Håkan Dyrhage

Rational Environment Training: Advanced Topics

Document Control Number: 8014A
Rev. 0.0, September 1986
Rev. 1.0, February 1987
Rev. 2.0, July 1987 (Delta)

This document subject to change without notice.

Note the Reader's Comments form on the last page of this book, which requests the user's evaluation to assist Rational in preparing future documentation.

Rational
1501 Salado Drive
Mountain View, California 94043

Rational Environment Training: Advanced Topics

Slides

Contents

Workspace Management
- Introduction 1
- General Aids 4
- More Window Management 20
- More Library Operations 25
- Sessions 35
- Switches 39
- Searchlists 45
- Login Procedures 49
- Job Control 53
- Access Control 57
- Archiving 70
- Basic Networking 78

Environment Command Interface
- Environment Facilities 87
- Objects and Naming 95
- Logs and Profiles 111
- Keymap Modifications 120
- Tool Building 128

Development Mechanisms
- Coding Aids 136
- Subunits 148

Debugger Operations 155
Link Operations 166
Incremental Operations 170
Testing 180
Reusable Components 185

Basic Subsystems and Configuration Management
Project Management Issues 190
Project Structuring with Subsystems 206
Subsystem Construction 222
Basic Development Methodology 232
Source Reservation with CMVC 243
Parallel Development with Subpaths 248

Seminar Outline

Workspace Management
- Introduction
  - General Aids
  - More Window Management
  - More Library Operations
  - Sessions
  - Switches
  - Searchlists
  - Login Procedures
  - Job Control
  - Access Control
  - Archiving
  - Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Course Objectives

- Introduces additional Environment concepts and mechanisms
- Broadens the experience of users with various Environment features and facilities
- Provides an introduction to project development using Rational Subsystems and CMVC

Course Materials

- *Rational Environment Training: Advanced Topics*
- *Rational Environment Basic Operations*
  - — Sequence of steps, commands, and keys used to perform common Environment functions
  - — Basic Keymap
- *Rational Environment Reference Manual* (11-volume set)
- Reference Summary (Volume 1 of Reference Manual)
  - — List of Environment commands
  - — Naming symbols and attributes
  - — Keymap
  - — Master Index

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Macros

- Provide a way of repeating a series of keystrokes
- Can be bound to any single key or key combination
- Are created with
  - — Begin the macro: Mark - Begin Of
  - — Enter all keystrokes in the macro
  - — Complete the macro: Mark - End Of
- Can be used in two ways
  - — Execute the macro: Mark - Promote or Meta X
  - — Bind the macro to any key: Mark - Definition and then press the key to which to bind the macro
- Can be used with argument prefix keys to execute the macro several times

Macros (cont.)

- Can be saved and reused: `Macro.Save`
- Example: A macro that displays both parts of the Ada unit at the cursor

Exercise: Using Macros

1. Build a macro that comments out a line of code in an Ada unit and adds your initials to identify your change. The macro should comment out the entire line regardless of the cursor's position on the line. The cursor should be moved to the next line in the Ada unit so that the macro can be executed several times to comment out several lines without additional keystrokes.

2. Test this macro on a unit in the **Experiment** world. Make sure the macro can be executed several consecutive times to comment out several lines.

3. Test the macro using an argument prefix key or keys. Does pressing [numeric 3] - [Mark] - [Promote] comment out three successive lines?

4. Bind the macro to [F1]. Verify that it works singly and with argument prefixes.

Exercise: Using Macros (cont.)

5. Return the unit to its original configuration.

Optional Exercise: Using Macros

1. Build a macro that removes the comment characters and your initials from a line in an Ada unit. The macro should remove the leading comment characters regardless of the cursor's position on the line. The cursor should be moved to the next line in the Ada unit so that the macro can be executed several times to remove the comment characters from several lines without additional keystrokes.

2. Test this macro on a unit in the `Experiment` world. Make sure the macro can be executed several consecutive times to remove comments from several lines.

3. Test the macro using an argument prefix key or keys. Does pressing [numeric 3] - [Mark] - [Promote] remove the comments from three successive lines?

Optional Exercise: Using Macros (cont.)

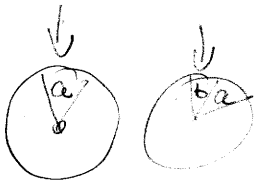
4. Bind the macro to Shift F1. Verify that it works singly and with argument prefixes.

5. Verify that both the original macro on F1 and the macro on Shift F1 work. Which macro is executed when you press Mark - Promote?

6. Return the unit to its original configuration.

Mark Stack

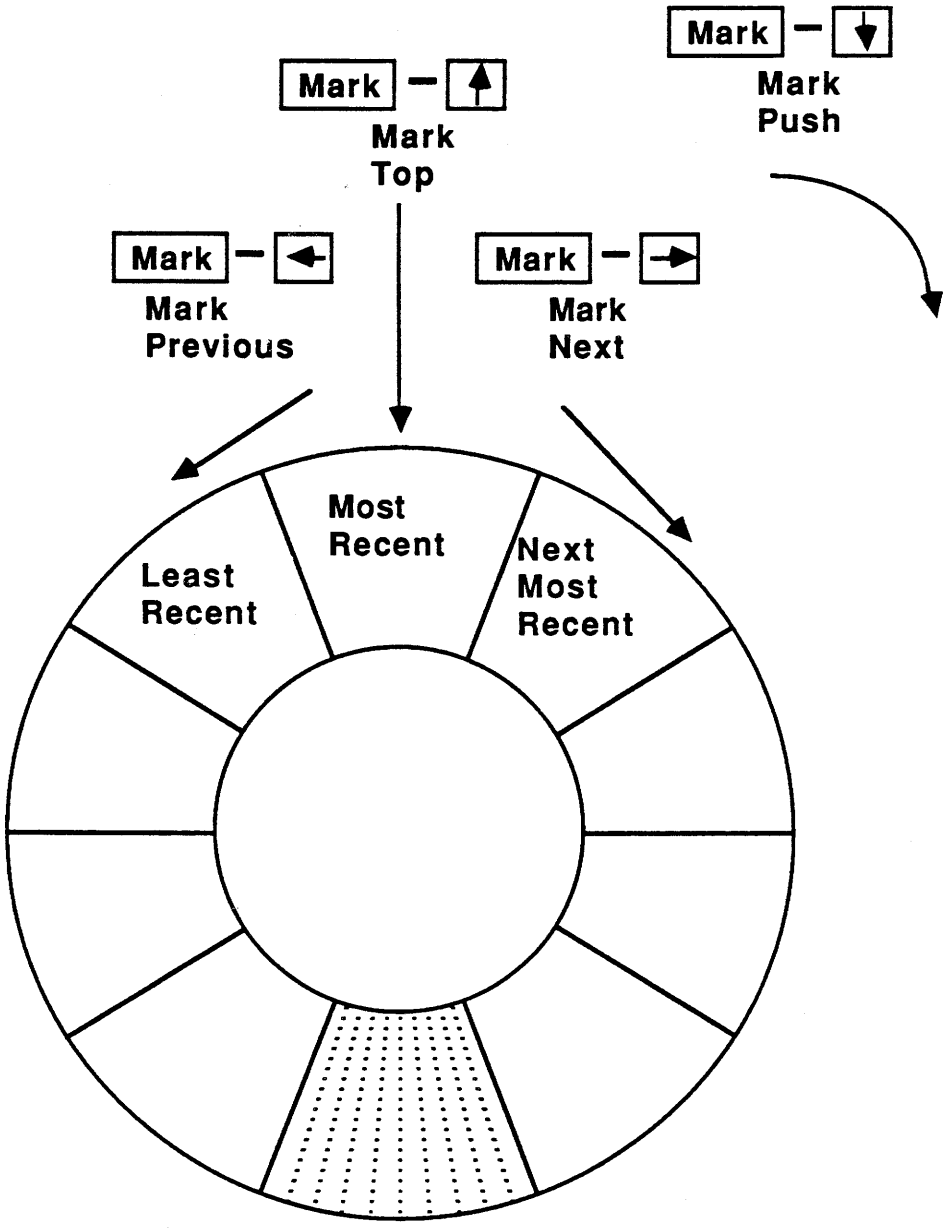
- Is often used to save and return to cursor locations in windows
- Contains the 100 most recently marked cursor locations in a ring buffer
- Commands
  - Explicitly push a mark onto the stack: Mark - ↓
  - Go to the mark at the top of the stack: Mark - ↑
  - Go to the next or previous mark in the stack: Mark - → or Mark - ←
  - Consult the Keymap for commands that edit the stack



Reference summary C-20 Purchase Mark

Purchase Region -- see comment proc for forse @else.

Mark Stack (cont.)



Exercise: Using the Mark Stack

1. Go to your home library and place the cursor at the beginning of the image.
2. Push a mark at the top of the image. mark ↓
3. Move down in the image several lines. Push another mark on the stack. MARK ↓
4. Cycle through the marks on the stack. Notice that there are only two marks on the stack and the remaining 98 stack locations are not used. MARK ←
5. Remove the window from the screen by pressing Window - D. Remember that this operation will leave the window in the Window Directory.
6. Return to the top mark by pressing Mark - ↑. Notice that the window containing the mark returns to the screen.

Exercise: Using the Mark Stack (cont.)

7. Delete the window by pressing Object - X or Object - G. This operation removes the window from the Window Directory.
8. Return to the top mark. Notice that the mark previously saved has been removed from the stack because the window has been deleted.

Region (Hold) Stack

holder dole of text
alt do deletes gutles go stack!

- Can be used to recover deleted regions of text
- Contains the 100 most recently used regions arranged in a ring buffer
- Also contains regions implicitly pushed onto the stack by all line and region deletions
- Commands
  - — Explicitly push a region onto the stack: Region - ↓
  - — Recover the top region from the stack: Region - ↑
  - — Recover the next or previous region from the stack: Region - → or Region - ←
  - — Consult the Keymap for commands that edit the stack

Exercise: Using the Region Stack

1. In the `Experiment` world, edit the `Line_Copy'body` unit.
2. Select the loop in the `Locate_Comment` function. Use either region or object keys. Select only the loop, not the whole function.
3. Push this region onto the region stack.
4. Move to the statement region of the `Strip_Blanks` function.
5. Recover the top of the region stack by pressing Region - ↑.

   Notice that the region is inserted into the function at the current cursor position.
6. Return to the `Locate_Comment` function and select just the *if* statement. Push it onto the region stack.

Exercise: Using the Region Stack (cont.)

7. Delete the *return* statement from the function using line, region, or object operations.

   Remember that deletions are automatically pushed onto the region stack.

8. Move to the statement region of the `Has_Comment` function.

9. Recover that top of the region stack. Then, without moving the cursor, recover the next region from the stack. Continue cycling through the regions on the stack, noticing that each replaces the previous region at the cursor position.

   If the cursor is moved in some way, then the region currently displayed from the stack will not be replaced by the next region recovered from the stack. Notice that regions recovered from the stack are not removed from the stack.

Screen Stack

- Can be used to save and return to screen configurations after browsing
- Contains the 100 most recently used screen configurations (windows and placement) in a ring buffer
- Commands
  - — Explicitly push a screen onto the stack: Control Meta S ift ↓
  - — Recover the screen on the top of the stack: Control Meta Shift ↑
  - — Recover the next or previous screen in the stack: Control Meta Shift → or Control Meta Shift ←
  - — Consult the Keymap for commands that edit the stack

Reference Summary / C-29
Package Screen

Exercise: Using the Screen Stack

1. Build a macro that first pushes the screen image and then does a definition. Bind it to F2.

   Build another macro that first pushes the screen image and then does an enclosing object. Bind this to F3.

2. Now browse around using the macros.

3. Return to the first screen image by pressing Control Meta Shift →.

4. Go back to the last screen and "unwind" through your browsing by pressing Control Meta Shift ← several times.

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Window Operations

- Controlling window replacement
  - — If the window is unlocked, make the current window the next window to be replaced: [Window] - [Edit] or [Window] - [Demote]
  - — If a window has been demoted, return it to a normal window and make another window the next one to be replaced: [Window] - [Promote]
- Removing a lock on a window
  - — Unlock a locked window: [Window] - [Demote]
  - — Transpose a locked window: [Window] - [T]
  - — Explicitly remove a locked window from the screen: [Window] - [D]

Exercise: Using Window Operations

This exercise should be completed without locking any windows.

1. Go to your home library.
2. Go to the `Experiment` world without creating another window by pressing [Window] - [Edit] - [Definition].

   Notice that the window's contents are replaced with the new world.
3. Look at the package spec of `Line` by pressing [Definition In Place].

   Again, the window's contents are replaced.
4. Return to the home library without creating another window by pressing [Enclosing In Place].

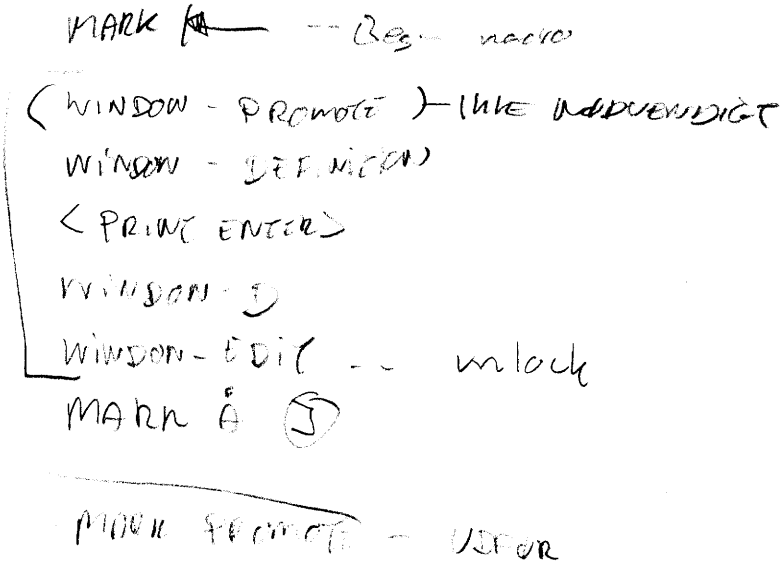
Window Directory Operations

- Go to the image pointed to by the cursor: Definition
- Save the selected image: Enter ex vee lock out
- Save all images (without selection): Enter
- Promote the selected object: Promote
- Demote the selected object: Demote
- Delete the selected image or window: Object - D or Object - K
- Edit the selected image: Edit
- Remove the Window Directory without choosing another window: Object - G or Object - X

Exercise: Using Window Directory Operations

1. Create a macro that saves the current window location, enters the Window Directory, saves all images, and returns to the previous window location.

2. Test the macro by making minor editing changes to units in the **Experiment** world. Verify that the changes are saved by the macro.



Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Library Commands

- Copy a library object from one place to another: `Library.Copy`
- Move a library object to another location: `Library.Move`
- Rename a library object: `Library.Rename`
- Delete a library object or set of objects: `Library.Delete`
- Freeze an object, preventing inadvertent changes: `Library.Freeze`
- Display the resolution of a naming expression: `Library.Resolve`
- Set the number of deleted versions retained in a library: `Library.Set_Retention_Count`
- Consult package `Library` for other operations

Library Object Operations

- Save the selected object: Enter
- Promote the selected object: Promote
- Demote the selected object: Demote
- Change the state of the selected object: Source, Install, or Code
- Edit the selected object: Edit
- Delete the selected object: Object - D or Object - K
- Copy a selected object to the cursor position: Object - C

Library Object Operations (cont.)

- Directly execute the selected coded procedure: Promote
  - — Creates a Command window for procedures with parameters without defaults
  - — Directly executes parameterless procedures or procedures with all default parameters
- Directly execute under Debugger control: Meta Promote ESC ^1
  - — Must be parameterless or have a complete set of defaults

Library Display Operations

- Library displays can contain units and versions of units not shown by default
- There are eight display combinations
  - — Include deleted units or not
  - — Include subunits or not
  - — Include previous versions or not
- Midpoint is the default display defined by your session switches
- Commands to cycle the display
  - — Cycle down to another display combination: Object - .
  - — Cycle up to another display combination: Object - !

Library Display Operations (cont.)

- Display combinations
  - — `{versions}`: Show all nondefault versions and deleted library units and subunits
  - — `versions`: Show all nondefault versions of library units and subunits (but not deleted units)
  - — `{lib vers}`: Show all nondefault versions and deleted library units (but not subunits)
  - — `lib vers`: Show library units and all nondefault versions (but not subunits or deleted units)
  - — `{units}`: Show library units, subunits, and deleted units
  - — (blank): Default

Library Display Operations (cont.)

— `{lib units}`: Show all library and deleted library units (but not subunits or nondefault versions)

— `lib units`: Show all library units (but not subunits or deleted units)

Library Display Operations (cont.)

- Library displays can show one of three sets of information for each listed object
  - — Basic display with object's name
  - — Standard information display with object's class
  - — Miscellaneous information display with update information, state of Ada units, size, retention count, and frozen status
- Visible library display information is controlled by
  - — Cycling between sets of information: ~~Explain~~ object ?
  - — Setting session switches for library operations

Exercise: Using Library Operations

1. Go to the `Experiment` world and use Explain to find out if the `Program_Profile` program is coded. Make the program coded if it is not.
2. Directly execute the main program by selecting the `Program_Profile` subprogram and pressing Promote.
3. Enter the unit `Test_Input_1` at the first prompt and press Promote again.
4. Demote the unit `Line'body` by selecting it and pressing Source Unit.
5. Cycle through the three different library displays.

   Notice that, with the cursor on a particular unit, only the display of that unit is changed.

Exercise: Using Library Operations (cont.)

6. Move the cursor to the top of the library display. Adjust the library display to show the state of the Ada units in the library.

   Notice that the display of the entire library is changed.

7. Delete the selected unit `Line'body` using [Object] - [D].

8. Cycle through the different levels of hiding in the library.

   Notice that the deleted object reappears at some levels of hiding.

9. Undelete the deleted object `Line'body` with [Object] - [U].

   Notice that the display of the unit `Line'body` changes.

10. Promote the previously deleted unit directly to coded.

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Sessions

- Users can create several sessions, each with different characteristics
  - — Each session is an object in a user's home library
  - — Users choose a session when logging in
- Sessions can be tailored to define
  - — Screen characteristics: scrolling behavior, number of windows, Message window size
  - — Command response characteristics: error responses, log generation
  - — Image characteristics: content of library image displays
  - — Initial workspace configuration: command visibility, initial library destination

Uses of Sessions

- Organize your workspace on an individual or project basis
- Tailor Environment behavior and setup for specific lifecycle phases of a project
- Customize your individual workspace
- Provide session-specific behavior in login procedures

User-Tailored Sessions

- Sessions are created and controlled by the Environment
  - — Are created when logging into a new session
  - — Can be created by the user: `Operator.Create_Session`
  - — Have default characteristics initially
- Sessions can be tailored by the user
  - — Each characteristic is tailored with facilities from packages `Switches`, `Log`, `Profile`, and `Search_List`
  - — Session characteristics are saved for use in all subsequent logins

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches SG, SZ i Reference Summary / Symbols + Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Characteristics of Switches

- Provide a way of tailoring the behavior of various Environment facilities
- Consist of two classes
  - — *Library switches* affect library operations, including compilation in that library and Ada unit formatting
  - — *Session switches* affect screen and command behavior

Operations on Switches

- Create a library switch file in the current library: `Switches.Create`
- Edit library switches: `Switches.Edit`
- Apply a library switch file to a library object: `Switches.Associate`
- Edit session switches: `Switches.Edit_Session_Attributes`
- Change a selected switch: Edit toggle value true/false
- Get help on the current switch: ~~Explain~~ object ?
- Save the changes to the switches: Enter

Exercise: Editing Session Switches

1. Edit your session switches with
   `Switches.Edit_Session_Attributes.`
2. Press ← when you are on the left margin. Notice that the Environment beeps or flashes.
3. Select the `Beep_On_Errors` switch and press Edit. Save your change by pressing Enter.
4. Now press ← again. Note that there is no beep this time.

Exercise: Editing Session Switches (cont.)

5. Search for the `Window_Command_Size` switch. This controls the size of Command windows when they are created.
6. Select and edit the switch. Note that a two-line Command window is created with the `Change` command.
7. Change the parameter to `4`. Promote the command and save your change.
8. Create another Command window somewhere and notice its new size.

Extra:

9. Edit the switch
   Library_Std_Show_Unit_State = true
   Save the changes
10. All [illegible] after step 9?
    [illegible]
    if you want your "old" [illegible] do a log out/log in.

Exercise: Editing Library Switches

1. Go to the `Experiment` world in your home library.
2. Create a switch file in that world and associate it with the `Experiment` world using the `Switches.Create` command.
3. Associate the new switch file with the `Experiment` world with the `Switches.Associate` command.
4. Edit the switch file.
5. Change the `Major_Indentation` switch from `4` to `2` and save your change.
6. Get the definition of the body of the `Line` package.
7. Select the entire body and execute the `Library.Reformat_Image` command.
8. Try changing other format switches.

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Characteristics of Searchlists

- Define visibility in Command windows
- Specify an ordered set of libraries to search for name resolution
- Are unique for each session
- Systemwide default:

```
$`
!COMMANDS
!LOCAL
!COMMANDS.ABBREVIATIONS`
!MACHINE.RELEASE.CURRENT.COMMANDS`
!IO
!TOOLS
!TOOLS.NETWORKING
```

  - — $ = Search current context
  - — ` = Also search through the library's links
  - — Located in `!Machine.Search_Lists.Default`

Searchlist Modification

- Modify a searchlist to provide visibility to project or user-defined tools in Command windows
- Edit a session's searchlist:

  `Search_List.Show_List`
- Modify the searchlist with common editing commands
  - — Prompt for and insert a searchlist entry at the cursor position: Object - I
  - — Delete a selected searchlist entry: Object - D
  - — Get the definition of a searchlist entry: Definition

Exercise: Using Searchlists

1. In a Command window, execute the command: `Text_Io.Put_Line ("test");`.

   Notice that you do not have to *with* `Text_Io` for this to work. This is true because `Text_Io` is referenced in your searchlist.

2. Edit the searchlist.

3. Remove the entry for `!Io`.

4. Attempt to reexecute the command.

   The attempt fails because `Text_Io` is no longer visible in the searchlist.

5. Replace the `!Io` entry in your searchlist. THIS IS IMPORTANT. GET YOUR INSTRUCTOR TO HELP YOU IF NECESSARY.

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

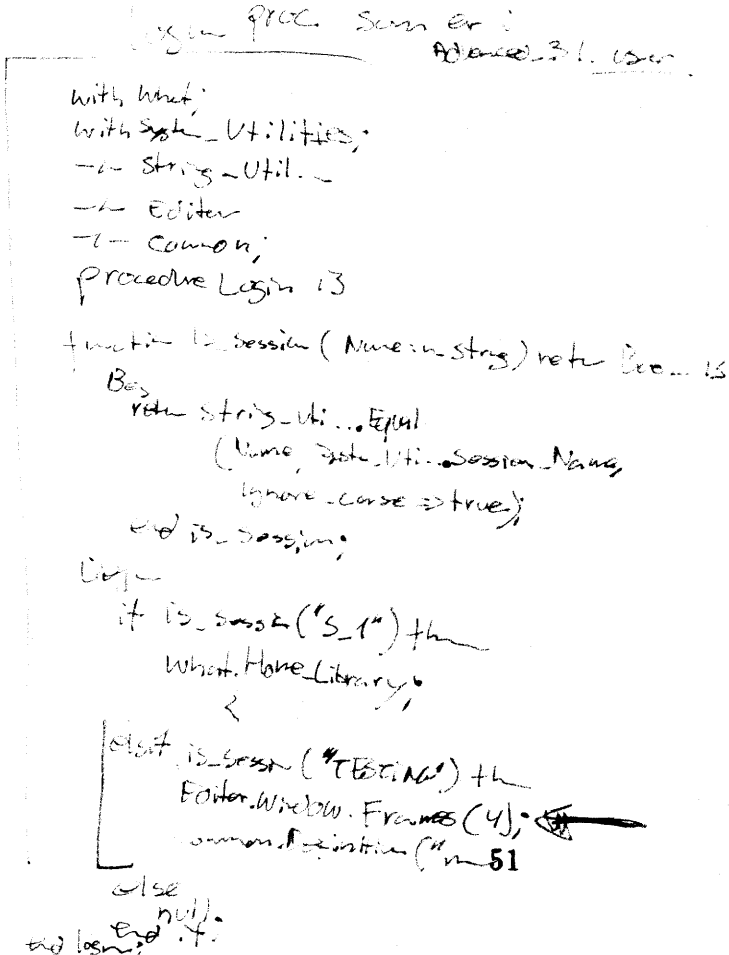
Basic Subsystems and Configuration Management

Login Procedure Basics

- Procedure executes automatically whenever you log into the Environment
- Basic method
  - — Create a parameterless procedure called `Login` in your home library
  - — Enter any Ada code and use any Environment facilities
  - — Promote the procedure to coded when complete

Some Uses of Login Procedures

- Call different setup commands based on the session name
- Examples
  - Set the number of windows
  - Display any status from overnight compiles or tests
  - Go to your project library
  - Change your keymap



Exercise: Using Login Procedures

1. Modify the login procedure in your home library. For a new session called **`Testing`**, create the following initial conditions:
   - The number of screen frames is 4.
   - The **`Program_Profile_System`** library is displayed.
   - A Command window appears off the **`Program_Profile_System`** library.
   - The users logged into the Environment are displayed.
2. Log out and back in with the new **`Testing`** session and verify that your new login procedure works.

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

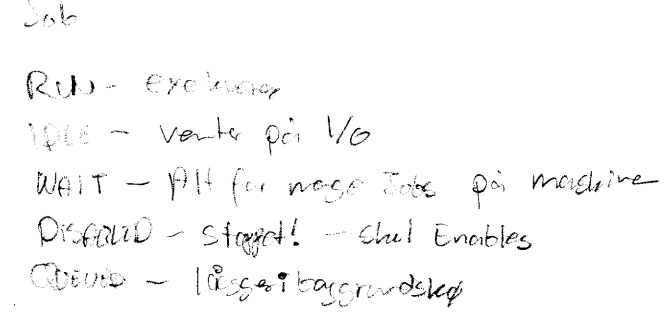
Basic Subsystems and Configuration Management

Basics of Job Control

- Any job can be created and disconnected to continue execution in the background
  - — Disconnect from any job: Control G
  - — Job becomes a background job; different scheduling applies
- Disconnected jobs can be reconnected
  - — Reconnect to specific job: Job Connect and enter job number
  - — Job number is available from the disconnect message
  - — Job scheduling returns to foreground characteristics
- Currently connected or most recently disconnected job can be killed: Meta G

Basics of Job Control (cont.)

- Procedures can be constructed to run as background jobs by including a call to `Job.Disconnect`
- Dynamic display of all jobs: `What.Jobs`
  - — Display idle or disabled jobs: Object - ! or Object - .
  - — Kill a selected job: Object - D
  - — Terminate the jobs display: Object - X
- You can log off with background jobs running
  - — Must redirect I/O to files: `Log.Set_Output`



Exercise: Controlling Jobs

Use the *Rational Environment Basic Operations*, the Rational Environment Reference Summary, and your notes to assist you.

1. Determine the users and jobs currently running on the system through the jobs display.

2. Start the following job: `delay 1000.0`.

   While watching the jobs display:

   — Disconnect from the job.

   — Reconnect to the job.

   — Disable the job.

   — Reenable the job.

   — Kill the job.

   — Eliminate the jobs display.

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

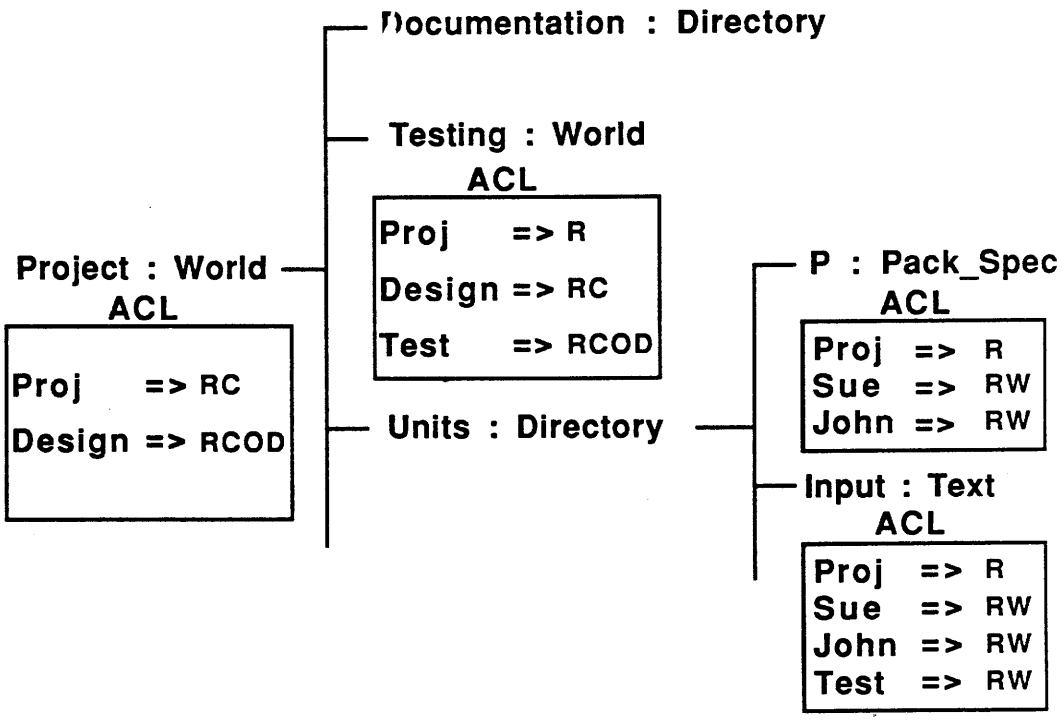
Basic Subsystems and Configuration Management

Access Control

- Controls access to worlds
- Controls access to individual objects
  - — Ada units
  - — Text and binary files
- Controls execution of programs and commands
- Can be used to
  - — Isolate projects from one another on the same machine
  - — Exclude unauthorized users from a project or machine
  - — Prevent accidental modifications or deletions
  - — Enforce design decisions
  - — Limit access to certain privileged commands

Access Lists (ACLs)

- ACLs are the basis of access control
- Each world and each version of Ada or file objects has its own ACL
- ACLs consist of a list of groups and the access rights granted to each group



Groups

- A group consists of a list of usernames
  - — Groups cannot reference other groups
- Every user belongs to at least one group, which has the same name as the user
- Having access to a world or object means that the user belongs to at least one group in the ACL that has been granted the required access rights
- Access rights for a job are determined by the access rights granted to the owner of the job

Special Groups

- Public: All users on the local machine
- Network_Public: All users on the local machine and all users on other machines
- Privileged: Users in this group can run jobs in *privileged* mode, which disables all access checks
- Operator: Users in this group have *operator capability*, which allows execution of system management commands such as enabling terminals, creating user accounts, and so on

Access to Worlds

- Four kinds of access: owner, read, create, delete
- Owner access
  - — Can change the ACL for the world and for any objects in the world
  - — Can change the links for the world
  - — Can associate/dissociate a switch file with the world
  - — Can freeze/unfreeze the world or objects in the world

Access to Worlds (cont.)

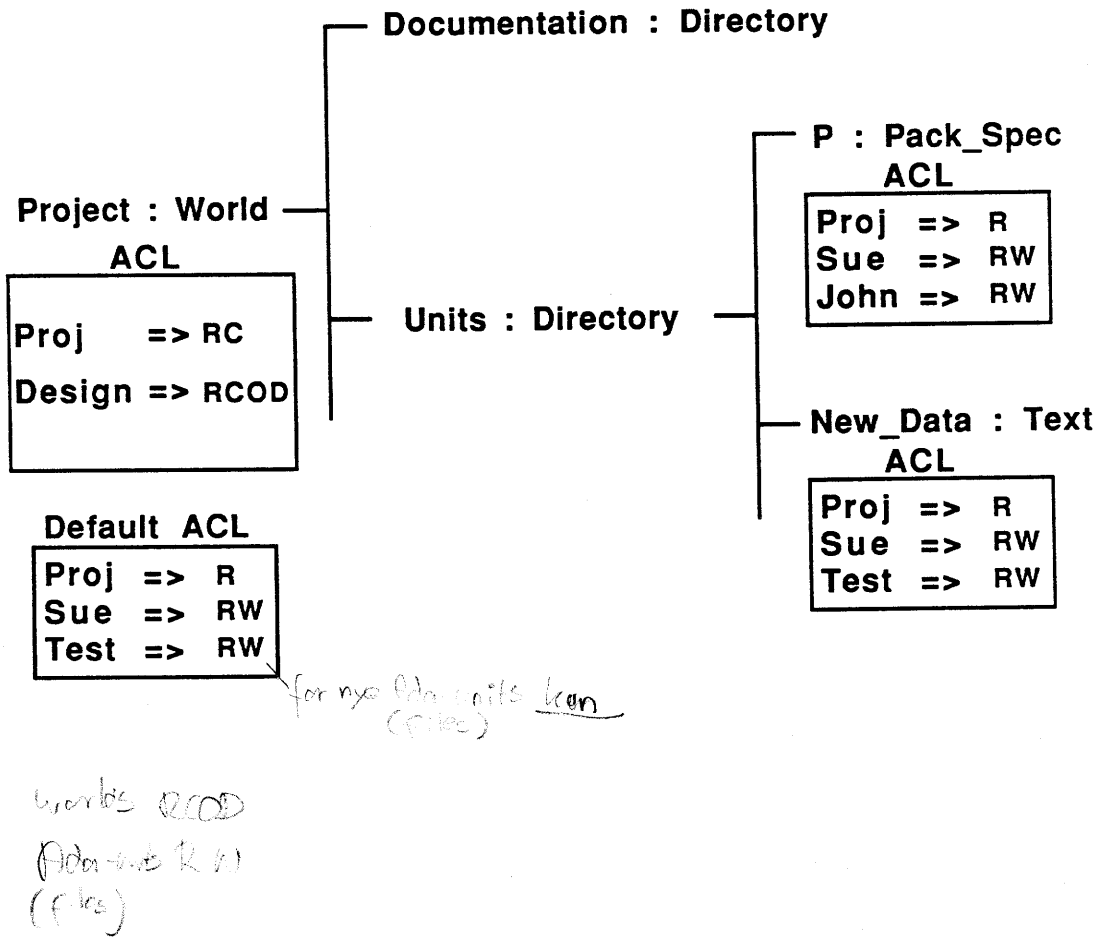
- Read access
  - — Can look at the world and its contents
  - — Must have read access to every library in the world's full pathname
  - — Must have read access to the switch file associated with the world
  - — If read access is not granted, Environment acts as if the world does not exist
- Create access
  - — Can create new objects in the world
  - — Can create new versions of existing objects in the world
- Delete access
  - — Can delete a world

Access to Objects

- Two kinds of access rights: read, write
- Read access
  - Can look at an object
  - Can demote Ada units
  - Must have read access to every library in the object's full pathname
  - If read access is not granted, Environment acts as if the object does not exist
  - If read access is not granted, the unit cannot be executed
- Write access
  - Can make changes to the object
  - Can delete the object
  - Can promote an Ada unit

Default ACLs for New Objects

- Each world has a *default* ACL
  - — Defines the initial ACL for new non-library objects created within the world
- New versions of existing objects inherit their ACL from the previous version



Default ACLs for New Objects (cont.)

- New worlds inherit their ACL from the ACL of the enclosing world and their default ACL from the default ACL of the enclosing world
- ACLs (and default ACLs) for user home libraries are created by concatenating
  - Read and write access granted to the newly created user group *with*
  - The machine default defined in `!Machine.User_Acl_Suffix`

Commands

- Display the ACL for an object: `Acl.Display`
- Set the ACL of an object: `Acl.Set`
- Set the default ACL of a world: `Acl.Set_Default`
- Add a group or list of groups to the ACL of an object: `Acl.Add`
- Create an ACL group: `Operator.Create_Group`
- Display the users who are members of a group: `Operator.Display_Group`
- Add a user or list of users to a group: `Operator.Add_To_Group`

Exercise: Using Access Control

1. Go to your home library.
2. Try to delete the world `Acl_World` in your home library. Notice the error message.
3. Display the ACL for `Acl_World` with the command:

```
Access_List.Display (For_Object => "Acl_World");
```

4. Notice that `Acl_World` has as an ACL that does not provide delete access
5. Give yourself access to `Acl_World` with the command:

```
Access_List.Set
    (To_List => "Network_Public => RWCOD",
     For_Object => "$.Acl_World",
     Response => "<PROFILE>");
```

6. Display the ACL for `Acl_World` again. Notice that the ACL has been changed.

Exercise: Using Access Control (cont.)

7. Now try to delete the world `Acl_World`. Notice that this time you can delete the world.

8. Find out which users belong to the group `Network_Public` with the command:

```
Operator.Display_Group
    (Group => "Network_Public",
     Response => "<PROFILE>");
```

multiuse

9. Find out which groups you belong to with the command:

```
Operator.Display_Group
    (Group => "Advanced_N",  -- Your username
     Response => "<PROFILE>");
```

Group Advanced_N
-- Network_Pub
-- Public

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

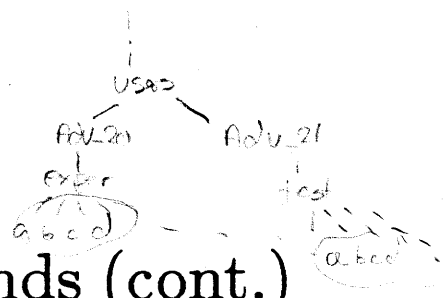
Basic Subsystems and Configuration Management

Archiving

- Provides mechanism for transferring entire library structures in original form
  - — Move structure to another place on the same machine
  - — Move structure to another R1000 via network or tape
- Provides archive storage on tape
  - — Structures can be deleted and restored later if necessary
- Provides additional backup mechanism with incremental recovery

Archive Commands

- `Archive.Save`
  - Saves a set of objects onto tape or into a library
  - `Options` parameter specifies tape format and object filters
- `Archive.Restore`
  - Restores one or more objects from a library or tape
  - `Use_Prefix` and `For_Prefix` parameters are used for placing the objects in a different place in the library structure
  - `Options` parameter specifies overwrite properties, promotion of Ada units, and resulting access control



Archive Commands (cont.)

- **`Archive.Copy`**

  — Transfers objects to new location without using a tape or intermediate library

  — New location can be on another R1000

  — **`Use_Prefix`** and **`For_Prefix`** parameters are used for placing the objects in a different place in the library structure

Options

- Many commands, including `Archive` commands, have an `Options` parameter
  - — Options are string parameters listing the options active for the command
  - — Options reduce the number of parameters on commands when many options are possible
  - — Individual options are separated by a comma or space
- Important save options
  - — `Nonrecursive`: Specifies that only the listed units and not the objects contained within them are to be saved
  - — `After = <time_expression>`: Specifies that only those objects modified after the specified time are to be saved

Options (cont.)

- Important restore options
  - `Promote`: Specifies that restored objects be promoted to their original state
  - `Overwrite = All_Objects`: Specifies that all specified objects be restored
  - `Overwrite = New_Objects`: Specifies that only specified objects that do not already exist are to be restored
  - `Replace`: Specifies that the object is to be replaced even if other objects need to be unfrozen or demoted
  - `Object_Acl = <acl_value>`: Specifies that the ACL for the object be set to the specified list
  - `Become_Owner`: Specifies that the ACLs be modified such that the restorer becomes the owner

Exercise: Using Archive

1. Create a new directory called `Archive` in your home library.

2. Save the `Program_Profile_System` world into the `Archive` directory with the following command:

```
Archive.Save
  (Objects => "Program_Profile_System",
   Options => "R1000",
   Device => "Archive",
   Response => "<PROFILE>");
```

3. Restore the world with a new name and promote the Ada units to their original state with:

```
Archive.Restore
  (Objects => "?",
   Use_Prefix => "!Users.Advanced_N.New_System",
   For_Prefix => "!Users.Advanced_N.Program_Profile_System",
   Options => "R1000, Promote",
   Device => "Archive",
   Response => "<PROFILE>");
```

Exercise: Using Archive (cont.)

4. Restore the `Program_Profile_System` world inside the `Archive` directory with:

```
Archive.Restore
  (Objects => "?",
   Use_Prefix => "!Users.Advanced_N.Archive",
   For_Prefix => "!Users.Advanced_N",
   Options => "R1000",
   Device => "Archive",
   Response => "<PROFILE>");
```

Seminar Outline

Workspace Management
- Introduction
- General Aids
- More Window Management
- More Library Operations
- Sessions
- Switches
- Searchlists
- Login Procedures
- Job Control
- Access Control
- Archiving
- Basic Networking

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

Telnet

- Provides a virtual terminal interface to other machines on the network
- Establish a session with another machine:
  `Telnet.Connect`

  Normally only the remote machine name is provided

  All defaults are derived from the session switches

  Important parameters:

  — `Remote_Machine`: Specifies network name of desired machine

  — `Session`: Specifies session when creating multiple Telnet sessions

  — `Escape`: Specifies character used to return to the original session; Control [is the default

Telnet (cont.)

- Break the Telnet session: `Telnet.Disconnect`

  Important parameters:

  — `Remote_Machine`: Specifies machine name of the connection

  — `Session`: Specifies session number if multiple sessions have been created

File Transfer

- File Transfer Protocol (FTP) provides the ability to move files from one machine to another
- Basic method
  - — Connect and log into remote machine
  - — Transfer files to/from remote machine
  - — Disconnect from remote machine
- Combined operations are available
- Session switches can be used to set default parameters for
  - — Remote machine name
  - — Login name and password
  - — Password prompting (available for additional security)

File Transfer (cont.)

- Establish a TCP/IP connection to a remote machine: `Ftp.Connect`

  Important parameters:

  — `To_Machine`: Specifies name of remote machine

  — `Auto_Login`: Specifies whether you want to log in (should be set to true)

  — `Username`: Specifies remote machine login name

  — `Password`: Specifies remote machine password

File Transfer (cont.)

- Move a file from the local machine to the remote machine: `Ftp.Store`

  Important parameters:

  — `From_Local_File`: Specifies name of local file

  — `To_Remote_File`: Specifies name of new file

- Move a file from the remote machine to the local machine: `Ftp.Retrieve`

  Important parameters:

  — `From_Remote_File`: Specifies name of remote file

  — `To_Local_File`: Specifies name of new file

- Log out from the remote session and break the FTP connection: `Ftp.Disconnect`

File Transfer (cont.)

- Other commands
  - `Ftp.Put`: Same as `Ftp.Store` but also does connection, login, and disconnect after the transfer
  - `Ftp.Get`: Same as `Ftp.Retrieve` but also does connection, login, and disconnect after the transfer
  - `Ftp.Get_List`: Same as `Ftp.Get` but transfers a specified list of files

Exercise: Transferring a File

Transfer a file to yourself by looping back over the network to the same machine you are currently logged into.

1. Go to your home library and create a Command window.
2. Enter the command: `Ftp.Get` Complete
3. Enter `!Users.>>Your_Username<<.Login'body` for the `Frc _Remote_File` parameter.
4. Enter `Login_Body` for the `To_Local_File` parameter
5. Enter appropriate values for the `Remote_Machine`, `Username`, and `Password` parameters.
6. Execute the command.
7. Compare the image of the Ada unit `Login'body` and the `Login_Body` file.

Rehosting Software

- Move files to R1000
  - — Multiple calls to `Ftp`
  - — Copy files from ANSI-labeled tapes: `Tape.Read`
- Convert the files into Ada units: `Compilation.Parse`
  - — Use wildcard to specify all files
  - — Use another library for the `Directory` parameter to avoid name conflicts
- Compile the software: Code (This World)

Seminar Outline

Workspace Management

Environment Command Interface

- Environment Facilities
  Objects and Naming
  Logs and Profiles
  Keymap Modifications
  Tool Building

Development Mechanisms

Basic Subsystems and Configuration Management

Environment Interfaces

- Common facilities are in libraries in world `!`
  - — Most commands are in world `!Commands`
  - — I/O facilities are in world `!Io`
  - — Tools and utilities are in world `!Tools`
  - — LRM-specified facilities other than I/O are in world `!Lrm`
- Interfaces for other products (for example, Networking) are in the worlds `!Commands` or `!Tools`

I/O Facilities

- Package `Io` provides streamlined operations similar to package `Text_Io`
  - — Append operation
  - — Standard error file in addition to standard input and output files
  - — Preinstantiated `Boolean`, `Integer`, and `Float` I/O operations
  - — Functional form of `Get_Line`
- Package `Polymorphic_Sequential_Io` provides reading and writing of several types of data into the same file
  - — Generic operations read or write multiple user-defined data types into a single file
  - — File must be read in exactly the same order in which it is written

I/O Facilities (cont.)

- Package `Pipe` provides message passing between jobs
  - — Higher performance than reading and writing files
  - — Implicit queuing of messages
  - — Correct synchronization properties (can be opened by two jobs simultaneously)
- Package `Window_Io` provides programmatic control of I/O to windows
  - — Allows creation of menus, custom displays, report templates, and so on
  - — Can control cursor placement and contents of window
  - — Allows reading of keystrokes from the terminal

Tools

- Package `String_Utilities` augments Ada's string-handling facilities
  - — Case conversion
  - — Conversion between numeric values and strings
  - — Substring location
- Packages `Unbounded_String` and `Bounded_String` provide dynamic-length string handling
  - — Conversion between strings and variable strings
  - — Copy, move, and append
  - — Insert, delete, and replace of characters or substrings

Tools (cont.)

- Generic packages `List`, `Set`, `Map`, `Queue`, and `Stack` provide abstract type operations
- Package `Table_Formatter` displays a formatted table of data
- Package `Table_Sort_Generic` sorts a table containing any type of data
- Package `Debug_Tools` provides programmatic access to Debugger functions to
  - — Set a breakpoint in the program
  - — Display messages in the Debugger window
  - — Specify symbolic task names that can be referenced when debugging
  - — Provide user-defined display of object values
  - — Get the name of any raised exception

Tools (cont.)

- Package `System_Utilities` provides access to system information
  - — Get CPU time consumed by job
  - — Get current user or session name
- Package `Time_Utilities` provides additional facilities for manipulating time
  - — Manipulate durations
  - — Convert between time formats and string representations
- Package `Profile` provides facilities for determining command error response and log formats

Stop monday

Optional Exercise: Using Environment Facilities

1. Using the `System_Utilities` and `Time_Utilities` packages in a Command window, write a program that measures and reports the elapsed and CPU time for the execution of a fragment of Ada code or procedure.

2. Test the following kinds of constructs:

   — A delay statement for 10 seconds

   — A rendezvous

   — A loop with a large number of iterations calling a simple procedure with a null body

3. Make your previous solution generic with a single parameterless procedure (the one to test). Use the `Table_Formatter` package to format the output.

Seminar Outline

Workspace Management

Environment Command Interface
- Environment Facilities
- Objects and Naming
- Logs and Profiles
- Keymap Modifications
- Tool Building

Development Mechanisms

Basic Subsystems and Configuration Management

Object Characteristics

- All objects have a name, class, and subclass
  - — Classes: Library, Ada, File, Session, and others
  - — Subclasses of class Library: World, Directory, Subsystem, Mailbox, and others
  - — Subclasses of class File: Text, Binary, Switch, Activity, Mail, and others
  - — Subclasses of class Ada: Package_Spec, Package_Body, Generic_Function, and others
- Ada and file objects have versions
  - — New versions are created whenever an object is edited

Object Characteristics (cont.)

- Ada units have four states:

  archived, source, installed, coded

- Archived state
  - — Lower space requirements
  - — No selection allowed
  - — Use `Compilation.Demote` to demote source units to archived
  - — Use Promote to restore units in the archived state to the source state
  - — Used for compact storage of deleted objects and versions

Special Naming Characters

- Provide convenient expression of object names in the Environment
- Specify a name context
  - — Root of the Environment: `!`
  - — Enclosing context: `~`
  - — Enclosing library: `$`
  - — Enclosing world: `$$`
- Specify names of deleted objects
  - — Resolve this name even if it is deleted: `{name}`

Wildcard Characters

- Are used to specify more than one object or to abbreviate parts of a name
- Wildcard character meanings
  - — `#`: A single character in a simple name
  - — `@`: Zero or more characters in a simple name
  - — `?`: Zero or more simple name segments (does not match worlds or their contents)
  - — `??`: Zero or more simple name segments (does match worlds)

/users.mws.baseball-system

?

Wildcard Characters (cont.)

- Examples
  - — `"Input#"`: All simple names with six characters beginning with *Input*
  - — `"In@"`: All simple names beginning with the characters *In*
  - — `"@"`: All simple names
  - — `"!Users.??"`: All objects and their contents inside *!Users*

Additional Naming Constructs

Mange kommandoer kræver denne "sets-notation" istedet for Wild-card

- Specifying sets of names
  - Use the contents of the specified file (or activity): `@Filename`
    indholdet i filename
  - Use the set of names in brackets:

Completion. Make `[name1, name2, ...]`

  - Exclude some names from a set:

```
[names@, ~name7]
```

bare ikke ~name7

- Specifying use of searchlists
  - Search the libraries in the session's searchlist: `\`
  - Example: `"\Projects"` means to find an object called `Projects` in one of the libraries on the current searchlist

package
↓
What.Does

Def ("What") virker ikke da der ikke er nogen konto til Pack What

Def ("\What") virker ☺

Additional Naming Constructs (cont.)

  - Another example: `Def ("\lib");` allows you to move to objects on the current session's searchlist without knowing the full pathname

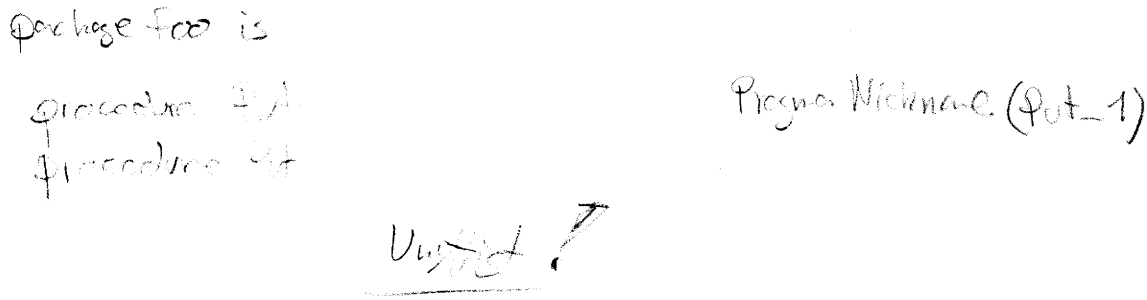
- Ada naming
  - Provides naming for Ada units not on your searchlist
  - Used for execution in Command windows or for binding keys
  - Example: `"!Commands".What.Time;`

Use of Attributes

- Attributes specify subsets of objects: Ada parts, versions, classes, subclasses, Ada unit state, and nicknames
  - Ada unit attributes specify either specifications or bodies of named units: `'spec` or `'body`
  - Version attributes select a particular version of an object: `'V(n)`
  - Class and subclass attributes select specific kinds of objects: `'C(class or subclass name)`
  - State attributes select objects in a specified state: `'S(archived or source or installed or coded)`

Use of Attributes (cont.)

— Nickname attributes allow selecting a specified declaration from a set of overloaded declarations: `'N(nickname)`



Exercise: Using Naming Expressions

1. Use the `Library.Resolve` command with several of the naming constructs introduced in the previous section.

   At a minimum, try the following in a Command window attached to your home library:

   — `Experiment.@`
   — `@'C(World)`
   — `[S@, L@]`
   — `^adv@`
   — `@`
   — `@io`

2. Create a text file in your home library and edit it to list some naming constructs. Submit this file, using the indirect filenaming mechanism, to `Library.Resolve`.

Use of Links in Naming

- Names can be resolved relative to the links in the nearest enclosing world
  - — Symbol used to include links in the resolution of names: `
- Example
  - — `!Users.Advanced_3.Login` refers to a login procedure locally declared in a home library
  - — `Login`` refers to the procedure `Login`, located either in the current context or in the set of links in the nearest enclosing world

Substitution Characters

- Specify how to form target (output) pathnames from wildcard expressions in source strings (input pathnames)
  - — Wildcard characters in source string identify substrings for replacement
  - — Substitution characters in target strings identify modifications to replacement substrings
- Substitution characters specify what part of a wildcard expression to include in the formation of target strings
- Are used primarily with `Library.Copy` and `Archive.Restore` commands

Substitution Characters (cont.)

- Substitution character meanings
  - `@`: Matches zero or more characters in a simple name
  - `#`: Matches a single segment of a pathname
  - `??`: Matches zero or more simple name segments
- Examples
  - Copy all objects in Sue's `Tools` library into Jim's `New_Tools` library:

```
Library.Copy
    (From => "!Users.Sue.Tools.@",
     To   => "!Users.Jim.New_Tools.#");
```

Substitution Characters (cont.)

— Restore all objects in the archive that match the wildcard name in the `For_Prefix` parameter and replace that prefix with `Use_Prefix`:

```
Archive.Restore
    (Objects => "?",
     For_Prefix => "!Users.B@.@_Documents.Chapter@",
     Use_Prefix => "!Users.#.@_New_Documents.#");
```

The new `Use_Prefix` maintains old usernames, renames `@_Document` directories to `@_New_Documents`, and retains all old chapter names

Def. summary

Symbols + attributes

Abbreviations

- Are found in `!Commands.Abbreviations`
  - — Placed on the default searchlist
- Are defined by
  - — Building skins with simpler names
  - — Links to procedures, again with simpler link names
- Commonly used abbreviations
  - — `Def: Common.Definition`
  - — `Send: Message.Send`
  - — `Diff: File_Utilities.Difference`
  - — `Sledit: Search_List.Show_List`
- Users also can define abbreviations

Seminar Outline

Workspace Management

Environment Command Interface
- Environment Facilities
- Objects and Naming
- Logs and Profiles
- Keymap Modifications
- Tool Building

Development Mechanisms

Basic Subsystems and Configuration Management

Log Generation

- Many commands in the Environment produce log output
  - — Compilation commands in package
    `!Commands.Compilation`
  - — Library commands in package
    `!Commands.Library`
- Logs are displayed in the current output window by default
- Types of messages are denoted by three-character symbols
  - — Eight types of Environment messages occur
  - — Additional symbols are provided for user-defined messages
  - — Procedure for adding a message to log:
    `Log.Put_Line(">>message<<")`

Log and I/O Redirection

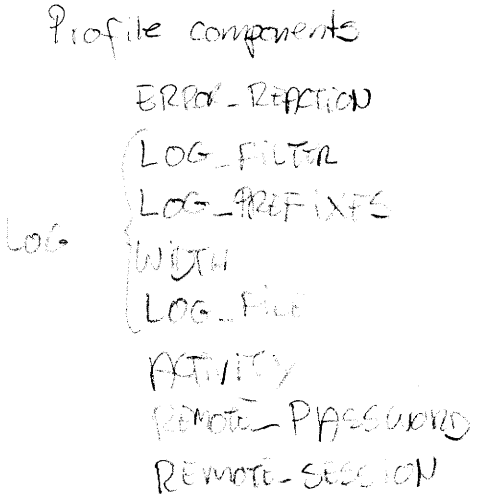
- Logs, and all I/O, can be redirected through commands in `!Commands.Log`
  - — Redirect output to a file: `Log.Set_Output`
  - — Close the output file, saving its contents: `Log.Reset_Output`
- Changes to source or destination of I/O can be saved on a stack
  - — Multiple I/O files can be switched with this stack
- Commands can set, reset, or pop I/O source or destination
- Commands individually control standard input, standard output, or standard error

Log Filters

- Logs can be filtered to remove unwanted types of messages
- Several predefined filter procedures exist in package `!Commands.Log`
  - Get all messages: `Log.Filter`
  - Get only summary messages: `Log.Summarize`
  - Get only error messages: `Log.Filter_Errors`
- Any filter can be created by changing parameter values

Profiles

- Profiles are used by many commands to determine
  - — Response to errors
  - — Format of logs
  - — Filtering of logs
  - — Activity to use (for Rational Subsystems development)



Profiles (cont.)

- One of three different profiles is used for a particular job
  - — Each job can have a specific profile
  - — Each session has a default profile that is used by that session's jobs if a job does not specify another profile
  - — The Environment has a default profile that is used by each session unless a session specifies another profile
- Facilities for manipulating any of the three profiles exist in `!Tools.Profile`
- Session profile can also be changed by editing session switches

Predefined Profiles

- Many Environment commands have `Response` parameters
  - These parameters are often defaulted with `"<PROFILE>"`
  - This default takes the currently defined profile for the job
- Other useful inputs
  - `"<QUIET>"`: Display no output
  - `"<VERBOSE>"`: Display all possible output
  - `"<SESSION_PROFILE>"`: Ignore the current profile and take the profile defined by the current session switches
  - Others are defined in package `!Tools.Profile`

Exercise: Using Logs and Profiles

1. Go to the `Program_Profile_System` directory in your home library.
2. Redirect the output of the `Compilation.Make` command to a file called `Compilation_Log`.
   - Create a Command window.
   - Enter the following Commands and promote:

```
Log.Set_Output ("Compilation_Log");
Compilation.Make;
```

3. Go to the file and verify the output.

Exercise: Using Logs and Profiles (cont.)

4. Return to the Command window and filter the log with:

```
Log.Summarize ("Compilation_Log");
```

   Compare the results with the original log.

5. Modify the log format by changing the prefixes:

   — Return to the Command window.

   — Enter and promote:

```
Profile.Set_Prefixes (Profile.Symbols,
                      Profile.Nil,
                      Profile.Nil);
Compilation.Make;
```

6. Repeat step 5 with your own combination of prefixes.

Seminar Outline

Workspace Management

Environment Command Interface
- Environment Facilities
- Objects and Naming
- Logs and Profiles
- Keymap Modifications
- Tool Building

Development Mechanisms

Basic Subsystems and Configuration Management

Key Bindings

- Users can define or redefine the meaning of any key
  - — Macro definitions
  - — Environment or user-defined procedures
- Bindings can be either temporary or permanent

Temporary Key Bindings

- Change a key binding for the duration of the current session
- Basic method
  - — Identify the procedure to bind to a key
  - — Add a link to the procedure in your home library
  - — Identify the name of the key to which to bind the procedure: Help On Key gives the name of any key
  - — Bind the procedure to the key from a Command window off your home library:

    `Editor.Key.Define`

Permanent Key Bindings

- Two choices for permanent key bindings
  - — Change key bindings at every login via `Login` procedure (simplest method for a few changes)
  - — Modify local `Rational_Commands` procedure (most efficient method for substantial changes)
- `Login` procedure can contain calls to `Editor.Key.Define`
  - — Key binding seems permanent because keys are defined at every login
  - — Keymap changes can be session specific by testing for session name with `System_Utilities.Session_Name` in a login procedure

Permanent Key Bindings (cont.)

- Local `Rational_Commands` procedure is used to create user-customized key bindings
  - Parse `!Machine.Editor_Data.Rational-_User_Commands` into your home library
  - Add case alternatives for new keys that you want to bind
  - Promote procedure to installed when omplete
- Legal key names for all supported terminals are defined in `!Machine.Editor_Data-.Visible_Key_Names`

Systemwide Changes to Key Bindings

- Default keymaps for the entire system are in the library `!Machine.Editor_Data`
  - — Standard Rational Terminal defaults are in the procedure `Rational_Commands`
  - — Other terminals can have default keymaps in this library
- Changing the appropriate procedure in this library changes the default keymap for all users of the system
- Changes take effect when the machine is rebooted

Exercise: Binding Keys

1. Create a new world called `Tools` in your home library.

2. In that world, create a procedure that contains the following statement:

```
Common.Definition
   ("!Users.>>your username<<.Tools");
```

   Substitute your username in the command. Call the procedure whatever you like.

3. Promote the procedure to the coded state when complete.

4. Return to your home library.

5. Temporarily bind this procedure to a key using `Key.Define`.

6. Verify that the key takes you to the `Tools` directory from any other window.

Exercise: Binding Keys (cont.)

7. Permanently bind the same operation to another key by creating your own `Rational_Commands` procedure in your home library.

Seminar Outline

Workspace Management

Environment Command Interface
- Environment Facilities
- Objects and Naming
- Logs and Profiles
- Keymap Modifications
- Tool Building

Development Mechanisms

Basic Subsystems and Configuration Management

Tool Building

- User-defined tools are written in Ada
  - — Language consistency
  - — Full power of Ada
  - — Full power of the Environment for development of tools
- Three major forms
  - — Scripts of commands
  - — System programming tools
  - — Customization of Environment interfaces and products

System Programming Interfaces

- Ada interfaces to I/O, editor, directory system, DIANA
- Package `!Io.Window_Io`
  - — Used for window-based user interfaces
  - — Supports menu, form, box graphics interfaces available in Software Library
  - — Provides access to raw character stream from keyboard
- Packages `!Io.Polymorphic_Sequential_Io` and `!Io.Polymorphic_Io`
  - — Support writing and reading of multiple, user-defined data types
  - — Used for archiving analysis databases for later use

System Programming Interfaces (cont.)

- Package `!Tools.Object_Editor`
  - — Provides access to Environment Editor
  - — Provides pathnames for selections, images, cursor position
  - — Allows tools to work relative to selection, image, cursor
- Package `!Implementation.Diana`
  - — Is used for building analysis tools
  - — Provides access to semantic resolution of the compilation system
  - — Package `Lrm_Interfaces` provides higher-level interface
- Package `!Tools.Link_Tools`
  - — Is used for traversing and decomposing the links associated with a world

System Programming Interfaces (cont.)

- Package `!Tools.Directory_Tools`
  - — Provides programmatic interface to the Environment library system
  - — Defines type `Object.Handle` similar to `Text_Io.File_Type`
  - — Defines type `Object.Iterator` for lists of handles
  - — Resolves pathnames to handles/iterators
  - — Provides traversal to parent, enclosing world, subunits
  - — Provides size and update statistics
  - — Generates Ada unit dependencies

Releasing Tools

- Various methods include
  - — Copy program into "release" library
  - — Add skin to "release" library
  - — Keep program in place and add searchlist entry
  - — Add link to a library already on the searchlist whose links are also searched
- Use of `Library.Freeze` procedure prevents inadvertent changes
  - — Frozen units can be viewed but not modified

Releasing Tools (cont.)

- Use of `pragma Main` improves performance
  - — Applies only to library subprograms
  - — Saves link time when executed
  - — Still requires loading
  - — Main programs are demoted to installed when any unit in their closure is demoted
- Create loaded main programs:
  `Compilation.Load`
  - — Loaded image is stored with an object of subclass `Loaded_Main`
  - — Similar to *.exe* objects on other systems
  - — Unit remains executable even if changes to closure are made
  - — Only subprogram specification is visible

Exercise: Using Directory Tools

1. In a Command window off your home library, enter the following code:

```
declare
    package Object renames Directory_Tools.Object;
    package Naming renames Directory_Tools.Naming;

    Iter : Object.Iterator;
    An_Object : Object.Handle;
begin
    Iter := Naming.Resolution ("@");
    while not Object.Done (Iter) loop
        An_Object := Object.Value (Iter);
        Io.Put_Line (Naming.Simple_Name (An_Object));
        Object.Next (Iter);
    end loop;
end;
```

2. Enter other naming expressions for the input to `Naming.Resolution`.

3. Try using the `Unique_Full_Name` function from the `Naming` package in directory tools.

4. Try some of the functions in the `Statistics` package.

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms

- Coding Aids
  Subunits
  Debugger Operations
  Link Operations
  Incremental Operations
  Testing
  Reusable Components

Basic Subsystems and Configuration Management

Templates

- Create template for unit body: Create Body
  - — Applies to library units or individual declarations
  - — Works on nested declarations in bodies
  - — Works on declarations added via incremental operations
  - — Handles arbitrary nesting and subunits
  - — Spec can be in any state but archived
  - — Unit name must be registered in library
- Create template for package private part: Create Private
  - — Creates template for all private types and deferred constants in package spec
  - — Spec must be in source state

Incomplete Programs

- Can be saved permanently (even with syntactic and semantic errors) with Enter
- Packages can be coded and executed with `[statement]` prompts
- Incomplete programs are typically used instead of stubbing a unit
  - Need not make all subprograms legal Ada units
  - Can easily find remaining `[statement]` prompts
- Execution of a `[statement]` prompt results in `Program_Error` exception

Registration of Unit Names in Libraries

- Register the unit name by promoting the unit
  - — This will not work if the unit contains semantic errors
- Register the unit name without promoting the unit: `Install_Stub`
  - — The unit name must be in the library for some commands
  - — The unit name should be registered as soon as reasonable

Modification of Unit Names

- Ada units and text objects can be renamed with the `Library.Rename` procedure
- A unit can also be withdrawn from the library to change its name
  - Unit kind and parameter profile can be changed without withdrawal
- Basic method
  - Select a unit to change and withdraw:

    Withdraw
  - Order is critical: bodies before specs and so on
  - Use `Install_Stub` or Promote when the name change is done

GODT!

Changes in Identifiers

- Change only identifiers, not substrings or comments: `Replace_Id`
  - — Operates on a selected part of a unit
  - — Example: Need to change index variable in *for* loop from `I` to `Index`

byte i for loop as procedure name!

- Basic method
  - — Ada unit must be in source state
  - — Select region of Ada unit
  - — Press Create Command, enter `Replace_Id`, and press Complete
  - — Provide old and new identifier names

Exercise: Using Coding Aids

1. Create a new package called `New_Package` in the `Experiment` world off your home library.
2. Add a type declaration called `My_Type` and several procedures and functions that reference it.
3. Install the unit in the library with `Install_Stub`.
4. Rename the package to `My_Type_Package` with `Library.Rename`.
5. Use `Replace_Id` to rename `My_Type` to `My_New_Type`.
6. What happens to the substring `My_Type` in the package name?

Conflicts in Error Displays

- Mixture of old and new syntactic errors can hide actual problems
  - — Unit has syntactic or semantic errors
  - — In fixing old errors, new syntactic errors are introduced
  - — Both sets of errors remain displayed
- Commands
  - — Remove all underlines: Underlines Off F10
  - — Redisplay underlines to display only new errors: Show Errors F10

Show Usage

- Provides another browsing facility complementing Definition
- Display usages of any selected identifier: Show Usage
  - — Underlines usages in the current unit if all usages are local to the unit
  - — Show Usage (Unit) underlines only usages in the current unit
  - — Show Usage (Indirect) includes indirect references via RENAME
  - — Creates a menu of units that have usages of the identifier if the usages occur in more than one unit
- Underline declarations that are not referenced: Show Unused

Exercise: Using Show Usage

1. Go to the library `Experiment` in your home library.
2. Code the library if it is not already consistent.
3. Get the definition of `Unit's spec`.
4. Select the identifier `Statistics` and press Show Usage (Unit).

   Notice that each local usage is underlined.
5. Select the first underlined `Statistics` and press Show Usage.

   Notice that this time a menu appears with several units listed. Each unit contains at least one reference to the selected identifier.

Exercise: Using Show Usage (cont.)

6. In the xref menu, place the cursor on a unit name and press Definition.

   Notice that the unit appears with all references to `Statistics` underlined.

Xref Tool

- Provides noninteractive cross-reference listings
- Commands in `!Tools.Xref_Utility`
  - — List all usages: `Xref.Used_By`
  - — List the defining occurrences: `Xref.Uses`
- Boolean parameters specify which references are included in the list

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms
- Coding Aids
- Subunits
- Debugger Operations
- Link Operations
- Incremental Operations
- Testing
- Reusable Components

Basic Subsystems and Configuration Management

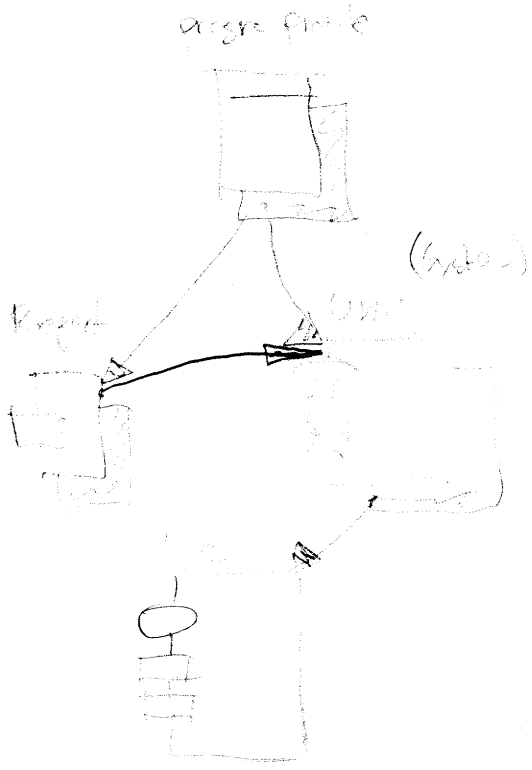
Subunit Operations

- Subunits can be created from an Ada stub
  - — Ada code is entered for stub (for example, `procedure Report is separate;`)
  - — Selecting and editing the stub creates the corresponding subunit: Edit
- Subunits also can be created by extracting code from the enclosing unit: Make Separate
  - — Enclosing unit must be source or installed
  - — Subunit is created in the source state
  - — Subunits cannot be overloaded (LRM rule)

Subunit Operations (cont.)

- Subunits appear in library displays as units nested under parent units
  - — Example

```
Some_Package
   .A_Subunit
     .Another_Nested_Subunit
   .Another_First_Level_Subunit
```



Subunit Management

- Subunits can be merged back into parent unit: Make Inline MAKE_INLINE
  Select separate_unit / create_command/ MAKE_INLINE
  - Parent unit must be source or installed
  - Subunit can be in any state
  - Merging may fail because of syntactic or semantic errors in subunit
- Subunits have a compilation state separate from their parent unit
  - Subunits must be installed after parent unit
  - Code (This World) can be used to complete coding of subunits in the correct order

Exercise: Using Coding Aids and Subunits

An incomplete version of the `Program_Profile` program is provided in the `Subunit_Exercise` world in your home library. Complete the program so that it exhibits the following characteristics:

— The package `System` is renamed to `Unit`. All existing references to `System` must be changed to `Unit` as well.

— The actual private type definition for `Statistics` in `Unit'spec` should be implemented in the private part as:

```
type Statistics is
    record
        Name     : String (1..100);
        Lines    : Natural := 0;
        Comments : Natural := 0;
    end record;
```

Exercise: Using Coding Aids and Subunits (cont.)

— `Line.Locate_Comment` is a subunit.

— The code for `Line.Kind'body` is in-line and completed as in the following:

```
if Clean_Line'Length = 0 then
    return Blank;
elsif Comment_Location = 1 then
    return Comment_Only;
else
    return Other_Line;
end if;
```

— `Line.Strip_Blanks` is added as a subunit local to the package body. The code for the subprogram is:

```
function Strip_Blanks (The_String : String) return
String is
begin
    for I in The_String'Range loop
        if The_String (I) /= ' ' then
            return The_String (I..The_String'Last);
        end if;
    end loop;
    return "";
end Strip_Blanks;
```

Exercise: Using Coding Aids and Subunits (cont.)

— `Unit.Analyze'body` is a subunit.

— All program units are coded.

— The `Program_Profile` program executes correctly using `Test_Input1` as a test case.

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms
- Coding Aids
- Subunits
- Debugger Operations
- Link Operations
- Incremental Operations
- Testing
- Reusable Components

Basic Subsystems and Configuration Management

Execution Commands

- Several forms
  - — Single-step: Run
  - — Run free until breakpoint or exception raise: Execute
  - — Execute entire local statement: Run Local
  - — Run until the current subprogram returns: Run Returned

Additional Debugger Commands

- Display the current source position: Show Source
- Display the current value of the selected identifier: Put
- Display the current execution stack: Show Stack
- Set a breakpoint at the selected statement: Break
- Activate all defined breakpoints: Activate
- Remove a breakpoint: Remove Breaks
- Propagate a named or selected exception: Propagate

Reference summery
tools.
Commands
C 27
Package Debug
C25
procedure Set_value
for non
ex:
Pointer_level address
--etc!

Debugger Options

- The Debugger contains options for altering
  - — Format of output
  - — Handling of stacks, tasks, and breakpoints
  - — Amount of detail in displays of objects
- Commands
  - — Change options specified in `type Option`: `Debug.Enable`
  - — Change options specified in `type Numeric`: `Debug.Set_Value`
  - — Three key bindings for `Debug.Set_Value`: Set Pointer Level, Set Element Count, Set First Element
  - — Change all other options: `Debug.Flag`

Exceptions

- The Debugger provides control of exception handling
  - — Stop when any exception is raised
  - — Stop when a specified exception is raised
  - — Ignore an exception when it is raised
  - — Stop or ignore an exception when raised by a specified task at a specified location
- Commands
  - — Stop when exception is raised:
    `Debug.Catch` (Catch)
  - — Propagate exception to enclosing scope:
    `Debug.Propagate` (Propagate)
  - — Ignore exception when raised:
    `Debug.Forget` (Forget)

Selection

- Selection in Ada units can identify the argument for Debugger commands
  - — For [Put] and [Modify], use selection of object identifiers
  - — For [Propagate], use selection of exception declarations and explicit *raise* statements
- Selection also works on output in the Debugger window
  - — Go to Ada unit for specific stack frame selection: [Show Source]
  - — Display the parameter values for a selected frame: [Put]
  - — Go to object declaration for a selected `Put` or `Modify` command: [Show Source]
  - — Dereference selected access values displayed in the Debugger window: [Put]

Multiple Tasks

- The Debugger provides control of individual tasks
  - Any task can be executed, stopped, single-stepped, displayed, or viewed
  - Task control can be done either asynchronously or synchronously across all tasks in a program
  - Exceptions, objects, breakpoints, stacks, traces, and histories can be specified or controlled on a per-task basis
- Commands
  - Display status of tasks: `Debug.Task_Display`
  - Stop individual task: `Debug.Stop`
  - Change task control model: `Freeze_Task` flag

Multiple Tasks (cont.)

- Control context
  - — The Debugger defaults to interacting with the main program task
- Basic commands work relative to the control context
  - — Single-step execution
  - — Source display
- Set the control context to another task:
  `Debug.Context`
  - — Need to specify task name
  - — Hex name taken from `Debug.Task_Display`

Special Displays

- Debugger allows users to define their own display routines for values of types
- Basic method
  - — Define an `Image` function (returning a string) for the type
  - — Instantiate the generic `Register` procedure in `!Tools.Debug_Tools`
  - — Call the `Register` procedure in the elaboration of the code of the program

Exercise: Using Debugger Selection

1. Go to the `Program_Profile_System` directory in your home library.
2. Execute `Test_Driver1` under control of the Debugger.
3. Single-step five times: 5 Run.
4. Select the case statement inside the `Analyze` function and set a breakpoint.
5. Execute to the breakpoint and single-step once more.
6. Display the stack.

Exercise: Using Debugger Selection (cont.)

7. Select the fourth stack frame and press Put.
8. Select the second stack frame and press Show Source.
9. Select the reference to the object `Unit_line` and press Put.
10. Select `Put ("%ROOT_TASK._3.UNIT_LINE");` in the Debugger window and press Show Source.

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms
- Coding Aids
- Subunits
- Debugger Operations
- Link Operations
- Incremental Operations
- Testing
- Reusable Components

Basic Subsystems and Configuration Management

Link Operations

- View a world's links: `Links.Display`
- Edit a world's links: `Links.Edit`
- Copy links from one world to another: `Links.Copy`
- Delete the selected link: Object - D or Object - K
- Insert a link: `Links.Add` or Object - I
- Change the source of the selected link: Edit
- Expand the display of links: Object - !
- Contract the display: Object - .
- View the source of the current link: Definition

Link Operations (cont.)

- Obtain information on the uses of a link:
  Explain ~~Explain~~ ?
  - — Adds comment lines below link showing units that utilize the link
  - — Removes comments when pressed again

Exercise: Using Link Operations

1. Execute `Links.Edit` on the `Experiment` world in your home library.
2. Place the cursor on the link for the `Line` unit and press [Definition].
3. Return to the links display and try the definition on an external link.
4. Press [Object] - [!] several times. Notice that the display returns to its original configuration after several changes.
5. Press [Object] - [.] several times. Notice that the links displayed return to the full set after several changes.
6. Try [Explain] on several links.
7. Delete the link for the `Line` unit.
8. Try to delete the link for `Text_Io`.

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms
- Coding Aids
- Subunits
- Debugger Operations
- Link Operations
- Incremental Operations
- Testing
- Reusable Components

Basic Subsystems and Configuration Management

Characteristics of Incremental Operations

- Are used to make changes to semantically valid programs to reduce turnaround time
  - — Add one or more statements, declarations, or comments
  - — Modify a statement, declaration, or comment
  - — Delete a statement, declaration, or comment
- Allow changes/additions of declarations to coded specifications
- Allow additions/changes to comments in coded units
- Allow changes to declarations and statements in bodies in the installed state

Obsolescence Menus

- List first level of units in all libraries made obsolete by attempted change
  - — Does not include full transitive closure
- Commands
  - — Change state of a listed unit: [Promote], [Demote], [Install], or [Source]
  - — View a listed unit: [Definition]
  - — Edit a listed unit: [Edit]
  - — Expand to see full pathname: [Object] - [!]
  - — Return to simple pathname: [Object] - [.]

Obsolescence Menus (cont.)

- Useful for incrementally modifying declarations in package specifications
- Basic method
  - Select the declaration to modify
  - Press [Edit] to get an obsolescence menu
  - Use menu demotion to remove dependencies
  - Extract the declaration ([Edit]) to make the change
  - [Promote] to replace modified declaration
  - Make consistent with [Code (All Worlds)]

Controls on Extent of Compilation Commands

- The `Promote_Scope` parameter limits the units affected by the compilation commands `Make` and `Promote`
  - — Value `Single_Unit` allows compilation of unit only
  - — Value `Unit_Only` allows compilation of unit spec and body and units in *with* clauses
  - — Value `Subunits_Too` allows compilation of unit, its subunits, and units in *with* clauses
  - — Value `All_Parts` allows compilation of unit and its body, its subunits, and units in *with* clauses

Controls on Extent of Compilation Commands (cont.)

- The `Limit` parameter controls the libraries affected by the compilation commands `Make`, `Promote`, and `Demote`
  - — Value `<DIRECTORIES>` allows changes only to units in current library
  - — Value `<WORLDS>` allows changes to units in enclosing world
  - — Value `<ALL_WORLDS>` allows compilations to cross world boundaries via external links

Controls on Extent of Compilation Commands (cont.)

- The `Effort_Only` parameter specifies whether to estimate the effort for a compilation or to do the compilation for the compilation commands `Make` and `Demote`
  - — False value executes compilation (the default)
  - — True value estimates compilation effort

Additional Key Bindings

- Additional key bindings have been defined with various forms of the compilation control parameters
  - — Code (This World) / Install (This World)
    Codes/installs all units within the current world
  - — Code (All Worlds) / Install (All Worlds)
    Codes/installs the transitive closure even if units are in other worlds
  - — Source (This World) / Uncode (This World)
    Demotes the transitive closure of all dependent units to the source/installed state within the current world only
  - — Source (All Worlds) / Uncode (All Worlds)
    Demotes the transitive closure of all dependent units to the source/installed state within all worlds

Exercise: Using Incremental Operations

Add the use of the package `String_Utilities` to the `Program_Profile` program using incremental operations in the `Incremental_Exercise` world.

1. Replace `Line.Locate_Comment` and `Line.Strip_Blanks` with comparable functions from package `String_Utilities`.

Exercise: More Incremental Operations

Rename the function `Analyze` in the package `Unit` to `Analyze_Statistics`.

1. Select the declaration of the function `Analyze` and press Edit to get an obsolescence menu.
2. Demote each dependency by selecting each dependent unit and pressing Source Unit.
3. Incrementally extract the function `Analyze` with Edit and change the name.
4. Promote the declaration back into the specification.
5. Change any references to the function in the demoted units.
6. Make the world consistent.

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms
- Coding Aids
- Subunits
- Debugger Operations
- Link Operations
- Incremental Operations
- Testing
- Reusable Components

Basic Subsystems and Configuration Management

Simple Automation of Testing

- Create test drivers
  - — Create prototype in Command window
  - — Move to library unit when stable
- Regression approach to testing
  - — Create correct results by hand *or*
  - — Manually verify the output of some test run
  - — Use file comparison operations in package `File_Utilities` to determine if program generated correct results

Use of File Utilities

- Package `File_Utilities` provides facilities for comparing and manipulating files such as test results
- Commands
  - Find differences between files: `Difference`
  - Merge two differing files: `Merge`
  - Append one file to another file: `Append`
  - Find a specified pattern in a file: `Find`
  - Check whether two files are identical: `Equal`

Exercise: Testing

Test the `Program_Profile` program in the `Program_Profile_System` world, previously completed. Automate the test process so regression testing can be done.

1. Go to the `Program_Profile_System` world in your home library.
2. Set up a golden results file by running `Program_Profile` on `Test_Input_1` units and copying the output into a `Golden_Results` file.
3. In a Command window, create a test driver that runs the `Program_Profile` program on `Test_Input_1` and compares the results to those in `Golden_Results`. The test driver should then print out a passed or failed message to the terminal.

Exercise: Testing (cont.)

4. When the test driver in the Command window works, create a library unit containing the driver.
5. Execute the library unit version of the test driver.
6. Modify the test driver to take a parameter that is the name of the unit to have statistics collected on its line characteristics. You will also need to generalize the rest of the test driver to operate on the parameter rather than `Test_Input_1`.

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms
- Coding Aids
- Subunits
- Debugger Operations
- Link Operations
- Incremental Operations
- Testing
- Reusable Components

Basic Subsystems and Configuration Management

Reusability

- Reuse of software can dramatically increase productivity
- `!Tools` provides numerous reusable packages
  - Environment execution is achieved by adding a link and a *with* clause in the standard way
  - Cross-Development Facilities provide bodies for inclusion in target applications
- Two kinds of reusable components are
  - Generic data abstractions
  - Utilities
- Rational Software Library provides
  - Public domain software
  - Tools developed by Rational technical representatives

Data Abstractions

- Packages in `Tools`
  - `Set_Generic`: Unordered sets
  - `List_Generic`: Ordered LISPlike lists
  - `Stack_Generic`: First in, last out
  - `Queue_Generic`: First in, first out
  - `Bounded_String`: Arbitrary-length strings with maximum
  - `Unbounded_String`: No maximum; uses heap allocation

Data Abstractions (cont.)

- `Map_Generic`: Maps domain values to range values
- `Concurrent_Map_Generic`: Supports concurrent access
- `String_Map_Generic`: Domain type is `String`
- `String_Table`: Table lookup for `String` values

Utilities

- Utility packages in `Tools`
  - `String_Utilities`: Manipulates `String` values
  - `Table_Formatter`: Formats data in labeled columns
  - `Table_Sort_Generic`: Sorts a table of objects according to a user-defined "<" function
  - `Time_Utilities`: Manipulates `type Time` and `type Duration`

Seminar Outline

Workspace Management

Environment Command Interface

Development Mechanisms

Basic Subsystems and Configuration Management

- Project Management Issues
  Project Structuring with Subsystems
  Subsystem Construction
  Basic Development Methodology
  Source Reservation with CMVC
  Parallel Development with Subpaths