of what everyone else is doing. The result is akin to what the students will experience in this activity.

HOMEWORK

Look up the definition of engineering in a dictionary. What does it say? Based on this definition, what do you think software engineering is?

Dictionaries define engineering as the application of science and mathematics (some add arts) in order to make properties of matter and nature useful to humanity in structures, machines, systems, or processes.

Software engineering, then, is the application of science, mathematics, and arts (to appreciate the artistic component, look at some modern multimedia applications) in order to make properties of matter and nature useful to humanity in creating ***software*** *systems. There are two types of properties:*

1. ***Properties of matter and nature.*** *These come from the problem you're solving. For example, if you are writing a program that calculates the time a ball takes to fall when dropped from a certain height, you use properties of gravity as determined by the laws of physics.*

2. ***Properties of software.*** *Software is not matter, and it does not occur in nature, so we must consider its properties separately. Software properties include algorithm execution speed (the big O notation) and memory use. These properties are what make up the discipline of* ***computer science.***

Unit 2

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Software complexity is a major issue facing large software development efforts. Abstraction is one of the fundamental ways that humans cope with complexity. An abstraction provides a simple view of a problem by summarizing the interesting and essential properties. In other words, you emphasize details that are significant and suppress details that are not.

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Explain what an abstraction is and how it helps us deal with complexity
- Explain how abstraction helps us deal with communication issues
- Explain how abstraction helps in managing change
- Explain what Ada mechanisms are available for creating abstractions
  - Ada packages
  - Concept of a well-defined interface
  - Separate compilation (package specification versus body)
  - Context clauses

1. Software Engineering
2. Abstraction
3. Information Hiding
4. Reuse

2
3
4

Unit 2

Abstraction

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

If you were planning to drive from Washington, D.C. to Los Angeles, CA, what maps would you need?

- Map of the U.S.: You'd need a map of the U.S. so that you could choose the major highways through the various states to get from Washington to California.
- Map of California: Once you get to California, you'll need to take a look at major and minor roads to get from the border to Los Angeles.
- Map of Los Angeles: Once you get to Los Angeles, you'll need to see all of the roads into Los Angeles so that you can navigate to the corner of Hollywood and Vine.

What is the difference between these maps? The level of detail they contain.

STUDENT INTERACTIONS

- To illustrate the need for abstractions in our every day lives, pose the following questions:
  - How big would the map be if it contained all of the details of all the city and states between Washington and California? (Too big to manage.)
  - Would you be able to use it? Why or why not? (Probably not because it would contain too much detail.)
  - How would you tell someone to use such a map? (It would be very difficult; imagine the size of the table of contents or index.)
- What are some abstractions that programming languages give you? (Procedures, functions, control loops.)

OBJECTIVE

The students should be able to:

- State how humans manage complexity (the level of detail)

Planning a Trip?

- What maps would you need to drive from Washington, D.C. to Los Angeles, CA?
- Why would you need a:
  - Map of the United States
  - Map of California
  - Map of Los Angeles
- What is the difference between these maps?

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

An abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose **(the essential information)** and suppress those aspects that are unimportant (e.g., level of detail). The purpose of an abstraction is to limit details so that we can do something with the details (e.g., understand them, communicate certain properties about them, implement them).

You use abstraction to aid in communication and to deal with complexity. For example, if you had to describe a car to someone who has never seen one, you describe a car at a very high level (e.g., a form of transportation). You would add details as the person began to understand; however, you would probably not start describing how an engine works.

Many abstractions can exist for the same object. What you emphasize and de-emphasize depends on the perspective of the viewer. For example, a car owner might emphasize the features of their car (e.g., leather seats, CD player, cruise control), whereas a mechanic would emphasize the various parts of a car and how they interact (e.g., radiator, carburetor, exhaust).

STUDENT INTERACTIONS

For the objects shown on the slide (*dictionary, house, computer*), describe the characteristics you would emphasize from the following viewpoints:

- Dictionary: user, editor. (User—words are in alphabetical order. Editor—words are in alphabetical order, definitions are correct and complete.)
- House: homeowner, builder. (Homeowner—number of rooms, kitchen appliances. Builder—materials needed to build home, contractors to help build the house.)
- Computer programmer, user. (Programmer—compilers, linkers, debuggers. User—word processors, games.)

Do some of the characteristics overlap?

(This interaction shows that the characteristics you emphasize vary, depending on your viewpoint.)

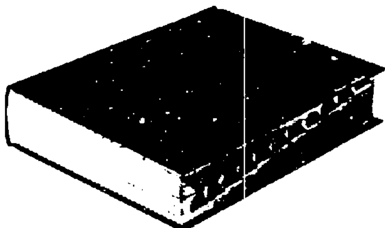
OBJECTIVE

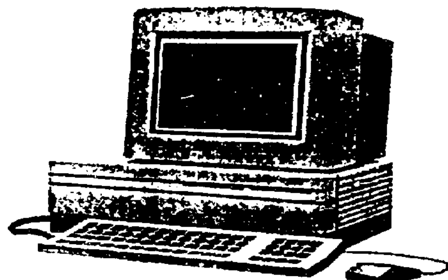
The students should be able to:

- Define abstraction

Abstraction

- A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information
- Emphasizes significant details
- Suppresses unimportant details





Examples of Abstraction

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Suppose you have to write a simple program that reads a series of integers from a file and prints them in reverse order to another file. Now suppose you had three people assigned to write such a program. One person would be assigned to read the integers, another to store them in reverse order, and finally, one to output them. Let's concentrate on the second person; the one responsible for storing the integers in reverse order.

Why should we concentrate on the second person? Because, potentially, the design of the second piece is important to both the first person and the third person. The first person will have to know how the third person expects the integers to be stored. This is the job of the second person.

STUDENT INTERACTIONS

- What would happen if you assigned only two people to solve this problem and told them that they could not communicate? What assumptions do you think they'd have? (They would probably be very different!)

OBJECTIVE

The students should be able to:

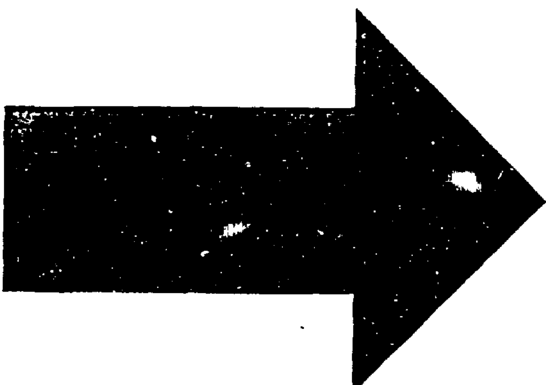
- Understand how abstraction can help with communication and managing complexity

Example of Abstraction: Requirements

Software Requirements:
Read, Reverse, Write

Input: File of Integers

1
2
3
4
5
6



Output: File of integers in reverse order

6
5
4
3
2
1

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Our design consists of three modules: a module to read the input file, a control module to store the numbers in reverse order, and a module to write the output file. Concentrating on the control module, what is a good abstraction for storing numbers in reverse order? A stack. Let's create a stack module to store the the numbers from the input file.

How do these modules aid in communication and dealing with complexity? Abstraction allows you to talk about what the modules will do without specifying how they will do their jobs. Abstraction allows groups of engineers to work together to come up with a solution as a set of cooperating modules.

You deal with complexity by omitting the insignificant details. In this case, the insignificant details are the actual algorithms and control logic that each of the abstractions will eventually take. The significant details are that a stack will be used to reverse the order of the input file. How the stack is implemented is insignificant at this point.

STUDENT INTERACTIONS

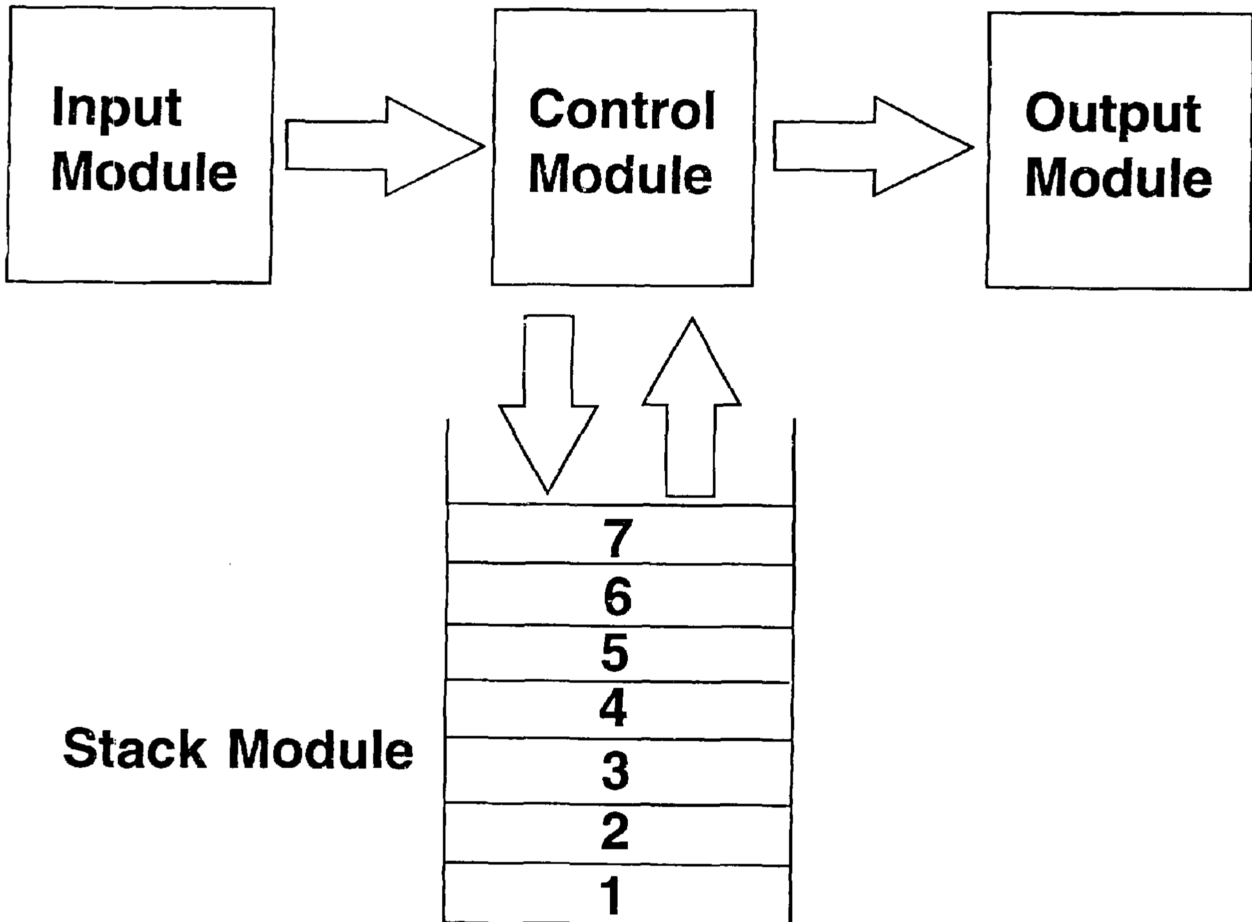
- Why is it important to talk about what these modules will do without specifying how they will do their work? (This is the way we manage complexity.)

OBJECTIVES

The students should be able to:

- Explain how abstraction is used to deal with complexity and communication
- Explain the importance of distinguishing between what needs to be done and how it will get done

Example of Abstraction: Design



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

A stack is a sequence of items in which items are added and removed from the top of the stack. The details or behavior that are important to us for a stack are:

- Every stack has a top (you can read the topmost item on the stack).
- You push an item onto a stack.
- You pop an item off of a stack.
- You can test whether a stack is empty.
- You can retrieve the size of the stack (i.e., the total number of items pushed onto the stack).
- You can test whether a stack is full.

The details that are not important to us are the elements or items of the stack (e.g., the behavior of the stack acts no different if it holds an integer, record, or real number). In computer science terms, the data structure we use to implement the stack is unimportant to characterize the behavior of a stack. A stack behaves the same way regardless of the items it stores or the mechanisms used to store the items (e.g., array, file, linked list).

STUDENT INTERACTIONS

- What are some examples of stacks in a daily situation? (Push-down stack of plates in the cafeteria.)
- How would you explain a queue to someone who is not familiar with computers? (Any line where you have to stand and wait for something, check-out line, stop light.)

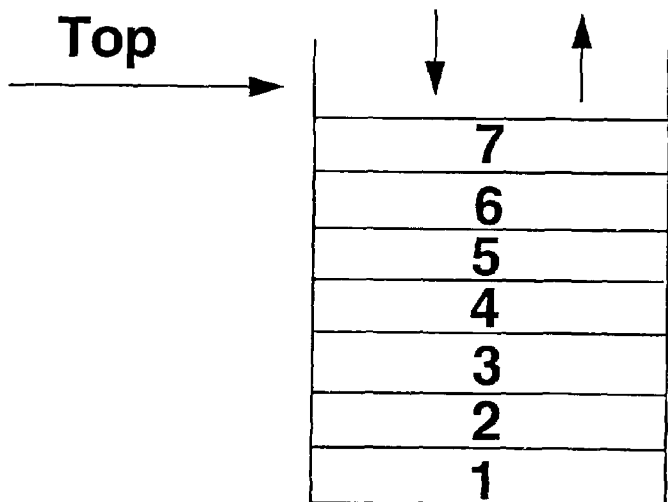
OBJECTIVES

The students should be able to:

- Explain how to use an abstraction to aid in reducing complexity and enhancing communication
- List everyday examples of abstractions

Using a Stack Abstraction

- Let's solve the problem by storing the integers in a stack as we need them
- Recall, a stack stores a sequence of items in which items are added and removed from the same end
- Linear list that grows and shrinks from one end



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

A specification defines the behavior of an abstraction. A specification defines the visible parts of the abstraction. A specification serves as a contract between the implementer of the abstraction and a user of the abstraction. The important point here is that a user of the specification defines the operations, not how those operations will be implemented. This means identifying the name of the operation and its input and output parameters, if any.

You use a specification to describe the abstraction. You are defining the essential information, e.g., that a stack has a push/pop operation, while suppressing the unimportant details, e.g., how you implement the push/pop operation.

A specification shows the requirements and design for an abstraction. Consider the example of reading from a file and printing in reverse order: you would take your specification to the people responsible for storing and printing the integers to make sure that the operations were sufficient for them to complete their tasks. What would happen if you took the actual code for a push or pop operation? There would be too much information for either person to know whether you had the right set of procedures available. You also want to make sure you have identified the right set of operations before coding. This is one of the important steps taken during design.

The definitions for the stack specification are:

Push	Pushes an integer onto the top of the stack
Pop	Removes the top integer from the stack
Is_Empty	Returns true if the stack contains no integers; false otherwise
Top	Returns the integer at the top of the stack
Size	Returns the number (count) of integers pushed onto the stack

STUDENT INTERACTIONS

- What would an example specification look like for a queue? (Add an element to the back of the queue, pop an element from the front of the queue, etc.)
- What is missing from the specification? (E.g., Is_Full)

OBJECTIVES

The students should be able to:

- Describe what a specification is and how to create one (*i.e.*, identify operations)
- Explain why a specification is important in communicating

Stack Specification

- **Let's list all of the operations (procedures and functions) for a stack, independent of how it will be implemented**
- **This is called a <u>specification</u>**

```
procedure Push (Element : in Integer);
procedure Pop;
function Is_Empty return Boolean;
function Top return Integer;
function Size return Integer;
```


Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Let's create a single, cohesive abstraction. We can do this by creating an Ada package. It allows you to group procedures, functions, types, and variables into one programming unit or module. An Ada package forms a collection of logically related type declarations, variables, procedures, and functions that manipulate the type declarations and variables. An Ada **package** consists of two parts: a **package specification** and a **package body**.

Let's package our stack specification into one unit (as opposed to a series of procedures/functions). We can package types, variables, procedures, and functions into one unit to create a single abstraction. This is called an Ada **package specification**.

The package specification identifies the visible parts of the package. This serves as the contract between the implementer of the package and a user of the package. A package specification provides an interface (e.g., what procedures and functions can be called by other procedures and functions or clients).

Through compilation, Ada enforces that a user of the package only has access to the explicit visible parts of a specification. The language rules do not permit a package user to do anything more than what the specification allows.

It is not important for a user to understand how these operations are actually implemented. We'll talk about this in the next unit, Information Hiding.

The package body is where you actually implement the procedures and functions.

STUDENT INTERACTIONS

- What is visible to clients of Integer_Stack? (All the procedures and functions.)
- What would be some meaningful package and operation names to describe a queue? (Add—insert an item at the back of the queue. Pop—remove an item from the front of the queue. Size—returns the number of items in the queue [count].)

OBJECTIVE

The students should be able to:

- Describe an Ada package and the purpose of an Ada package specification

Stack Specification (cont.)

Let's package the procedures and functions together to indicate their related purpose

```
package Integer_Stack is
   procedure Push (Element : in Integer);
   procedure Pop;
   function Is_Empty return Boolean;
   function Top return Integer;
   function Size return Integer;
end Integer_Stack;
```

Serves as a contract between user of the package and the implementer of the package

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

Eventually, you have to implement the various operations for the stack program. You do this by writing the Ada **package body**.

The Ada package body provides the implementation of the subprograms defined in the package specification. For every procedure and function you define in the specification, you must implement each one in the package body.

There are significant advantages to separating the specification from the implementation. Consider again our 2 million lines of code example. A clear interface (without regard to implementation) is important when you have many potential users of the package. This gives you a chance to verify that you are providing the right subprograms for the potentially many different users of your package.

Another advantage comes into play when you decide on the algorithms you'll use to implement the subprograms. Suppose you have several different users of the Integer_Stack package. You could change any of your algorithms (e.g., Push) without affecting any of the users. The only time a user of a package is affected is when you change the specification. More on this concept will be presented in the next unit.

STUDENT INTERACTIONS

- How would this separation (specification from implementation) help with programming in-the-large? (This helps with communication because you do not have to look at all the details.)

OBJECTIVES

The students should be able to:

- Explain what an Ada package body is
- Understand that types, variables, procedures, and functions declared solely in the package body are only visible to the package body

Stack Implementation

```
package body Integer_Stack is
   Index : Integer := 1;
   type Stack_Type is array (1..100) of Integer;
   Stack : Stack_Type;
   procedure Push (Element : in Integer) is
   begin
      Stack (Index) := Element;
      Index := Index + 1;
   end Push;
   procedure Pop is
   begin
      Index := Index - 1;
   end Pop;
   . . .
end Integer_Stack;
```

This information is not visible outside of package body

Implementation of the specification

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

When one package or subprogram uses operations, types, or variables provided by another package, it must name that package in a "with" clause. A with clause allows you to include a package specification into your unit. This allows a package or subprogram to gain visibility into the Ada specification being named. Once other developers have "withed" the compilation unit, they can invoke or use anything in the specification.

The with clause for procedure Read_Input allows Read_Input to call any subprogram in the package specification of Integer_Stack. The same holds true for Write_Output. To invoke a subprogram from Integer_Stack, you specify the fully qualified name: package name followed by a period (.) followed by the procedure, function, type, or variable name.

There are several benefits associated with context clauses:

- See the effect of change: For example, if you change the specification of Pop to return the element it popped, which procedures would have to change? Obviously, you would need to update the specification for procedure Pop so that it will return an integer; also you would have to update procedure Write_Output (remove the call to Top and update the call to Pop).
- Aid in communication: Because you must "with" all packages you use, this helps in communicating parts of the design of the package. You can look at the context clause to get a sense of how complicated the package is and the set of utilities needed. For example, if you saw a context clause that contained 100 packages, how complex would you think the package was?

STUDENT INTERACTION

- You can also "with" procedures in Ada. What must the Control_Module with?
- Suppose you changed the package body for Integer_Stack (e.g., made Stack_Type a linked list). What changes would you need to make to Read_Input and Read_Output? (The answer is none!)

OBJECTIVES

The students should be able to:

- Understand how to call a subprogram in another package
- Understand how context clauses aid in change and communication

Clients of Integer_Stack

Include package specification

```
with Integer_Stack;
procedure Read_Input is
   Element : Integer;
begin
   Open (Input_File);
   while not End_Of_File loop
      Get (Element, Input_File);
      Integer_Stack.Push (Element);
   end loop;
   Close (Input_File);
end Read_Input;
```

```
with Integer_Stack;
procedure Write_Output is
   Element : Integer;
begin
   Open (Output_File);
   while (Integer_Stack.Size > 0) loop
      Element := Integer_Stack.Top;
      Put (Element, Output_File);
      Integer_Stack.Pop;
   end loop;
   Close (Output_File);
end Write_Output;
```

Call appropriate subprograms using fully qualified names

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

A package specification allows us to deal with complexity by implementing a problem separately from the solutions to other problems. As a programmer, you design the interface to your solution. (For the Integer_Stack package, this is when we specified the procedures and functions we would need to define a stack.) Again, a user of the package does not need to know how you have solved the problem (*e.g.*, the kinds of data structures you use, your choice of algorithms, etc.).

Once you have written your specification (packaging the procedures, functions, and data into a package), you compile it into an Ada library. Once your specification is in the library, users have access or visibility to the specification. Therefore, a package specification can have multiple users.

The package body must implement all of the procedures and functions from the package specification. The package body must implement all of the operations as stated in the package specification. Thus, you could not have *Pop return nothing in* the specification by an integer in the body. The Ada compiler would give you a compilation error. The package body may also add additional types, variables, procedures, and functions; however, they are not visible outside of the body. The package body is then compiled into an Ada library.

STUDENT INTERACTIONS

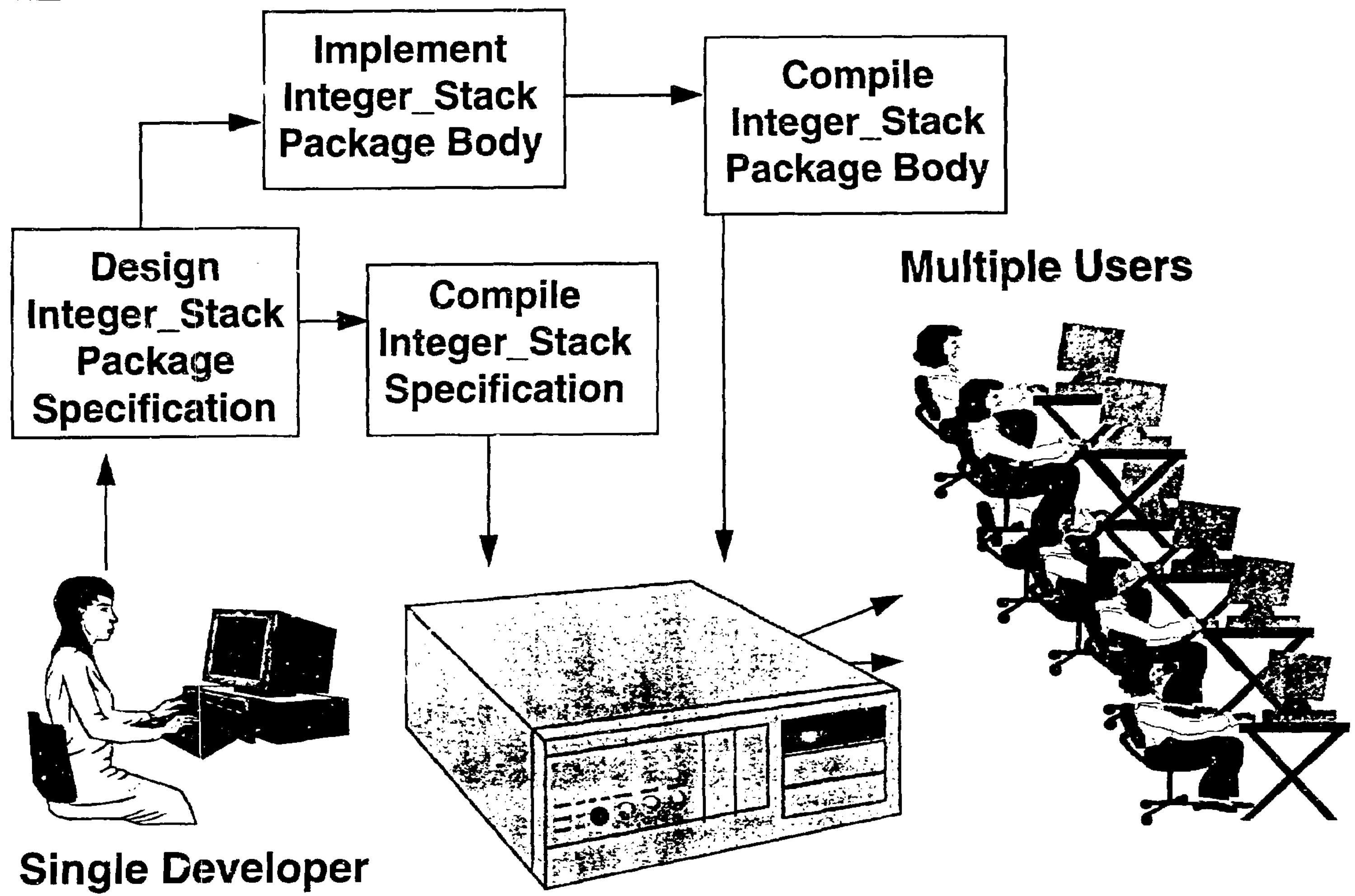
- Can you list other advantages of writing a specification? (Maybe you haven't decided on which algorithms or data structures you are going to use.)

OBJECTIVES

The students should be able to:

- Explain how Ada packages help form abstractions
- Understand the mechanism by which you compile code into an Ada library

Total Picture for Ada Packages



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide summarizes the topics covered in Unit 2.

Abstraction allows you to manage complexity and enhance communication by concentrating on essential information. Think about how lucky we are that we can defer tough decisions until later (e.g., an actual implementation of a stack). We use abstraction in our every day lives, not just to develop software.

STUDENT INTERACTIONS

How does abstraction help in managing complexity and aiding communication? (Allows you to ignore nonessential details.)

OBJECTIVES

Students should be able to:

- Explain abstraction
- Explain how abstraction helps in software engineering
- Explain what an interface is

Summary

- Abstraction allows us to manage complexity by concentrating on essential information
- Abstraction allows us to separate an interface from its implementation
  - Aids in communication
  - Aids in managing complexity
- Ada supports abstraction via packages
  - Package specification implements the interface
  - Package body implements the specification

Copyright © 1995, Software Productivity Consortium.

UNIT 2: ABSTRACTION

UNIT SUMMARY

ABSTRACTION

Abstraction is a technique employed during software design. It lets the developer temporarily suppress irrelevant details so he or she can concentrate on essential information. Developing software requires defining a great deal of detail, so any techniques that can be used to consider information selectively are of great value. Abstraction is one such technique.

What is "essential information" and what are "irrelevant details"? Typically, essential information is the data you need in your program and **what** you will do with it. The irrelevant details are **how** you will represent that data. In general, you can use this division into **what** versus **how** to help you differentiate between essential information and irrelevant details.

For example, suppose a program is to create a file of integers whose content is that of another file of integers, but in reverse order. You can design a program that does this as follows. The program will read the integers from the input file, storing each one on a stack as it is read. When all integers have been read, the program will pop each integer off the top of the stack and write it to the output file. In this design, you have created four modules: one to read input, one to write output, one to hold the stack, and one to control the others (see Figure 1).

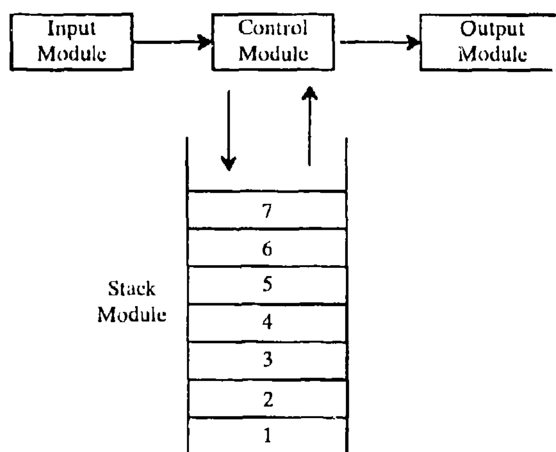


Figure 1. Module Design for Reversing a File of Integers

When you're working in a team, it's important to create the module design. You can assign each person one or more modules. This is a good way for team members to work together.

Recall that a stack is a linear list of values accessible only through a fixed set of operations: Push, Pop, Top, Is_Empty, and Size. This statement of a stack is an abstraction. It proclaims the essential information—namely, what five operations can be used to access a stack. It also defines what kind of descriptive information other packages can see, for example, the stack's size. It suppresses irrelevant details, such as how the stack will be represented.

This essential information is the abstraction's **specification**. In Ada, you can **package** the essential information to show its interrelatedness.*

```
package Integer_Stack is
   procedure Push(Element: in Integer);
   procedure Pop;
   function Is_Empty return Boolean;
   function Top return Integer;
   function Size return Integer;
end Integer_Stack;
```

In Ada, the package construct groups together a set of procedures and functions. (You can also include constants, variables, and data types, as will be shown later.) Everything between the first line and the **end** line is declared to be part of the **package specification**. This package specification declares two procedures and three functions. The first procedure, `Push`, has a single parameter, `Element`. This parameter is declared **in**, which means that you must supply a value for it when you invoke it. Furthermore, its value will be unchanged when `Push` finishes.

The specification gives you enough essential information to let you write most of the program.**

```
with Integer_Stack;
procedure Read_Input is
   Element: Integer;
begin
   Open(Input_File, In_File,
         File_Name);
   Set_Input(Input_File);
   while not End_Of_File loop
      Get(Element);
      Integer_Stack.Push(Element);
   end loop;
   Close(Input_File);
end Read_Input;
```

```
with Integer_Stack;
procedure Write_Output is
   Element: Integer;
begin
   Open(Output_File, Out_File,
         File_Name);
   Set_Output(Output_File);
   while Integer_Stack.Size > 0 loop
      Element := Integer_Stack.Top;
      Put(Element);
      Integer_Stack.Pop;
   end loop;
   Close(Output_File);
end Write_Output;
```

These two procedures both begin with the line **with** `Integer_Stack`, meaning that the information in the package specification of `Integer_Stack` is within their scope. They can, therefore, within invoke `Push`, `Pop`, `Top`, and `Size`. Notice that references to these procedures and functions are preceded by the package's name and a period; e.g., `Integer_Stack.Pop`. This is Ada's way to avoid ambiguities, since other packages might have procedures and functions with the same names as those found in `Integer_Stack`.

The package specification serves as a contract with other modules in the program. When you write it, you are suppressing the implementation as an irrelevant detail but promising that you will develop an implementation that provides the functions stated in the specification. In Ada, you place this implementation in a **package body**, which is separate from the package specification.

* In the code fragments, Ada reserved words are shown in **boldface** type.

** For simplicity and clarity, the code examples omit details of file input and output.

```
package body Integer_Stack is
   Index: Integer := 1;
   type Stack_Representation is array (1..100) of Integer;
   Stack: Stack_Representation;

   procedure Push(Element: in Integer) is
   begin
      Stack(Index) := Element;
      Index := Index + 1;
   end Push;

   procedure Pop is
   begin
      Index := Index - 1;
   end Pop;

   function Top return Integer is
   begin
      return Stack_Contents(Index-1);
   end Top;

   function Size return Integer is
   begin
      return Index-1;
   end Size;

end Integer_Stack;
```

Other modules, such as `Read_Input` and `Write_Output`, do not need to know any details of the implementation. They only need the information in the specification. The Ada programming language enforces this. `Read_Input` and `Write_Output` can access the information in the package specification, but cannot access the information in the package body. They can invoke `Size` but cannot determine that `Stack` is an array or that `Size` works by accessing the variable `Index`. The designer of `Integer_Stack` has hidden the irrelevant details. This shows how you can use abstraction to write a module that shows to other modules only what you consider to be essential information.

Ada packages help teams design and implement programs using abstraction. A team will assign a single developer the responsibility to develop a module such as a stack. The developer will design an Ada package specification for the stack. He or she will then compile the stack's specification and place it in a central library that all the other team members can access. The other members who need a stack can reference the abstraction as they develop their own programs. This gives them access to exactly enough information to design and implement their own modules. Meanwhile, the stack developer will implement the package body for the stack, then compile the package body and place it in the library. Note that other team members can compile their modules without the stack package body, but they can't execute them until the stack package body has been placed in the library!

This page intentionally left blank.

UNIT 2: ABSTRACTION

GROUP ACTIVITY

Split the class into two-person teams. One member of the team should examine the following code:

```
type A is array (<>) of Integer;
procedure p(p1: in A;
            p2: in Integer;
            p3: out Integer) is
   u, m, l: Integer;
begin
   l := a'first;
   u := a'last;
   while l < u loop
      m := (l+u)/2;
      if p1(m) = p2 then
         p3 := m;
         return;
      elsif p1(m) > p2 then
         u := m-1;
      else
         l := m + 1;
      end if;
   end loop;
   p3 := a'first-1;
end p;
```

Describe the irrelevant information in this code to your partner. In other words, do not discuss what you believe is the purpose of this code. Instead, discuss only the algorithms and the data it uses.

Your partner is to guess the essential information from your description: what purpose the code accomplishes. The essential information should be described in terms of two things:

- The value of p3 when the procedure finishes executing
- The name of the procedure

This page intentionally left blank.

UNIT 2: ABSTRACTION

TEACHER NOTES FOR GROUP ACTIVITY

Split the class into two-person teams. One member of the team should examine the following code:

```
type A is array (<>) of Integer;
procedure p(p1: in A;
            p2: in Integer;
            p3: out Integer) is
   u, m, l: Integer;
begin
   l := a'first;
   u := a'last;
   while l < u loop
      m := (l+u)/2;
      if p1(m) = p2 then
         p3 := m;
         return;
      elsif p1(m) > p2 then
         u := m-1;
      else
         l := m + 1;
      end if;
   end loop;
   p3 := a'first-1;
end p;
```

A few words are in order to the teacher who knows Pascal but not Ada. You may wish to rewrite this example in Pascal for your students, since the purpose of the activity is to understand the algorithm rather than to learn Ada. In any case, here is some explanation of the code.

- *The notation <> in the top line is Ada's notation for unconstrained array bounds that are determined when a procedure is called. Thus, any array of integers can be passed to* `p`*. The notations* `a'first` *and* `a'last` *(read "a tic first" and "a tic last," respectively) are the upper and lower indixes of whatever array is passed to* `p`*.*

- *Instead of Pascal's*

```
while condition do begin
   statement1;
   ...
   statementn
end
```

Ada uses

```
while condition loop
   statement1;
   ...
```

```
      statementn;
   end loop;
```

- *Ada's* `return` *statement causes control to return immediately from the procedure in which it's executed.*

- *Instead of Pascal's*

```
   if condition1 then begin
      statement1; ... ; statementn
   end
   else if condition2 then begin
      statementa; ... ; statementz
   end
   else begin
      statementA; ... ; statementZ
   end
```

Ada uses

```
   if condition1 then
      statement1; ... statementn;
   elsif condition2 then
      statementa; ...; statementz;
   else
      statementA; ...; statementZ;
   end if;
```

Matching each `if` *with an* `end if` *eliminates the need for begin blocks. Note also the* `elsif`*, which clearly shows that* `condition2` *logically matches* `condition1`*.*

Describe the irrelevant information in this code to your partner. In other words, do not discuss what you believe is the purpose of this code. Instead, discuss only the algorithms and the data it uses.

Your partner is to guess the essential information from your description: what purpose the code accomplishes. The essential information should be described in terms of two things:

- The value of `p3` when the procedure finishes executing
- The name of the procedure

The procedure `p` *is a binary search algorithm. The identifier names are deliberately abbreviated to make the activity more challenging. A better declaration would be:*

```
procedure Perform_Binary_Search(Values: in A;
              Element_To_Search_For: in Integer;
                Location_Of_Element: out Integer);
```

This procedure specification succinctly captures that essential information which must be known to developers that use this procedure. However, as they write their modules, they do not care about details of the implementations like the identifier names.

The student asked to listen to her or his partner should, in effect, come up with this declaration. In other words, the procedure takes as input a sorted array of integers (p1) and an integer value (p2). It returns in p3 the index of the value if the value exists in the array. If the value is not in the array, it returns in p3 the integer value that is one less than the first valid index into p1. The student presenting the irrelevant details might say something like:

> *"The procedure has two inputs. One is an array of integers. The second is an integer. It declares three integer variables.*
>
> *"It begins by assigning two of the variables the lower and upper bounds of the array. It then checks to see if the second parameter equals the value midway between these two bounds. If it does, then the procedure assigns the index of the middle value to the third parameter and exits.*
>
> *"If the second parameter does not equal the value in the middle of the array, the procedure resets the bounds it will check. If the second parameter is greater than the value in the middle of the array, the bounds are reset to the first half of the array. Otherwise, they are reset to the second half of the array.*
>
> *"This process repeats until a value matching the second parameter is found, or until the difference between the bound is 0 or less. In the former case, the third parameter is set to the index of the matching value. In the latter case, it is set to one less than the array's lower bound."*

This activity will be very difficult if students do not know the algorithm. If your students have not learned about binary searching, you should substitute an algorithm you have previously taught them.

This page intentionally left blank.

Unit 3

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

A program is a solution to a problem. Software engineers work by **designing** a solution and then implementing it. Design is an important part of engineering. You need a design to reduce complexity. Abstraction plays a critical role here. The design lets you think of the solution as a set of interacting parts. You can understand the essential information and irrelevant details of a single part, but in a large program you can't grasp all the details of all parts. Abstraction reduces the details you must consider at any time.

In a team, you also need a design to apportion the program among members. Your design should let you assign each member a set of parts that he or she can implement. When everyone is done, they should be able to assemble their parts to form the whole program.

To create a design, you need a **design method** (a set of guiding principles) that helps you:

- Break up the problem into a set of parts, where each team member can work on some subset.
- Ensure that each team member can work independently of her or his colleagues. More independence means less communication and, consequently, less time consumed.

In this unit, you will see the problems that change and complexity cause when you don't design properly. You will then learn about the **information hiding** design method and how it helps you create designs. Finally, you will see how Ada supports information hiding.

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Explain the stepwise refinement design method and its limitations
- Explain the principles of information hiding
- Understand the relationship between information hiding and abstraction
- Explain how information hiding reduces the chance that a change in one part of a program necessitates changes in other places
- Understand how the Ada packages and **private** mechanisms support information hiding

1. Software Engineering
2. Abstraction
3. Information Hiding
4. Reuse

Unit 3

Information Hiding

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide depicts the objective of design: to define a set of parts and a structure that can guide coding.

During design, you break up a problem into a set of smaller parts, called **modules**. You also describe the relationships between these modules; this is the **structure** of the software. Many things are designed this way. A house is broken into rooms; you can think of the layout of the rooms as the structure. A car consists of an engine, drive train, wheels, etc.; you can think of the locations (engine in front or rear, front or rear-wheel drive) as the structure. In software, modules can be procedures or functions, and the structure describes which procedures call which. A module at a black arrow's tail calls the module at its head; thus Module 1 calls Module 2 and two unseen modules.

You break up a problem for at least two reasons. First, as the notes for Slide 3-1 mention, you can more readily grasp the purpose of a module than of a whole system. That's because each module is an abstraction. This helps you deal with complexity.

Second, you must break up a problem if you are working in a group. Each member of the group needs to be assigned a portion of the solution. This means you must break up the problem so each member can work independently.

This leads to the issue of how you break up the problem. And once you've done so, how can you tell whether your modules and structure are "good" or "bad"?

STUDENT INTERACTIONS

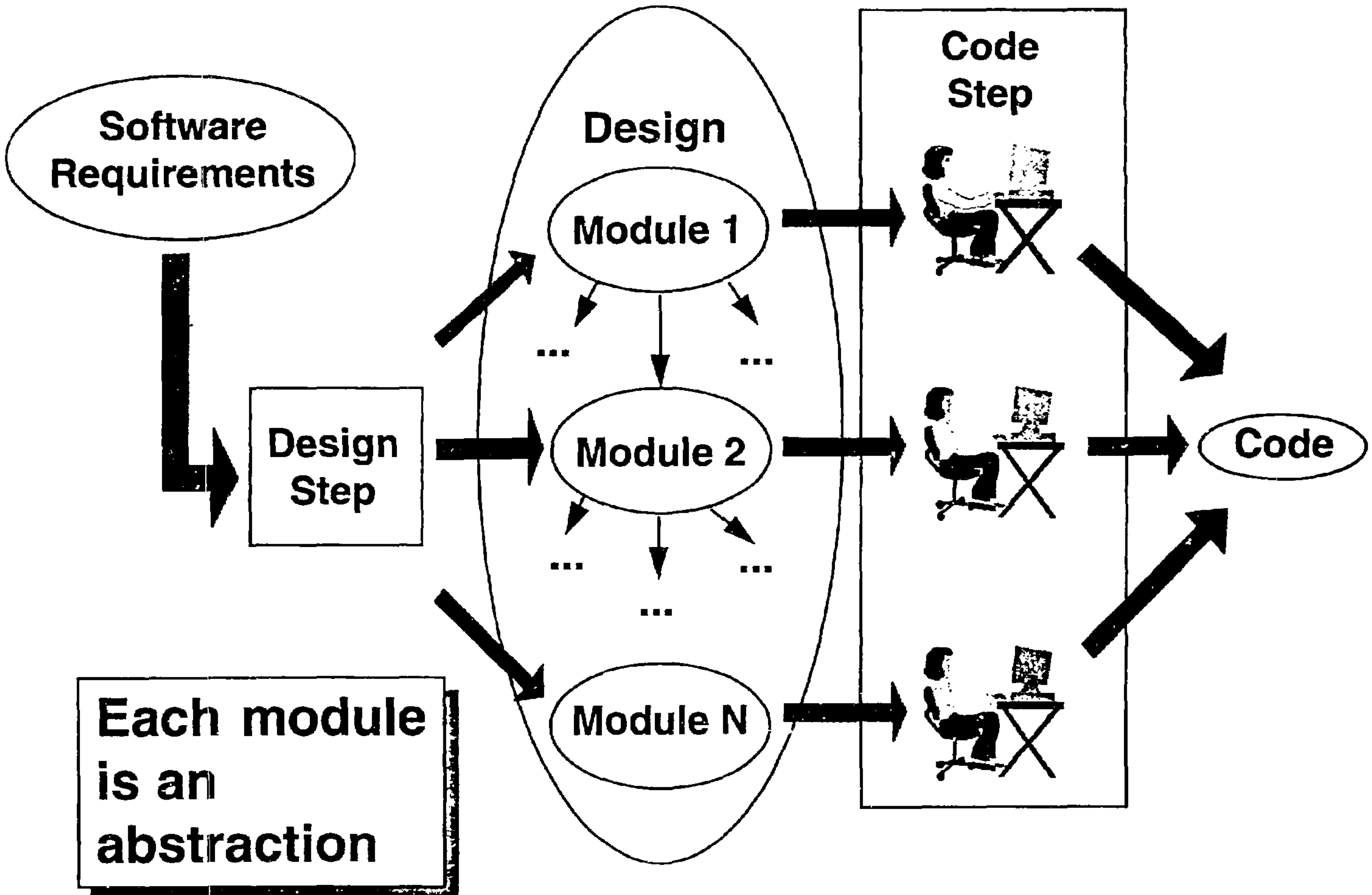
- What types of structures do you think exist in software? (The calling chain among procedures is one example. Another is which Ada packages use which, as determined by **with** clauses.)
- Try describing your house without discussing its rooms or structure. (This shows the difficulty of describing something without a good design.)

OBJECTIVES

Students should be able to:

- State the products that constitute a design
- Describe where design fits into software development

Design Means Decomposition



Each module is an abstraction

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide discusses the stepwise refinement design method. Stepwise refinement is the first method most students learn.

In this slide, stepwise refinement is used to create a program that solves a problem similar to the one from Unit 2: create a text file named by the variable Output_File_Name whose content is the lines of Input_File_Name in reverse order. (The variables' declarations and values aren't shown; in reality, the program would need to obtain them from somewhere.)

You perform stepwise refinement by making a chain of design decisions. First you decide the algorithms and data structures for your main module—that is, the main program. You design the algorithm as a set of processing steps. Here, you decide that an appropriate solution is to read the input file and store it in a variable called Data, reverse the lines in Data, then write the contents of Data into the output file. (The workbook's summary explains why the stack isn't used in this unit.)

You then "refine" each step into a module that describes an algorithm and data structures to perform the step. For example, you decide you will implement the Read_File step by opening the named file, then using a while loop to read each line into Data. (The notation Data'first means the lower bound of the Data array. The variable I tracks the number of lines read from the file. It's an **in out** parameter of the Read_File procedure. See the workbook Unit Summary for the complete Ada code.)

In a team, everyone might help design the main module. Then each member might be assigned one processing step and be responsible for refining that step into a module—and perhaps refining each processing step of that module into more modules, each refined by other members.

STUDENT INTERACTIONS

Can you think of other algorithms that might be used to implement the Reverse module? (You could push the data into a stack, then pop it back out. The point is to realize you can make many decisions, even in simple programs.)

OBJECTIVE

Students should be able to:

- Explain the actions each module performs

Stepwise Refinement Design

Software Requirements: Read, Reverse, Write

Main Module

```
Data: array(1..1000) of String(1..255);
I: Integer;
Read_File(Data, Input_File_Name, I);
Reverse(Data, I);
Write_File(Data, Output_File_Name, I);
```

Read File Module

```
Open(F, In_File,
        Input_File_Name);
Set_Input(F);
I := Data'first;
while not End_Of_File loop
   Get_Line(Data(I), Length);
   I := I + 1;
end loop;
Close(F);
```

Reverse Module

```
Copy_Of_Data := Data;
for J in Data'first .. I-1 loop
   Data(J) := Copy_Of_Data(I-J);
end loop;
```

Write File Module

```
Create(F, Out_File,
        Output_File_Name);
Set_Output(F);
for J in Data'first .. I-1 loop
   Put_Line(Data(J));
end loop;
Close(F);
```

Refined Modules

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows a stepwise refinement hazard: problems can arise from changing an early decision.

Suppose you change the representation of Data. For instance, suppose it's an array and you want to change it to a linked list. Look at all the places where it's referenced (shown in **boldface**). You will need to change almost all of them, because they all depend on the representation. You potentially have to make a change in every module.

This is particularly problematic in a team. Usually, one person will recognize the need for a change. That person must notify all relevant people, who in turn may need to notify other people, all the way along the chain. Moreover, if you haven't explicitly recorded the decision chain, you may not realize all the people you need to notify! Think of the difficulties this causes a large project with many thousand team members.

Let's see why stepwise refinement tends to yield designs where a change on one place necessitates changes throughout the program.

STUDENT INTERACTIONS

- Can you think of reasons why you might want to change the representation of Data? (One possibility: suppose you wrote your program for an IBM PC, where array sizes are limited by available memory, but decided to run it on a Macintosh where arrays can't be larger than 32,767 bytes? In other words, the 1000-element array of 255-character strings won't fit on a Macintosh.)

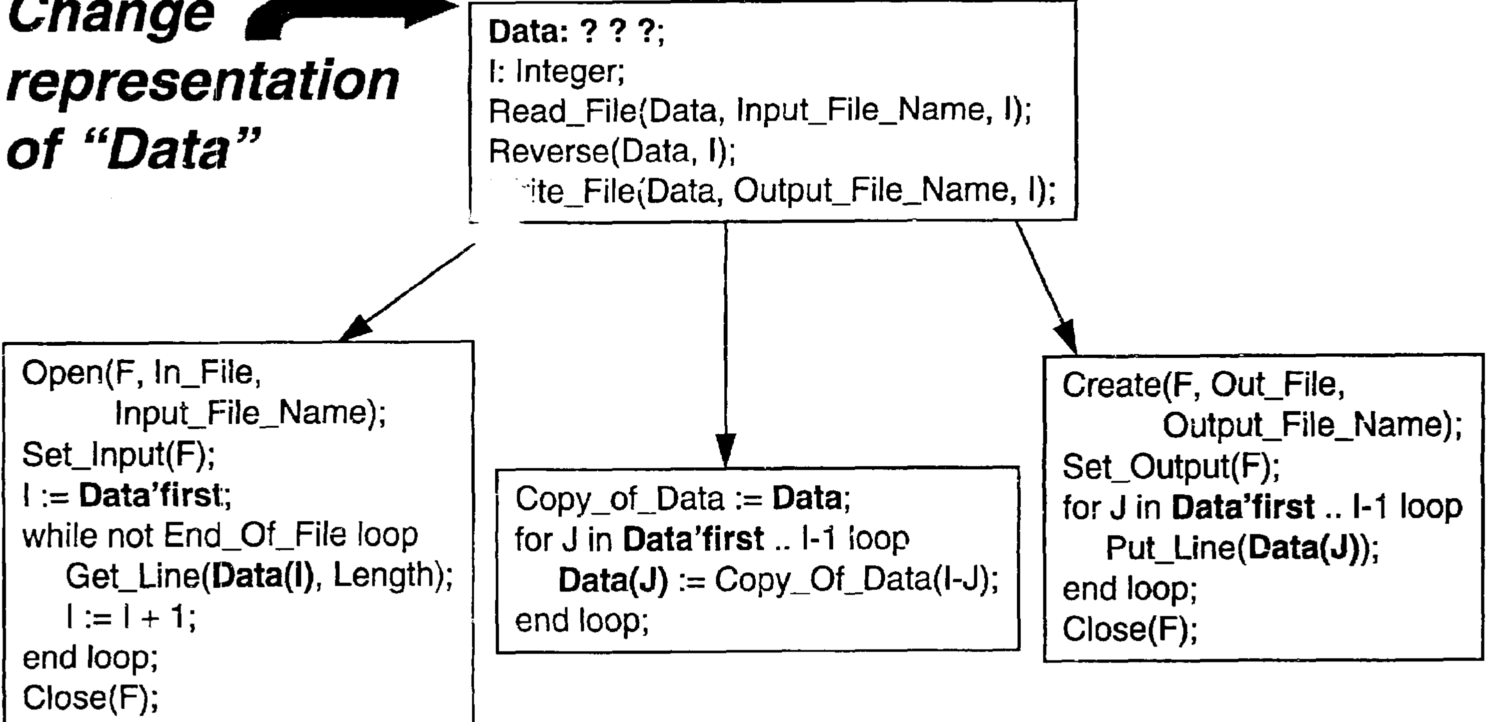
OBJECTIVE

Students should be able to:

- Explain how to determine the modules affected by a change

Hazard of Stepwise Refinement

Change representation of "Data"



The change affects every module!

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows the chain of decisions the designer made when creating the design for this program.

The slide shows both the decisions and the order in which the designer made them. Any time you design a program, you follow a thought process like what this slide depicts.

The representation of Data depends on the algorithm for the main module. If you had decided on an algorithm that pushed each line onto a stack as it was read, you could not have used just an array—you would have needed a stack top variable too. Similarly, the algorithm for reading data depends on the representation of Data. If you study the decision chain, you will find that each decision except the first depends on an earlier (in the picture on this slide, higher) decision.

In general, design decisions depend on previous decisions. The early decisions are, therefore, very important. They influence the overall design more than the late ones. **If you make a mistake in an early decision, you may need to rethink all the subsequent decisions which depend on that decision.** This is why the change to Data on Slide 3-4 causes so many other changes.

STUDENT INTERACTIONS

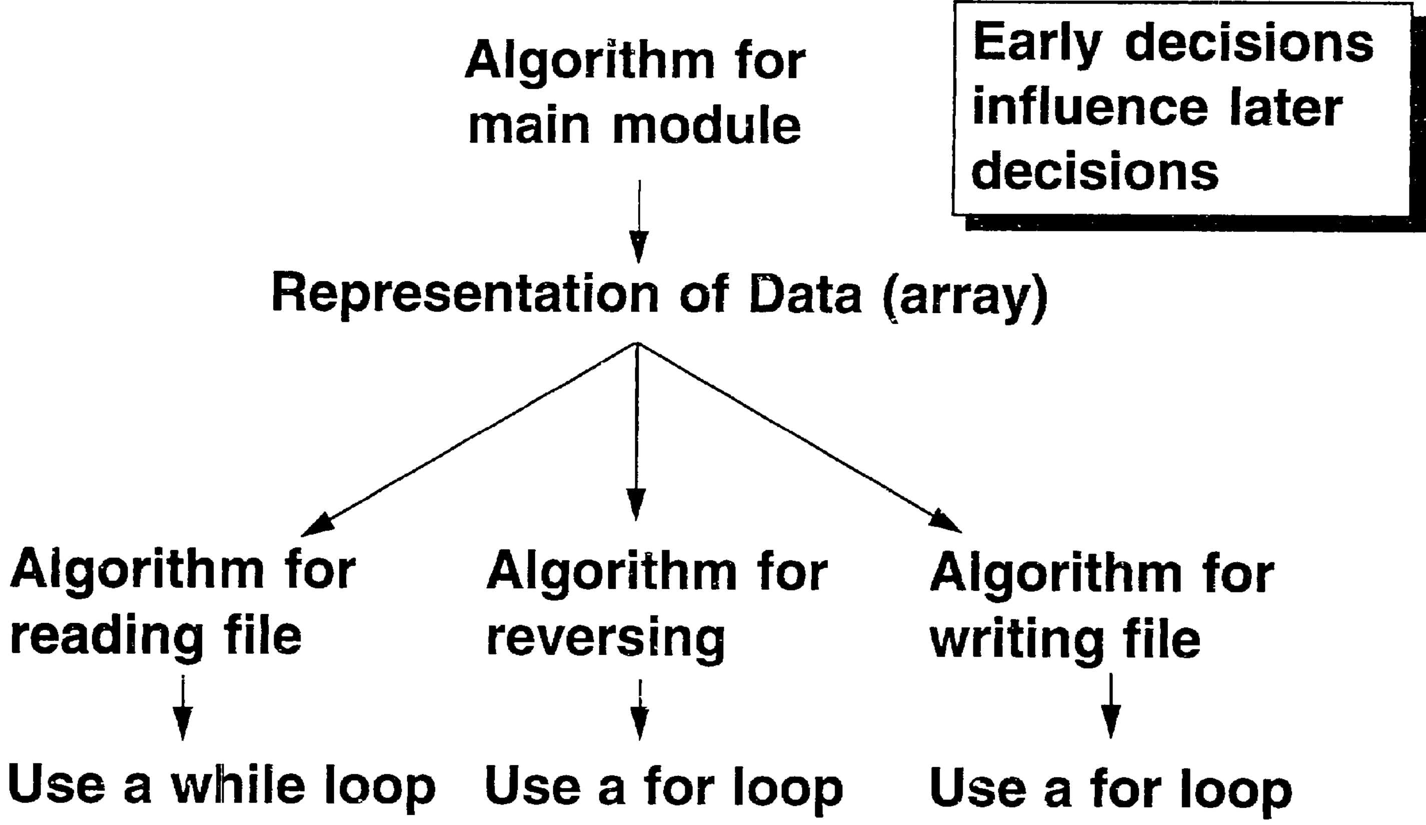
- What is an example of a decision you could change without changing the first two decisions in the chain? (In the Reverse module, you can reverse the data without copying it. You could change that algorithm.)
- Demonstrate that decisions in different branches of the tree are independent. (For instance, the Reverse module's algorithm does not influence the algorithm in the Read_File module. By the time you design that algorithm, you have already decided its result, which is all you need to design the Reverse algorithm.

OBJECTIVE

Students should be able to:

- Explain why early decisions have farther-reaching consequences than late ones

The Design Decision Chain



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide is a view of *Slide 3-3 showing only* the modules and which call which. This is a common abstraction for understanding program structure.

This slide shows the program as a set of modules, *i.e.*, work assignments. In a 4-person team, for example, the manager would assign each person one module. Each person would then independently write and test their assigned module. The team would then integrate all the modules to form the *complete program*.

This slide also shows how stepwise refinement emphasizes actions, not data. The decomposition shows the functions each module performs and shows which module calls which; but it says nothing about the data on which the modules operate. Moreover, it doesn't associate the data with any one module. Each module is *responsible for performing a specific set of actions*. However, no single module is responsible for the data. We have decomposed the problem such that control is allocated across modules, but data is global.

Slide 3-5 and this one have presented two basic problems with the stepwise refinement design of this program. First, the design decision on the representation of Data was made too early. Second, the modules are *designed based on actions the program performs*, and this forces Data to be global. We shall now study the information hiding design method and see how it addresses these problems.

STUDENT INTERACTIONS

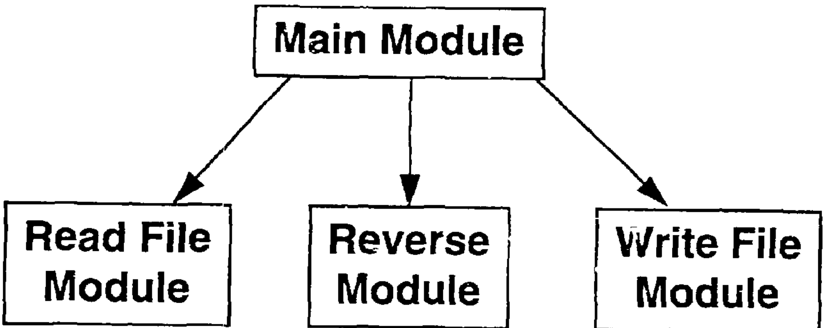
In general, must data be global in stepwise refinement? (No—it can be associated with a single module or with a subtree.)

OBJECTIVES

Students should be able to:

- Explain how the modules derived from stepwise refinement can be allocated to team members
- Explain how stepwise refinement emphasizes control more tha data

Modules From Stepwise Refinement



Each algorithm and refinement is a module

The main module controls the others (structure information)

Where are the data structures?

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows another decomposition, created using the **information hiding** design method.

This structure is similar to the one we created using stepwise refinement. (Again, an arrow means the module at the tail calls the module at the head.) However, we'll add a module called Line Holder. We'll design this module using the abstraction principles discussed in Unit 2. We'll determine its essential information and design an Ada package specification describing that information, as we did for the stack on Slide 2-8. We'll place the irrelevant details on representing a list of lines in the package body.

We'll make sure Line Holder is the only module where decisions are made about how to represent a list of lines. That way, if we change the representation, we'll only need to change this module. No other module will depend on the representation of lists of lines. This means making a change is much less work than in stepwise refinement.

This offers many benefits to a software development team. The person building Line Holder can experiment with different representations without interrupting the work of the people writing the Read File, Reverse, or Write File modules. Compare this to stepwise refinement and you'll see that the need for communication among team members is greatly reduced.

Let's now study the decision chain we followed to create this structure.

STUDENT INTERACTIONS

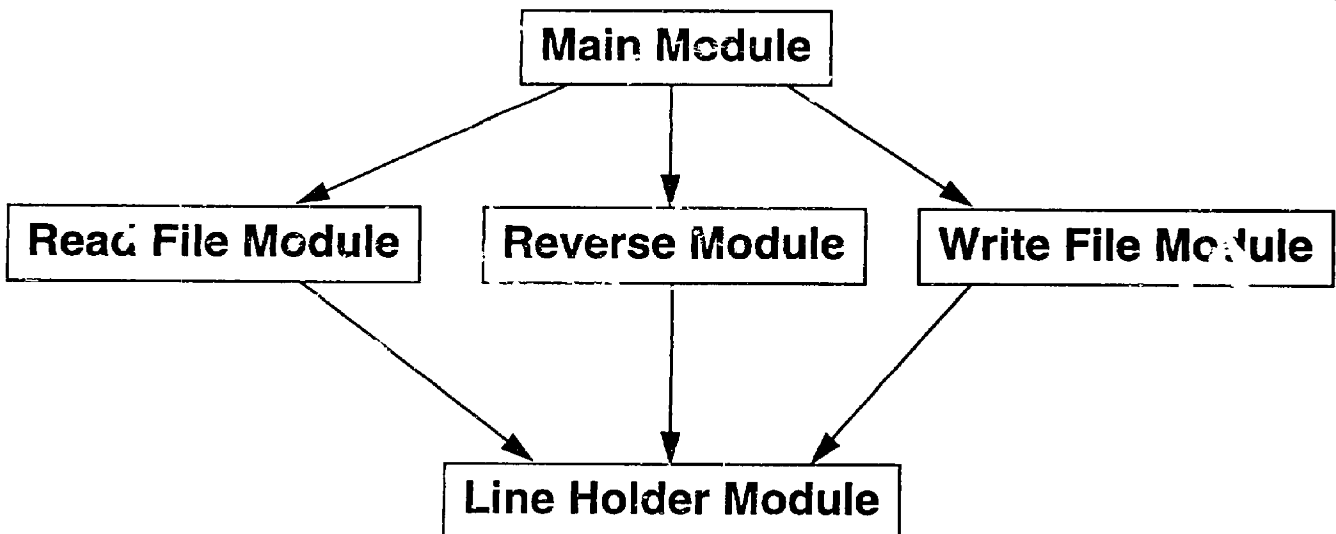
The main module has no arrow to the Line Holder module. What does this imply? (The main module does not invoke the Line Holder. Instead, it only invokes modules that invoke the Line Holder. See Slide 3-12 for details.)

OBJECTIVES

Students should be able to:

- Understand why module independence is important and why a design method should achieve it
- Explain the goal of this decomposition versus the one created using stepwise refinement

A Better Set of Modules



Let's use abstraction principles to design a Line Holder module

Only Line Holder knows how a list of lines is represented

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows the decision chain for information hiding.

In stepwise refinement, you start by thinking of algorithms and data structures for the main module (see Slide 3-5). In information hiding, you begin by thinking about the problem and how to break it up into a set of modules. You use abstraction principles to create two sets of modules. One deals directly with the requirements (read, reverse, and write). The other supports the first set. This way:

- If the requirements change, you only need to change modules in the first set.
- If you change your implementation strategy, you only need to change modules in the second set.

Stepwise refinement, by contrast, generally forces you to change many of your modules in either case.

Late decisions still depend on early ones However, the early decisions concern structure, not algorithms or data representations. Experience shows that program structure changes less than algorithms or data representations. Early decisions are thus less likely to change in information hiding than in stepwise refinement. In practice, then, only a few decisions will have global effects if changed.

This is very important for dealing with change. Designers of large programs, which change frequently, must create designs that accommodate change. Information hiding helps them do so.

STUDENT INTERACTIONS

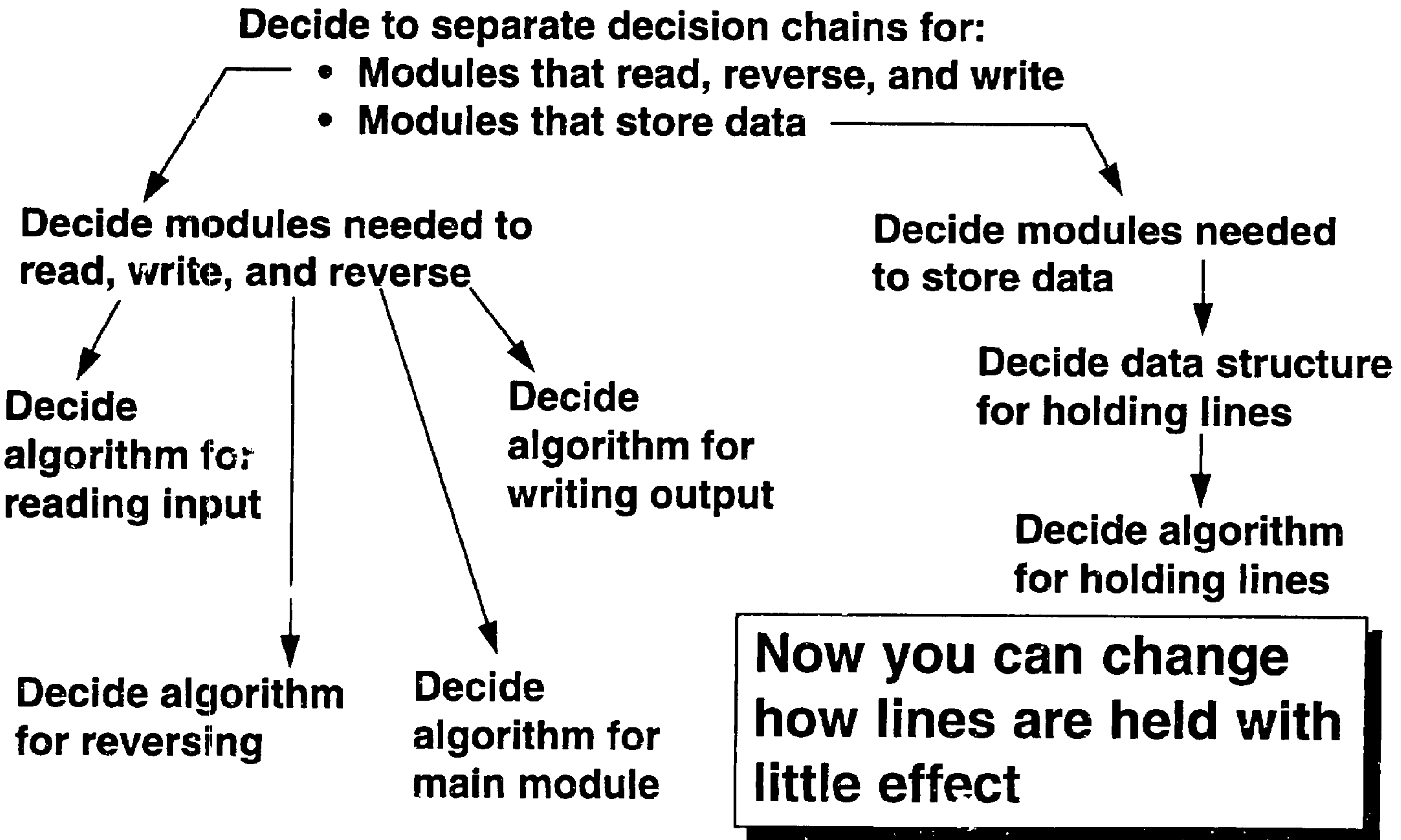
- What decisions change if the data structure changes? (Data structure and algorithm for holding lines.)
- Suppose you change the requirements: the program is to write the lines in the order in which they appear in the input file, but reversed (e.g., abcd as dcba). What decision changes? (The algorithm for reversing.)

OBJECTIVE

The students should be able to:

- Explain the first decisions you make in information hiding

A Better Decision Chain



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide presents principles of the information hiding design method. Information hiding yields software that is easier to change than other design methods. The principles on this slide were used to create the decision chain on Slide 3-8.

In information hiding, you break up your software into a set of modules. Each hides some set of design decisions. This is the module's **hidden information**. A module also has an **interface**. The interface is the portion of the module that other modules may reference. This separation of interface and hidden information comes from the abstraction principles covered in Unit 2. Notice, however, that whereas abstraction tells how to view a single module, information hiding tells how to break up a problem into modules. You can't use abstraction alone to design a program.

Information hiding also provides criteria to judge a design's quality. You try to minimize the number of modules that depend on other modules' hidden information. A small number of such modules means the modules in your design are relatively independent. Changes to your program will involve neither great effort nor extensive communication among team members. A design with many modules that violate information hiding principles means you can expect change to be difficult.

The usual things you want to keep as hidden information are data representation, as in our reverser program example, and algorithms used. We've seen why keeping data representation hidden information is important. Similarly, other modules generally have no need to know the algorithms used. For instance, we should be able to change the algorithm in the Reverse module without affecting the main module.

STUDENT INTERACTION

Why is information difficult to hide in stepwise refinement? (Because you must make decisions on representation before you decide on the algorithms that manipulate the representation. This forces you to embed the representation in your algorithms.)

OBJECTIVE

Students should be able to:

- Explain the major principles of information hiding

Information Hiding Principles

- Each module has “hidden information”—something no other module may know
- Each module has an “interface”—defining <u>exactly</u> what other modules may know about it
- A module is badly designed if the decisions in it depend on decisions in other modules
- A module is well designed if it does not know any of the hidden information of other modules—only the things that are in their interface

> Information hiding tells you how to use abstraction during design

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows an interface for the Line Holder module.

The Line Holder holds a list of lines. You need it to provide the same capabilities the array did in the stepwise refinement-derived program. It must let you store lines, access lines stored, and determine the number of lines stored. It must also remove the limitations of an array. This is the essential information of the abstraction. The principles are the same as those used to derive Integer_Stack on Slide 2-8.

A programming language can help you implement a module designed using information hiding if it can show the module's interface to other modules, but prevent them from accessing the hidden information.

You can use Ada packages to do this. You place the interface in the package specification. Here, you declare that the interface consists of two procedures and two functions, plus a data type List_Of_Lines. To use the Line Holder, you declare a variable of type List_Of_Lines and invoke Initialize. You then use Add_Line_To_List to add a line to it. Once you have added lines, you can access the i'th line using the Line_Number function, and you can determine how many lines you have added to it using Number_Of_Lines.

Note that this module is a set of procedures, functions, and data types. This is typical in information hiding. In stepwise refinement, a module is usually a single procedure or function.

List_Of_Lines is a **private** type. Notice that its representation isn't shown. Where does it go?

STUDENT INTERACTIONS

Does the interface imply anything about the representation of a list of lines? (No.)

OBJECTIVE

Students should be able to:

- Explain how a collection of types, procedures, and functions can constitute a module's interface

Interface Example: Line Holder

```
package Line_Holder is
    type List_Of_Lines is private;
    subtype Line is String(1..255);

    procedure Initialize(Lines: out List_of_Lines);

    procedure Add_Line_To_List(
                    Line_To_Add: in Line;
                        Lines: in out List_Of_Lines);

    function Line_Number(Lines: in List_Of_Lines; I: in integer)
        return Line;

    function Number_Of_Lines(Lines: in List_Of_Lines)
        return Integer;
private
    . . .
end Line_Holder;
```

This defines a line holder to other packages

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide continues the example on Slide 3-10. It shows how Ada supports representing a module's hidden information.

You hide two kinds of information: a data type's representation and the algorithms you use.

Ada lets you hide the representation in a package specification's **private** part. This is a portion that is in the package specification but can't be accessed by other packages. This is just enough information for Ada to compile the specification, so teams can work as shown in Slide 2-11. Here, other packages can know List_Of_Lines is the name of the data type for a line holder and can declare variables of that type. However, they can't directly manipulate the underlying array. They must use the procedures and functions in the package specification.

Compare this package to the Integer_Stack package on Slide 2-8, which defines a single stack. The Line_Holder package defines a data type. Other packages can declare variables of this type.

In the package body, you hide information other packages don't need, such as algorithms.

The package body can reference the information in the package specification's private part (for instance, Initialize references the record structure). This is the only place in an Ada program that is allowed to do so.

In a team, each member could develop each module as a package. He or she would show other members only the specification. Other members would have no way of knowing the hidden information of the implementation. This helps prevent team members from accidentally violating information hiding principles.

STUDENT INTERACTIONS

Can you think of other places where private types would be useful? (Just about any common data structure, like a stack or a queue.)

OBJECTIVES

Students should be able to:

- Explain what a package is and how it separates specification from implementation
- Explain how packages and private types can be used to preserve hidden information

The Hidden Information

```
package Line_Holder is
   . . .
private
   Max_Lines: constant Integer := 10000;
   type Lines is array(1..Max_Lines) of Line;
   type List_Of_Lines is record
         Number: 0..Max_Lines;
         Values: Lines;
   end record;
end Line_Holder;
```

The package body hides the irrelevant details from other packages

```
package body Line_Holder is
   procedure Initialize(Lines: out List_Of_Lines) is
   begin
      Lines.Number := 0;
   end Initialize;

   procedure Add_Line_To_List(
                     Line_To_Add: in Line;
                           Lines: in out List_Of_Lines) is
   begin
      Lines.Number := Lines.Number + 1;
      Lines.Values(Lines.Number) := Line_To_Add;
   end Add_Line_To_List;

   function Line_Number(
                     Lines: in List_Of_Lines;
                         I: in integer)
      return Line is
   begin
      return Lines.Values(I);
   end Line_Number;
   . . .
end Line_Holder;
```


Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows four modules of the program and the algorithms that implement them. Moreover, it shows how these modules access the Line Holder module.

The top module is the main module. It calls the other three modules. Its flow of control is analogous to that used in stepwise refinement (as is true for the other three modules). It uses four variables, whose declarations aren't shown: Input_File_Name and Output_File_Name, which are strings, and Input_Lines and Output_Lines, which are of type List_Of_Lines. The main module calls Read_File. Read_File fills Lines_Read with the contents of the named file, using the Initialize and Add_Line_To_List procedures from the interface of the Line Holder module. The main module then calls the Reverse_Lines procedure with these lines. Reverse_Lines iterates over all the original lines; it knows how many there are through the Line Holder's Number_Of_Lines function. It adds the last line to the Reversed_Lines list, then the next to last line, and so on, using the Line_Number function. In this way, Reversed_Lines becomes the reverse of original_lines. Finally, the main module calls Write_File. This procedure opens the output file and writes each line of Lines_To_Write into that file.

STUDENT INTERACTIONS

Why are the parameters of Reverse named Original_Lines and Reversed_Lines, whereas the corresponding variables in the main module are named Input_Lines and Output_Lines? (The purpose of the Reverse module is to reverse lines. It has nothing to do with files. The main module, though, knows that the lines come from and go to files.)

OBJECTIVES

Students should be able to:

- Explain the algorithm used in each module
- Explain the interactions between modules
- Explain how each module makes use of the Line Holder module

Using the Line Holder

Main Module

```
Read_File(Input_File_Name, Input_Lines);
Reverse_Lines(Input_Lines, Output_Lines);
Write_File(Output_File_Name, Output_Lines);
```

Read_File(Input_File_Name, Lines_Read)

```
Open(F, In_File, Input_File_Name);
Set_Input(F);
Initialize(Lines_Read);
while not End_Of_File loop
   Get_Line(L, Length);
   Add_Line_To_List(L, Lines_Read);
end loop;
Close(F);
```

Write_File(Output_File_Name, Lines_To_Write)

```
Open(F, Out_File, Output_File_Name);
Set_Output(F);
for I in 1..Number_Of_Lines(Lines_To_Write) loop
   Put_Line(Line_Number(Lines_To_Write, I));
end loop;
```

Reverse_Lines(Original_Lines, Reversed_Lines)

```
Initialize(Reversed_Lines);
for I in reverse 1..Number_Of_Lines(Original_Lines) loop
   Add_Line_To_List(Line_Number(Original_Lines, I), Reversed_Lines);
end loop;
```

None of these modules relies on the representation of the lines

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows how information hiding facilitates change by switching implementations.

We have said information hiding helps in groups because it yields designs where modules do not depend on implementation decisions. When you designed the program, you created the Line Holder abstraction, with a clearly defined interface. In this slide, you replace the Line Holder module's package body. The interface stays exactly the same, but now in the private part, List_Of_Lines is a pointer to a record that forms one element of a linked list. Similarly, in the package body, the Initialize procedure now uses the usual algorithm for creating an empty linked list.

Because the interface stays the same, modules that use Line Holder do not need to change. In this way, you achieve your goals of independence among team members and reduce the likelihood that change will adversely affect you. Look at Slide 2-11; developers could work with the compiled package specification while the person responsible for Line_Holder experiments with various implementations to see which one is most efficient.

STUDENT INTERACTIONS

- Examine Slide 3-8. What decision(s) changed? (The data structure for holding lines and the algorithm for holding lines.)
- Which of the modules on Slide 3-12 need to change? (None!)

OBJECTIVE

Students should be able to:

- Explain why the new implementation does not require any other modules in the program to change

Another Implementation

```
package Line_Holder is
    type List_Of_Lines is private;
    subtype Line is String(1..255);

    procedure Initialize(Lines: out List_Of_Lines);
    ...
private
    type Line_List_Element;
    type List_Of_Lines is access Line_List_Element;
    type Line_List_Element is record
        Value: Line;
        Next: List_Of_Lines;
    end record;
end Line_Holder;

package body Line_Holder is
    procedure initialize(Lines: out List_Of_Lines) is
    begin
        Lines := null;
    end Initialize;
    . . .
end Line_Holder;
```

The interface stays the same

The interface's private portion uses a linked list

The implementation uses new algorithms

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide summarizes the topics covered in Unit 3.

In software design, you break up a problem into a set of modules that, working together, solve the problem. You should do this to help you manage the complexity of the solution. Think of how hard it would be to write programs if you couldn't use procedures or functions.

As you break up your problem, you must consider whether you have done a good or bad job. Design methods provide criteria that help you assess the quality of your design. Stepwise refinement is one such method. Information hiding is another.

Stepwise refinement is best for individuals designing small programs. Information hiding, which isolates the portions of a program likely to need modification in response to some change, is better suited to team projects and large programs. It's also good for anyone who expects their programs to change—and everyone should expect this!

Ada's package mechanism helps you implement modules designed using information hiding. A package clearly separates the interface from the hidden information. You can prevent other packages from accidentally accessing information that is intended to be hidden.

STUDENT INTERACTIONS

Is information hiding or stepwise refinement a better description of the design of an automobile? (There isn't really a right or wrong answer, but it's fun to try both methods and see what happens.)

OBJECTIVES

Students should be able to:

- Explain the relative advantages of stepwise refinement and information hiding
- Explain the importance of a design method

Summary

- Design requires creating a solution to a problem as a set of modules
- Some module decompositions are more adversely affected by change than others
- Especially in a team setting (but also for an individual), using information hiding reduces the:
  - Need to communicate
  - Effort to make changes
- Ada packages provide a simple, effective way to implement information hiding principles

Copyright © 1995, Software Productivity Consortium.

UNIT 3: INFORMATION HIDING

UNIT SUMMARY

The first step of developing software from requirements is to create a design. In design, you break a problem into a set of modules and a structure. The modules, working together in accordance with this structure, form a solution to the problem. Breaking a problem into modules is important for two reasons. First, it helps you deal with the complexity of software because you can grasp the workings of a small module more easily than you can grasp the workings of an entire program. Second, if you are part of a team, it lets you allocate work assignments among the team members.

Stepwise refinement is a popular design method. In stepwise refinement, you break up a problem by creating an algorithm to solve that problem. At first, you keep each of the steps of the algorithm abstract. If you cannot express a step as a single instruction in a programming language, you refine it into another algorithm. You do this for each step and for each step of each algorithm you create. You keep refining until you can express every step as a single computer instruction.

At each refinement, you are making a decision. Once you have made the decision, subsequent decisions will depend on that decision. You can think of design as a chain of decisions.

As an example, let's consider the problem from Unit 2 of reading, reversing, and writing. In this unit, we shall change the example slightly:

- The program will read and reverse a set of lines, not integers. The reason for this change will become clear in Unit 4.
- A stack is not used to solve the problem. The implementation chosen here more clearly compares and contrasts stepwise refinement and information hiding.

You begin stepwise refinement by creating a main module. This module describes an algorithm that will solve the problem and the data structures used in the algorithm. You do not state the algorithm exactly, but rather use procedures and functions to describe large processing steps. For example, the main module for the Reverser program is:

```
procedure Reverse_File is
   type List_Of_Lines is array(1..1000) of String(1..255);
   Data: List_Of_Lines;
   Input_File_Name: constant String := "input.txt";
   Output_File_Name: constant String := "output.txt";
   I: Integer;
begin
   Read_File(Input_File_Name, Data, I);
   Reverse_Lines(Data, I);
   Write_File(Output_File_Name, Data, I);
end Reverse_File;
```

This describes a straightforward algorithm: read a file's contents into a variable named `Data`, reverse the contents of `Data`, and write `Data` to another file.

You next decompose each procedure. You might decompose `Read_File` as follows:*

```
procedure Read_File(Input_File_Name:in String;
                                    Data: out List_Of_Lines;
                                       I: in out Integer) is
   F: file_type;
   Length. Integer;
begin
   Open(F, In_File, Input_File_Name);
   Set_Input(F);
   I := Data'first;
   while not End_Of_File loop
      Get_Line(Data(I), Length);
      I := I + 1;
   end loop;
end Read_File;
```

In Ada, a procedure may have parameters that are declared `in`, `out`, and `in out`. The Unit 2 Summary described `in` parameters. An `out` parameter is one whose value may be set but not accessed. A procedure can both set and access an `in out` parameter.

The notation `Data'first` is the lower index bound of the array `Data`. This notation helps you write code that does not need to change if, for some reason, you decide to change the index range of an array.

You follow this strategy until no procedure contains any procedures that need to be decomposed. Here are `Reverse` and `Write_File`:

```
procedure Reverse_Lines(Data:in out List_Of_Lines;
                           I: in integer) is
   Copy_Of_Data: List_Of_Lines;
begin
   Copy_Of_Data := Data;
   for J in Data'first .. I-1 loop
      Data(J) := Copy_Of_Data(I-J);
   end loop;
end Reverse_Lines;

procedure Write_File(Output_File_Name:in String;
                                 Data: in List_Of_Lines;
                                    I: in Integer) is
   F: file_type;
begin
   Create(F, Out_File, Output_File_Name);
   Set_Output(F);
   for J in Data'first .. I-1 loop
      Put_Line(Data(J));
   end loop;
end Write_File;
```

You can see from the above discussion that Figure 1 is the decision chain followed to derive this program using stepwise refinement.

* For simplicity and clarity, the code examples omit details of file input and output.

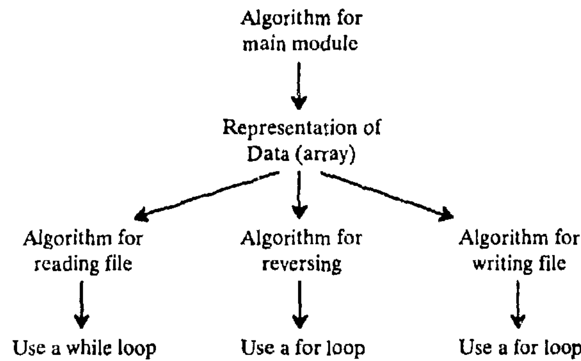


Figure 1. Decision Chain for Stepwise Refinement

Stepwise refinement has a hazard. The decisions made early are often crucial ones and, also, are often subject to change. Because decisions made late in the design depend on decisions made early, changing a decision made early often has global repercussions and necessitates many changes to the software. Such a change is especially troublesome in a team because news of the change must be communicated to all relevant team members. Think of how difficult this must be on a project involving teams scattered across the nation at several companies. You can easily see why change is one of the great contributors to faulty and costly software.

Information hiding is a design method that helps you deal with change. When you perform information hiding, you use the principles of abstraction covered in Unit 2. You describe each module in terms of an **interface** and **hidden information**. The interface defines exactly what other modules can know and use: the essential information. The hidden information states things no other module may assume: the irrelevant details (i.relevant insofar as other modules are concerned). This leads to designs where changes are confined to a few modules.

In stepwise refinement, the early decisions focus on algorithms. In information hiding, they focus more on modules. Figure 2 shows the decision chain for the Reverser program derived using information hiding.

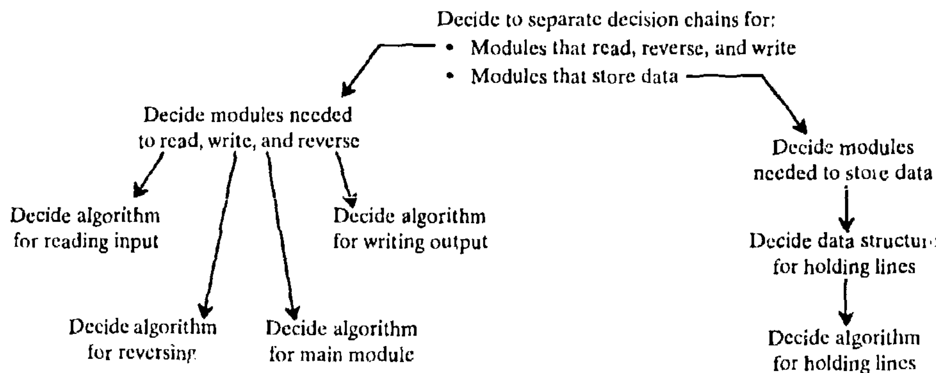


Figure 2. Decision Chain for Information Hiding

Information hiding gives you criteria you can use to judge the quality of your design:

- A module is designed well if it does not know any of the hidden information of other modules.
- A module is designed badly if the decisions in it depend on decisions in other modules (e.g., if `Reverse_File` assumes stack is implemented using an array).
- Your objective during design is to minimize the number of badly designed modules.

The ability to judge a design's quality is an important part of sound engineering practice.

Once you have created your design, you must implement it. The Ada programming language has features that help you hide information as you write modules:

- You can use a **package** to implement a module defined by information hiding. You can use the **package specification** to show the interface. You can use the **package body** to implement the hidden information. The rules of Ada allow other packages to access any information in a package specification, but do not allow other modules to access information in the package's body.
- You can use **private types** to ensure that data type representations are hidden information. Private types declare a data type name that other packages may access, but they forbid other packages from making any assumptions about the representation of the data.

Here is the Line Holder module implemented as an Ada package. First, the package specification:

```
package Line_Holder is
   type List_Of_Lines is private;
   subtype Line is String(1..255);

   procedure Initialize(Lines: out List_Of_Lines);

   procedure Add_Line_To_List(Line_To_Add: in Line;
                              Lines: in out List_Of_Lines);

   function Line_Number(Lines: in List_Of_Lines; I: in Integer)
      return Line;

   function Number_Of_Lines(Lines: in List_Of_Lines) return Integer;

private
   Max_Lines: constant Integer := 10000;
   type Lines is array(1..Max_Lines) of Line;
   type List_Of_Lines is record
      Number: Integer range 0..Max_Lines;
      Values: Lines;
   end record;
end Line_Holder;
```

The type `List_Of_Lines` is declared private. This means other packages may reference its name and declare variables of the type, but they cannot reference its representation. This forces them to manipulate variables of type `List_Of_Lines` through the procedures and functions provided in the

package specification, the only ones permitted to access the representation. In this way, the designer of the `Line_Holder` package controls what is essential information and what is hidden.

The package's **private part** contains the representation of `List_Of_Lines`. The private part is that part following the reserved word **`private`** up to the end of the package specification. The private part declares a list of lines to be a record containing an array of lines and an integer variable that can store how much of the array is in use.

Next, the package body:

```
package body Line_Holder is
   procedure Initialize(Lines: out List_Of_Lines) is
   begin
      Lines.Number := 0;
   end Initialize;

   procedure Add_Line_To_List(Line_To_Add:in Line;
                                      Lines: in out List_Of_Lines) is
   begin
      Lines.Number := Lines.Number + 1;
      Lines.Values(Lines.Number) := Line_To_Add;
   end Add_Line_To_List;

   function Line_Number(Lines: in List_Of_Lines; I: in integer)
      return Line is
   begin
      return Lines.Values(I);
   end Line_Number;

   function Number_Of_Lines(Lines: in List_Of_Lines) return Integer is
   begin
      return Lines.Number;
   end Number_Of_Lines;

end Line_Holder;
```

The interface to the `Line_Holder` package gives you enough information and functionality to write the other modules in the program. The main module uses it only to declare variables of type `List_Of_Lines`. Unlike the stepwise refinement version, it has no knowledge of how a list of lines is represented, so changes to the representation won't affect the main module—or any other module. Here is the main module:

```
with Line_Holder, Read_File, Write_File, Reverse_Lines;
procedure Reverse_File is
   Input_File_Name: constant String := "input.txt";
   Output_File_Name: constant String := "output.txt";
   Input_Lines, Output_Lines: Line_Holder.List_Of_Lines;
begin
   Read_File(Input_File_Name, Input_Lines);
   Reverse_Lines(Input_Lines, Output_Lines);
   Write_File(Output_File_Name, Output_Lines);
end Reverse_File;
```

Notice the use of the **with** clause, explained in Unit 2, to allow this procedure to reference the interfaces of the `Line_Holder` package interface and the procedures it calls.

The other modules can also be implemented as Ada procedures. For example, the Reverse module can be implemented as follows:

```
with Line_Holder;
procedure Reverse_Lines(Original_Lines:in Line_Holder.List_Of_Lines;
                        Reversed_Lines: in out Line_Holder.List_Of_Lines)
is
begin
   Line_Holder.Initialize(Reversed_Lines);
   for I in reverse 1..Line_Holder.Number_Of_Lines(Original_Lines) loop
      Line_Holder.Add_Line_To_List(
         Line_Holder.Line_Number(Original_Lines, I), Reversed_Lines);
   end loop;
end Reverse_Lines;
```

Compare this to the stepwise refinement version. Notice how it uses the same algorithm but does not depend on how a list of lines is represented. This is especially valuable in a team, where developers need freedom to experiment with different implementation strategies and cannot risk disturbing other developers. Since other developers have made decisions based only on the interface, and since implementation strategies are hidden information, the information hiding method grants developers this freedom.

Information hiding is a good design method for individuals too. You may have already encountered a situation where a change you thought would affect only one part of your program required much more work than you thought. Information hiding helps you avoid this.

UNIT 3: INFORMATION HIDING

GROUP ACTIVITY

HIDING INFORMATION

The implementation of the vending machine software was developed using information hiding and includes a module **Change Calculation**. The purpose of this module is to calculate the change required from the purchase of some item in the machine. The interface of this module is as follows:

```
package Change_Calculation is
   type Coin_Value_And_Number is record
      Value: Positive;
      Number: Positive;
   end record;

   Number_Of_Coins_Used_In_Dispensing_Change: constant Integer := 3;

   type Coin_Money is array(1..Number_Of_Coins_Used_In_Dispensing_Change)
                  of Coin_Value_And_Number;

   procedure Calculate_Change(Price:in Positive;
                  Money_Received: in positive;
                        Change: in out Coin_Money);
end Change_Calculation;
```

Procedure `Calculate_Change`, given a price for an item in the vending machine and an amount of money received (both in cents), returns in `Change` coinage that makes up the difference. For example, if the price is 60 cents and the money received is 75 cents, the contents of the array `Change` will be:

	Value	Number
Change(1)	25	0
Change(2)	10	1
Change(3)	5	1

That is, the change is 0 quarters, 1 dime, and 1 nickel.

The information hidden in this module is the algorithm used to calculate how many coins of each type to dispense.

Here is the implementation of the module's hidden information, that is, the package's body:

```
package body Change_Calculation is

   procedure Calculate_Change(Price:in Positive;
                  Money_Received: in positive;
                        Change. in out Coin_Money) is
      Change_To_Dispense: Natural;
   begin
```

```
      Change(1).Value := 25;
      Change(2).Value := 10;
      Change(3).Value := 5;

      Change_To_Dispense := Money_Received - Price;
      Change(1).Number := Change_To_Dispense/Change(1).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(1).Value;
      Change(2).Number := Change_To_Dispense/Change(2).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(2).Value;
      Change(3).Number := Change_To_Dispense/Change(3).Value;
   end Calculate_Change;
end Change_Calculation;
```

Consider the following problems:

1. Identify the hidden information in the `Change_Calculation` module.

2. Can you think of another algorithm to implement `Calculate_Change`?

3. You are to build a new version of the vending machine. This version will be sold in Germany. The German monetary system differs from that of the United States. It is based on the Deutsche Mark (DM). There are 100 pfennigs in a DM. Germany has 1 pfennig, 5 pfennig, 10 pfennig, 50 pfennig, 1 DM, 2 DM, and 5 DM coins. German vending machines don't dispense as change paper money or coins less than 10 pfennigs.

   German vending machines dispense drinks, but dispensing food hasn't caught on in Germany or most other European countries. Drinks cost anywhere from 50 pfennigs to 1.2 DM.

   Create a new version of the `Change_Calculation` module that calculates change for a machine that receives and dispenses German money. Make as few changes to the interface as you can.

4. Why is it important that you change the interface as little as possible?

HOMEWORK

1. Adapt the change calculation module of the vending machine for use in another country.

2. A rational number is a number that can be expressed as the ratio of two integers. Use information hiding to design and implement a program that reads two rational numbers, adds them, and prints the result:

   a. Decompose the problem into a set of modules. For each module, state its interface and its hidden information. Describe the interface as an Ada package specification; describe the hidden information in English.

   b. Implement each module. Represent a rational number as a pair of integers.

   c. Change the implementation of rational numbers to a single floating-point value. What are each implementation's relative advantages and disadvantages? In what programs would you use one or the other?

TEACHER NOTES FOR GROUP ACTIVITY

The package `Change_Calculation` *presented here illustrates how you can use the programming language Ada to implement modules designed using the information hiding design method. The package specification is the module's interface. It provides a single procedure that another module may use to determine what coins, and how many of them, must be dispensed to provide a person with change for their purchase. The package also has several data and type definitions. This is usual in information hiding: developers will accompany the procedures and functions with data types that support their use.*

1. Identify the hidden information in the `Change_Calculation` module.

   The algorithm is the hidden information, so it's the implementation of calculate_change.

2. Can you think of another algorithm to implement `Calculate_Change`?

   Here's one. Though slightly more complex than the first algorithm, it's actually easier to change when you do activity 3. The reason is that the original algorithm made a design assumption that change was always dispensed using three coins.

```
procedure Calculate_Change(Price:in Positive;
                 Money_Received: in positive;
                         Change: in out Coin_Money) is
   Change_To_Dispense: Natural;
begin
   Change(1).Value := 25;
   Change(2).Value := 10;
   Change(3).Value := 5;

   Change_To_Dispense := Money_Received - Price;
   for C in Change'range loop
      Change(C).number := Change_To_Dispense / Change(C).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(C).Value;
   end loop;
end Calculate_Change;
```

   In Ada, you use the notation `Change'range` *in a for loop to index each value in the array* `Change`*. In this case, the loop index variable* `C` *assumes the values 1, 2, and 3, in that order.*

3. You are to build a new version of the vending machine. This version will be sold in Germany. The German monetary system differs from the United States'. It is based on the Deutsche Mark (DM). There are 100 pfennigs in a DM. Germany has 1 pfennig, 5 pfennig, 10 pfennig, 50 pfennig, 1 DM, 2 DM, and 5 DM coins. German vending machines don't dispense as change paper money or coins less than 10 pfennigs.

   German vending machines dispense drinks, but dispensing food hasn't caught on in Germany or most other European countries. Drinks cost anywhere from 50 pfennigs to 1.2 DM.

   Create a new algorithm that calculates change for a machine that receives and dispenses German money. Make as few changes to the interface as you can.

Here is one answer. The lines in italics are the lines that differ from the original. The algorithm now calculates change for a machine that dispenses coins from 10 pfennigs to 2 DM. The interface had to be changed slightly to support this, since the U.S. version assumed that 3 types of coins (quarters, dimes, and nickels) were dispensed, whereas the German version dispenses 4 types (10 pfennig, 50 pfennig, 1 DM, and 2 DM).

```
package Change_Calculation is
   type Coin_Value_And_Number is record
      Value: Positive;
      Number: Positive;
   end record;

   Number_Of_Coins_Used_In_Dispensing_Change: constant Integer := 4;

   type Coin_Money is
array(1..Number_Of_Coins_Used_In_Dispensing_Change)
                     of Coin_Value_And_Number;

   procedure Calculate_Change(Price:in Positive;
                     Money_Received: in positive;
                              Change: in out Coin_Money);
end Change_Calculation;

package body Change_Calculation is

   procedure Calculate_Change(Price:in Positive;
                     Money_Received: in positive;
                              Change: in out Coin_Money) is
      Change_To_Dispense: Natural;
   begin
      Change(1).Value := 200;
      Change(2).Value := 100;
      Change(3).Value := 50;
      Change(4).Value := 10;

      Change_To_Dispense := Money_Received - Price;
      Change(1).Number := Change_To_Dispense/Change(1).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(1).Value;
      Change(2).Number := Change_To_Dispense/Change(2).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(2).Value;
      Change(3).Number := Change_To_Dispense/Change(3).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(3).Value;
      Change(4).Number := Change_To_Dispense/Change(4).Value;
   end Calculate_Change;
end Change_Calculation;
```

- Why is it important that you change the interface as little as possible?

Suppose you are part of a team and are writing a module that uses this module. You will have made some design decisions based on the interface you expect. If that interface changes, you may have to rethink your decisions.

You will often find you cannot avoid making any *changes to a module's interface. However, you can plan ahead when you design a module by thinking of the things that are likely to change. This is part of what information hiding is all about. The things you think are most likely to change are the things you hide behind an interface. You may have to make certain parts of the interface susceptible to change. Here, the number of coins in the monetary system had to change. However, by making that value a constant, you can let other modules rely on the constant rather than on a literal. In this way you can lower the likelihood of inadvertent effects.*

This page intentionally left blank.

TEACHER NOTES FOR HOMEWORK

Students may do the homework problems in any programming language, although using Ada will help them separate interface from implementation. If they use Pascal, encourage them to use units (if your compiler supports them.)

1. Adapt the change calculation module of the vending machine for use in another country.

   In France, the monetary system is based on the French Franc. The coins are 1, 5, 10, 20, and 50 centimes, and 1, 2, and 5 Francs; 100 centimes equals 1 Franc. French vending machines dispense coins from 20 centimes to 2 Francs, inclusive. Products cost between 1.2 and 8 Francs.

```
package Change_Calculation is
   ...
   Number_Of_Coins_To_Dispense: constant Integer := 4;
   ...
end Change_Calculation;

package body Change_Calculation is

   procedure Calculate_Change(Price:in Positive;
                  Money_Received: in positive;
                          Change: in out Coin_Money) is
      Change_To_Dispense: Natural;
   begin
      Change(1).Value := 200;
      Change(2).Value := 100;
      Change(3).Value := 50;
      Change(4).Value := 20;

      Change_To_Dispense := Money_Received - Price;
      Change(1).Number  := Change_To_Dispense/Change(1).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(1).Value;
      Change(2).Number  := Change_To_Dispense/Change(2).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(2).Value;
      Change(3).Number  := Change_To_Dispense/Change(3).Value;
      Change_To_Dispense := Change_To_Dispense mod Change(3).Value;
      Change(4).Number  := Change_To_Dispense/Change(4).Value;
   end Calculate_Change;
end Change_Calculation;
```

2. A rational number is a number that can be expressed as the ratio of two integers. Use information hiding to design and implement a program that reads two rational numbers, adds them, and prints the result:

   a. Decompose the problem into a set of modules. For each module, state its interface and its hidden information. Describe the interface as an Ada package specification; describe the hidden information in English.

      It's important that the students try to design the modules before plunging into the implementation. They should try to come up with the answer to this question before

tackling Part b. However, it's also okay if they don't get the correct answer on the first try. Engineering design is an iterative activity.

There are two modules, Rational_Number and Main. Rational_Number has the following interface:

```
package Rational_Number is
   type Rational is private;
   function Rational_Number(Numerator: in Integer;
                            Denominator: in Integer)
      return Rational;
   function Add(r1, r2: Rational) return Rational;
   function Numerator(r: Rational) return Integer;
   function Denominator(r: Rational) return Integer;
private
   type Rational is record
      Numerator: Integer;
      Denominator: Integer;
   end record;
end Rational_Number;
```

The hidden information of this module is the representation of rational numbers and the algorithms used by the functions that manipulate them.

The main module has the following interface:

```
procedure Read_Sum_And_Print_Rational_Numbers;
```

The hidden information of this module is the algorithm it uses.

b. Implement each module. Represent a rational number as a pair of integers.

The following implementation takes the trouble to normalize rational numbers after each operation—that is, it divides the numerator and denominator by their greatest common divisor. If your students have not yet encountered this algorithm, you may want to provide it for them.

```
package body Rational_Number is

   -- This package implements rational numbers as pairs of integers.
   -- The advantage to this scheme is that rational numbers are
   -- represented exactly.  The disadvantage is that numbers must
   -- be normalized after creation and each arithmetic operation,
   -- requiring some extra time.

   procedure Normalize(r: in out Rational) is
      Numerator, Denominator: Natural;
      Remainder: Natural;
   begin
      Numerator := abs(r.Numerator);        -- Find the greatest common
      Denominator := abs(r.Denominator);    -- divisor of the numerator
      while Denominator /= 0 loop           -- and denominator using
         Remainder := Numerator rem Denominator;    -- Euclid's
         Numerator := Denominator;          -- algorithm. The algorithm
         Denominator := Remainder;          -- ends with numerator
      end loop;                             -- holding the GCD.

      r.Numerator := r.Numerator / Numerator;
      r.Denominator := r.Denominator / Numerator;
   end Normalize;

   function Rational_Number(Numerator: in Integer;
                            Denominator: in Integer)
      return Rational is
      r: Rational;
   begin
      r.Numerator := Numerator;
      r.Denominator := Denominator;
      Normalize(r);
      return r;
   end Rational_number;

   function Add(r1, r2: Rational) return Rational is
      r: Rational;
   begin
      r.Numerator := r1.Numerator*r2.Denominator
                   + r2.Numerator*r1.Denominator;
      r.Denominator := r1.Denominator * r2.Denominator;
      Normalize(r);
      return r;
   end Add;

   function Numerator(r: Rational) return Integer is
   begin
      return r.Numerator;
   end Numerator;

   function Denominator(r: Rational) return Integer is
   begin
      return r.Denominator;
   end Denominator;
end Rational_Number;
```

This implementation of the main module goes through the rigamarole necessary to instantiate the generic packages needed for integer input and output. That's not important for the purposes of this assignment. Rather, have the students concentrate on the algorithm.

```
with Rational_Number, Text_IO;
procedure Read_Sum_And_Print_Rational_Numbers is
   package Integer_IO is new Text_IO.Integer_IO(Integer);
   r1, r2, Sum: Rational_Number.Rational;
   Numerator, Denominator: Integer;
begin
   Text_IO.put("Enter the first number: ");
   Integer_IO.get(Numerator); Integer_IO.get(Denominator);
   r1 := Rational_Number.Rational_Number(Numerator, Denominator);

   Text_IO.put("Enter the second number: ");
   Integer_IO.get(Numerator); Integer_IO.get(Denominator);
   r2 := Rational_number.Rational_number(Numerator, Denominator);
   Sum := Rational_Number.Add(r1, r2);
   Text_IO.put("The sum is ");
   Integer_IO.put(Rational_Number.Numerator(Sum));
   Text_IO.put('/');
   Integer_IO.put(Rational_Number.Denominator(Sum));
end Read_Sum_And_Print_Rational_Numbers;
```

c. Change the implementation of rational numbers to a single floating-point value. What are each implementation's relative advantages and disadvantages? In what programs would you use one or the other?

The package specification is identical, except that the representation of `Rational` *changes. Replace the lines:*

```
type Rational is record
      Numerator: Integer;
      Denominator: Integer;
   end record;
```

with:

```
type Rational is new float;
```

Here is the package body:

```
package body Rational_Number is

      -- This package implements Rational numbers as floating-point
      -- quantities.  The advantage to this scheme is that it is
      -- very fast for performing arithmetic operations.  The
      -- disadvantage is that the numerator and denominator must be
      -- approximated rather than computed exactly.

   Number_Of_Denominators: constant := 10;

   Denominators: constant array(1..Number_Of_Denominators) of Integer
      := (1, 2, 3, 5, 7, 11, 13, 17, 19, 23);
```

```
function Rational_Number( Numerator: in Integer;
                          Denominator: in Integer)
   return Rational is
begin
   return Rational(Numerator)/Rational(Denominator);
end Rational_Number;

function Add(r1, r2: Rational) return Rational is
begin
   return r1+r2;
end Add;

   -- Compute the numerator and denominator of r.  The algorithm
   -- is to choose for denominator the value in the denominators
   -- array such that numerator = floor(r*denominator) and
   -- numerator/denominator is closer to r than for any other
   -- value of denominator in the array.
procedure Compute_Numerator_And_Denominator_Of_Rational(
   r: in Rational;
   Numerator, Denominator:     Integer) is
   d, n: Integer;
   dprime, nprime: Integer;
begin
   d := Denominators(Denominators'first);
   n := Integer(r*Rational(d));
   for i in Denominators'first+1 .. Denominators'last loop
      dprime := Denominators(i);
      nprime := Integer(r*Rational(dprime));
      if abs(Rational(nprime)/Rational(dprime) - r)
       < abs(Rational(n)/Rational(d) - r) then
         d := dprime;
         n := nprime;
      end if;
   end loop;
   Numerator := n;
   Denominator := d;
end Compute_Numerator_And_Denominator_Of_Rational;

function Numerator(r: Rational) return Integer is
   d, n: Integer;
begin
   Compute_Numerator_And_Denominator_Of_Rational(r, n, d);
   return n;
end Numerator;

function Denominator(r: Rational) return Integer is
   d, n: Integer;
begin
   Compute_Numerator_And_Denominator_Of_Rational(r, n, d);
   return d;
end Denominator;

end Rational_Number;
```

The differences between the two implementations may be summarized as follows:

- *The first implementation is slower. It must execute the normalization operation after creation and every arithmetic operation. Moreover, adding two rational numbers using the first implementation requires three multiplications and one addition, whereas adding them using the second implementation requires only a single addition.*

- *The second implementation is less exact. The first always represents a rational number as the ratio of two integer quantities, which is the definition of a rational number. The second, which uses a floating-point value, represents a fixed number of significant digits. Furthermore, its algorithm for determining the numerator and denominator is accurate only to two significant digits.*

You would use the first implementation in programs where accuracy is of more concern than speed. You would use the second implementation when speed is of more concern than accuracy. You must also assess what operations you will use most frequently. The first implementation will be both faster and more accurate if you invoke the numerator and denominator functions more often than the addition and creation functions.

Unit 4

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This unit discusses software reuse. Software reuse is using previously written software on a new project. More generally, software reuse also entails thinking of how to write software so it will be easy to reuse in the future.

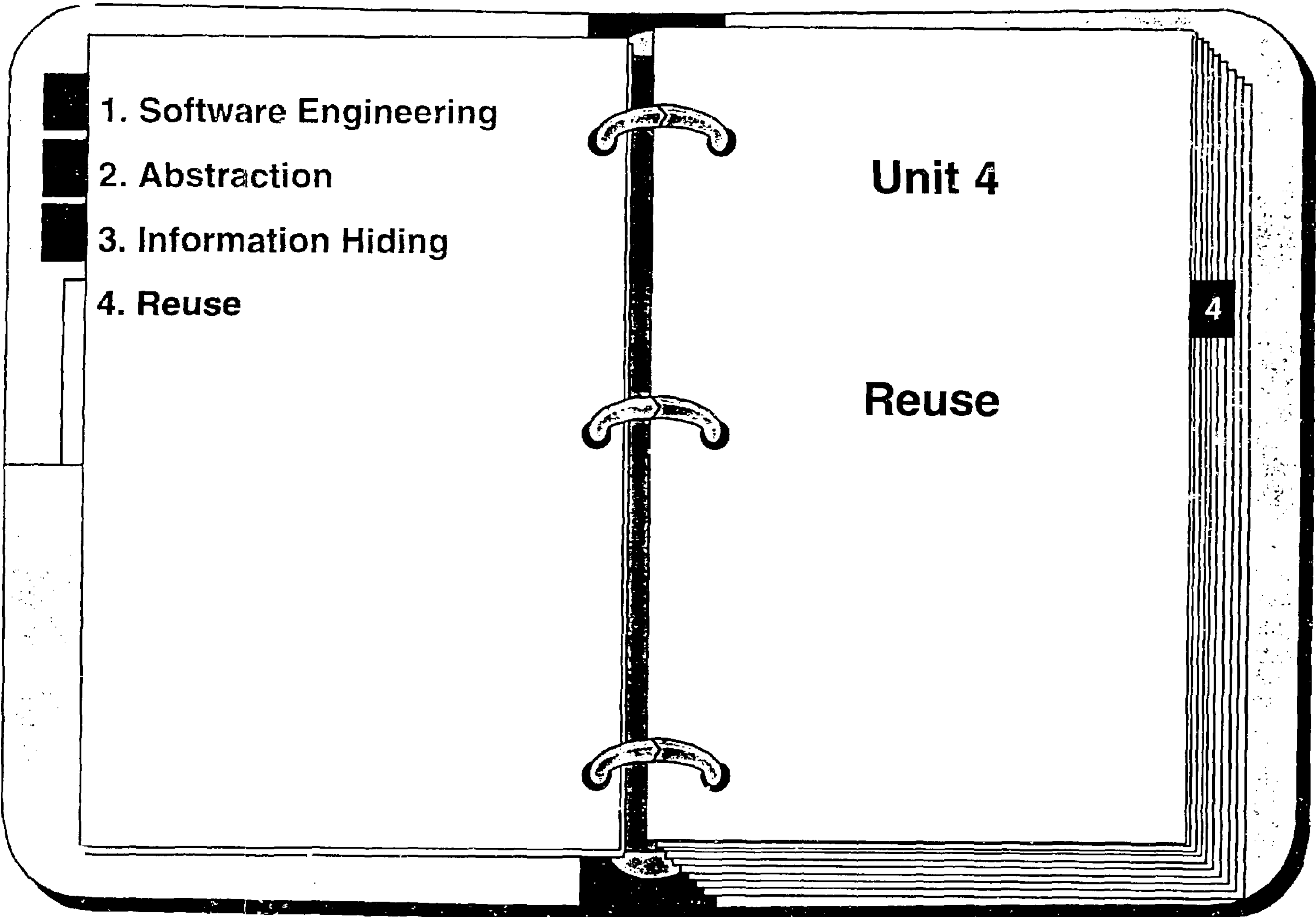
Reuse is very important. Engineers can greatly reduce the effort required to create a program through reuse. Slide 4-4 presents some statistics on just how much effort you can save when you reuse software. (By custom, "reuse" is both a verb and a noun.)

Reuse is, however, more difficult than it might seem. This unit will explain some of the reasons reuse is difficult. It will present strategies for dealing with reuse and illustrate those strategies using examples in the Ada programming language. It will also discuss how the information hiding design method presented in Unit 3 leads to reusable software.

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Define reuse
- Explain the role information hiding plays in reuse
- Understand how the Ada **generic** mechanism supports reuse



Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide depicts a common software development situation. You have implemented the Reverser program from Unit 3, which reverses the lines in a file. Now suppose you're developing another program, and it needs to store a list of integers. The Line Holder module has a similar, but not quite identical purpose: it stores lines, not integers. Can you save yourself work by using it in this program?

Clearly, you can't use the Line Holder directly. Can you simply replace the Line data type with Integer everywhere?

Perhaps, but this illustrates a general problem. You have a module close to, but not identical to, what you need. Which will be easier—using the existing module or writing what you need from scratch?

Consider a specific example. Suppose you have implemented the Reverser program from Unit 3 and someone asks you to implement the program from Unit 2, where you reverse a file of integers rather than text lines. Should you adapt the program from Unit 3 to solve the problem or write the program in Unit 2 from scratch?

Without more information, it's hard to tell. But a better answer is that using the existing module will be easier **if** you anticipated the potential for its **reuse** when you wrote it. You can save yourself a lot of future effort by planning ahead when you write programs. You can practice **software reuse**: writing a module that you can easily use in programs other than the one for which you originally wrote it.

Some programming languages help you do so. In this unit, we'll study how to "plan ahead." We'll also see how Ada supports reuse.

STUDENT INTERACTIONS

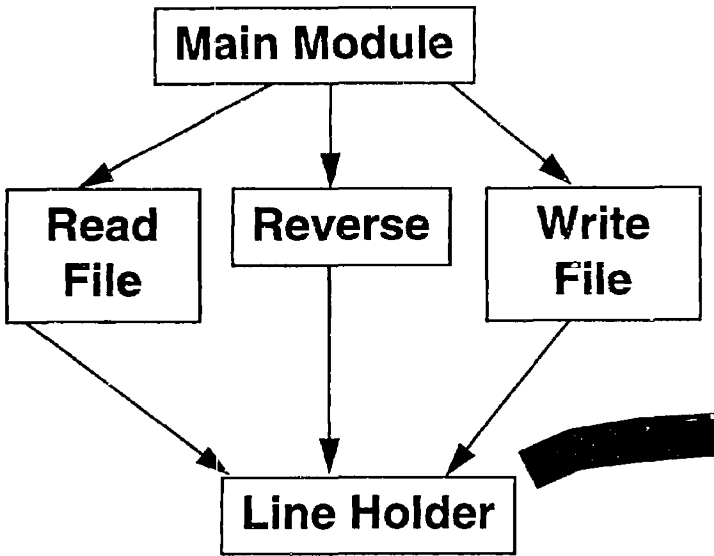
- Have you ever faced a situation similar to the one on this slide? Describe your experience.

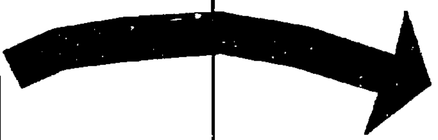
OBJECTIVES

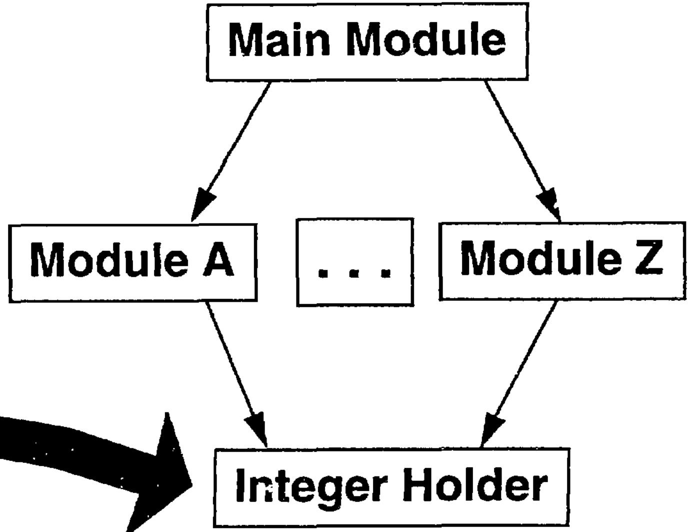
Students should be able to:

- Define reuse
- Understand that reuse involves planning ahead

A Common Situation







Will the Line Holder work here?

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide presents the prevailing theme of this unit: you should capitalize on others' work when developing software. This is known as practicing **software reuse**.

Moreover, you should develop the attitude that you have little excuse for not doing so. Whatever program you write bears some resemblance to a program someone has written already. At least some portion of it will be similar to existing software.

Professional engineers always try to reuse their own and others' work. Reuse helps them get their job done quicker. Moreover, they're using software someone has already tested, so they're confident that most of the bugs have been eliminated already. They couldn't be so confident if they had just written it. Schools may frown upon using others' work, of course, but they encourage students to build on their knowledge and previous work.

Reuse is possible because of similarities:

- Similarities among problems give you clues about viable solutions. If you realize the problem you're solving is similar to one you've solved in the past, you can look to the previous solution for ideas.
- Similarities among solutions help you identify the modules and structures you'll need and provide code you can reuse in the program you're writing.

Similarities can be fairly trivial, like two routines that both need the same trigonometric formula. In such a case, you can't reuse much, just a line or two of code. However, the similarities are usually more complex. Two programs may both share a stack data structure, for instance, or they may both use a mouse as an input device and require code to control it.

STUDENT INTERACTION

- What similarities do you see between the reversing problems in Units 2 and 3? (They both reverse a file's contents.)
- What similarities do you see between the solutions to the reversing problems? (They both share the idea of a main module controlling modules that read and write, although the details of the algorithms differ.)

OBJECTIVES

Students should be able to:

- Explain why reuse is advantageous
- Explain why reuse is possible

Two Software Maxims

- Every problem you solve is similar to a problem someone has solved before
- Every solution you create has similarities to an existing solution

You should practice <u>software reuse</u>!

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide presents some data that shows how much effort reuse can save you.

Software developers often achieve 70% or higher reuse—that is, of all the lines of code in a program, they write only 30% themselves, taking the other 70% from existing systems. The graph on this slide illustrates the result. Studies have shown that if you divide the total lines of code in a system by the number of days from the time design starts to the time testing ends (see Slide 1-7), then divide that number by the number of software developers who worked on the project, the result will be about 40. If you expect your software to be about 100,000 lines, then you should expect the project to need about 2,400 days of work. If you divide this work among a 10-person team, your project should take 240 days.

If, however, you can obtain 50% of these 100,000 lines from existing programs, you only have to write 50,000 lines. In that case you need only 1,200 days of work, or 120 days on a 10-person team. If you can reuse 70% of the software, you reduce your effort to 75 days. The graph shows clearly the difference reuse makes as the project size grows. Its effect is less dramatic on small projects, but still significant.

STUDENT INTERACTIONS

Do you think that the number of lines of code per day rises or falls on team projects? (It falls. The necessity for communication among developers steals time that would otherwise be spent designing, implementing, and testing software.)

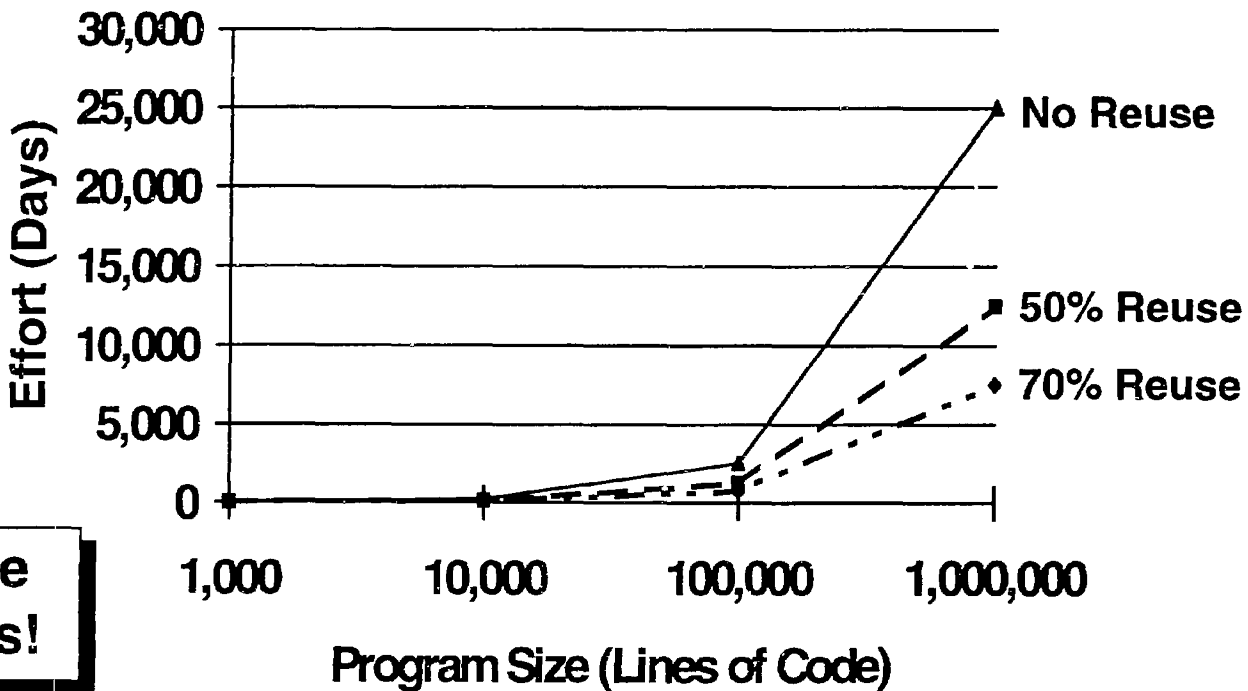
OBJECTIVE

Students should be able to:

- Explain the data in the graph on this slide

Some Software Reuse Data

The average software developer writes 40 <u>correct</u> lines of code per day over the course of a project



Reuse Saves!

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide carries Slide 4-2 one step further by showing a general desired capability.

We don't just want to adapt things once. Often, we'll find we need the same general capability of a module with many variations. Programs often need a "holder," i.e., a list of things. Here, we see that three programs need a holder. Our Reverser program needs to hold lines. A second program needs to hold a record type. A third program needs to hold integers. We do not want to have to write a completely new holder for each program. Indeed, we want to do as little work as possible to reuse the holder in all three programs. We want an *X* holder, i.e., something we can easily adapt to an arbitrary data type, either built-in or defined by the software developer.

This is especially true in a team. A large program typically needs many different holders. It makes sense to have one person responsible for developing them all, rather than having several people building independent holders.

Although this unit uses holders as the example, programs share many other types of modules.

STUDENT INTERACTIONS

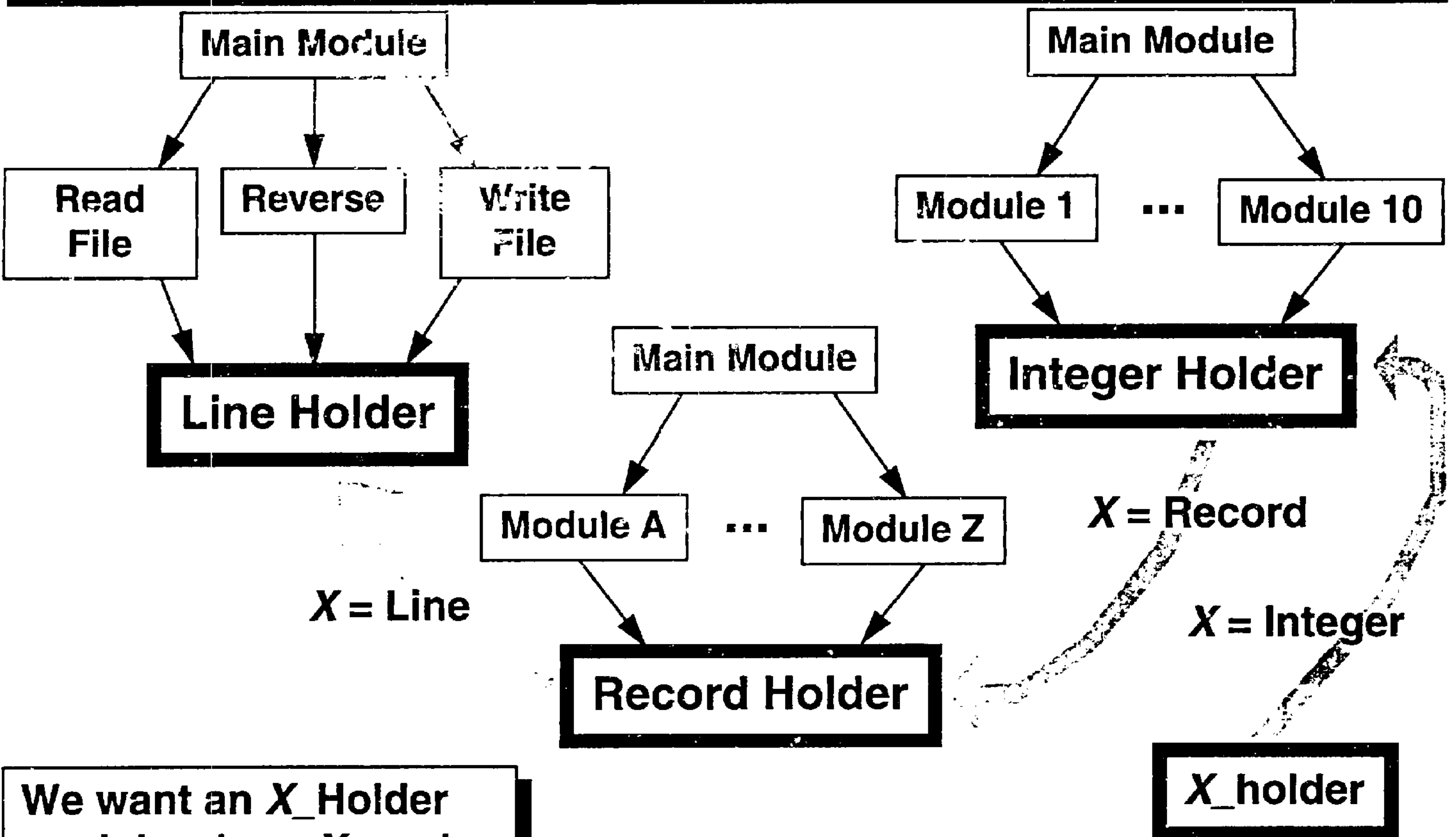
- What do you think would be likely to happen if a project had several people develop several different holders? (Probably they would all make, and have to fix, the same programming errors. It would be much more effective if only one person made and fixed an error!)
- What other modules can you think of that would be reusable in many programs? (Just about any data structure, like a stack or queue; code to perform common functions, like checking if the characters of a string form an integer; user interface code.)

OBJECTIVES

Students should be able to:

- Explain why a module might be useful in several programs
- Understand the notion of a general holder of things, adapted to a particular thing for a particular program

Reusing Modules: The Second Maxim at Work



We want an *X*_Holder module where *X* can be any data type we need.

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows how Ada supports reusable modules through its **generic** mechanism.

We change the Line_Holder package to a **generic package** called Holder. The second line adds a **generic parameter** named **Item**. This replaces the Line data type from the Line_Holder package. Note how Item appears in the private part instead of Line in the Values field of the List record.

Now look below the horizontal line, to the two **instantiations**. An instantiation yields a package from a generic package. The first one yields a new package, Line_Holder. The Item => Line says we want the value of the generic parameter Item to be Line. The effect is to create Line_Holder from Holder essentially by substituting Item with Line everywhere Item appears in the Holder package. Do this substitution, and you'll see that the record in the private part is the same as in the Line_Holder package on Slide 3-11.

The second instantiation shows the power of generics. We can create a holder of integers just as easily as lines by using Item => Integer. Again, you can see how this works by substituting Integer for Item everywhere Item appears.

STUDENT INTERACTIONS

The type List_Of_Lines has been changed to List, and the constant Max_Lines has been changed to Max_Values. Why? (Because those names reflected the module designer's intent to store only lines. The new names reflect the increased generality.)

OBJECTIVES

Students should be able to:

- Explain what it means to "instantiate" a package
- Understand that the Line_Holder package is now instantiated from the Holder package
- Understand the Integer instantiation
- Explain how to instantiate the Holder package to create holders of other simple types

Ada Generic Package

```
generic
    type Item is private;
package Holder is
    type List is private;

procedure Initialize(Items: out List);
    ...
private
    Max_Values: constant Integer := 10000;
    type Items is array(Integer range 1..Max_Values) of Item;
    type List is record
        Number: Integer range 0..Max_Values;
        Values: Items;
    end record;
end Holder;
```

Let's add a generic parameter

Remove references to lines

The record now stores values of type Item

```
subtype Line is String(1..255);
package Line_Holder is new Holder(Item => Line);

package Integer_Holder is new Holder(Item => Integer);
```

Now we instantiate the generic package

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide explains how a software developer declares and instantiates a generic package.

A generic package consists of a **generic specification** and a package body. A generic specification is a package specification that begins with the keyword **generic**, followed by one or more **generic parameters**. Slide 4-6 showed one type of generic parameter. Slides 4-8 and 4-10 will show two others.

The generic specification differs from a nongeneric package specification only in the addition of generic parameters. The package body for a generic package is identical to the package body of a nongeneric package.

You cannot access the interface of a generic package *in a program. You must* **instantiate** *a generic* package. You declare a new package (here, Instantiated_P) from a generic package. Other packages may access the interface of the instantiated package, but not the generic package. So if package Generic_P contains a procedure X, other packages can call Instantiated_P.X, *but not* Generic_P.X.

When you instantiate a generic package, you must provide a value for each generic parameter.

STUDENT INTERACTIONS

What are the implications of using a private type as a generic parameter on Slide 4-6? (The generic package can make no assumptions about the representation of the generic parameter.)

OBJECTIVES

Students should be able to:

- Explain how to declare and instantiate a generic package
- Define “generic parameter” and “instantiation”

Concepts of Generics

```
generic
   generic parameter P1;
   generic parameter P2;
   ...
package Generic_P is
   ...
end Generic_P;
```

Generic Specification:
Package specification plus generic parameters

```
package Instantiated_P is
   new Generic_P(
      P1 => V1,
      P2 => V2,
      ...
   );
```

Generic Instantiation:
Create an instance of a package according to the generic parameters

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide presents **generic formal objects**, another type of generic parameter.

You may want to parameterize a generic package with things other than data types. For instance, the designer of the Holder package might want to let other packages estimate the maximum number of things their module can hold. (This is realistic since the computer's finite RAM and disk space always impose an upper bound.) You can use generic formal objects to do this.

In this slide, the Holder package declares Max_Values as a generic formal object parameter. It must be a positive integer (you never want a module that can't hold at least one value.) When you instantiate the Holder with a value for this generic parameter, the value is substituted everywhere the parameter appears. Therefore, the first instantiation creates a holder capable of holding 1,000 values; the second, 30; the third, 50,000.

STUDENT INTERACTIONS

Why is Max_Values declared as positive? What do you think would happen if you declared it as an integer and instantiated Holder with Max_Values = –1? (It's declared to constrain it as much as possible to the valid range. If you made it an integer and instantiated the package with Max_Values = –1, you would get a compile-time error, because the type declarations for the fields of the List record wouldn't make sense.)

OBJECTIVES

Students should be able to:

- Explain the effect of instantiating the Holder package with Max_Values = N for any integer N
- Understand the concept of a generic formal object parameter

Generic Formal Objects

```
generic
   Max_Values: Positive;
   type Item is private;
package Holder is
   type List is private;

   procedure Initialize(Items: out List);
   ...
private
   type Items is array(Integer range 1.. Max_Values) of Item;
   type List is record
      Number: Integer range 0..Max_Values;
      Values: Items;
   end record;
end Holder;
```

Let's have other packages control the maximum possible number of things held

Here's how we reuse the holder 3 times

```
package Line_Holder is new Holder(Max_Values => 1000, Item =>Line);
package TinyInt_Holder is new Holder(Max_Values => 30, Item => Integer);
package BigInt_Holder is new Holder(Max_Values => 50000, Item => Integer);
```


Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide presents a problem that will motivate the need for increased flexibility in declaring a generic formal object.

Suppose you want to store students' grades. Moreover, you decide that you want to store them by year, such that you can easily access all students' grades for 1993, 1994, etc. You have records going back to 1980 and want to build a system that will be able to keep records for 50 years.

You declare a type Student that can store one student's grades in all subjects for a single year (this school's curriculum is very homogeneous, apparently). You instantiate the generic Holder package to create a package Students that can hold the grades for all students in a single year. You then declare the Yearly_Record type, associating a year with a Students Holder, and instantiate the Holder package again to create the School_History package. This is a module that can store 50 years of students' grades.

This works, *but clumsily*. For instance, how do you retrieve the grades for 1992? You might expect 1992 to be the thirteenth item in the Holder, but you can't be sure it was stored that way—that is, the order in which items are stored is likely to be hidden information, not accessible to the module that retrieves items.

This is a flaw in the design of the Holder. Its indexes always start at 1 and end at Max_Values. This sort of situation is what always tempts software developers to rewrite software. The Holder would be more flexible and reusable if, when you instantiated it, you could make its indexes run between any two values.

OBJECTIVE

Students should be able to:

- Explain how the structures on this slide can be used to store and access student grades.

Recording Student Grades

**We record 50 years—
but in what order?**

```
with Holder;
package Student_Grades is
   type Grade is (A, B, C, D, F);
   type Student is record
         Name: String(1..20);
         Math_Grade, CS_Grade, History_Grade: Grade;
   end record;
   package Students is new Holder(Max_Values => 50, Item => Student);
   type Yearly_Record is record
         Year: Integer range 1980..2029;
         Accomplishments: Students.List;
   end record;

   package School_History is new Holder(Max_Values => 50,
                                        Item => Yearly_Record);
end Student_Grades;
```


Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide presents **generic range parameters**. A generic range parameter solves the problem discussed on Slide 4-9.

You could solve the problem with the indexes by adding to the Holder package another generic parameter, Min_Value. You might then declare:

```
type Values is array (Min_Value..Max_Values) of Item;
```

However, Ada offers a simpler approach. You can replace Max_Values with a new parameter, Index. The range of Index is given as <> meaning you must declare it when you instantiate it. You do so by providing any two integers. (Slide 4-11 has examples.)

This requires modifying the Item_Number function. Previously, its second parameter was an Integer that had to be between 1 and Max_Values. Let's restrict it to the range specified by Index. Thus, the first item added to the Holder will be indexed by the lowest value in the range of Index. The second item added will be indexed by the second lowest value in the range of Index, and so on.

STUDENT INTERACTIONS

- Must you change Number_Of analogously to Item_Number? (No. Number_Of returns the number of items currently stored in a Holder—always an ordinal number—so there is no need to change it.)
- What does the declaration of the Number component of the List record do? (Number's range is from the smallest value of Index minus 1 to the largest value of Index. This is equivalent to the previous declaration.)

OBJECTIVES

Students should be able to:

- Understand how the new definition lets you define a holder with an index of a specified range
- Explain how to instantiate a holder with an index in a specified range

Generic Range Parameters

```
generic
   type Index is range <>;
   type Item is private;
package Holder is
   type List is private;

   procedure Initialize(Items: out List);
   procedure Add_Item_To_List(Item_To_Add: in Item; Items: in out List);
   function Item_Number(Items: in List; i: in Index) return Item;
   function Number_Of(Items: in List) return Integer;
private
   type Items is array(Index) of Item;
   type List is record
      Number: Integer range Integer(Index'first)-1 .. Integer(Index'last);
      Values: Items;
   end;
end Holder;
```

A generic range lets us define the minimum and maximum index values

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows how to use the new Holder module.

The slide shows two generic instantiation examples. The first, Students, is equivalent to the example on Slide 4-9. However, the equivalence is obtained by specifying a range and, in particular, specifying that 1 is the lower bound of the range.

The second, School_History, creates a Holder that holds a list of students. The index is from 1980 to 2029. This means you provide Item_Number with a year, representing the year you want to access, instead of providing it with an ordinal number that's an offset from 1980 as before.

STUDENT INTERACTIONS

How would you reference a student in the inner **for** loop? (Students.Item_Number(Single_Year, S), where Single_Year is declared to hold the records of students. The assignment statement assigns it a single year's worth of records.)

OBJECTIVES

Students should be able to:

- Understand the example use of the School_History package
- Explain how to instantiate the Holder package

Using the School History

```
subtype Student_Index is Integer range 1..50;
package Students is new Holder(Index => Student_Index,
                               Item => Student);
subtype Year_Index is Integer range 1980..2029;
package School_History is new Holder(
                               Index => Year_Index,
                               Item => Students.List);

All_Students_Ever: School_History.List;
Single_Year: Students.List;
...
for Year in 1980 .. 1980 + School_History.Number_Of(All_Students_Ever) - 1
loop
    Single_Year := School_History.Item_Number(All_Students_Ever, Year);
    for S in 1 .. Students.Number_Of(Single_Year)
    loop
        ...
    end loop;
end loop;
```

You specify the ranges for students and years explicitly

You index school history starting from 1980

You index students within a year starting from 1

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide relates reuse to abstraction and information hiding.

Reusability comes from adaptability. The more adaptable a module, the more likely it'll be reusable.

You can make a module more adaptable by keeping in mind how reuse occurs. When you are designing software, you are thinking about the functions your software performs. You will consider reusing a module if it seems to perform a function you need.

The obstacle to reuse is that modules often perform a function in a specific way—with a specific data type as a parameter, for instance. However, if your module is adaptable, you can expect that it can be adapted to perform the same basic function, but in a range of conditions.

When you design a module, you should think about this. Consider the basic function your module performs—in other words, the module's essential information. Now look at the module again. How might other programs want to use this same function? Your goal should be to design the module so it always performs the basic function, but can do so in many ways.

You can achieve this by hiding the basic function within the module so other modules don't have to worry about how it's performed and striving to make the nonessential characteristics of the module adaptable and placing them in the interface. In other words, you can apply information hiding principles to help create reusable modules.

Ada language features also help you create reusable modules:

- You can use packages to show an interface to the basic function but hide the algorithm that implements it.
- You can use generics to show, in a module's interface, the ways the module may be adapted.

OBJECTIVE

Students should be able to:

- Explain why information hiding principles are important in the design of generic packages

Designing Reusable Modules

- Your goal: Flexibility
- You:
  - Provide other modules an exact description of the allowed flexibility (the generic parameters)
  - Hide, within the private part and the package body, how the parameters provide this flexibility

In other words:

Abstraction and information hiding principles help you design reusable modules

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

This slide shows the role reuse plays in the software development process covered in Unit 1.

Reuse does not occur by chance. It happens through disciplined use of sound engineering practices.

- You must systematically catalog all your modules and place them in a "library" so you will have ready access to them.
- During design, you carefully study the contents of the library to see what modules you might be able to reuse.
- During coding, you reuse those modules.
- Also during coding, you create new modules. Some of these might be reusable and so should be placed in the library.

STUDENT INTERACTIONS

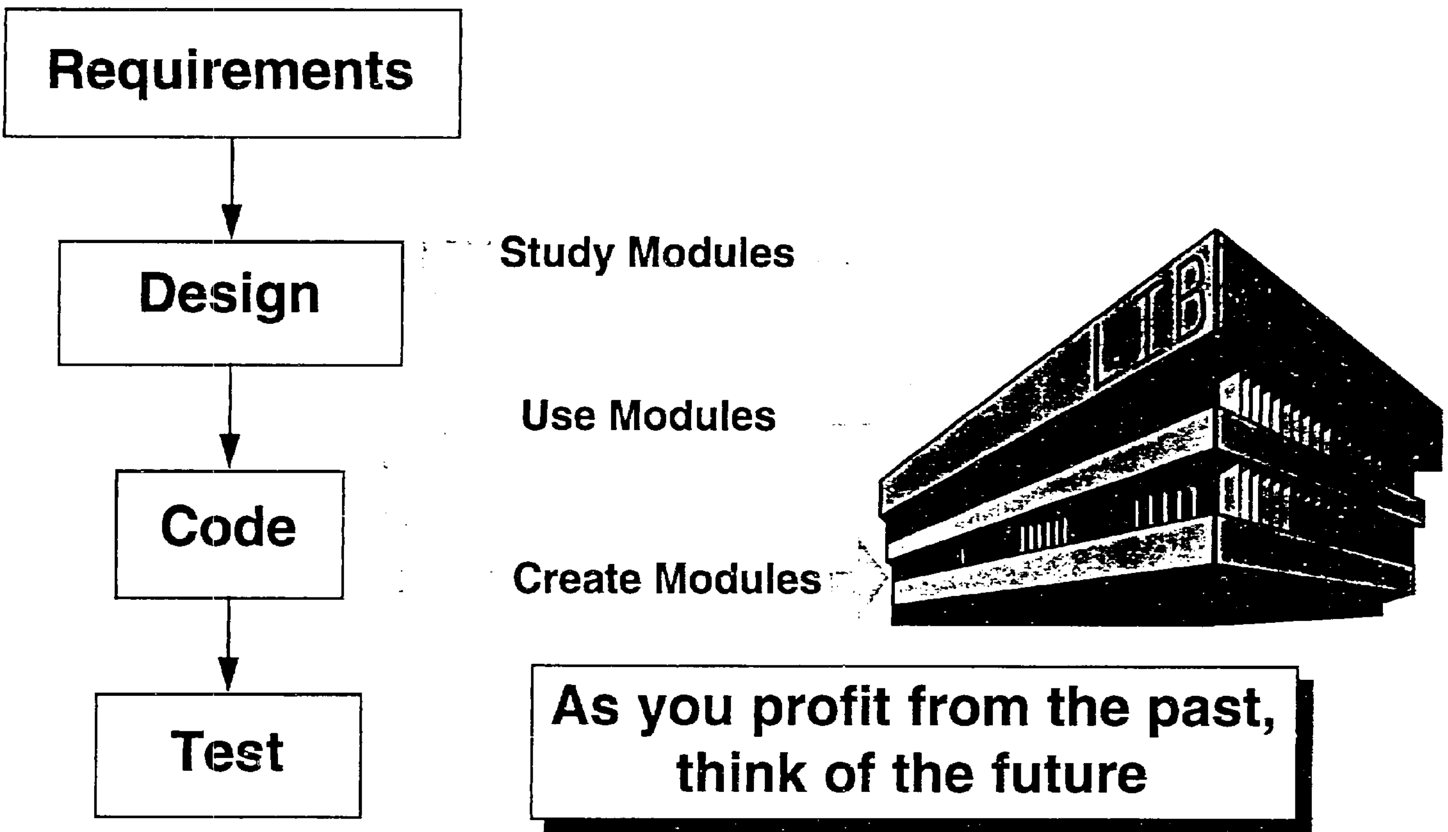
- Do you think that the existence of reusable modules might influence your design? (Yes. If you can make use of an existing module, you should make design decisions that will facilitate that use.)
- Your school library lets you search for books based on author, title, or subject. What do you think might be appropriate for modules? (Software libraries are similar to regular libraries. Title and subject indexes are used most.)

OBJECTIVES

Students should be able to:

- Explain where in the software process you reuse software
- Explain where in the software process you create reusable software

Total Picture for Reuse



As you profit from the past, think of the future

Copyright © 1995, Software Productivity Consortium.

DISCUSSION

The discipline of software engineering emerged to solve the problems with developing large software systems. The main objective of software engineering is to facilitate the production of high quality software systems within budget and on time. You want to stress that solutions should be engineered, not "hacked" together.

We concentrated on four aspects of software engineering in this course:

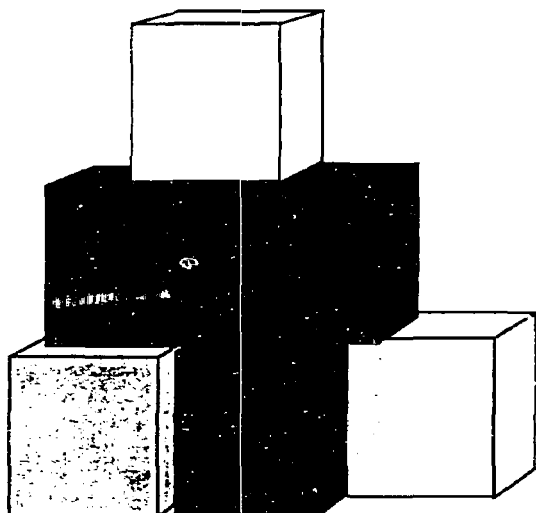
- How software engineering helps with complexity
- How software engineering helps with communication
- How software engineering helps with modularity or the ability to minimize the effect of change
- How you realize these software engineering principles with the Ada programming language

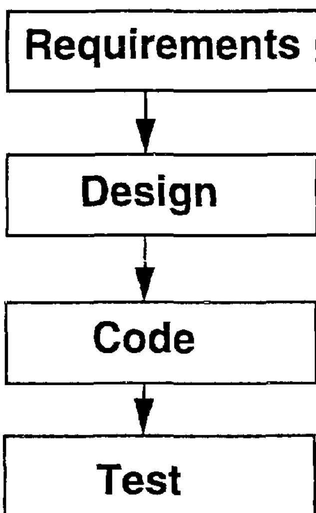
You use design methods as your building blocks to deal with complexity. Design methods help you break the problem into manageable subsets so that individual software designers can design and implement a piece of the problem. You use abstraction and information hiding to help manage complexity and change. You use Ada packages (both specification and body) to realize abstraction and information hiding. You also consider reuse in managing complexity. Software engineers strive for being able to reuse previous solutions. Ada provides a very powerful mechanisms to help achieve this goal in Ada generics.

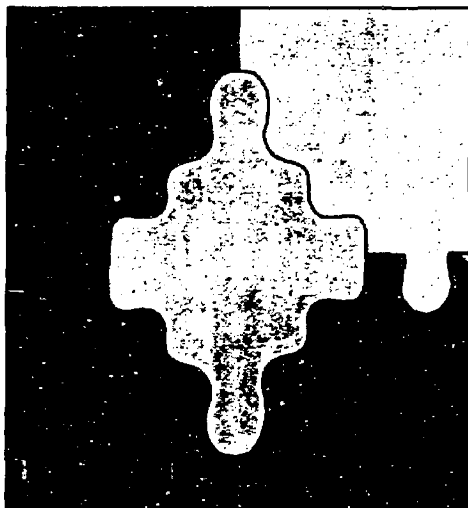
Communication is important for all phases of the software development life cycle. Requirements specify what you are to build. Design specifies how you will build the software. Code is the actual implementation of your design. Finally, test allows you to verify that you have built the software according to the requirements. To minimize the effect of change, you want to solve relatively independent pieces of the problem and bring them together as a set of cooperating modules. Again Ada helps in communication by managing the complexity by having a mechanism for separate work units (e.g., packages). Think of the puzzle pieces as parts of the software problem you would divide among various teams.

Course Summary

Complexity	Communication	Change
• Design methods • Reuse	• All phases of software development	• Modularity







Ada Programming Language Features

Copyright © 1995, Software Productivity Consortium.

UNIT 4: REUSE

UNIT SUMMARY

As you develop software, you will often find that you need a module that offers functions and data types similar to, but not exactly matching, one you have developed in the past. The more experience you gain with software, the more you will find this to be true. You will also find that other people have developed software you might be able to use. In other words, no program is totally unique. It solves a problem related to problems that have already been solved, and its modules and structure resemble, in part, modules and structures of existing programs.

You should try to **reuse** existing software whenever you can. Studies have shown that in many programs, especially larger ones, 50% of the lines of code can easily come from existing programs. Reuse of 70% of the code is not uncommon. Since a software developer produces an average of 40 lines of code per day over the course of a project, it's easy to calculate just how quickly the savings will add up.

Unfortunately, reuse is harder than it might seem. You'll find the primary reason is that you developed your modules for use in a specific program. When you try to use them in another program, you often realize you need something slightly different: a function must operate on a string rather than an integer, for instance. You may find that writing a new module from scratch is easier than modifying an existing module.

There are several things you can do to improve the chances that a module will be reusable. One is to use **generic** packages. When you write a generic package, you declare **generic parameters** that specify the different ways you expect you might want to use the package. You can then **instantiate** the generic package by providing values for the parameters that meet the needs of specific programs. For example, you can use Ada generic packages to rewrite the Line_Holder package so it can hold integers, strings, or any object that is a valid Ada data type:

```
generic
   type Item is private;
package Holder is
   type List is private;

   procedure Initialize(Items: out List);
private
   Max_Values: constant Integer := 10000;
   type Items is array(Integer range 1..Max_Values) of Item;
   type List is record
      Number: Integer range 0..Max_Values;
      Values: Items;
   end record;
end Holder;
```

If you then declare a data type representing a line:

```
subtype Line is String(1..255);
```

you can create a `Line_Holder` package that behaves identically to the one in Unit 3 using the following **generic instantiation**:

```
package Line_Holder is new Holder(Item => Line);
```

Because Holder is a generic package, you can use it to hold objects of any type. For example, you can use the following generic instantiation to hold a list of integers:

```
package Integer_Holder is new Holder(Item => Integer);
```

The Ada compiler cranks out a new package based on the parameters to the generic instantiation. It's as if you took the `Holder` package and substituted `Line` or `Integer` everywhere `Item` appears. See Figures 1 and 2, drawn from the Ada code on Slide 4-6. In each figure, the generic `Holder` package on the left side, with its generic parameter `Item`, is instantiated to yield the package on the right.

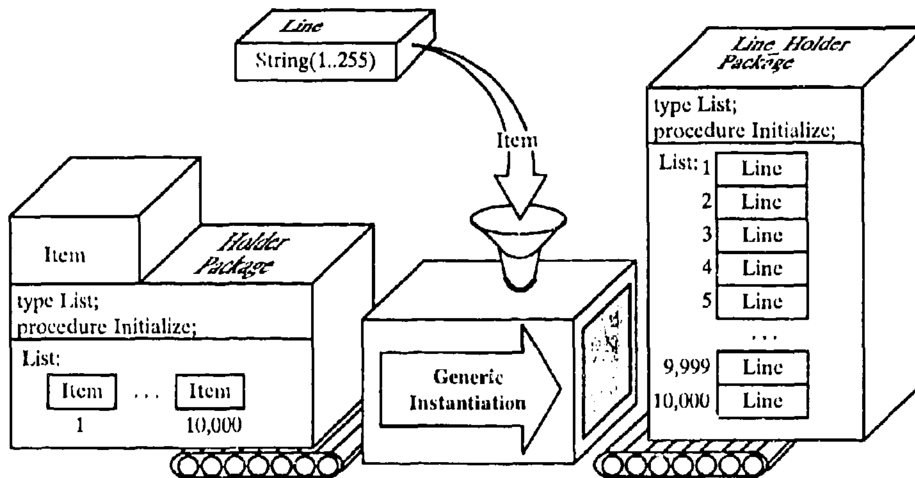


Figure 1. Generic Instantiation of the Holder Package With Item=>Line

Data types and numeric values are examples of parameters you often use to make a module more reusable.

Ada has many kinds of generic parameters. Figures 1 and 2 illustrate a **generic type parameter**. You can also write generic packages with **generic formal object parameters**. A generic formal object is a parameter that's a constant value, such as an integer or a character. For example, you could add a parameter `Max_Values` that controls the maximum number of values a holder can store. See Figures 3, 4, and 5, drawn from the Ada code on Slide 4-8. A second generic parameter, `Max_Values`, has been added to the `Holder` package. This parameter lets you specify the maximum number of values an instantiation of the `Holder` package can store. You can now control both the type and size. Figure 3 creates a package that stores 1,000 lines. Figure 4 creates a package that stores 30 integers. Figure 5 creates a package that stores 50,000 integers.

A generic type parameter can be any valid Ada data type, even one declared from a generic instantiation. Figure 7, drawn from the Ada code on Slide 4-9, illustrates this point. Here, the item used in the lower generic instantiation is of type `List` from the package `Students`, which was instantiated from `Holder`.

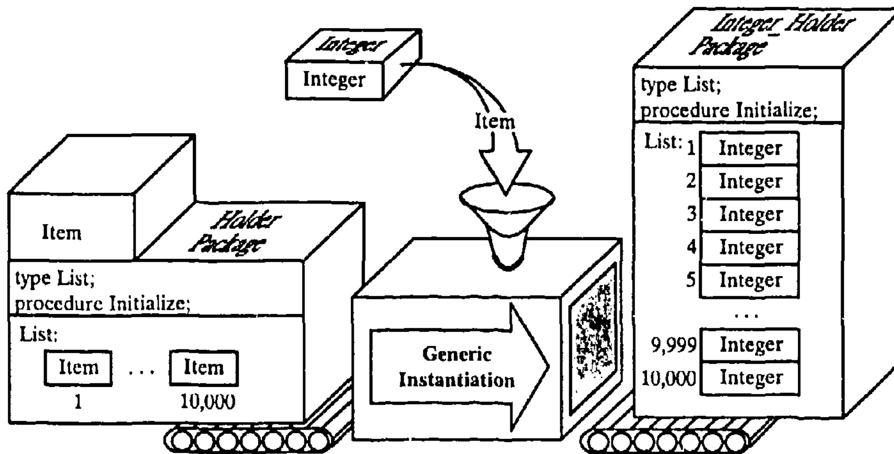


Figure 2. Generic Instantiation of the Holder Package With Item=>Integer

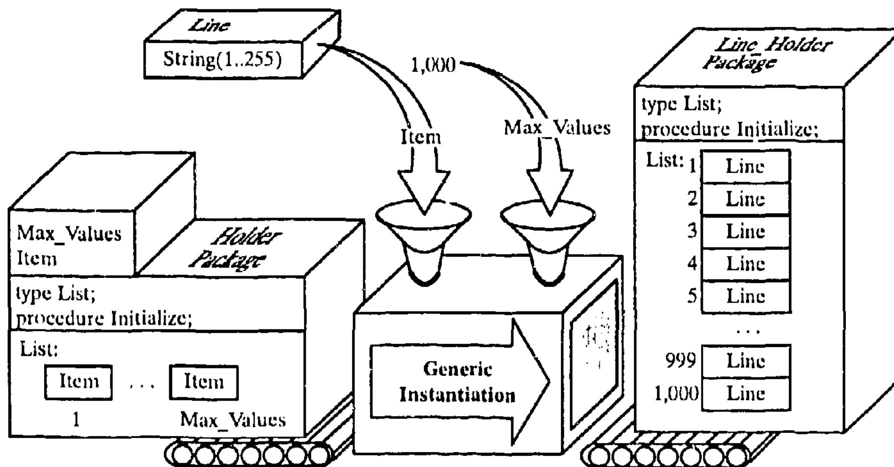


Figure 3. Generic Instantiation of the Holder Package to Hold 1,000 Lines

Another useful kind of generic parameter is the **generic range parameter**. It is a special case of the generic type parameter, where the type you provide is a subtype of integers. Using subtypes helps you overcome the situation in Figure 7, where to index calendar years you must create a new data type to hold the years of a school's existence. Figure 6 uses the `Year_Index` subtype in the lower generic instantiation to create a holder module whose indexes range from 1980 to 2029.

You can also improve the chances that a module will be reusable by spending some extra time thinking about reuse as you develop the module. Ponder the functions that the module offers. Think about what is essential to the module and what is incidental to the program for which you are developing it. This will help you realize what generic parameters are appropriate.

If you follow this advice, over time you will build up a "library" of reusable modules. You will find that your software development process automatically incorporates reuse. As you design your software, you will consider what modules are in your library and base your design on what you can reuse, realizing that reuse will save you considerable amounts of time.

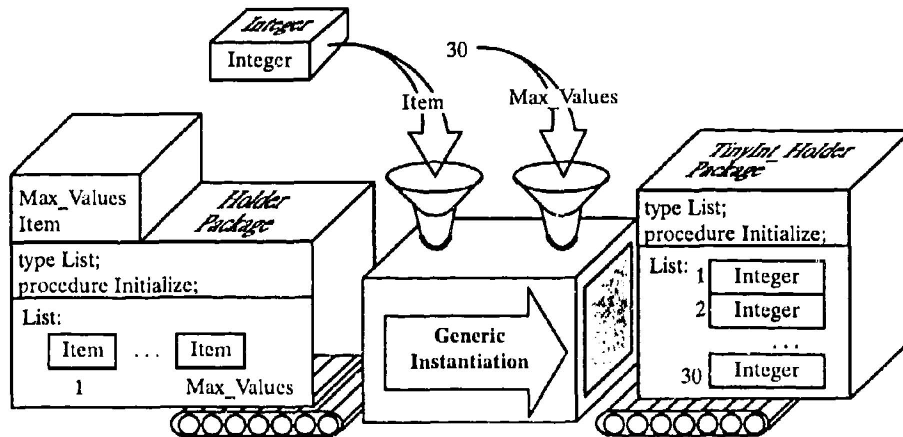


Figure 4. Generic Instantiations of the Holder Package to Hold 30 Integers

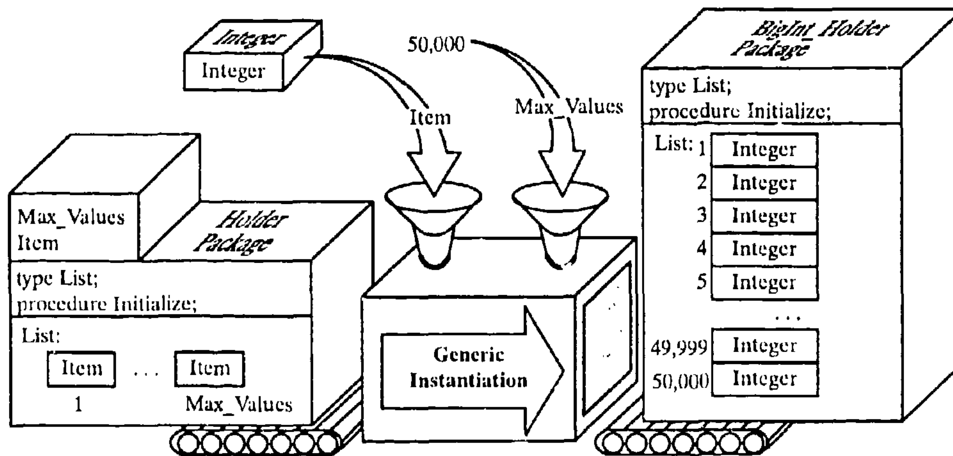


Figure 5. Generic Instantiation of the Holder Package to Hold 50,000 Integers

Reuse goes hand in hand with the information hiding design method covered in Unit 3. To create reusable software, you must make it adaptable to a range of situations in which it will likely be used. You can do this by hiding how the essential functions work, but showing, on the interface, the exact ways in which the adaptation is possible. This separation of interface and hidden information comes directly from information hiding.

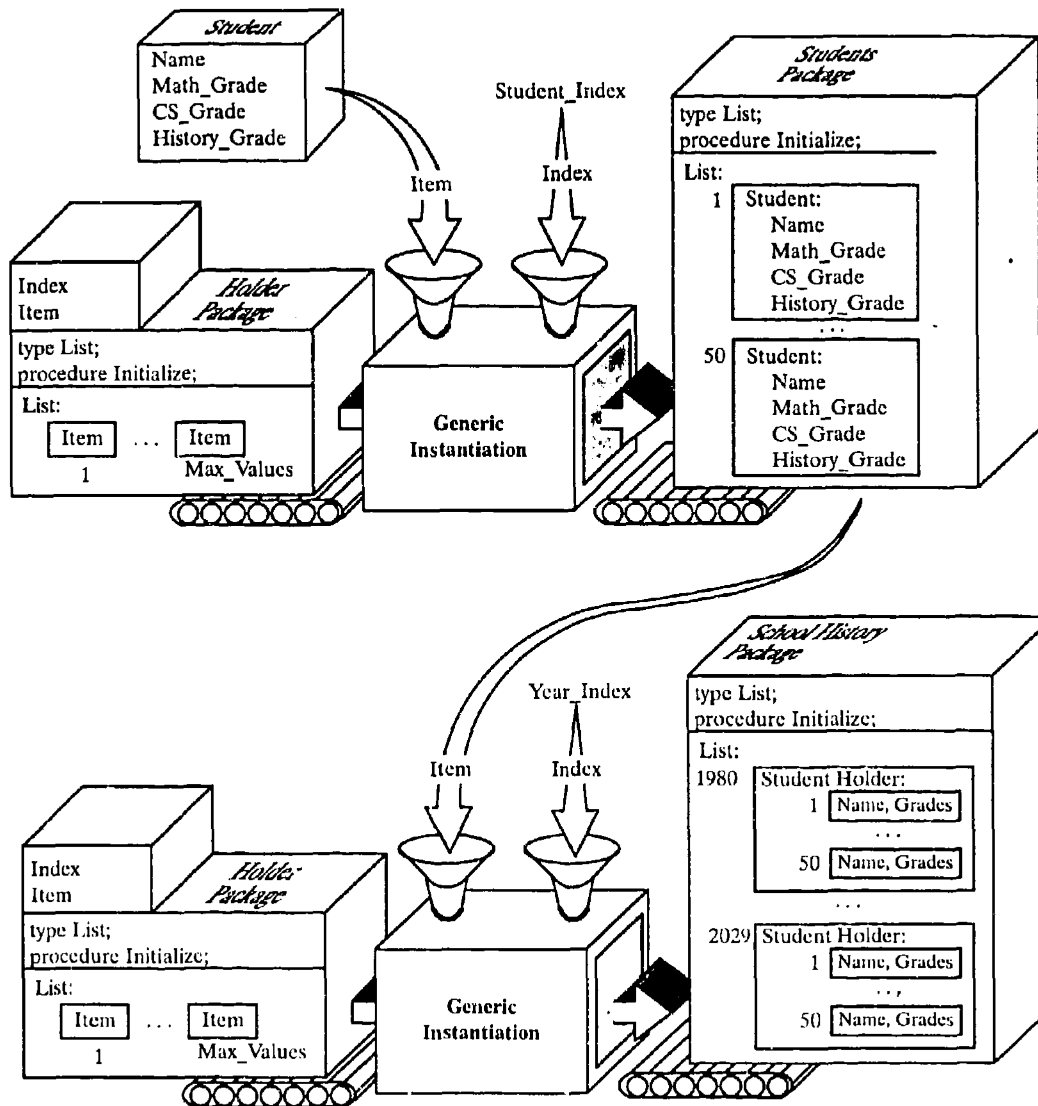


Figure 6. Instantiating Student Grades and School History Using Generic Range Parameters

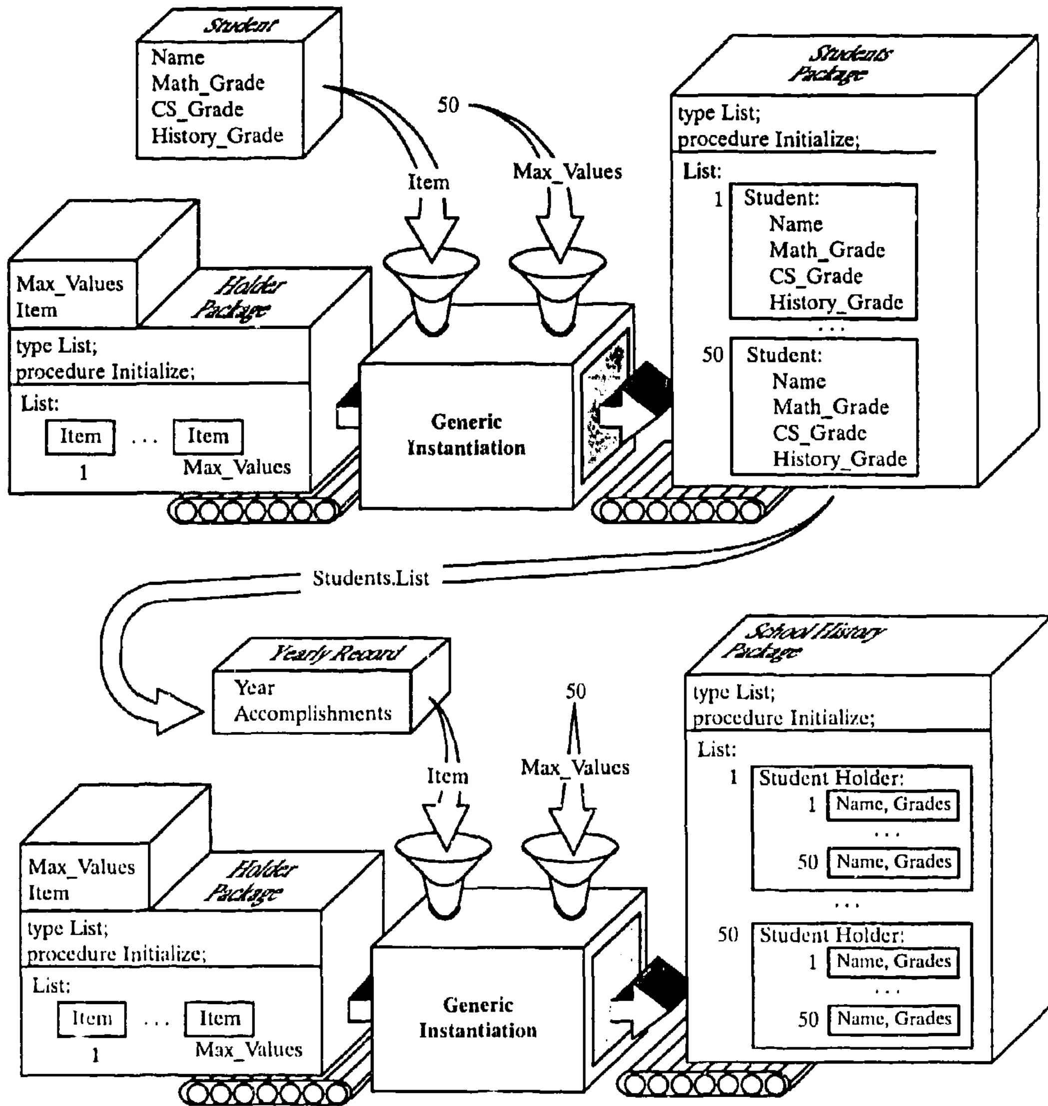


Figure 7. Instantiating Student Grades and School History

UNIT 4: REUSE
LABORATORY SPECIFICATION

NOTE: This laboratory is not being produced for the pilot offerings. The laboratory will appear in a later version of the course, based on comments received from the teachers of the pilot offerings. This specification provides a definition of the current vision for such a laboratory. Because the laboratory has not been built, not all issues have been resolved. Unresolved issues are shown in *italic* text.

PART 1: BACKGROUND

In this laboratory, you will assemble vending machine software. You will not write this software yourself; you will use the reuse techniques you learned in your lecture to create it. You will work in groups, jointly reusing and developing the software.

Suppose you are an employee of the Press 'n Gobble Vending Machines Company. One day, your sales department informs you that it has located two potential markets for the vending machines your company builds. However, none of Press 'n Gobble's current machines quite fits either market. Management has decided to develop new ones and assigns you to develop the software the machines will need.

Here is a description of each machine:

1. The first machine is to be sold in the United States. It will dispense a variety of food products. It will look as shown in Figure 8. Items in the second row cost 65¢. All other items cost 55¢.

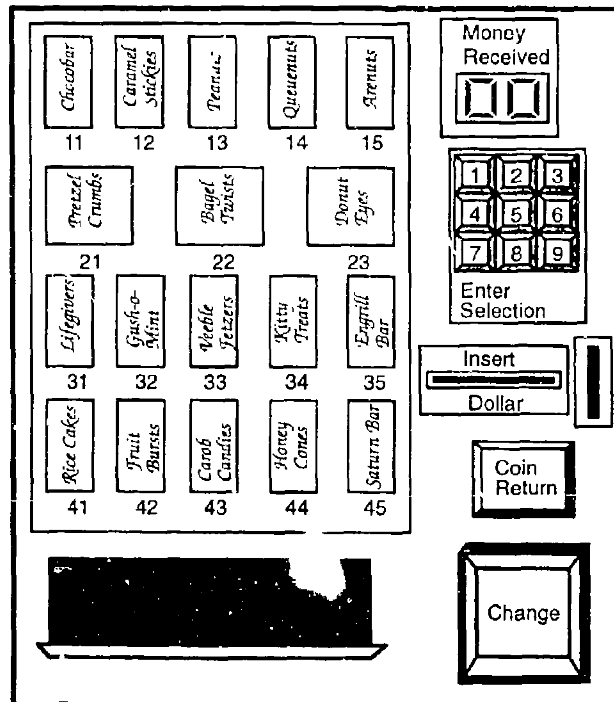


Figure 8. A Food-Dispensing Vending Machine

2. The second machine is to be sold in Germany. It is shown in Figure 9. This machine is to dispense hot beverages: coffee, decaffeinated coffee, tea, and espresso. Because it dispenses only a few items, it does not have a numeric keypad for selection. Instead, each item has a button; you push that button to get the item. Of course, German labels will be substituted for the English ones when the machine is placed in final production. You can insert 10 pfennig, 50 pfennig, 1 DM, 2 DM, or 5 DM coins. The machine dispenses change using 1 DM, 50 pfennig, and 10 pfennig coins.

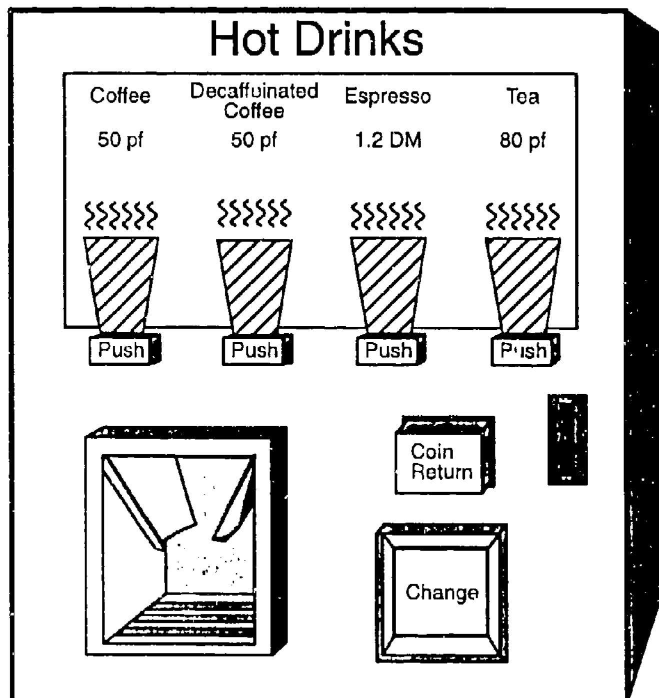


Figure 9. A Drink-Dispensing Vending Machine

Your assignment for this laboratory is to generate the software for both vending machines. You will do so by choosing the necessary software modules, as explained in the exercises below. You must compile and link these modules. You can then run the software.

The Software Design

Press 'n Gobble's software developers maintain an extensive, well-organized reuse library of the modules they have developed over the years. Furthermore, they have developed a general design for vending machine software. When presented with the requirements for a new vending machine, they can quickly determine the modules they need. Every program they develop always has the modules listed in Table 1.

Table 1. Software Modules Used In All Press 'n Gobble Vending Machines

Module Name	Module Description
Change Return Button	Signal that the money the person has entered so far is all to be returned. The hidden information of this module is how it is determined that the button has been pressed.
Coin Return	Dispense a selected amount of money, in coins. The hidden information of this module is how the hardware that dispenses coins is activated.
Coin Acceptor	Accept coins and provide to the software the value of the coin. The hidden information is the means by which it is determined what coin was entered.
Item Dispenser	Dispense a product to the person. The hidden information of this module is how the hardware dispenses products.
Money Accumulator	Maintain a record of how much money the person has entered so far. The hidden information is the means to calculate and represent this information.
Item Selector	Signal a selection. The hidden information of this module is how it is determined that the person has pressed the buttons to make a selection.
Input Event Handler	Collect and respond to the signals issued by other modules. The hidden information of this module is the algorithm for collecting and responding to signals.
Change Calculator	Determine the amount of change needed for a purchase. The hidden information of this module is the algorithm for calculating change.
Price Information	Maintain the price for items dispensed by the machine, and allow determination of whether a specified amount of money is sufficient to purchase a specified item. The hidden information is the representation of the prices and the algorithm for determining whether the purchase price is enough.
Holder	Maintain a list of items. The hidden information is the representation of the list and the algorithms for accessing it.

The details of some of these modules may vary between machines, but a form of each module exists in the software of any vending machine Press 'n Gobble sells. *This list of modules is not intended to be complete, just illustrative. They were derived using information hiding. Moreover, many of them are used in both vending machines. This module sharing is the primary requirement for any design.*

Press 'n Gobble's software library has other parts too. These parts are only needed in certain vending machines, as described in Table 2.

Table 2. Software Modules in Specific Press 'n Gobble Vending Machines

Module Name	Module Description	Include If...
Bill Acceptor	Accept bills, and provide to the software the value of the bills. The hidden information is the means to determine what bill was entered.	The vending machine is to accept both coins and bills.
Money Display	Display an amount of money. The hidden information is the algorithms used to activate the display.	The vending machine is to display the amount of money the person has entered so far.

PART 2: LABORATORY EXERCISES

EXERCISE 1: UNITED STATES VENDING MACHINE

You are to create the software needed for the vending machine to be sold in the United States. You must perform the following steps:

1. Read the description of the food-dispensing vending machine on Page 7.

2. Determine the modules you will need for the vending machine's software.

3. Assign a set of modules to each member of your group. You should bear in mind that some modules are larger than others and partition them equally across your group so everyone has approximately the same workload. Use the figures in the last column of Tables 1 and 2 as a rough guide to the relative time each person will need for each module. *This column isn't in place yet and can't be until the software is written. We shall need to time each module's compilation and prepare the figures based on that information. Note that Steps 1 through 3 would make an excellent prelaboratory homework assignment.*

4. Each person must perform the following steps. See the accompanying write-up on using the laboratory for instructions on how to do so.

   a. Create a directory in which to work with her or his assigned set of modules.

   b. Copy the modules assigned to her or him from Press 'n Gobble's library of reusable modules to the directory created in Step a.

   c. Write generic instantiations for the following modules: . . . *We shall ask the students to write a few generic instantiations, just so they get the feel of adapting reusable modules to fit a specific need. We shall provide them with templates, and we shall provide the teacher with the answers.*

   d. Create an Ada library. *This is assuming that the Ada compiler does not support concurrent compilation using a single library.*

   e. Link her or his library with the library of everyone else in their group.

   f. Use an Ada compiler to compile her or his modules.

   There is one complication to Step 4.f The modules are represented as Ada packages. As Unit 1 mentioned, Ada packages must be compiled in a particular order. You and your fellow group members must observe the rules in Table 3 as you compile your modules.

Table 3. Compilation Dependencies Among Press 'n Gobble Software Modules

Module Name	Compilation Dependencies
This column lists a module that's dependent on at least one other module.	*This column lists all dependencies.*

5. Your group is now ready to create an executable program. The person who compiled module *main program* must invoke the Ada linker.

You may now execute your program, *using the following input data:* . . . .

After you have finished executing your program, answer the following questions:

1. What communication difficulties did you encounter and how did you overcome them?

2. How would you compare this to your experience with software development?

Exercise 2: German Vending Machine

Repeat Steps 1 through 5, this time creating software for the vending machine Press 'n Gobble will sell in Germany.

When you have built the software, execute your program, *using the following input data:* . . .

Now answer the following questions:

1. How many modules from the first assignment did you reuse without any additional work?

2. How many modules from the first assignment did you reuse by performing different generic instantiations?

PART 3: INSTRUCTIONS FOR LABORATORY

NOTE: This laboratory will ultimately be available for a variety of platforms (IBM PC, Macintosh, etc.). This write-up, which describes how to use the laboratory, is specific not only to each platform but to the institution in which it is used. A separate version of this write-up is therefore needed for each platform, and instructors must tailor it to their own institutions. In all cases, students must:

- Have an Ada compiler
- Be able to create files
- Be able to read files created by other students
- Be able to read a set of files created by the instructor

For simplicity's sake, this write-up is written as if the laboratory were being run in the following environment:

- Each student has access to an IBM PC (or compatible) computer with a 286 or compatible processor.
- Each PC is connected to a file server on drive S.
- Each student has permission to create files in a subdirectory of drive S.
- Each student can create and edit text files (Microsoft's **edit** application or most Pascal compilers would do).
- Each student has access to an Ada compiler.

As in the laboratory descriptions, unresolved issues appear in *italic* text.

This write-up describes how to use your computer to perform the vending machine laboratory exercises. The emphasis is on Steps 4 and 5, since these are the steps that involve using the computer.

1. To perform this step, you must log on to your computer. Then perform each of the following steps:

   a. Create a directory in which to work with your assigned set of modules. For this laboratory, you will work in the directory `S:\adalab`. Create a directory whose name is your last name:

   ```
   C:\>S:
   S:\>mkdir \adalab\yourname
   S:\>chdir \adalab\yourname
   ```

   b. Copy the modules assigned to you modules from Press 'n' Gobble's library of reusable modules to the directory you created in Step 1.a. You will find these modules in the directory `S:\adalab\pressgobble_modules`. The modules are in the following files:

Here we include a table listing all the modules shown in the tables in the laboratory. For each module, we state the file or files holding its code.

```
S:\adalab\yourname>copy \adalab\file1.ada .
```

Perform a copy command for each module assigned to you.

c. If you have been assigned module *X*, you must write a generic instantiation of package *Y* named *Z*. A generic instantiation of *Y* has the form:

```
package Z is new Y(P1 => V1, P2 => V2);
```

Use *text editing application* to create a file named `z.ada` that contains the above line. Use `value1` for `V1` and `value2` for `V2`.

d. Create an Ada library, using the following command:

```
S:\adalab\yourname>mklib
```

e. Link your library to that of other members of your group. For example, if your partners are hername and hisname, issue the following two commands:

```
S:\adalab\yourname>linklib s:\adalib\hername\ada.lib
S:\adalab\yourname>linklib s:\adalib\hisname\ada.lib
```

f. Compile your assigned set of modules. For instance, if you are assigned modules stored in files `x.ada`, `y.ada`, and `z.ada`, issue the following commands:

```
S:\adalab\yourname>ada x.ada
S:\adalab\yourname>ada y.ada
S:\adalab\yourname>ada z.ada
```

Be sure to observe the dependency rules! If you do not, you will get an error message from the compiler:

The error message when a package can't be found.

2. Whoever in your group was assigned to compile the file `main.ada` must now link together all the modules:

```
S:\adalib\yourname>link main
```

This will produce a file called `main.exe`. You can execute this file by typing the command:

```
S:\adalib\yourname>main
```

This page intentionally left blank.

UNIT 4: REUSE
LABORATORY SPECIFICATION

TEACHER NOTES FOR LABORATORY

This section must describe to the instructor how to conduct the laboratory. Topics include:

- *Suggestions on how to make the example seem more realistic by inventing a background tailored to the school in which the course is being taught*
- *Answers to the laboratory exercises*
- *Additional questions the teacher may want to ask students*

This page intentionally left blank.

TEST FOR *MEGAPROGRAMMING IN ADA COURSE*

1. True/False Software developers spend the majority of their time writing code.

2. True/False The majority of software changes result from the need to enhance the software.

3. True/False A programming language can help developers manage change and communication.

4. True/False The only information in a package that is visible to other packages is that contained in the package specification, outside the private part.

5. Abstraction helps developers separate the ________________________________ from the ________________________________.

6. True/False A developer who builds an Ada package must write both the specification and the body before it is useful to other developers.

7. True/False The stepwise refinement design method results in designs that are easy to change.

8. True/False The first decisions you make when following the information hiding design method concern the modules in your program.

9. True/False Software developers usually find similarities between the programs they are developing and programs they have developed previously.

10. Ada ____________ help software developers build packages that other software developers can reuse.

11. Describe the purpose of software design.

12. Using the principles of abstraction and information hiding, design the interface for a module that implements a counter—that is, something another module might use to maintain a count of the number of times some event or situation occurs.

13. Consider the following specification of a package for searching an array of integers:

```
package Integer_Array_Search is
   subtype Array_Index is Integer range 1..1000;
   type Integer_Array is array (Array_Index) of Integer;

   procedure Search_Array(
              Array_To_Search: in Integer_Array;
           Number_Of_Elements: in Array_Index;
        Element_To_Search_For: in Integer;
                Element_Found: out boolean;
               Index_If_Found: out Array_Index
   );
end Integer_Array_Search;
```

Use generics to rewrite this package to be more reusable.

TEST FOR
MEGAPROGRAMMING IN ADA COURSE

TEACHER ANSWERS

1. True/**[False]** Software developers spend the majority of their time writing code.
2. **[True/]**False The majority of software changes result from the need to enhance the software.
3. **[True/]**False A programming language can help developers manage change and communication.
4. **[True/]**False The only information in a package that is visible to other packages is that contained in the package specification, outside the private portion.
5. Abstraction helps developers separate the *essential information* from the *irrelevant details*.
6. True/**[False]** A developer who builds an Ada package must write both the specification and the body before it is useful to other developers.
7. True/**[False]** The stepwise refinement design method results in designs that are easy to change.
8. **[True/]**False The first decisions you make when following the information hiding design method concern the modules in your program.
9. **[True/]**False Software developers usually find similarities between the programs they are developing and programs they have developed previously.
10. Ada *generics* help software developers build packages that other software developers can reuse.
11. Describe the purpose of software design.

    Software design lets software developers decompose a problem into a set of modules. Each of these modules is simpler than the whole. This is necessary to reduce the complexity of the overall system, making it easy for individuals to understand portions of a system.

    Another reason for software design is to break a problem into parts that can be assigned to a set of individuals. In other words, software design is necessary for large programs to ensure that each person on a team has a coherent development assignment.

12. Using the principles of abstraction and information hiding, design the interface for a module that implements a counter—that is, something another module might use to maintain a count of the number of times some event or situation occurs.

The following package specification provides other packages the ability to initialize the count to 0, to increment the count, and to determine its current value. This is the essence of counting.

```
package Counter is
   procedure Set_To_Zero;
   procedure Increment;
   function Current_Value return Integer;
end Counter;
```

13. Consider the following specification of a package for searching an array of integers:

```
package Integer_Array_Search is
   subtype Array_Index is Integer range 1..1000;
   type Integer_Array is array (Array_Index) of Integer;

   procedure Search_Array(
             Array_To_Search: in Integer_Array;
          Number_Of_Elements: in Array_Index;
       Element_To_Search_For: in Integer;
               Element_Found: out boolean;
              Index_If_Found: out Array_Index
   );
end Integer_Array_Search;
```

Use generics to rewrite this package to be more reusable.
You can make the array's base data type and index generic. Note the name change for the second parameter. The old name was based on an ordinal counting system. In the generic version, the array's lower bound might not be 1.

```
generic
   type Item is private;
   type Array_Index is range <>;
package Array_Search is
   type Generic_Array is array (Array_Index) of Item;

   procedure Search_Array(
             Array_To_Search: in Generic_Array;
                Last_Element: in Array_Index;
       Element_To_Search_For: in Item;
               Element_Found: out boolean;
              Index_If_Found: out Array_Index
   );
end Array_Search;
```

SURVEY FOR
MEGAPROGRAMMING IN ADA COURSE

Please answer the following questions. The organization that developed the course material will use this information to improve the course.

1. Do you feel that you understand basic software engineering principles (abstraction, information hiding, and reuse) after taking this course?

2. Do you see value in these principles? Why or why not?

3. Do you see value in using a programming language such as Ada that helps you express these principles? Why or why not?

4. Would you like to learn more?

5. What activity(ies) or example(s) was most helpful to you in understanding the basic software engineering principles?

6. Do you have any other suggestions for how the course can be improved?

SURVEY FOR *MEGAPROGRAMMING IN ADA COURSE*

TEACHER ANSWERS

There are no right or wrong answers on this section. A suggestion for this survey would be to hand it to the students after they have completed the test and give them extra credit if they fill it out and hand it in the next day.

This page intentionally left blank.

Megaprogramming in Ada Course: Laboratory for Unit 3

SPC-95013-CMC

Version 01.01.04

November 1995

Prepared for the
Department of Defense Ada Joint Program Office

Produced by the
SOFTWARE PRODUCTIVITY CONSORTIUM

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

Copyright © 1995, Software Productivity Consortium, Herndon, Virginia. This document can be copied and distributed without fee in the U.S., or internationally. This is made possible under the terms of the DoD Ada Joint Program Office's royalty-free, worldwide, non-exclusive, irrevocable license for unlimited use of this material. This material is based in part upon work sponsored by the DoD Ada Joint Program Office under Advanced Research Projects Agency Grant #MDA972-92-J-1018. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred. The name Software Productivity Consortium shall not be used in advertising or publicity pertaining to this material or otherwise without the prior written permission of Software Productivity Consortium, Inc. SOFTWARE PRODUCTIVITY CONSORTIUM, INC. MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THIS MATERIAL FOR ANY PURPOSE OR ABOUT ANY OTHER MATTER, AND THIS MATERIAL IS PROVIDED WITHOUT EXPRESS OR IMPLIED WARRANTY OF ANY KIND.

PREFACE

This laboratory and teacher notes are part of the *Megaprogramming in Ada Course* (SPC-94094-CMC, version 01.01.04) produced by the Software Productivity Consortium. The course, which is a short course aimed at high school students, consists of four units: software engineering, abstraction, information hiding, and software reuse. The laboratory described in this write-up should be performed at the end of the information hiding unit (Unit 3), preferably after the students have completed Homework Assignment 2 of Unit 3. Assignment 2 deals with concepts of the software that students will use in the laboratory and, therefore, serves as a good introduction to the laboratory material.

This page intentionally left blank.

UNIT 3: INFORMATION HIDING

LABORATORY

In this laboratory, you will compile and execute an implementation of the rational number package from Homework Assignment 2 in Unit 3.

The software you will use is in three files:

- RATNUM.ADA, which contains the package specification for rational numbers.
- RATNUM_B.ADA, which contains the package body for rational numbers.
- READ_SUM.ADA, which contains the procedure Read_Sum_And_Print_Rational_Numbers. This procedure uses the rational number package to read, sum, and print two rational numbers.

You must first compile the software. Perform the following steps:

1. Create a directory called RATNUM on your C drive:

```
C:> md ratnum
```

2. Change your directory to RATNUM:

```
C:> cd ratnum
```

3. Copy the software to your current directory. Your teacher will provide you with the location of the software. For example, if it is located in S:\ADA\RATNUM, you would execute the following command:

```
C:\RATNUM> copy s:\ada\ratnum\*.ada
```

4. Compile the software. You must first compile the rational number package specification, then the rational number package body, then the Read_Sum_And_Print_Rational_Numbers procedure:

```
C:\RATNUM> janus ratnum.ada
C:\RATNUM> janus ratnum_b.ada
C:\RATNUM> janus read_sum.ada
```

   You **must** type the .ADA file name suffix.

   The Ada compiler will print diagnostic information as it compiles each file. This information, not shown here, should indicate that compilation is progressing without errors. If you see any error messages, contact your teacher.

5. Link the software:

```
C:\RATNUM> jlink read_sum
```

Do **not** type a file name suffix.

After you successfully complete these steps, there will be an executable file called READ_SUM.COM in your directory.

You may now execute the software:

```
C:\RATNUM> read_sum
```

You will be prompted to enter two rational numbers. You will be asked for the first number's numerator, then its denominator, then the second number's numerator, and finally the second number's denominator. Enter each number as an integer. For instance, the following shows how to instruct the program to compute 1/7 + 3/5:

```
C:\RATNUM> read_sum
Enter the numerator for the first number: 1
Enter the denominator for the first number: 7
Enter the numerator for the second number: 3
Enter the denominator for the second number: 5
The sum is 26/35
```

Exercises

1. Use the software to compute 3/18 − 10/7.

2. Try using the software to compute 1/1000 + 1000/1.

   a. Why do you think the software fails? (Hint: Examine the Add function to discover how two rational numbers are added.)

   b. The lectures on abstraction and information hiding covered the need to express a module's functionali[illegible] package specification. Based on this laboratory, what else do you think must be in a package specification?

UNIT 3: INFORMATION HIDING

TEACHER NOTES FOR LABORATORY

NOTE: This course contains a simplified version of a planned software laboratory. A more elaborate version may be available at a later date.

You should have received, along with these instructions, a floppy diskette containing the software solving Homework Assignment 2 in Unit 3. This software is almost identical to that shown in the Unit 3 Teacher Notes, with the following exceptions:

- The software on the floppy diskette includes some error-handling code that lets the compiled program terminate gracefully under abnormal conditions.
- The software on the floppy diskette uses a friendlier input paradigm.

The floppy diskette includes the three files of Ada source code discussed in the laboratory write-up: RATNUM.ADA, RATNUM_B.ADA, and READ_SUM.ADA. You must provide each student with a copy of these files. If your computers are linked together on a network and have access to a central file server, you can place them on that server. Each student can then copy the files directly from that server to her or his own computer, as shown in the laboratory write-up. You can also provide each student with a floppy diskette containing the source files and ask them to copy the files from that diskette to their hard drive.

Each student must be able to use an Ada compiler. The instructions in the laboratory write-up use the Janus/Ada compiler from R&R Software, Inc. See the file READ_ME.TXT on the floppy diskette for more information on using this compiler.

EXERCISES

1. Use the software to compute 3/18 − 10/7.

   This simple exercise ensures that students know how to use the program they have just compiled. Be sure they enter the input values correctly: only integers are accepted. Entering anything else will cause the program to stop prematurely.

2. Try using the software to compute 1/1000 + 1000/1.

   a. Why do you think the software fails? (Hint: Examine the Add function to discover how two rational numbers are added.)

      The Add function uses the following formula to add two rational numbers R1 and R2:

```
R.Numerator := R1.Numerator * R2.Denominator
             + R2.Numerator * R1.Denominator;
R.Denominator := R1.denominator * R2.denominator;
```

      Evaluating the first assignment statement using 1/1000 and 1000/1 yields:

$$\begin{aligned} & 1 \times 1 + 1000 \times 1000 \\ = \; & 1 + 1000000 \end{aligned}$$

An examination of the representation of a rational number in the package specification reveals that Numerator and Denominator are values of type Integer. An Integer value can range from −32,768 to 32,767. Since 1,000,000 is greater than 32,767, evaluating the expression causes an overflow. The Ada language requires that a compiler generate code to detect these conditions. This is an instance of language standardization, discussed in Unit 1.

b. The lectures on abstraction and information hiding covered the need to express a module's functionality in a package specification. Based on this laboratory, what else do you think must be in a package specification?

The package specifications shown include information on the procedures and functions, and how to use them. The specifications should also show the ways in which the procedures and functions can fail!